Keith Cooper
John Mellor-Crummey
Vivek Sarkar (Eds.)

LNCS 6548

Languages and Compilers for Parallel Computing

23rd International Workshop, LCPC 2010
Houston, TX, USA, October 2010
Revised Selected Papers

Springer

Lecture Notes in Computer Science 6548

Keith Cooper John Mellor-Crummey
Vivek Sarkar (Eds.)

Languages and Compilers for Parallel Computing

23rd International Workshop, LCPC 2010
Houston, TX, USA, October 7-9, 2010
Revised Selected Papers

Springer

Volume Editors

Keith Cooper
Rice University, Department of Computer Science
6100 Main Street, Houston, TX 77005-1892, USA
E-mail: keith@rice.edu

John Mellor-Crummey
Rice University, Department of Computer Science
6100 Main Street, Houston, TX 77005-1892, USA
E-mail: johnmc@cs.rice.edu

Vivek Sarkar
Rice University, Department of Computer Science
6100 Main Street, Houston, TX 77005-1892, USA
E-mail: vsarkar@rice.edu

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-19594-5 e-ISBN 978-3-642-19595-2
DOI 10.1007/978-3-642-19595-2
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011922053

CR Subject Classification (1998): D.1.3, C.2.4, D.4.2, H.3.4, D.2

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

It is our pleasure to present the papers accepted for the 23rd International Workshop on Languages and Compilers for Parallel Computing (LCPC), held during October 7–9, 2010 at Rice University in Houston, Texas, USA. Since 1988, the LCPC workshop has emerged as a major forum for sharing cutting-edge research on all aspects of parallel languages and compilers, as well as related topics including runtime systems and tools. The scope of the workshop spans foundational results and practical experience, and targets all classes of parallel platforms including concurrent, multithreaded, multicore, accelerated, multiprocessor, and cluster systems. Given the rise of multicore processors, LCPC is particularly interested in work that seeks to transition parallel programming into the computing mainstream. This year's LCPC workshop was attended by 66 registrants from across the globe. There were two co-located events held during October 5–6, 2010, prior to the start of LCPC — the Workshop on Language-level Approaches for Retargetable High-Performance Computational Science Applications, and the Second Annual Concurrent Collections Workshop.

This year, the workshop received 47 submissions with coauthors from 16 countries — Australia, Brazil, Canada, China, France, Germany, India, Israel, Japan, Netherlands, Poland, Spain, Sweden, Switzerland, the UK, and the USA. Of these submissions, the program committee selected 18 papers for presentation at the workshop, representing an acceptance rate of 38%. Each selected paper was presented in a 30-minute slot during the workshop. In addition, 9 submissions were selected for presentation as posters during a 90-minute poster session. Each submission received at least three reviews. The program committee held an all-day meeting on August 26, 2010 to discuss the papers. Whilst each paper was discussed, PC members who had a conflict of interest with the paper were asked to temporarily leave the meeting. Decisions for all PC-authored submissions were made by PC members, who were not coauthors of any submissions.

We were fortunate to have two keynote speakers at this year's LCPC workshop. Keshav Pingali, who holds the W.A.“Tex” Moncrief Chair of Grid and Distributed Computing at UT Austin, gave a talk titled “Why Compilers Have Failed, and What We Can Do about It”. His talk summarized major accomplishments of compiler research in the past 25 years, as well as lessons learned from major failures of compilers during that period. After presenting insights gained from his group's Galois project, Keshav challenged the LCPC community with a research goal to build a general-purpose automatically parallelizing compiler system for future 1000-core processors. Steve Wallach, Chief Scientist, Co-founder and Director of Convey Computers, gave a talk titled “Computer Software: The ‘Trojan Horse’ of HPC”. He highlighted the increasing gap between advances in hardware technologies such as heterogeneous accelerators and setbacks in their accompanying software, and observed that ultimately hardware that is easier to program will win over hardware that is more difficult to program. After discussing the approach being taken by Convey Computers with application engines that implement application-specific instructions, Steve concluded that heterogeneous processors are here to stay and that smarter compilers must be developed to ensure that they can be programmed more easily than with current approaches. The keynote talks, technical sessions, and poster session led to several interesting discussions among participants during breakfasts, lunches, coffee breaks, evening receptions, and the workshop banquet held on October 7, 2010 evening at the Houston Museum of Natural Science.

We would like to conclude by thanking the many people whose dedicated time and effort helped make LCPC 2010 a success. The hard work invested by program committee members and external reviewers in reviewing the submissions helped ensure a high-quality technical program for the workshop. The steering committee members and the LCPC 2009 organizing committee provided valuable guidance and answered questions that arose during preparations for LCPC 2010. All participants in the workshop contributed directly to the technical vitality of the event either as presenters or as audience members. We would also like to thank workshop sponsors ET International and Reservoir Labs for their financial support. Finally, the workshop would not have been possible without the tireless efforts of the entire local arrangements team at Rice University — Kathryn O'Brien, Darnell Price, Marilee Dizon, Jennifer Harris, Karen Lavelle, and Amanda Nokleby.

October 2010

Keith Cooper
John Mellor-Crummey
Vivek Sarkar

Organization

LCPC 2010 was organized by the Department of Computer Science at Rice University.

Steering Committee

Rudolf Eigenmann	Purdue University, USA
Alex Nicolau	University of California at Irvine, USA
David Padua	University of Illinois at Urbana-Champaign, USA
Lawrence Rauchwerger	Texas A&M University, USA

Program Committee

Jose Nelson Amaral	University of Alberta, Canada
John Cavazos	University of Delaware, USA
Barbara Chapman	University of Houston, USA
Keith Cooper	Rice University, USA
Guang R. Gao	University of Delaware, USA
Xiaoming Li	University of Delaware, USA
Calvin Lin	University of Texas at Austin, USA
John Mellor-Crummey	Rice University, USA
Sanjay Rajopadhye	Colorado State University, USA
Lawrence Rauchwerger	Texas A&M University, USA
Vivek Sarkar	Rice University, USA
Michelle Strout	Colorado State University, USA

Proceedings Chair

Jun Shirako	Rice University, USA

Web and Publicity Chair

Zoran Budimlić	Rice University, USA

Sponsoring Institutions

ET International, Inc., USA
Reservoir Labs, Inc., USA

Table of Contents

McFLAT: A Profile-Based Framework for MATLAB Loop Analysis and Transformations*

Amina Aslam and Laurie Hendren

School of Computer Science, McGill University, Montreal, Quebec, Canada
amina.aslam@mail.mcgill.ca, hendren@cs.mcgill.ca

Abstract. Parallelization and optimization of the MATLAB programming language presents several challenges due to the dynamic nature of MATLAB. Since MATLAB does not have static type declarations, neither the shape and size of arrays, nor the loop bounds are known at compile-time. This means that many standard array dependence tests and associated transformations cannot be applied straight-forwardly. On the other hand, many MATLAB programs operate on arrays using loops and thus are ideal candidates for loop transformations and possibly loop vectorization/parallelization.

This paper presents a new framework, MCFLAT, which uses profile-based training runs to determine likely loop-bounds ranges for which specialized versions of the loops may be generated. The main idea is to collect information about observed loop bounds and hot loops using training data which is then used to heuristically decide upon which loops and which ranges are worth specializing using a variety of loop transformations.

Our MCFLAT framework has been implemented as part of the McLAB extensible compiler toolkit. Currently, MCFLAT, is used to automatically transform ordinary MATLAB code into specialized MATLAB code with transformations applied to it. This specialized code can be executed on any MATLAB system, and we report results for four execution engines, Mathwork's proprietary MATLAB system, the GNU Octave open-source interpreter, McLAB's McVM interpreter and the McVM JIT. For several benchmarks, we observed significant speedups for the specialized versions, and noted that loop transformations had different impacts depending on the loop range and execution engine.

1 Introduction

MATLAB is an important programming language for scientists and engineers [17]. Although the dynamic nature and lack of static type declarations makes it easy to define programs, MATLAB programs are often difficult to optimize and parallelize. The McLAB system [2] is being defined to provide an open and extensible optimizing and parallelizing compiler and virtual machine for MATLAB and extensions of MATLAB such as ASPECTMATLAB [7]. As an important part of McLAB, we are developing a framework for loop dependence tests and loop transformations, MCFLAT, which is the topic of this paper.

* This work was supported, in part, by NSERC.

K. Cooper, J. Mellor-Crummey, and V. Sarkar (Eds.): LCPC 2010, LNCS 6548, pp. 1–15, 2011.

Due to the dynamic nature of MATLAB, there is very little static information about array dimensions and loop bounds. Furthermore, many of the scientific codes written in MATLAB can be applied to very different sized data sets. Thus, our design of MCFLAT is based on a profiling phase which collects information about loop bounds over many different runs. We then have a heuristic engine which identifies important loop bound ranges and then a specializer which produces specialized code for each important range. The specializer applies loop dependence tests and loop transformations specific to the input range. Currently, for each important range, we exhaustively generate all legal specializations, but the ultimate goal is to combine this framework with a machine learning approach which will automatically generate a good specialization for the given range.

This paper describes our initial design and implementation of MCFLAT and provides some exploratory experimental data obtained by using MCFLAT to generate different versions of code which we execute on four different systems, Mathworks' MATLAB implementation (which includes a JIT), the GNU Octave open-source interpreter [1], our McVM interpreter and our McVM JIT [13]. Interestingly, this shows that different optimizations are beneficial for different ranges and on different MATLAB execution engines. This implies that specialization for both the range and intended execution engine is a good approach in the context of MATLAB.

The remainder of this paper is organized as follows. In Section 2 we give a high-level view of MCFLAT, and in Section 3 we provide more details of each important component. We apply our framework to a selection of benchmarks and report on the experimental results in Section 4. Finally, we discuss related work in Section 5 and conclude in Section 6.

2 Overview of Our Approach

The overall structure of the MCFLAT framework is outlined in Figure 1. Our ultimate goal is to embed this framework in our McJIT system, however currently it is a standalone source-to-source framework which uses the McLAB front-end. The user provides both the MATLAB program which they wish to optimize and a collection of representative inputs (top of Figure 1). The output of the system is a collection of specialized programs (bottom of Figure 1), where each specialized program has a different set of transformations applied. The system also outputs a dependence summary for each loop, which is useful for compiler developers.

The design of the system is centered around the idea that a MATLAB program is likely to be used on very different sized inputs, and hence at run-time loops will have very different loop bounds. Thus, our objective is to find important ranges for each loop nest, and to specialize the code for those ranges. Knowing the ranges for each specialization also enables us to use very fast and simple dependence testers.

The important phases of MCFLAT, as illustrated in Figure 1, are the *Instrumenter*, which injects the profiling code, the *Range Estimator* which decides which ranges are important, and the *Dependence Analyzer and Loop Transformer Engine*. In the next section we look at each of these components in more detail.

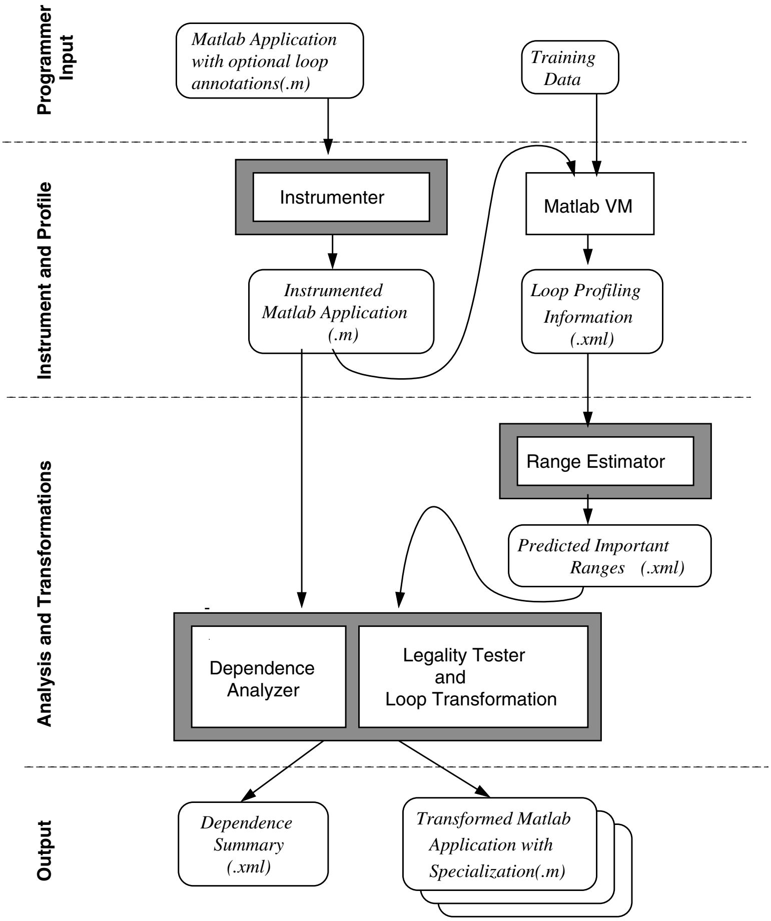

Programmer Input
Matlab Application with optional loop annotations(.m)
Training Data
Instrument and Profile
Instrumenter
Matlab VM
Instrumented Matlab Application (.m)
Loop Profiling Information (.xml)
Analysis and Transformations
Range Estimator
Predicted Important Ranges (.xml)
Dependence Analyzer
Legality Tester and Loop Transformation
Output
Dependence Summary (.xml)
Transformed Matlab Application with Specialization(.m)

Fig. 1. Structure of the McFLAT Framework

3 Important Components of McFLAT

In this section we provide an overview of the key components of our McFLAT framework, and we briefly discuss parallel loop detection and some current limitations of the framework.

3.1 Instrumenter

As illustrated in the phase labeled Instrument and Profile in Figure 1, the *Instrumenter* component is used to automatically inject instrumentation and profiling code into a MATLAB source file. This injection is done on the high-level structured IR produced by the McLAB front-end. In particular, we inject instrumentation to associate a unique loop number to each loop, and we inject instrumentation to gather, for each loop, the lower bound of the iteration, the loop increment, the upper bound of the iteration, the nesting level of the loop, the time spent executing the loop, and a list of variables that are written to in the loop body.

The MATLAB program resulting from this instrumentation is functionally equivalent to the original code, but emits additional information that generates training data required for the next phase.

When the instrumented program is executed using a MATLAB virtual machine, the profile information is written to an .xml file. This .xml file is persistent, and so multiple runs can be made, and each run will add new information to the .xml file. The loop profiling information .xml file is then used as an input to the next component.

3.2 Range Estimator

The *Range Estimator* is the first important component of the main part of MCFLAT, the Analysis and Transformations phase in Figure 1. The Range Estimator reads the loop profiling information and determines which are the important ranges for each loop. The important ranges are identified using Algorithm 1. The input to this algorithm is a hash table containing all the observed values for all the loops and the output is a list of important ranges. The basic idea is that for each loop, we extract the observed values for that loop, partition the value space into regions and subregions, and then identify subregions which contain more values than a threshold.

Algorithm 1. Algorithm for range estimation

```
Data Items
H (K,V) : Hash table with loop numbers as keys and list of observed values
Procedure processLoopData(LoopID)
l ← lookup(LoopID, H) // get all observed values for loop with LoopID
sort(l)
importantRanges ← empty
R ← computeRegions(min(l), max(l))
// for each large region
for all r in R do
   // for each subregion (divide R into 10 equal parts)
   for all sR in R do
      if numInRegion(l,sR) ≥ threshold then
         PredVal ← maxval(sR)
         add PredVal to importantRanges
      end if
   end for
end for
return(importantRanges)
```

We determine the regions and subregions as illustrated in Figure 2. The regions are powers of 10, starting with the largest power of 10 that is less than the smallest observed value, and ending with the smallest power of 10 that is greater than the highest observed value. For example, if the observed upper bounds were in the range 120 to 80000, then

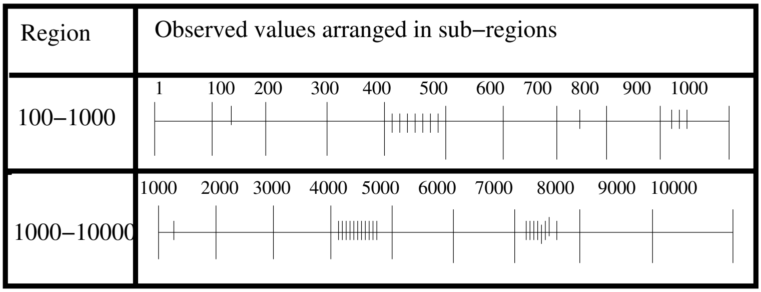


Fig. 2. Pictorial Example of Ranges and Subranges

we would choose regions of size 100, 1000, 10000 and 100000. Each region is further subdivided into 10 subregions. A subregion is considered important if the number of observed values are above a threshold, which can be set by the user. For our experiments we used a threshold of 30 % . When an important region is identified, the maximum observed value from the region is added to the list of important ranges.

3.3 Dependence Analysis

During this phase, McFLAT calculates dependences between all the statements in the loop body against all the predicted important ranges for that loop. It maintains various data structures supporting dependence analysis. This information is used in subsequent loop transformation phases.

The data dependence testing problem is that of determining whether two references to the same array within a nest of loops may reference to the same element of that array [21,4].

Since we have identified the upper loop bounds via our profiling, we have chosen very simple and efficient dependence testers: the *Banerjee's Extended GCD(Greatest Common Divisor) test* [8] and the *Single Variable Per Constraint Test* [4]. Currently, we have found these sufficient for our small benchmarks, but we can easily add further tests as needed.

3.4 Loop Transformations

In our framework programmers can either suggest the type of transformation that they need to apply through optional loop annotations, or it will automatically determine and apply a transformation or a combination of transformations which are legal for a loop.

McFlat implements following loop transformations that have been shown to be useful for two important goals, parallelism and efficient use of memory hierarchy [15]: *loop interchange* and *loop reversal*. For automatic detection and application of above mentioned loop transformations, we use the unimodular transformation model presented in [20]. Loop interchange and reversal are modeled as elementary matrix transformations, combinations of these transformations can simply be represented as product of elementary transformation matrices. An elementary transformation or a compound transformation is considered to be legal if the transformed distance vectors are lexicographically positive.

Apart from automatically testing the legality of loop interchange and reversal, our framework supports a larger set of transformations which can be specified by the user. This allows us to use our system as a testbed for programmers with which they can suggest different transformations and observe the effect of different transformations on different loops. Programmers just have to annotate the loop body with the type of transformation that they need to apply on the loop. Our framework checks for the presence of annotations, if a loop annotation is present it computes the dependence information using the predicted loop bounds for that loop and applies the transformations if there is no dependency between the loop statements. The current set of transformations supported by annotations is: *loop fission*, *loop fusion*, *loop interchange* and *loop reversal*.

3.5 Parallelism Detection

Efficient parallelization of a sequential program is a challenging task. Currently our McFLAT framework automatically detects whether a **for** loop can be automatically converted to a **parfor** loop or not. The framework performs parallelization tests on the loops based on the dependence information calculated in the dependence analysis and instrumentation phase. A loop is classified as a parallel loop according to MATLAB's semantics [17], since the generated code is targeted for the MATLAB system. Thus, a loop is classified as a parallel for-loop if it satisfies the following conditions.

- There should be no flow dependency between the same array access within the loop body. i.e. distance vectors for all the same array accesses should be zero.
- Within the list of indices for the arrays accessed in the loop, exactly one index involves the loop index variable.
- Other variables used to index an array should remain constant over the entire execution of the loop. The loop index variable cannot be combined with itself to form an index expression.
- Loop index variables must have consecutively increasing integers.
- The value of the loop index variable should not be modified inside the loop body.

3.6 Current Limitations of McFLAT

At present, our framework implements a limited set of loop transformations. It only handles perfectly nested loops which have affine accesses and whose dependences can be summarized by distance vectors. As we develop the framework we will add further dependence tests and transformations, as well as transformations to enable more parallelization. However, since we also wish to put this framework into our JIT compiler, we must be careful not to include overly expensive analyses.

4 Experimental Results

In this section we demonstrate the use of McFLAT through two exploratory performance studies on a set of MATLAB benchmarks. Our ultimate goal is to integrate

McFlat with a machine learning approach, however these example studies provide some interesting initial data. The first study examines performance and speedups of transformed programs, applying our dependence testers and standard loop transformations for a variety of input ranges. The second study looks at the performance of benchmarks when we automatically introduce **parfor** constructs.

4.1 Benchmarks and Static Information

Table 1 summarizes our collection of 10 benchmarks, taken from the McLab and University of Stuttgart benchmark suites. These benchmarks have a very modest size, but yet perform interesting calculations and demonstrate some interesting behaviours. For each benchmark we give the name, description, source of the benchmark, the number of functions, number of loop nests, number of loops that can be automatically converted to parallel for loops.

Table 1. Benchmarks

Benchmark Name	Source of Benchmark	# Lines Code	# Func.	# Loops	# Par. Loops	Benchmark Description
Crni	McLab Benchmarks	65	2	4	1	Finds the Crank-Nicholoson Sol.
Mbrt	McLab Benchmarks	26	2	1	0	Computes mandelbrot set.
Fiff	McLab Benchmarks	40	1	2	0	Finds the finite-difference solution to the wave equation.
Hnormal	McLab Benchmarks	30	1	1	1	Normalises array of homogeneous coordinates.
Nb1d	McLab Benchmarks	73	1	1	0	Simulates the gravitational movement of a set of objects.
Interpol	Uni of Stutt	187	5	5	0	Compares the stability and complexity of Lagrange interpolation.
Lagrcheb	Uni of Stutt	70	1	2	2	Computes Lagrangian and Chebyshev polynomial for comparison.
Fourier	Uni of Stutt	81	3	3	2	Compute the Fourier transform with the trapezoidal integration rule.
Linear	Uni of Stutt	56	1	2	1	Computes the linear iterator.
EigenValue	Uni of Stutt	50	2	1	0	Computes the eigenvalues of the transition matrix.

4.2 Performance Study for Standard Loop Transformations

For our initial study, we ran the benchmarks on an AMD Athlon^{TM} 64 X2 Dual Core Processor 3800+, 4GB RAM computer running the Linux operating system; GNU Octave, version 3.2.4; Matlab, version 7.9.0.529 (R2009b) and McVM/McJIT, version 0.5.

For each benchmark we ran a number of training runs through the instrumenter and profiler. For these experiments instrumented code was executed only on Mathworks' Matlab to generate profile information. Then we used our dependence analyzer and

Table 2. Mathworks' MATLAB Execution Times and Speedups

Benchmark	Trans	Pred. Range 1		Pred. Range 2		Pred. Range 3	
Name	Applied	Time	% Speedup	Time	% Speedup	Time	% Speedup
Crni	N	**60ms**		3.41s			
	R	60ms	0.0 %	**3.21s**	**5.8** %		
Mbrt	N	**1.91s**		9.40s			
	I	1.98s	-3.6 %	9.55s	-1.6%		
	R	1.91s	0.0 %	**9.25s**	**1.5**%		
	(I+R)	1.97s	-3.4%	9.32s	0.8%		
Fiff	NN	**400ms**		880ms			
	RN	405ms	-1.25%	**830ms**	**5.6**%		
Hnormal	N	1.85s		4.52s			
	R	**1.84s**	**0.5**%	**4.48s**	**0.8**%		
Nb1d	N	40ms		2.53s			
Interpol	N	44.70s		60.35s			
Lagrcheb	NN	140ms		280ms		450ms	
	RR	**138ms**	**1.4**%	**270ms**	**3.5**%	**420ms**	**6.6**%
	RN	143ms	-2.1%	280ms	0.0%	450ms	0.0%
	NR	143ms	-2.1%	280ms	0.0%	430ms	4.4%
Fourier	NNN	50ms		1.31s			
	FN	**40ms**	**20.0**%	1.49s	-13.7%		
	RRN	50ms	0.0%	1.25s	4.5%		
	(F+R)N	60ms	-20.0%	1.31s	0.0%		
	RNN	50ms	0.0%	**1.21s**	**7.6**%		
	NRN	50ms	0.0%	1.25s	4.5%		
Linear	NN	336ms		640ms		2.60s	
	IN	566ms	-68.4%	890ms	-39.0%	3.67s	-38.4%
	IR	610ms	-81.5%	850ms	-32.8%	3.42s	-31.5%
	NR	**320ms**	**4.7**%	**600ms**	**6.2**%	**2.51s**	**3.4**%
EigenValue	N	**80ms**		310ms		1.10s	
	I	100ms	-25.0%	370ms	-19.3%	1.18s	-7.27%
	R	90ms	-12.5%	290ms	6.4%	1.10s	0.0%
	(I+R)	90ms	-12.5%	**280ms**	**9.6**%	**1.08s**	**1.81**%

loop transformer to generate a set of output files, one output file for each combination of possible transformations. For example, if the input file had two loops, and loop reversal could be applied to both loops, then we would produce four different output files corresponding to: (1) no reversals, (2) reversing only loop 1, (3) reversing only loop 2, and (4) reversing both loops. For our experiments, we used a combination of both the modes that McFLAT provides for applying loop transformations i.e. *Automatic mode* and *Programmer-annotated mode*.

Each output file has a specialized section for each predicted important range, plus a dynamic guard around each specialized section to ensure that the correct version is run for a given input.

Table 3. Octave Execution Times and Speedups

Benchmark	Trans	Pred. Range 1		Pred. Range 2		Pred. Range 3	
Name	Applied	Time	% Speedup	Time	% Speedup	Time	% Speedup
Crni	N	**5.46s**		1102s			
	R	5.46s	0 %	**1101s**	**0.09**%		
Mbrt	N	**289.8s**		**2000s**			
	I	300s	-3.5 %	2000s	0%		
	R	289.8s	0 %	2000s	0%		
	(I+R)	300s	-3.5%	2000s	0%		
Fiff	NN	6.44s		**251s**			
	RN	**6.41s**	**0.46**%	253s	-0.7%		
Hnormal	N	**7.34s**		**13.4s**			
	R	7.48s	-1.9%	13.6s	-1.4%		
Nb1d	N	2.56s		7.89s			
Interpol	N	3524s		5238s			
Lagrcheb	NN	**630ms**		1.28s		1.95s	
	RR	630ms	0%	**1.27s**	**0.7**%	**1.94s**	**0.51**%
	RN	630ms	0%	1.27s	0.7%	1.94s	0.51%
	NR	630ms	0%	1.27s	0.7%	1.94s	0.51%
Fourier	NNN	120ms		4.24s			
	FFN	120ms	0%	4.28s	-0.9%		
	RRN	120ms	0%	4.31s	-1.6%		
	FRN	120ms	0%	**4.19s**	**1.1**%		
	RNN	**110ms**	**8.3**%	4.26s	-0.4%		
	NRN	120ms	0%	4.25s	-0.2%		
Linear	NN	6.58s		**352s**		1496s	
	IN	6.65s	-1.0%	381s	-8.2%	1443s	3.5%
	IR	6.65s	-1.0%	382s	-8.5%	1422s	4.9%
	NR	**6.56s**	**0.3**%	369s	-4.8%	**1389s**	**7.1**%
EigenValue	N	240ms		**106s**		**460s**	
	I	**230ms**	**4.1**%	127s	-19.8%	502s	-9.1%
	R	230ms	4.1%	116s	-9.4%	486s	-5.6%
	(I+R)	230ms	4.1%	126s	-18.8%	507s	-10.2%

We report the results for four different MATLAB execution engines, the Mathworks' MATLAB (which contains a JIT) (Table 2), the GNU Octave interpreter (Table 3), the McVM interepreter, and the McVM JIT (McJIT) (Table 4).

In each table, the column labeled *Trans. Applied* indicates which transformations are applied to the loops in the benchmark, where *N* indicates that no transformation is applied, *R* indicates Loop Reversal is applied, *F* represents Loop fusion and *I* is representative of Loop Interchange. *NN* indicates that there are two loops in the benchmark and no transformation is applied on any of them. Similarly, *IR* shows there are two loops, Interchange is applied on the first loop and reversal on the second loop. *I+R* indicates one loop nest on which interchange is applied and then reversal.

Depending on the benchmark we had two or three different ranges that were identified by the range predictor. The ranges appear in the tables in increasing value, so *Pred.*

Table 4. McVM Execution Times and Speedups

		McVm(JIT)				McVM(Interpreter)			
Benchmark	Trans	Pred. Range 1		Pred. Range 2		Pred. Range 1		Pred. Range 2	
Name	Applied	Time	% Speedup	Time	% Speedup	Time	%Speedup	Time	% Speedup
Crni	N	**4.00s**		1074s		7.12s		1386.2s	
	R	4.00s	0.0 %	**820s**	**23.6** %	**6.35s**	**10.8** %	**1341.5**	**3.2** %
Mbrt	N	**98.37s**		**675s**		384s		2491s	
	I	101s	-3.3 %	714s	-5.8%	344s	10.4 %	**2286s**	**8.2**%
	R	110s	-12.6 %	781s	-15.6%	**342s**	**10.9** %	2370s	4.8%
	(I+R)	106s	-8.16%	738s	-9.35%	346s	9.8%	2375s	4.6%
Fiff	NN	**260ms**		500ms		7.38s		7.46s	
	RN	260ms	-1.95%	**460ms**	**8**%	**6.95s**	**5.8**%	**7.25s**	**2.8**%
Hnormal	N	5.00s		8.93s		7.23s		**11.6s**	
	R	**4.96s**	**0.8**%	**8.05s**	**10.9**%	**7.11s**	**1.6**%	12.24s	-5.5%
Nb1d	N	850ms		4.10s		1.41s		4.24s	

Range 1 corresponds to the lowest range and *Pred Range 3* corresponds to the highest range. We chose one input for each identified range and timed it for each loop transformation version. In each table we give the speedup (positive) or slowdown (negative) achieved as compared to the version with no transformations. We indicate in bold the version that gave the best performance for each range.

Let us consider first the execution time for Mathworks' MATLAB, as given in Table 2. Somewhat surprisingly to us, it turns out that loop reversal alone always gives performance speed-up on the higher ranges. Whereas, on lower ranges there is either no speed up or performance de-gradation in some of the benchmarks. This implies that it may be worth having a specialized version of the loops, with important loops reversed for higher data ranges.

MATLAB accesses arrays in column-major order, and MATLAB programmers normally write their loops in that fashion, so always applying loop interchange degrades the performance of the program. Performance degrades more for loops which involve array dependencies. However, the degradation impact is lower at higher ranges perhaps due to cache misses in both the transformed and original loop. The loop interchange degradation impact is less for loops that invoke a function whose value is written to an array, for example, Mbrt.

Loop fusion was only applied once (in Fourier) where it gives a performance speed-up on lower ranges. However, as the loop bounds and accessed arrays get bigger then performance degrades.

Now consider the execution time for Octave, given in Table 3. Octave is a pure interpreter and you will note that the absolute execution times are often an order of magnitude slower than Mathworks' system, which has a JIT accelerator. The applied transformations also seem to have very little impact on performance, particularly on the lower ranges. For higher ranges, no fixed behavior is observed, for some benchmarks there is a performance improvement whereas for others performance degrades.

We were also interested in how the transformations would impact our group's McVM, both in pure interpreter mode, and with the JIT. We couldn't run all the benchmarks on McVM because the benchmarks use some library functions which are not currently supported. However, Table 4 lists the results on the subset of benchmarks currently supported. Once again loop reversal can make a significant impact on the higher ranges for the JIT, and actually also seems beneficial for the McVM(interpreter).

4.3 Performance Study for Parallel For Loops

In Table 5 we report the execution time and speedups with MATLAB's `parfor` looping construct. We ran the benchmarks on an Intel ™Core(TM) i7 Processor (4 cores), 5.8GB RAM computer running a Linux operating system; MATLAB, version 7.9.0.529 (R2009b). For these experiments we initialized the MATLAB worker pool to size 4.

Table 5. Mathworks' MATLAB Execution Times and Speedups with Parallel Loops

Benchmark	Trans	Pred. Range 1		Pred. Range 2		Pred. Range 3	
Name	Applied	Time	% Speedup	Time	% Speedup	Time	% Speedup
Crni	N	**280ms**		13.41s			
	pN	1.03s	-257%	14.20s	-5.9%		
	R	290ms	-3.5 %	**13.30s**	**0.8** %		
Hnormal	N	800ms		1.70s			
	pN	70.5s	-8712 %	71.3s	-4094%		
	R	**780ms**	**2.5**%	**1.68s**	**1.1**%		
Lagrcheb	NN	120ms		200ms		280ms	
	(pN)(pN)	140ms	-16.6%	**180ms**	**10.0**%	**250ms**	**10.7**%
	N(pN)	**110ms**	**8.3**%	180ms	10.0%	250ms	10.7%
	(pN)N	120ms	0.0%	180ms	10.0%	260ms	7.1%
	R(pN)	120ms	0.0%	180ms	10.0%	250ms	10.7%
	(pN)R	120ms	0.0%	180ms	10.0%	250ms	10.7%
	RR	120ms	0.0%	200ms	0.0%	270ms	3.5%
	RN	130ms	-8.3%	200ms	0.0%	270ms	3.5%
	NR	130ms	-8.3%	200ms	0.0%	270ms	3.5%
Fourier	NNN	170ms		**680ms**			
	(pN)NN	50ms	70%	720ms	-5.8%		
	(pN)(pN)N	200ms	-17.6%	720ms	-5.8%		
	N(pN)N	50ms	70%	720s	-5.8%		
	(pF)N	50ms	70%	720ms	-5.8%		
	R(pN)N	50ms	70%	710ms	-4.4%		
	(pN)RN	50ms	70%	680ms	0.0%		
	FN	**20ms**	**88.2**%	690ms	-1.4%		
	RRN	170ms	0.0%	680ms	0.0%		
	(F+R)N	170ms	0.0%	680ms	0.0%		
	RNN	170ms	0.0%	680ms	0.0%		
	NRN	170ms	0.0%	680ms	0.0%		
Linear	NN	150ms		7.40s		29.8s	
	N(pN)	**150ms**	**0.0**%	**7.20s**	**2.7**%	30.2s	-1.3%
	I(pN)	390ms	0.0%	10.30s	-39.1%	40.2s	-34.8%
	IN	370ms	-146.6%	10.30s	-39.1%	37.6s	-26.1%
	IR	370ms	-146.6%	10.30s	-39.1%	37.6s	-26.1%
	NR	160ms	-6.6%	7.20s	2.7%	**29.4s**	**1.34**%

The term pN indicates that there is one loop in the benchmark, which is parallelized and no loop transformation is applied on it. (pF) means two loops are fused and then fused loop is parallelized. Note that it is not possible to combine loop reversal and parallelization with the MATLAB `parfor` construct as the MATLAB specifications require that the loop index expression must increase.

We have reported execution times of various combinations of parallel and sequential loops, to study the effect of parallelizing a loop in the context of MATLAB programming language.

For most of the benchmarks we observed that MATLAB's `parfor` loop does not often give significant performance benefits, and in some cases causes severe performance degradation. This is likely due to the parallel execution model supported by MATLAB which requires significant data copying to and from worker threads.

5 Related Work

Of course there is a rich body of research on the topics of dependence analysis, loop transformations and parallelization. In our related work we attempt to cover a representative subset that, to the best of our knowledge, covers the prior work in the area of our paper.

Banerjee [9], Wolfe and Lam [20,21] have modeled a subset of loop transformations like loop reversal, loop interchange and skewing as unimodular matrices and have devised tests to figure out the legality of these transformations. Our framework also uses unimodular transformations model to apply and test the legality of a loop transformation or a combination of loop transformations, but our intent is to specialize for different predicted loop bounds.

Quantitative models based on memory cost analysis have been used to select optimal loop transformations [18]. Memory cost analysis chooses an optimal transformation based on the number of distinct cache lines and the number of distinct pages accessed by the iterations of a loop. Our framework is a preliminary step towards building a self-learning system that selects optimal transformations based on loop bounds and profiled program features that have been beneficial in the past for a transformation or a combination of transformations.

A dimension abstraction approach for vectorization in MATLAB presented in [10] discovers whether dimensions of an expression will be legal if vectorization occurs. The dimensionality abstraction provides a representation of the shape of an expression if a loop containing the expression was vectorized. To improve vectorization in cases which have incompatible vectorized dimensionality, a loop pattern database is provided which is capable of resolving obstructing dimensionality disagreements.

Another framework, presented in [22], predicts the impact of optimizations for some objective (e.g., performance, code size or energy). The framework consists of three types of models: optimization models, code models and resource models. By integrating these models, a benefit value is produced that represents the benefit of applying an optimization in a code context for the objective represented by the resources. MCFLAT is the first step towards developing a self-learning system which would use its past experience in selecting optimal loop transformations.

5.1 Automatic Parallelization

Static automatic parallelism extraction have been achieved in the past [11, 16]. Unfortunately, many parallelization opportunities could still not be discovered by static analysis approach due to lack of information at the source code level. Tournavitis et. al. have used a profiling-based parallelism detection method that enhances static data dependence analysis with dynamic information, resulting in larger amounts of parallelism uncovered from sequential programs [19]. Our approach is also based on profiling-based parallelism detection but in the context of MATLAB programming language and within the constraints of MATLAB parallel loops.

5.2 Adaptive Compilation

Heuristics and statistical methods have already been used in determining compiler optimization sequences. For example, Cooper et. al. [14] developed a technique using genetic algorithms to find ”good” compiler optimization sequences for code size reduction. Profile-based techniques have also been used in the past to suggest recompilation with additional optimizations. The Jalopeño JVM uses adaption system that can invoke a compiler when profiling data suggests that recompiling a method with additional optimization will be more beneficial [6]. Our work is a first step towards developing an adaptive system that applies loop transformations based on predicted data from previous execution runs and profiled information about the programs.

Previously work has been done on JIT compilation for MATLAB. MaJIC, combines JIT-compilation with an offline code cache maintained through speculative compilation of Matlab code into C/Fortran. It derives the most benefit from optimizations such as array bounds check removals and register allocation [5]. Mathworks introduced the MATLAB JIT-Accelerator [3], in MATLAB 6.5, that has accelerated the execution of MATLAB code. McVM [13, 12] is also an effort towards JIT compilation for MATLAB, it uses function specializations based on run-time type of their arguments. The McVM(JIT) has shown performance speed-ups against MATLAB for some of our benchmarks. MCFLAT, the framework presented in this paper uses profiled program features and heuristically determines loop bounds ranges to generate specialized versions of loops in the program.

6 Conclusions and Future Work

In this paper, we have described a new framework, MCFLAT, which uses profile-based training runs to collect information about loop bounds and ranges, and then applies a range estimator to estimate which ranges are most important. Specialized versions of the loops are then generated for each predicated range. The generated MATLAB code can be run on any MATLAB virtual machine or interpreter.

Results obtained on four execution engines (MATLAB, GNU Octave, McVM(JIT) and McVM(interpreter) suggest that the impact of different loop transformations on different loop bounds is different and also depends on the execution engine. We were somewhat surprised that loop reversal was fairly useful for several execution engines, especially on large ranges. Although the tool detected quite a few parallel loops and transformed them to MATLAB's `parfor` construct, the execution benefit was very limited and sometimes very detrimental. Thus, our McJIT compiler will likely support a different parallel implementation which has lower overheads.

Although MCFLAT is already a useful stand-alone tool, in our overall plan it is a preliminary step towards developing a self-learning system that will be part of McJIT and which will decide on whether to apply a loop transformation or not depending on the benefits that the system has seen in the past. Our initial exploratory experiments validate that different loop transformations are beneficial for different ranges. Future work will focus on extracting more information about the program features from profiling, maintaining a mapping between loop bounds, program features and effective loop transformations and making use of past experience to make future decisions on whether to apply transformations or not.

References

1. GNU Octave, `http://www.gnu.org/software/octave/index.html`
2. McLab: An Extensible Compiler Framework for Matlab. Home page, `http://www.sable.mcgill.ca/mclab/`
3. Accelerating Matlab (2002), `http://www.mathworks.com/company/newsletters/digest/sept02/accel_matlab.pdf`
4. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Addison Wesley, Reading (1985)
5. Almasi, G., Padua, D.A.: MaJIC: A MATLAB Just-In-Time Compiler. In: Midkiff, S.P., Moreira, J.E., Gupta, M., Chatterjee, S., Ferrante, J., Prins, J.F., Pugh, B., Tseng, C.-W. (eds.) LCPC 2000. LNCS, vol. 2017, p. 68. Springer, Heidelberg (2001)
6. Arnold, M., Fink, S., Grove, D., Hind, M., Sweeney, P.F.: Adaptive Optimization in the Jalapeño JVM. In: OOPSLA 2000: Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 47–65. ACM, New York (2000)
7. Aslam, T., Doherty, J., Dubrau, A., Hendren, L.: AspectMatlab: An Aspect-Oriented Scientific Programming Language. In: Proceedings of 9th International Conference on Aspect-Oriented Software Development, pp. 181–192 (March 2010)
8. Banerjee, U.K.: Dependence Analysis for Supercomputing. Kluwer Academic Publishers, Norwell (1988)
9. Banerjee, U.K.: Loop Transformations for Restructuring Compilers: The Foundations. Kluwer Academic Publishers, Norwell (1993)
10. Birkbeck, N., Levesque, J., Amaral, J.N.: A Dimension Abstraction Approach to Vectorization in Matlab. In: CGO 2007: Proceedings of the International Symposium on Code Generation and Optimization, Washington, DC, USA, 2007, pp. 115–130. IEEE Computer Society, Los Alamitos (2007)
11. Burke, M.G., Cytron, R.K.: Interprocedural Dependence Analysis and Parallelization. SIGPLAN Not. 39(4), 139–154 (2004)
12. Chevalier-Boisvert, M.: McVM: An Optimizing Virtual Machine for the MATLAB Programming Language. Master's thesis, McGill University (August 2009)
13. Chevalier-Boisvert, M., Hendren, L., Verbrugge, C.: Optimizing MATLAB through Just-In-Time Specialization. In: International Conference on Compiler Construction, pp. 46–65 (March 2010)

14. Cooper, K.D., Schielke, P.J., Subramanian, D.: Optimizing for Reduced Code Space using Genetic Algorithms. In: LCTES 1999: Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems, pp. 1–9. ACM, New York (1999)
15. Lam, M.S., Wolf, M.E.: A Data Locality Optimizing Algorithm. In: PLDI 1991: Programming Language Design and Implementation, vol. 39, pp. 442–459. ACM, New York (2004)
16. Lim, A.W., Lam, M.S.: Maximizing Parallelism and Minimizing Synchronization with Affine Transforms. In: POPL 1997: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 201–214. ACM, New York (1997)
17. Matlab. The Language Of Technical Computing. Home page, `http://www.mathworks.com/products/matlab/`
18. Sarkar, V.: Automatic Selection of High-Order Transformations in the IBM XL FORTRAN compilers. IBM J. Res. Dev. 41(3), 233–264 (1997)
19. Tournavitis, G., Wang, Z., Franke, B., O'Boyle, M.F.: Towards a Holistic Approach to Auto-Parallelization: Integrating Profile-Priven Parallelism Detection and Machine-Learning based Mapping. In: PLDI 2009: Programming Languages Design and Implementation, vol. 44, pp. 177–187. ACM, New York (2009)
20. Wolf, M.E., Lam, M.S.: A Loop Transformation Theory and an Algorithm to Maximize Parallelism. IEEE Trans. Parallel Distrib. Syst. 2(4), 452–471 (1991)
21. Wolfe, M.J.: Optimizing Supercompilers for Supercomputers. MIT Press, Cambridge (1990)
22. Zhao, M., Childers, B., Soffa, M.L.: Predicting the Impact of Optimizations for Embedded Systems. In: Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems, San Diego, CA, USA, vol. 38, pp. 1–11. ACM, New York (2003)

Static Analysis of Dynamic Schedules and Its Application to Optimization of Parallel Programs[*]

Christoph M. Angerer and Thomas R. Gross

ETH Zurich, Switzerland

Abstract. Effective optimizations for concurrent programs require the compiler to have detailed knowledge about the scheduling of parallel tasks at runtime. Currently, optimizations for parallel programs must define their own models and analyses of the parallel constructs used in the source programs. This makes developing new optimizations more difficult and complicates their integration into a single optimizing compiler.

We investigate an approach that separates the static analysis of the dynamic runtime schedule from subsequent optimizations. We present three optimizations that are based on the information gathered during the schedule analysis. Variants of those optimizations have been described in the literature before but each work is built upon its own highly specialized analysis. In contrast, our independent schedule analysis shows synergistic effects where previously incompatible optimizations can now share parts of their implementation and all be applied to the same program.

1 Introduction

With the arrival of multicore systems, parallel programming is becoming increasingly mainstream. Despite this, compilers still remain largely ignorant of the task scheduling at run-time. Absent this knowledge, however, a compiler is missing important optimization and verification opportunities.

Because compilers do not have a good understanding of the runtime scheduling of tasks, researchers developing optimizations for parallel programs must additionally develop their own model and analysis of the parallel constructs in use. The overhead of defining a full-fledged analysis, however, obfuscates the actual optimization and prohibits synergistic effects that might emerge by combining different optimizations.

In this paper, we present a static analysis for parallel programs that is independent from any concrete optimization. By using our schedule analysis, the core algorithms of existing optimizations can often be implemented in only a few simple rules. A small algorithmic core not only helps with a better understanding of the optimization but also supports their integration into a single optimizing compiler. The contributions of this paper are:

[*] Supported, in part, by the Swiss National Science Foundation grant 200021_120285.

K. Cooper, J. Mellor-Crummey, and V. Sarkar (Eds.): LCPC 2010, LNCS 6548, pp. 16–30, 2011.

- we describe a model for fine-grained parallelism based on lightweight tasks with explicit scheduling. This model is powerful enough to express a wide variety of existing parallelism constructs (Section 2);
- we develop a schedule analysis that computes an abstract schedule from a program containing explicit scheduling instructions (Section 3);
- we present three optimizations for parallel programs that make use of the results of the schedule analysis. Two optimizations work with fundamentally different synchronization primitives: locks and transactional memory. The common schedule analysis enables the optimization of programs that intermix both those synchronization paradigms (Section 4).

The core of the schedule analysis has been implemented in a prototype and is available online at *http://github.com/chmaruni/XSched* (Section 5).

2 A Program Model with Explicit Scheduling

In this section we describe a model for fine-grained parallelism based on lightweight tasks with explicit scheduling. This model is general enough to express a wide variety of existing concurrency patterns, from structured fork-join style parallelism to unstructured threads. The explicit happens-before relationships simplify the analysis of parallel program schedules while avoiding the limitations of lexically scoped parallelism.

The basic building block of our execution model is a *task*. A task is similar to a method in that it contains code that is executed in the context of a `this`-object (or the class, in the case of `static` methods/tasks). Unlike a method, however, one does not *call* a task, which would result in the immediate execution of the body, but instead *schedules* it for later execution.

As an example, consider a task `t()` that starts a long-running computation `compute()` and schedules a task `print()` that will print the result after the computation has finished:

```
task t() {
    Activation aCompute = schedule(this.compute());
    Activation aPrint = schedule(this.print());
    aCompute→aPrint;
}
```

A schedule is represented as a graph of $\langle object, task() \rangle$ pairs. The statement `schedule(this.print())`, e.g., creates a new node with the `this` object and the `print()` task and returns an object of type `Activation` representing that node. Like any other object, `Activation` objects can be kept in local variables, passed around as parameters, and stored in fields.

At runtime, a scheduler constantly chooses activations that are eligible for execution and starts them. The order in which the scheduler is allowed to start the activations is specified by the edges in the schedule graph. If the schedule contains a happens-before edge $\langle o1, t1() \rangle \rightarrow \langle o2, t2() \rangle$, the scheduler must guarantee that activation $\langle o1, t1() \rangle$ has finished execution before activation $\langle o2, t2() \rangle$

```
 1  class ParallelOrderedMap {
 2      Vector out;
 3
 4      private task doMap(Object data) {
 5          Object mappedData = //complex computation using data
 6          now.result = mappedData;
 7      }
 8      private task doWrite(Activation mapActivation) {
 9          out.add(mapActivation.result);
10      }
11      task mapInput(Vector input) {
12          Activation lastWrite = now;
13          for(Object data : input) {
14              Activation map = schedule(this.doMap(data));
15              Activation write = schedule(this.doWrite(map));
16
17              map → write;
18              lastWrite → write;
19
20              lastWrite = write;
21      } } }
```

Fig. 1. Example of a parallel ordered mapping operation

is started. The statement `aCompute`→`aPrint` creates an explicit happens-before relationship between the two activation objects `aCompute` and `aPrint`.

In a program, the currently executing activation can be accessed through the keyword `now`. Whenever a new task is scheduled, the scheduler automatically adds an initial happens-before relationship between `now` and the new activation node. This initial edge prevent the immediate execution of the new activation and allows the current task to add additional constraints to the schedule before it finishes. Therefore, in the example the scheduler implicitly creates two additional edges `now`→`aCompute` and `now`→`aPrint`.

2.1 Example of a Parallel Ordered Mapping Operation

Figure 1 shows a more complex example of a class implementing a parallel ordered mapping operation. When the `mapInput()` task is activated, passing a `Vector` of input elements, this class will apply a (possibly expensive) mapping operation to each input element in parallel and write the resulting mapped values into the `out` vector in the original order.

The loop on line 13 iterates through every element in the input vector and, for each element, schedules the `doMap()` task on line 14, passing the `data` element to the activation. Line 15 then activates the `doWrite()` task for every element. A chain of `doWrite()` activations write the mapped values in the correct order into the `out` vector.

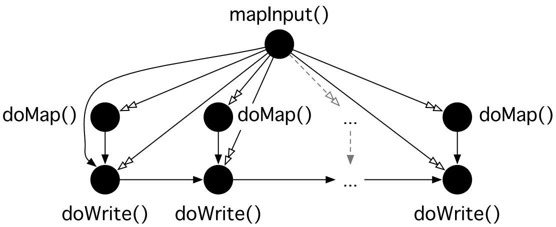


Fig. 2. The schedule created by `mapInput()` from Figure 1

For `doWrite()` to get to the result of `doMap()`, we pass the `map` activation object as a parameter to `doWrite()`. Inside `doMap()` we use the `result` field provided by `Activation` to store the result of the mapping operation which can then be read in `doWrite()`. The explicit happens-before relationship added on line 17 ensures that the mapped value has been stored in the `result` field before `doWrite()` executes.

So far we ensured the ordering between the mapping operation and the write operation for each single element. The correct ordering of the writes is achieved by an additional happens-before relationship between the current `write` activation and the `lastWrite` activation on line 18. Initially, we set `lastWrite` to `now` in line 12 and then update it to the most recent `write` activation on line 20.

Figure 2 shows the schedule that is created by `mapInput()`. The double-headed arrows are created implicitly by the `schedule` statements whereas the single-headed arrows stem from the →-statements.

After `mapInput()` has finished setting up the schedule, the scheduler can choose any of the `doMap()` activations for parallel execution because they are not ordered with respect to one another. The `doWrite()` activations must be executed in order, however, which guarantees the correct ordering of the written values in the `out` vector.

2.2 Additional Synchronization Primitives

Explicit scheduling alone is expressive enough to model many concurrency patterns such as fork-join, bounded-buffer producer-consumer, or fuzzy barriers [9].

There are cases, however, that require nondeterministic choice—which cannot be expressed with explicit scheduling alone. Non-deterministic access to shared resources requires additional synchronization primitives such as atomic compare-and-swap operations, locks, or transactional memory. Take, for example, a shared resource such as a printer and two parallel tasks waiting for user input before accessing the printer. There is no point in the program where we can define an ordering between the two tasks beforehand, because the timing of the user input is unknown.

3 Schedule Analysis

The core of our approach is a schedule analysis that can determine whether two activations are executed in parallel, sequentially, or exclusively. Schedule analysis thus computes the function $Activation \times Activation \rightarrow Relation$ where $Relation$ is one of the following:

Sequential: Two activations are sequential if their execution is strictly ordered.

Exclusive: Two activations are exclusive if they can never co-exist in a single run of the program (e.g., they are scheduled in different branches of a conditional statement).

Parallel: If two activations are neither sequential nor exclusive, they are considered (potentially) parallel.

In addition, schedule analysis computes the sets of objects that are read and/or written by each activation. The information about the read- and write-sets together with the information about the relative ordering of activations can then be used by subsequent optimizations such as the ones from Section 4.

The algorithm presented in this paper computes the abstract schedule and the read-/write-sets in the four phases shown in Figure 3. Given the bytecode of a whole program as input, a pre-processing phase extracts static information about the program structure. This information is used by a standard points-to analysis to propagate alias information and compute points-to sets for object fields and program variables. Points-to information is necessary because activations are normal objects that can be stored in fields and passed as parameters.

The third phase is the schedule analysis. During this phase we compute the read- and write-sets for each activation and extract an abstract schedule from the unconditional →-statements present in the program. The abstract schedule is a directed graph with different node and edge types. The fourth phase takes this graph and flattens it into the binary relations for sequential, exclusive, and parallel activations.

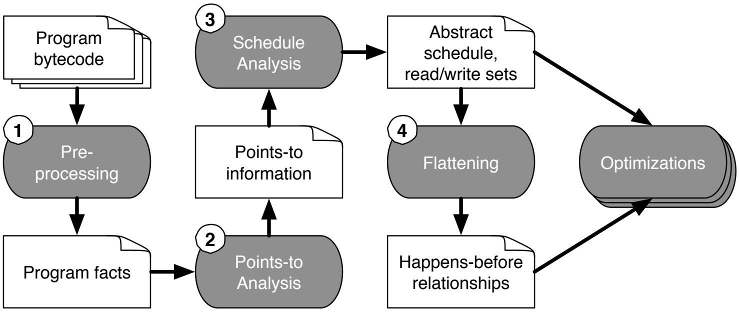


Fig. 3. The phases of a schedule analysis

V	the domain of variables. V contains all the allocation sites, formal parameters, return values, thrown exceptions, cast operations, and dereferences in the program.
H	the domain of heap objects. Heap objects are named by the invocation sites of object creation operations.
F	the domain of fields in the program. There is a special field *elements* to denote an array access.
M	the domain of implemented methods in the program. It does not include abstract or interface methods.
N	the domain of virtual method names used in invocations.
BC	the domain of bytecodes in the program.

Fig. 4. The Datalog domains

3.1 Datalog

Most phases of our analysis are formulated and implemented as Datalog programs. We chose Datalog because it is a concise high-level specification language that has been shown to be well-suited for dataflow analyses and scalable to even large real-world programs [18].

The basis of Datalog are two-dimensional tables called *relations*. In a relation, the columns are the attributes, each of which is associated with a finite domain defining the set of possible values, and the rows are the tuples that are part of this relation. Figure 4 shows the domains that we use in this paper.

If tuple (x, y, z) is in relation A, we say that predicate $A(x, y, z)$ is true. A Datalog program consists of a set of rules that compute new members of relations if the rule body is true. E.g., the rule:

```
D(w, z) : A(w, x), B(x, y), !C(y, z).
```

says that "tuple (w, z) is added to D if $A(w, x)$, $B(x, y)$, and not $C(y, z)$ are all true." The Datalog runtime will apply rules until a fixed point has been reached and no more tuples can be added to the relations.

3.2 Pre-processing and Points-to Analysis

The first two phases of the analysis implement a standard points-to analysis. For space reasons, we only describe briefly the outcomes of both phases. An in-depth explanation of the algorithms we use is given in [18].

Before the points-to analysis (implemented as a Datalog program) starts, a pre-processor extracts information about the analyzed program and generates input tuples that can be read by Datalog. The pre-processor generates many different relations encoding information about type hierarchies, virtual method calls, object creation, and more; in this paper, however, we are mostly interested in the following two relations:

$\mathit{store} : BC \times V \times F \times V$ represents store statements. $\mathit{store}(bc, v1, f, v2)$ says that bytecode bc is a statement "v1.f = v2".

$load : BC \times V \times F \times V$ represents load statements. $load(bc, v1, f, v2)$ says that bytecode bc is a statement "v2 = v1.f".

The goal of the points-to analysis is to compute the following three relations from the input relations generated by the pre-processor:

$variablePT : V \times H$ is the variable points-to relation. $variablePT(v, obj)$ means that variable v can point to heap object obj.

$heapPT : H \times F \times H$ is the heap points-to relation. $heapPT(obj1, f, obj2)$ means that field f of heap object $obj1$ may point to heap object $obj2$.

$invocationEdge : BC \times M$ is the relation of the resolved targets of invocation sites. $invocationEdge(bc, m)$ says that invocation bytecode bc may invoke the method implementation m.

During the points-to analysis we treat `schedule`-statements as normal method calls. This works because all the parameters for the activation are bound when the `schedule` statement is executed even though the exact time when the activation will execute is not known. →-statements are ignored during this phase.

Whaley and Lam [18] describe various points-to analyses with a variety of trade-offs between precision and computational cost. For the rest of the paper we assume a context insensitive analysis with on-the-fly call graph discovery, but other, more precise variants can be used.

3.3 Computing and Flattening the Abstract Schedule

The main task of the schedule analysis is to compute an abstraction of the scheduling graphs that can occur at runtime. For this, the analysis must take the `schedule`-statements for the initial creation edges and all →-statements for additional happens-before edges into account.

In general, the safe and conservative assumption is to over-approximate parallelism. As an example, take the detection of data races. Two activations are allowed to write to the same data if and only if they are sequentially ordered. If the sequential execution cannot be guaranteed we must assume that both tasks are potentially executed in parallel and report a data race if they access the same data.

Because the analysis cannot rely on happens-before relationships that are created conditionally, we only consider unconditional →-statements. Further, if for a statement `lhs→rhs` the points-to analysis found that one or both variables `lhs` and `rhs` may point to more than one activation object, the abstract schedule must over-approximate parallelism by ignoring the happens-before edge because it cannot guarantee the exact ordering of the involved activations.[1]

The core of the abstract schedule computation can be expressed in the following Datalog rules:

[1] In this case, increasing the context-sensitivity can result in higher precision and therefore smaller points-to sets which may allow the analysis to drop less edges.

```
multiple(v)  :- variablePT(v, obj1), variablePT(v, obj2), obj1 != obj2.
singleton(v) :- !multiple(v).

happensBeforeEdge(source, target) :-
                 arrowStatement(lhs, rhs),
                 variablePT(lhs, source), variablePT(rhs, target),
                 singleton(lhs), singleton(rhs).
```

The relation *multiple* : V contains all the variables that may point to two (or more) different objects whereas the relation *singleton* : V contains all variables not in *multiple* and thus pointing to at most one object. Given an arrow-statement `lhs`→`rhs`, the $happensBeforeEdge : H \times H$ relation contains the tuple `(source, target)` if the variables `lhs` and `rhs` point to the singleton objects `source` and `target` respectively.

The points-to analysis can only track a finite number of heap objects (including activation objects) but a program that contains loops and recursion can create a potentially infinite number of objects. For this reason, filtering out ambiguous →-statements is a necessary but not sufficient condition for computing a conservative abstract schedule because a single object at analysis time may represent multiple runtime objects.

In the example from Figure 1, there are potentially many activation objects created at lines 14 and 15. Therefore, the happens-before edge `map`→`write` on in line 17 is only valid without restrictions inside the same loop iteration. An activation `doMap()` of a later iteration, e.g., is not guaranteed to happen before a `doWrite()` activation of an earlier iteration.

To address this problem, we make use of the fact that a program in static single assignment form (SSA) captures the flow of values between loop iterations in the form of explicit Φ operations. The pre-processor described in Section 3.2 treats a statement `var3 =` Φ`(var1, var2)` in the source program similar to an object creation site. That is, it adds a new object `phiObj` to H and records the assignment of `phiObj` to `var3`. Additionally, the preprocessor adds the facts `varIntoPhi(var1, phiObj)` and `varIntoPhi(var2, phiObj)` to a relation *varIntoPhi* : $V \times H$ indicating that variables `var1` and `var2` flow into the `phiObj`.

With this information in place, the schedule analysis can compute the relation *phiEdge* : $H \times H$ with the following rule:

```
phiEdge(actObj, phiAct) :-
                 varIntoPhi(actVar, phiAct), variablePT(actVar, actObj).
```

A fact `phiEdge(act, phi)` means that the activation heap object `act` flows into the phi heap object `phi`.

Figure 5 shows the `mapInput()`-task from Figure 1 and the abstract schedule that is computed by the schedule analysis. Solid nodes represent normal activation nodes and dashed nodes are Φ activation objects. The dashed edges are computed by the above `phiEdge` rule. In addition to the activation nodes, the graph contains dashed boxes indicating loop boundaries: a solid node is inside a

```
task mapInput(Vector input) {
    Activation lastWrite0 = now;
    Iterator iterator = input.iterator();
label0:
    Activation lastWrite1 = Φ(lastWrite0, lastWrite2);

    if iterator.hasNext() == 0 goto label1;
    data = iterator.next();

    Activation map = schedule this.doMap(data);
    Activation write = schedule this.doWrite(map);

    map → write;
    lastWrite1 → write;

    lastWrite2 = write;
    goto label0;
label1:
    return;
}
```

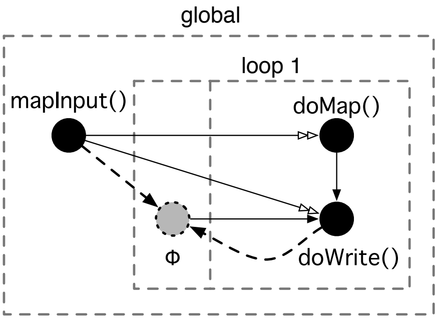


Fig. 5. The `mapInput()` task from Figure 1 in SSA form and the abstract schedule

dashed box if the `schedule`-statement is inside the loop body. The *Φ* nodes that play the role of loop variables are placed in a special "header" area in the loop. The loop information is computed by a *structural analysis* on a global interprocedural SSA graph [10]. A special `global` box represents the whole program.

With the graph in Figure 5, we can deduce that the `mapInput()` activation always happens before both the `doMap()` and the `doWrite()` activations. This is because `mapInput()` is in the `global` context and connected to the other two activations by creation edges. Further, the `doWrite()` activation is sequential to itself because of the recursion through the *Φ* node. The recursion loop encodes the fact that all `doWrite()` activations are ordered with respect to one another.

The `doMap()` activation, on the other hand, is parallel to itself, because it is created inside a loop, as well as parallel to `doWrite()` because the happens-before edge between them is created inside the `loop 1` box and therefore has no effect on the `global` context.

This example demonstrates that the effect of a happens-before edge generally depends on the loop context: in `loop 1`, we can say that `doMap()` always happens-before `doWrite()` because those objects are created inside this loop; but this is not true for the `global` perspective. Only *Φ* nodes allow us to establish happens-before relationships across loop iterations.

Most optimizations work in the `global` context, because they transform the source code which affects the whole program and not only a single loop iteration. Therefore, the last phase of the schedule analysis flattens the abstract schedule in the `global` context. The flattening process creates the three relations *parallel* : $H \times H$, *sequential* : $H \times H$, and *exclusive* : $H \times H$ by using the abstract schedule to find the type of relationship for each pair of activation objects.[2]

[2] Deciding exclusivity requires further flow-sensitive analysis of the source code but it can reduce the number of activations that are unnecessarily classified as being potentially parallel.

3.4 Computing Read- and Write-Sets

The second part of the schedule analysis is to compute the read- and write-sets for each activation. We capture the read and write sets in relations *read* : $H \times BC \times H$ and *write* : $H \times BC \times H$. A tuple `read(act, bc, obj)`, e.g., states that activation object `act` may reach bytecode `bc` and this bytecode is a `load` that may access object `obj`. The computation of the read and write sets is straightforward and can be expressed in the following two Datalog rules (where an underscore '_' means "any"):

```
read(act, bc, obj) :-
      activationReaches(act, bc), load(bc, v, _, _),  variablePT(v, obj).
write(act, bc, obj) :-
      activationReaches(act, bc), store(bc, v, _, _), variablePT(v, obj).
```

The relation *activationReaches* : $H \times BC$ is a simple reachability predicate that starting from the `task` of an activation object follows all invocation edges in the call graph to find all bytecodes that this activation may execute.

4 Optimizations Based on Schedule Analysis

In this section, we present three sample optimizations for parallel programs that are all based on the same schedule analysis from Section 3. The first two optimizations have been taken from the literature and target the two main synchronization primitives, locks and transactional memory. The third optimization is specific to our explicit scheduling model and tries to reduce the number of happens-before relationships in a program thus reducing scheduling overhead and potentially increasing parallelism.

4.1 Synchronization Removal

Like many imperative and object-oriented languages, Java provides a synchronization mechanism based on locks. Whenever a method or block may access data structures that are shared between multiple threads, the programmer must guard the critical section with a lock, e.g., using the `synchronized` keyword. Because a thread-safe library cannot know the context it is used in, it must conservatively assume a multi-threaded environment and guard all critical sections that potentially access shared data. In many programs, however, a large number of the locking operations may safely be removed because two parallel tasks never contend for the same locks.

A critical section is required if two parallel activations `act1` and `act2` may try to acquire a lock on the same object `lockObj`. Acquiring a lock requires the execution of a dedicated monitor enter instruction that is associated with a variable pointing to the lock object. Conversely, a critical section is unnecessary if its guarding monitor enter is not required. The following Datalog rules compute the set of required and unnecessary monitor enter bytecodes:

```
lockObject(monitorEnterBC, obj) :-
                lockVariable(monitorEnterBC, v), variablePT(v, obj).
requiredMonitorEnter(monitorEnterBC1) :-
                parallel(act1, act2),
                activationReaches(act1, monitorEnterBC1),
                activationReaches(act2, monitorEnterBC2),
                lockObject(monitorEnterBC1, lockObj),
                lockObject(monitorEnterBC2, lockObj).
unnecessaryMonitorEnter(monitorEnterBC) :-
                !requiredMonitorEnter(monitorEnterBC).
```

If, in the example from Figure 1, the programmer had guarded the call `out.add()` in the `doWrite()` task with a lock, the analysis would consider this lock as unnecessary because all activations of `doWrite()` are ordered and the `parallel(act1, act2)` clause is always false for two `doWrite()` activations. If the programmer had also guarded the body of task `doMap()` with the same lock, the above rules would consider all locks to be required because `doMap()` activations can happen in parallel with other `doMap()` and `doWrite()` activations.

Ruf [13] describes the same optimization but based on an analysis algorithm that is specialized to the task of synchronization removal. One of the achievements is that this approach can remove 100% of all synchronization for the special case of single threaded programs. Looking at the rules above, we can see that our optimization has the same property. In a single threaded program, the clause `parallel(act1, act2)` is always false and therefore all monitor enter bytecodes will be classified as unnecessary.

4.2 Reducing Strong Atomicity Overhead

Software Transactional memory is a promising alternative to synchronization that avoids many of the problems associated with locks. In an STM system, an atomic region `atomic{b}`, where the block `b` is a list of statements, requires the runtime to execute the sequence `b` *as though* there were no interleaved computation. When the transaction inside the atomic region completes, it either *commits*, thus making the changes visible to other processes, or it *aborts*, causing the transaction to be rolled back and the atomic region to be re-executed.

A transactional system is said to have *weak* atomicity semantics if it allows computations outside of transactions to be interleaved with transactions. Weak semantics allow for a more efficient implementation but it sacrifices ordering and isolation guarantees which can lead to incorrect execution of programs that are correctly synchronized under locks [16].

Strong atomicity, on the other hand, requires memory accesses outside of transactions to be accompanied by memory barriers, and this setup greatly increases the overhead of strong atomicity. Guarding a memory access with a barrier, however, is only necessary if it may conflict with a memory access inside a transaction that may be executed in parallel.[3]

[3] Strong atomicity semantics do not cover conflicting memory access outside transactions.

The following Datalog rules decide for a given read- or write-bytecode (outside a transaction) whether it requires a read or write barrier:

```
readInsideTransaction(act, obj:H) :-
                bcGuardedByAtomic(act, readBC), read(act, readBC, obj).
writtenInsideTransaction(act, obj:H) :-
                bcGuardedByAtomic(act, writeBC), write(act, writeBC, obj).
requiresReadBarrier(readBC) :-
                read(act1, readBC, obj),
                writtenInsideTransaction(act2, obj),
                parallel(act1, act2).
requiresWriteBarrier(writeBC) :-
                read(act1, writeBC, obj),
                writtenInsideTransaction(act2, obj),
                parallel(act1, act2).
requiresWriteBarrier(writeBC) :-
                read(act1, writeBC, obj),
                readInsideTransaction(act2, obj),
                parallel(act1, act2).
```

The relation *bcGuardedBy* : $H \times BC$ is a simple reachability predicate that contains all bytecodes that, starting from the `task` of a given activation object, may be executed inside an `atomic` block.

Modulo the exact points-to analysis used[4], the analysis presented here is almost the same as the optimizations presented by Hindman and Grossman [6]. The difference is the additional clause `parallel(act1, act2)` in each of the above rules. This means that if in the worst case the schedule analysis cannot compute any happens-before relationships (and therefore conservatively classifies all activations as *parallel*) our analysis is equivalent to [6]. If the schedule analysis can compute relevant edges, however, our analysis is more precise allowing the optimizer to remove more read- and/or write barriers.

4.3 Dependence Reduction

Dependence reduction aims at removing →-statements from the source code. This can be beneficial in two ways:

- Removing a →-statement that creates a happens-before relationship between two activations that are already (transitively) ordered can improve the performance of later analyses as well as improve the generated code. Unnecessary transitive →-statements can be found by looking for transitive edges in the schedule.
- Removing a →-statement between two activations that are otherwise not ordered can increase the parallelism in a program.

[4] Hindman and Grossman use a points-to analysis that distinguishes objects by type, not by creation site [6].

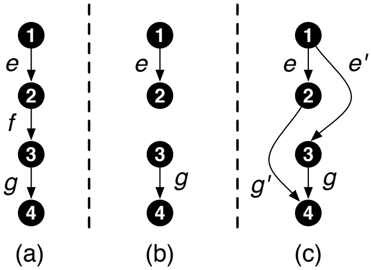


Fig. 6. Fixing the transitive ordering after removing the edge `f`

Removing a non-transitive edge between two activations may be allowed if the read- and write-sets of both activations are disjoint.

```
requiredEdge(act1, act2) :-
                happensBeforeEdge(act1, act2),
                write(act1, obj),
                readOrWrite(act2, obj).
requiredEdge(act1, act2) :-
                happensBeforeEdge(act1, act2),
                readOrWrite(act1, obj),
                write(act2, obj).
unnecessaryEdge(act1, act2) :- !requiredEdge(act1, act2).
```

When such an edge is removed, however, we must ensure that the transitive ordering is kept intact. Take, for example, the schedule shown in Figure 6(a). If the analysis finds that edge `f` is unnecessary, simply removing it results in the schedule shown in Figure 6(b). This schedule is broken, because by removing `f` the transitive ordering between node 1 and node 3 as well as the transitive ordering between node 2 and node 4 that was present before the removal is missing. After adding the additional edges `e'` and `g'` as shown in Figure 6(c) the transitive ordering is correct again. The parallelism has been increased, however, because activations 2 and 3 can now be executed in parallel. In [3] we present more details about this optimization.

5 Implementation and Future Work

The Datalog parts of our schedule analysis have been implemented and can be found on *http://github.com/chmaruni/XSched*. We use the *bddbddb* Datalog system. *bddbddb* is backed by binary decision diagrams (BDDs) and has been shown to scale to large programs using over 10^{14} contexts [18].

The next step is to integrate our optimizations with a compiler to produce optimized code and to empirically measure how our optimizations compare to the ones from the original papers.

6 Related Work

The *happens-before* ordering was first formulated by Lamport [7] and is the basis of the Java memory model [8]. Despite its significance in the memory model, in Java happens-before edges can be created only implicitly, e.g., by using `synchronized` blocks or `volatile` variables.

The goal of a pointer analysis is to statically determine when two pointer expressions refer to the same memory location. Steengaard [17] and Andersen [2] laid the groundwork for the flow-insensitive analysis of single threaded programs. Because points-to analysis is undecidable in the general case, however, researchers developed a large collection of approximation algorithms specialized for different problem domains [5], including parallel programming.

Rugina and Rinard [14] describe a pointer analysis for programs with structured fork-join style concurrency. For each program point, their algorithm computes a points-to graph that maps each pointer to a set of locations. By capturing the effects of pointer assignments for each thread, their algorithm can compute the interference information between parallel threads. Computing the interference information relies on the lexical scoping of the parallel constructs; it cannot handle unstructured parallelism.

By combining pointer and escape analysis, subsequent projects were able to extend their analyses beyond structured parallelism [15,11]. Both analyses compute points-to information but do not directly answer as to how two tasks are executed with respect to each other. Further, the tight integration of the pointer analysis with the escape analysis and concurrency analysis is contrary to our goal of separating the concerns of schedule analysis from points-to analysis.

A *may-happen-in-parallel* (MHP) analysis can be used to determine what statements in a program may be executed in parallel [12]. Without flow sensitivity, relating two program statements is of limited use for analyzing programs with unstructured parallelism. If two threads execute the same statements but in different contexts, for example, a context insensitive MHP analysis might unnecessarily classify the statements as parallel. When the programming language is restricted to structured parallelism, as is the case for X10, an intra-procedural MHP analysis can achieve good results, however [1].

Barik [4] describes a context and flow-sensitive may-happen-before analysis that distinguishes threads by their creation site. By using threads as their model, however, they must conservatively assume that a parent thread in the tree runs in parallel with each child thread. In our model a parent activation is known to happen before any child activation because the creation tree is a spanning tree embedded in the schedule.

7 Concluding Remarks

In this paper we showed how an independent schedule analysis can form the basis for different optimizations of parallel programs.

In previous compilers, each optimization had to come with its own model and analysis of concurrent computation. The introduction of an independent schedule analysis factors out the common aspects of these optimizations, making it easy to not only combine multiple optimizations but also to derive new ones. Combining the optimizations discussed in this paper, for example, allows the optimization of programs that intermix transactional memory with traditional locking. Moreover, the optimization for synchronization removal could be easily adapted to remove atomic sections as well.

The key factor that enabled this approach was a model of parallel computation that allowed a static analysis of the dynamic schedules to be encountered at runtime. Exposing the schedule (and allowing a compiler to analyze and optimize it) is a necessary step in the path towards improving the optimization of parallel programs.

References

1. Agarwal, S., Barik, R., Sarkar, V., Shyamasundar, R.K.: May-happen-in-parallel Analysis of X10 Programs. In: PPoPP (2007)
2. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. Ph.D thesis, DIKU, University of Copenhagen (1994)
3. Angerer, C.M., Gross, T.R.: Parallel Continuation-Passing Style. In: PESPMA (2010)
4. Barik, R.: Efficient computation of may-happen-in-parallel information for concurrent java programs. In: Ayguadé, E., Baumgartner, G., Ramanujam, J., Sadayappan, P. (eds.) LCPC 2005. LNCS, vol. 4339, pp. 152–169. Springer, Heidelberg (2006)
5. Hind, M.: Pointer Analysis: Haven't We Solved This Problem Yet? In: PASTE (2001)
6. Hindman, B., Grossman, D.: Strong atomicity for Java without virtual-machine support. Tech. Rep. UW-CSE- 06-05-01 (May 2006)
7. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Commun. ACM 21(7) (1978)
8. Manson, J., Pugh, W., Adve, S.V.: The Java Memory Model. In: POPL (2005)
9. Matsakis, N., Gross, T.: Programming with intervals. In: Gao, G.R., Pollock, L.L., Cavazos, J., Li, X. (eds.) LCPC 2009. LNCS, vol. 5898, pp. 203–217. Springer, Heidelberg (2010)
10. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, San Francisco (1997)
11. Nanda, M.G., Ramesh, S.: Pointer Analysis of Multithreaded Java Programs. In: Matsui, M., Zuccherato, R.J. (eds.) SAC 2003. LNCS, vol. 3006, Springer, Heidelberg (2004)
12. Naumovich, G., Avrunin, G., Clarke, L.: An Efficient Algorithm for Computing MHP Information for Concurrent Java Programs. In: ESEC/FSE-7 (1999)
13. Ruf, E.: Effective Synchronization Removal for Java. In: PLDI (2000)
14. Rugina, R., Rinard, M.: Pointer Analysis for Structured Parallel Programs. In: TOPLAS (2003)
15. Salcianu, A., Rinard, M.: Pointer and Escape Analysis for Multithreaded Programs. In: PPoPP (2001)
16. Shpeisman, T., et al.: Enforcing Isolation and Ordering in STM. In: PLDI (2007)
17. Steensgaard, B.: Points-to Analysis in Almost Linear Time. In: POPL (1996)
18. Whaley, J., Lam, M.S.: Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In: PLDI (2004)

Lowering STM Overhead with Static Analysis

Yehuda Afek, Guy Korland, and Arie Zilberstein

Computer Science Department, Tel-Aviv University, Israel
{afek,guy.korland}@cs.tau.ac.il, zilbers@post.tau.ac.il

Abstract. Software Transactional Memory (STM) compilers commonly instrument memory accesses by transforming them into calls to STM library functions. Done naïvely, this instrumentation imposes a large overhead, slowing down the transaction execution. Many compiler optimizations have been proposed in an attempt to lower this overhead. In this paper we attempt to drive the STM overhead lower by discovering sources of sub-optimal instrumentation, and providing optimizations to eliminate them. The sources are: (1) redundant reads of memory locations that have been read before, (2) redundant writes to memory locations that will be subsequently written to, (3) redundant writeset lookups of memory locations that have not been written to, and (4) redundant writeset record-keeping for memory locations that will not be read. We describe how static analysis and code motion algorithms can detect these sources, and enable compile-time optimizations that significantly reduce the instrumentation overhead in many common cases. We implement the optimizations over a TL2 Java-based STM system, and demonstrate the effectiveness of the optimizations on various benchmarks, measuring up to 29-50% speedup in a single-threaded run, and up to 19% increased throughput in a 32-threads run.

Keywords: Transactional Memory, Optimization, Static Analysis.

1 Introduction

Software Transactional Memory (STM) [15, 24] is an emerging approach that provides developers of concurrent software with a powerful tool: the *atomic block*, which aims to ease multi-threaded programming and enable more parallelism. Conceptually, statements contained in an atomic block appear to execute as a single atomic unit: either all of them take effect together, or none of them take effect at all. In this model, the burden of carefully synchronizing concurrent access to shared memory, traditionally done using locks, semaphores, and monitors, is relieved. Instead, the developer needs only to enclose statements that access shared memory by an atomic block, and the STM implementation guarantees the atomicity of each block.

In the past several years there has been a flurry of software transactional memory design and implementation work; however, with the notable exception of transactional C/C++ compilers [23], many of the STM initiatives have remained academic experiments. There are several reasons for this; major among

K. Cooper, J. Mellor-Crummey, and V. Sarkar (Eds.): LCPC 2010, LNCS 6548, pp. 31–45, 2011.

them is the large performance overhead [8]. In order to make a piece of code transactional, it usually undergoes an instrumentation process which replaces memory accesses with calls to STM library functions. These functions handle the book-keeping of logging, committing, and rolling-back of values according to the STM protocol. Naïve instrumentation introduces redundant function calls, for example, for values that are provably transaction-local. In addition, an STM with homogeneous implementation of its functions, while general and correct, will necessarily be less efficient than a highly-heterogeneous implementation: the latter can offer specialized functions that handle some specific cases more efficiently. For example, a homogeneous STM may offer a general `STMRead()` method, while a heterogeneous STM may offer also a specialized `STMReadThreadLocal()` method that assumes that the value read is thread-local, and as a consequence, can optimize away validations of that value.

Previous work has presented many compiler and runtime optimizations that aim to reduce the overhead of STM instrumentation. In this work we add to that body of knowledge by identifying additional new sources of sub-optimal instrumentation, and proposing optimizations to eliminate them. The sources are:

- **Redundant reads of memory locations that have been read before.** We use load elimination, a compiler technique that reduces the amount of memory reads by storing read values in local variables and using these variables instead of reading from memory. This allows us to reduce the number of costly STM library calls.
- **Redundant writes to memory locations that will be subsequently written to.** We use scalar promotion, a compiler technique that avoids redundant stores to memory locations, by storing to a local variable. Similar to load elimination, this optimization allows us to reduce the number of costly STM library calls.
- **Redundant writeset lookups for memory locations that have not been written to.** We discover memory accesses that read locations that have not been previously written to by the same transaction. Instrumentation for such reads can avoid writeset lookup.
- **Redundant writeset record-keeping for memory locations that will not be read.** We discover memory accesses that write to locations that will not be subsequently read by the same transaction. Instrumentation for such writes can therefore be made cheaper, e.g., by avoiding insertion to a Bloom filter.

Not all STM designs can benefit equally well from all the optimizations listed; For example, STMs that employ in-place updates, rather than lazy updates, will see less benefit from the redundant memory reads optimization. From here on, we restrict the discussion to the Transactional Locking II (TL2) [10] protocol, which benefits from all of the optimizations.

In addition to the new optimizations, we have implemented the following optimizations which have been used in other STMs: 1. Avoiding instrumentations of accesses to immutable and transaction-local memory; 2. Avoiding lock acquisitions and releases for thread-local memory; and 3. Avoiding readset population in read-only transactions.

To summarize, this paper makes the following contributions:

- We implement a set of common STM-specific analyses and optimizations.
- We present and implement a set of new analyses and optimizations to reduce overhead of STM instrumentation.
- We measure and show that our suggested optimizations can achieve significant performance improvements - up to 29-50% speedup in some workloads.

We proceed as follows: Section 2 gives a background of the STM we optimize. In Section 3 we describe the optimization opportunities that our analyses expose. In Section 4 we measure the impact of the optimizations. Section 5 reviews related work. We conclude in Section 6.

2 Background - Deuce, a Java-Based STM

In this section we briefly review the underlying STM protocol that we aim to optimize. We use the Deuce Java-based STM framework. Deuce [19] is a pluggable STM framework that allows different implementations of STM protocols; a developer only needs to implement the `Context` interface, and provide his own implementation for the various STM library functions. The library functions specify which actions to take on reading a field, writing to a field, committing a transaction, and rolling back a transaction.

Deuce is non-invasive: it does not modify the JVM or the Java language, and it does not require to re-compile source code in order to instrument it. It works by introducing a new `@Atomic` annotation. Java methods that are annotated with `@Atomic` are replaced with a retry-loop that attempts to perform and commit a transacted version of the method. All methods are duplicated; the transacted copy of every method is similar to the original, except that all field and array accesses are replaced with calls to the `Context` interface, and all method invocations are rewritten so that the transacted copy is invoked instead of the original.

Deuce works either in *online* or *offline* mode. In online mode, the entire process of instrumenting the program happens during runtime. A *Java agent* is attached to the running program, by specifying a parameter to the JVM. During runtime, just before a class is loaded into memory, the Deuce agent comes into play and transforms the program in-memory. To read and rewrite classes, Deuce uses ASM [6], a general-purpose bytecode manipulation framework.

In order to avoid the runtime overhead of the online mode, Deuce offers the offline mode, that performs the transformations directly on compiled `.class` files. In this mode, the program is transformed similarly, and the transacted version of the program is written into new `.class` files.

Deuce's STM library is homogeneous. In order to allow its methods to take advantage of specific cases where optimization is possible, we enhance each of its

STM functions to accept an extra incoming parameter, *advice*. This parameter is a simple bit-set representing information that was pre-calculated and may help fine-tune the instrumentation. For example, when writing to a field that will not be read, the advice passed to the STM write function will have 1 in the bit corresponding to "no-read-after-write".

In this work we focus on the Transactional Locking II (TL2) [10] protocol implementation in Deuce. In TL2, conflict detection is done by using a combination of versioned write-locks, associated with memory locations or objects, together with a global version clock. TL2 is a lazy-update STM, so values only written to memory at commit time; therefore locks are held for a very short amount of time. Our version of TL2 is word-based, and supports weak isolation [21] and flat nesting [3].

3 Optimization Opportunities

The following are optimization opportunities we have detected.

3.1 Preventing Redundant Memory Accesses

Load Elimination. Consider the following code fragment that is part of an atomic block (derived from the Java version of the STAMP suite):

```
for (int j = 0; j < nfeatures; j++) {
  new_centers[index][j] = new_centers[index][j] +
      feature[i][j];
}
```

A naïve STM compiler will instrument every array access in this fragment. However, the memory locations `new_centers[index]` and `feature[i]` are loop-invariant. We can calculate them once, outside the loop, and re-use the calculated values inside the loop. The technique of re-using values is a form of Partial Redundancy Elimination (PRE) optimization and is common in modern compilers. When PRE is applied to memory loads, it is called *Load Elimination*. The optimized version of the code will be equivalent to:

```
if (0 < nfeatures) {
  nci = new_centers[index];
  fi = feature[i];
  for (j = 0; j < nfeatures; j++) {
    nci[j] = nci[j] + fi[j];
  }
}
```

Many compilers refrain from applying the technique to memory loads (as opposed to arithmetic expressions). One of the reasons is that such a code transformation may not be valid in the presence of concurrency; for example, the compiler must make sure that the `feature[i]` memory location cannot be concurrently modified by a thread other than the one executing the above loop. This constraint, however, does not exist inside an atomic block, because the atomic block guarantees isolation from other concurrent transactions. An STM compiler can therefore enable PRE optimizations where they would not be possible with a regular compiler that does not support atomic blocks.

We note that this optimization is sound for all STM protocols that guarantee isolation. The performance boost achieved by it, however, is maximized with lazy-update STMs as opposed to in-place-update STMs. The reason is that lazy-update STMs must perform an expensive writeset lookup for every memory read, while in in-place-update STMs, memory reads are relatively cheap since they are done directly from memory. In fact, in such STMs, reads of memory locations that were read before can be optimized [29, 12] to perform just a direct memory read together with a consistency check. By using load elimination of memory locations, memory reads are transformed into reads of a local variable; as a result, such reads are made much faster and a lazy-update STM operates at in-place STM speeds.

Scalar Promotion. The dual to load elimination is *Scalar Promotion*. Consider the following code fragment, also from STAMP:

```
for (int i = 0; i < num_elts; i++) {
  moments[0] += data[i];
}
```

If this fragment appeared inside an atomic method, an STM compiler could take advantage of the isolation property to eliminate the multiple writes to the same memory location. An optimized version of the code would be equivalent to:

```
if (0 < num_elts) {
  double temp = moments[0];
  try {
    for (int i = 0; i < num_elts; i++) {
      temp += data[i];
    }
  } finally {
      moments[0] = temp;
  }
}
```

The advantage of the optimized version is that multiple memory writes are replaced with just one.

3.2 Preventing Redundant Writeset Operations

Redundant Writeset Lookups. Consider a field read statement `v = o.f` inside a transaction. The STM must produce and return the most updated value of `o.f`. In STMs that implement lazy update, there can be two ways to look up `o.f`'s value: if the same transaction has already written to `o.f`, then the most updated value must be found in the transaction's writeset. Otherwise, the most updated value is the one in `o.f`'s memory location. A naïve instrumentation will conservatively always check for containment in the writeset on every field read statement. With static analysis, we can gather information whether the accessed `o.f` was possibly already written to in the current transaction. If we can statically deduce that this is not the case, then the STM may skip checking the writeset, thereby saving processing time.

Redundant Writeset Record-Keeping. Consider a field write statement `o.f = v` inside a transaction. According to the TL2 protocol, the STM must update the writeset with the information that `o.f` has been written to. One of the design goals of the writeset is that it should be fast to search it; this is because subsequent reads from `o.f` in the same transaction must use the value that is in the writeset. But, some memory locations in the writeset will never be actually read in the same transaction. We can exploit this fact to reduce the amount of record-keeping that the writeset data-structure must handle. As an example, TL2 suggests implementing the writeset as a linked-list (which can be efficiently added-to and traversed) together with a Bloom filter (that can efficiently check whether a memory location exists in the writeset). If we can statically deduce that a memory location is written-to but will not be subsequently read in the same transaction, we can skip updating the Bloom filter for that memory location. This saves processing time, and is sound because there is no other purpose in updating the Bloom filter except to help in rapid lookups.

4 Experimental Results

In order to test the benefit of the above optimization opportunities, we used Deuce [19], a Java-based STM framework.

PRE optimizations (section 3.1) require no change to the actual Deuce runtime; they only require an extra preliminary optimization pass.

The optimization of preventing redundant writeset operations (section 3.2) needs to actually change the instrumentation. To do it, we enhance each of Deuce's STM library methods to accept an extra bit-set parameter, *advice*, every bit of which denotes an optimization opportunity.

We devised compile-time analyses that discover the opportunities and supply the *advice* parameters to the STM library method calls. The STM library methods detect the enabled bits in the *advice* parameters and apply the relevant optimizations. Specifically, the STM read method, upon seeing a 1 in the bit corresponding to "no-write-before-read", will avoid looking up the memory location in the writeset. Similarly, the STM write function, upon seeing a 1 in the bit corresponding to "no-read-after-write", will avoid updating the Bloom filter.

In order to discover which memory locations are read before they are written to, we perform an interprocedural, flow-sensitive, forward data flow analysis [17], that simulates the runtime contents of the readset. The analysis uses points-to information [17] to associate each abstract memory location, in each program point, with a tag representing whether the memory location was written to already or not. A similar, backward analysis, is used to discover memory locations that will not be read after they are written to.

Out test environment is a Sun UltraSPARC T2 Plus multicore machine with 2 CPUs, each with 8 cores at 1.2 GHz, each core with 8 hardware threads to a total of 128 threads.

4.1 Optimization Levels

We compared 5 levels of optimizations. The levels are cumulative so that every level includes all the optimizations of the previous levels. The *None* level is the most basic code, which blindly instruments every memory access. The *Common* level adds several well-known optimizations that are common in STMs. These include 1. Avoiding instrumentations of accesses to immutable and transaction-local memory; 2. Avoiding lock acquisitions and releases for thread-local memory; and 3. Avoiding readset population for read-only transactions. The *PRE* level consists of load elimination and scalar promotion optimizations. The *ReadOnly* level avoids redundant readset lookups for memory locations that have not been written to. Finally, the *WriteOnly* level avoids redundant writeset record-keeping for memory locations that will not be read.

4.2 Benchmarks

We experimented on a set of data structure-based microbenchmarks and several benchmarks from the Java version [9] of the STAMP [7] suite.

Data Structures Microbenchmarks. In the following microbenchmarks, we exercised three different data structures: *LinkedList* represents a sorted linked list implementation. *SkipList* represents a skiplist with random leveling. *Hash* represents an open-addressing hash table that uses a fixed-size array with no rehashing. Every data structure supports three atomic operations: adding an item, removing an item, and checking for containment of an item. The test consists of threads attempting to perform as many atomic operations as possible on a shared data structure; each thread chooses its next operation randomly, with 90% chance selecting the lookup operations, and 10% chance selecting addition or removal. The threads are stopped after 20 seconds.

In the microbenchmarks, we measure throughput, that is, the total number of operations performed. Each value is normalized relative to the results of a single-threaded run with no optimizations. The results appears in Figure 1. Each bar represents the median value of at least 10 runs.

STAMP Benchmarks. We tested four STAMP [7] benchmarks. *K-Means* implements k-means clustering. *Vacation* simulates an on-line travel reservation system. *Ssca2* performs several graph operations. In our tests we focused on Kernel 1, which generates a graph, and Kernel 2, which classifies large sets. *MatrixMul* is part of the Java version of the STAMP suite. It performs matrix multiplication. We could not test the other benchmarks from STAMP due to technical limitations: STAMP is a C library, and its Java port [1] is written in a special dialect of Java that requires manual conversion in order to compile on standard Java. After conversion, the tests ran with incorrect results.

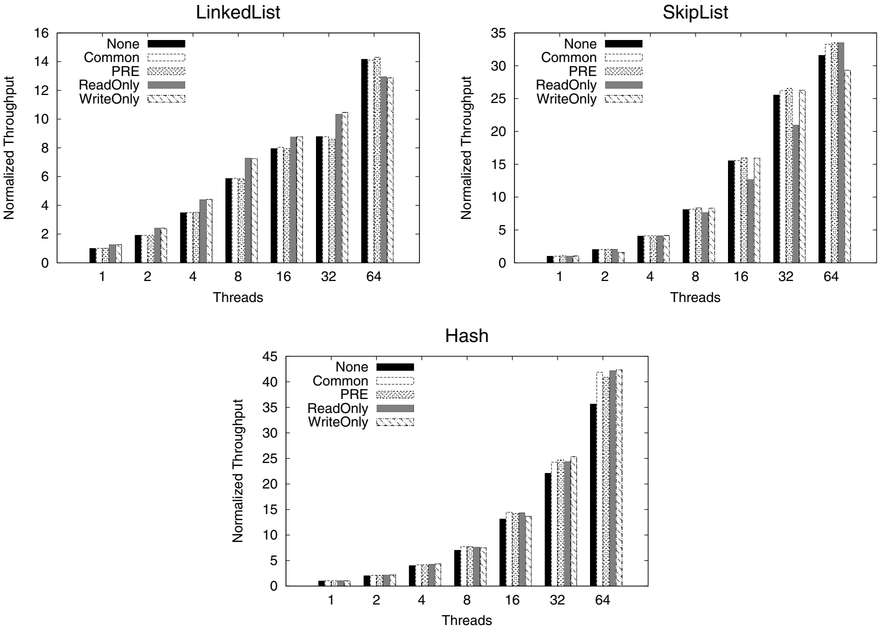


Fig. 1. Microbenchmarks comparative results (higher is better)

In the STAMP benchmarks we measured the time it took for each test to complete.[1] As before, the values are normalized relative to the single-threaded, no-optimizations run. The results appears in Figure 2.

4.3 Optimization Opportunities Breakdown

To understand to what extent optimizations are applicable to the benchmarks, we compared optimization-specific measures on single-threaded runs. The results appear in tables 1, 2. The measure for PRE is the percent of reads eliminated by load elimination and scalar promotion (compared to the Common level). The measure for ReadOnly is the percent of read statements that access memory locations that have not been written to before in the same transaction. The measure for WriteOnly is the percent of write statements that write to memory that will not be read afterwards in the same transaction. All numbers are measured dynamically at runtime. High percentages represent more optimization opportunities. Low percentages mean that we could not locate many optimization opportunities, either because they do not exist, or because our analyses were not strong enough to find them.

[1] Parameters used for the benchmarks: K-Means: -m 40 -n 40 -t 0.001 -i random-n16384-d24-c16.input; Vacation: -n 4 -t 5000000 -q 90 -r 65536 -u 80; Ssca2: -s 18; MatrixMul:130.

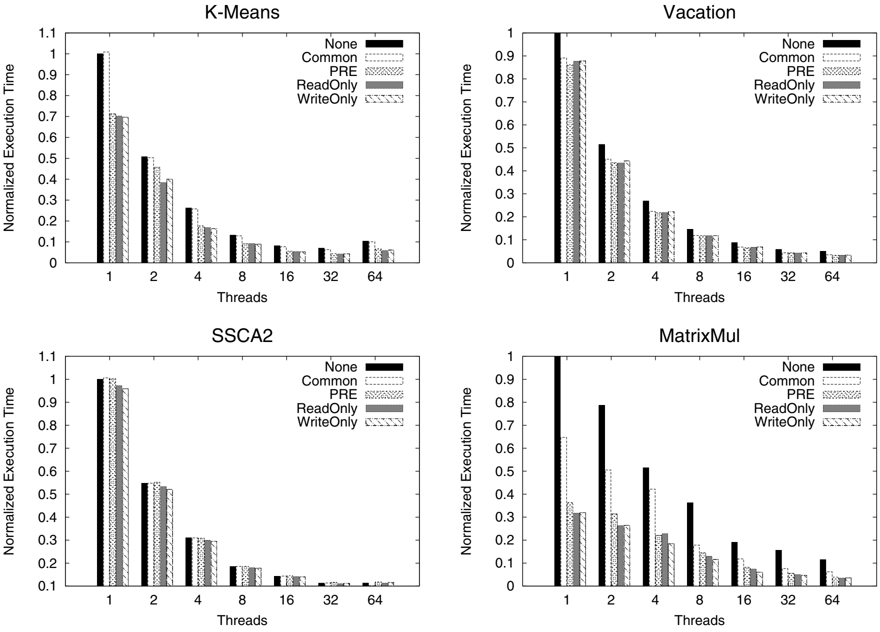


Fig. 2. STAMP Benchmarks comparative results (lower is better)

Table 1. Optimization opportunity measures per microbenchmark on a single-threaded run

	LinkedList	SkipList	Hash
PRE (% of reads eliminated)	0.17%	0.48%	0%
ReadOnly (% of reads from locations not written to before)	100%	3.56%	100%
WriteOnly (% of writes to locations not read from thereafter)	100%	0%	100%

4.4 Analysis

Our benchmarks show that the optimizations have improved performance to varying degrees. The most noticeable performance gain was due to PRE, especially in tight loops where many memory accesses were eliminated.

PRE. K-Means benefits greatly (up to 29% speedup) from load elimination: the above example (section 3.1) is taken directly from K-Means. MatrixMul also benefits from PRE due to the elimination of redundant reads of the main matrix object. Vacation achieves 4% speedup in the single-threaded run, but sees little to no speedup as the number of threads rises; this is because the eliminated loads exist outside of tight loops.

Our Scalar Promotion analysis, which focuses on finding loops where the same memory location is re-written in every iteration, was not able to find this pattern in any of the tested benchmarks. A more thorough analysis, that also considers

Table 2. Optimization opportunity measures per STAMP benchmark on a single-threaded run

	K-Means	Vacation	SSCA2	MatrixMul
PRE (% of reads eliminated)	56.86%	4%	0%	33.46%
ReadOnly (% of reads from locations not written to before)	56.05%	1.56%	100%	100%
WriteOnly (% of writes to locations not read from thereafter)	5.26%	0.02%	100%	100%

writes outside of loops, may have been able to detect some opportunities for enabling the Scalar Promotion optimization.

ReadOnly. LinkedList benefits (throughput increased by at up 28%) from the ReadOnly optimization, which applies to reading the next node in the list. This optimization is valid since traversal is done prior to updating the next node. We note that the read of the head of the list is also a read-only memory access; however this is subsumed by the Common optimizations because the head is immutable. Hash's throughput is increased by up to 4% due to ReadOnly opportunities in the `findIndex()` method, which is called on every transaction. SSCA2 and MatrixMul see modest benefits.

Our analysis discovered that in 4 benchmarks: LinkedList, Hash, SSCA2 and MatrixMul, all reads are from memory locations that have not been written to before in the same transaction. We suspect that reading before writing is the norm in almost all transactions, but our analyses could prove it only in these 4.

WriteOnly. The WriteOnly optimization is effective on transactions with a high number of successful writes. Hash, which makes over 11 million writes in the single-threaded run, shows up to 1.7% improvement, while SSCA2 with a similar number of writes, is up to 4% faster. Other benchmarks with 100% optimization of writes, MatrixMul and LinkedList, perform much less writes (only around 15,000) and therefore see very little benefit from the WriteOnly optimization.

We conclude that PRE shows the most impressive gains while ReadOnly follows next. The impact of the WriteOnly optimization is relatively small. The reason that the WriteOnly optimization is less effective is that the insertion to the Bloom filter is already a very quick operation. In addition, our tested workloads have a relatively low amount of writes.

The effectiveness of the optimizations varies widely with the different workloads and benchmarks. For example, the fully optimized LinkedList is 27% faster (compared to Common) on 1 thread, and 19% faster on 32 threads. MatrixMul is 50% faster on a single-threaded run. However, SkipList and Vacation shows no more than 1% gain on any number of threads due to lack of optimization opportunities.

While generally the optimizations improve throughput and save time, at some workloads their effects are detrimental. For example, the optimized versions of

LinkedList on 64 threads, or SkipList on 8 threads, perform worse than their non-optimized versions. We suspect that, on some specific workloads, making some transactions faster could generate conflicts with other advancing transactions.

5 Related Work

The literature on compiler optimizations that are specific to transactional memory implementations revolves mostly around in-place-update STMs [27]. Harris et al. [14] presents the baseline STM optimizations that appear in many subsequent works. Among them are: Decomposition of STM library functions to heterogeneous parts; code motion optimizations that make use of this decomposition to reduce the number of "open-for-read" operations; early upgrade of "open-for-read" into "open-for-write" operation if a write-after-read will occur; suppression of instrumentation for transaction-local objects; and more. In [2], immutable objects are also exempt from instrumentation. In addition, standard compiler optimization techniques (see [17] for a full treatment), such as loop peeling, method inlining, and redundancy elimination algorithms are applied to atomic blocks.

Eddon and Herlihy [12] apply fully interprocedural analyses to discover thread-locality and subsequent accesses to the same objects. Such discoveries are exploited for optimized "fast path" handling of the cases. Similar optimizations also appear in Wang et al. [29], Dragojevic et al. [11]. We note that the above works optimize for in-place-update STMs. In such an STM protocol, once a memory location is "open-for-write", memory accesses to it are nearly transparent (free), because the memory location is exclusively owned by the transaction. Our work is different because it targets lazy-update STMs, where subsequent instrumented accesses to memory cannot be made much cheaper than initial accesses; e.g., the writeset must still be searched on every read. We solve this problem by transforming instrumented reads and writes, that access shared memory, into uninstrumented reads and writes that access local variables.

Spear et al. [27] proposes several optimizations for a TL2-like STM: 1. When multiply memory locations are read in succession, each read is instrumented such that the location is pre-validated, read, and then post-validated. By re-ordering the instructions such that all the pre-validations are grouped together, followed by the reads, and concluded by the post-validations, they increase the time window between memory fences, such that the CPU could parallelize the memory reads. 2. Post-validation can sometimes be postponed as long as working with "unsafe" values can be tolerated; This eliminates or groups together expensive memory barrier operations. Shpeisman et al. [25]'s *barrier aggregation* is a similar, but simpler, optimization that re-uses barriers inside a basic block if they guard the same object.

Beckman et al. [5]'s work provides optimizations for thread-local, transaction-local and immutable objects that are guided by *access permissions*. These are Java attributes that the programmer must use to annotate program references. For example, the `@Imm` attribute denotes that the associated reference variable is immutable. Access permissions are verified statically by the compiler, and then used to optimize the STM instrumentation for the affected variables.

Partial redundancy elimination (PRE) [18, 22] techniques are widely used in the field of compiler optimizations; however, most of the focus was at removing redundancies of arithmetic expressions. Fink et al. [13] and Hosking et al. [16] were the first to apply PRE to Java access path expressions, for example, expressions like a.b[i].c . This variant of PRE is also called *load elimination.* As a general compiler optimization, this optimization may be unsound because it may miss concurrent updates by a different thread that changes the loaded value. Therefore, some works [4, 28] propose analyses that detect when load elimination is valid. *Scalar promotion*, which eliminates redundant memory writes, was introduced by Lu and Cooper [20], and improved by later works (e.g. [26]).

6 Conclusions and Further Work

We showed that two pre-existing optimizations, load elimination and scalar promotion, can be used in an optimizing STM compiler. Where standard compilers need perform an expensive cross-thread analysis to enable these optimizations, an STM compiler can rely on the atomic block's isolation property to enable them. We also highlighted two redundancies in STM read and write operations, and showed how they can be optimized.

We implemented a compiler pass that performs these STM-specific code motion optimizations, and another pass that uses static analysis methods to discover optimization opportunities for redundant STM read and write operations. We have augmented the interface of the underlying STM compiler, Deuce, to accept information about which optimizations to enable at every STM library method call, and modified the STM methods themselves to apply the optimizations when possible.

The combined performance benefit of all the optimizations presented here varies with the workload and the number of threads. While some benchmarks see little to no improvement (e.g., SSCA2 and SkipList), we have observed speedups of up to 50% and 29% in other benchmarks (single-threaded MatrixMul and K-Means, respectively).

There are many ways to improve upon this research. For example, a drawback of the optimizations presented here is that they require full interprocedural analysis to make sound decisions. It may be interesting to research which similar optimizations can be enabled with less analysis work, for example, with running only intraprocedural analyses, or with partial analysis data that is calculated at runtime.

Acknowledgments. The authors thank the anonymous reviewers of this paper's draft for their clear and spot-on remarks. This paper was supported in part by the European Union grant FP7-ICT-2007-1 (project VELOX), by grants from Intel Corporation and Sun Microsystems, and by the Laura Schwarz-Kipp Institute of Computer Networks.

References

[1] http://demsky.eecs.uci.edu/software.php

[2] Adl-Tabatabai, A.-R., Lewis, B.T., Menon, V., Murphy, B.R., Saha, B., Shpeisman, T.: Compiler and runtime support for efficient software transactional memory. In: PLDI 2006: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 26–37. ACM, New York (2006) ISBN 1-59593-320-4, doi http://doi.acm.org/10.1145/1133981.1133985

[3] Agrawal, K., Leiserson, C.E., Sukha, J.: Memory models for open-nested transactions. In: MSPC 2006: Proceedings of the 2006 Workshop on Memory System Performance and Correctness, pp. 70–81. ACM, New York (2006) ISBN 1-59593-578-9, doi http://doi.acm.org/10.1145/1178597.1178610

[4] Barik, R., Sarkar, V.: Interprocedural load elimination for dynamic optimization of parallel programs. In: PaCT 2009, pp. 41–52. IEEE Computer Society, Los Alamitos (2009) ISBN 978-0-7695-3771-9, doi http://dx.doi.org/10.1109/PACT.2009.32

[5] Beckman, N.E., Kim, Y.P., Stork, S., Aldrich, J.: Reducing STM overhead with access permissions. In: IWACO 2009: International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming, pp. 1–10. ACM, New York (2009) ISBN 978-1-60558-546-8, doi http://doi.acm.org/10.1145/1562154.1562156

[6] Bruneton, E., Lenglet, R., Coupaye, T.: ASM: A code manipulation tool to implement adaptable systems. In: Adaptable and Extensible Component Systems (2002)

[7] Cao Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: IISWC 2008: Proceedings of The IEEE International Symposium on Workload Characterization (September 2008)

[8] Cascaval, C., Blundell, C., Michael, M., Cain, H.W., Wu, P., Chiras, S., Chatterjee, S.: Software transactional memory: why is it only a research toy? Commun. ACM 51(11), 40–46 (2008) ISSN 0001-0782, doi http://doi.acm.org/10.1145/1400214.1400228

[9] Demsky, B., Dash, A.: Evaluating contention management using discrete event simulation. In: Fifth ACM SIGPLAN Workshop on Transactional Computing (April 2010)

[10] Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)

[11] Dragojevic, A., Ni, Y., Adl-Tabatabai, A.-R.: Optimizing transactions for captured memory. In: SPAA 2009: Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures, pp. 214–222. ACM, New York (2009) ISBN 978-1-60558-606-9, doi http://doi.acm.org/10.1145/1583991.1584049

[12] Eddon, G., Herlihy, M.: Language support and compiler optimizations for STM and transactional boosting. In: Janowski, T., Mohanty, H. (eds.) ICDCIT 2007. LNCS, vol. 4882, pp. 209–224. Springer, Heidelberg (2007)

[13] Fink, S.J., Knobe, K., Sarkar, V.: Unified analysis of array and object references in strongly typed languages. In: SAS 2000. LNCS, vol. 1824, pp. 155–174. Springer, Heidelberg (2000) ISBN 3-540-67668-6

[14] Harris, T., Plesko, M., Shinnar, A., Tarditi, D.: Optimizing memory transactions. In: Conference on Programming Language Design and Implementation. ACM SIGPLAN, pp. 14–25 (June 2006)

[15] Herlihy, M., Eliot, J., Moss, B.: Transactional memory: Architectural support for lock-free data structures. In: Proceedings of the 20th Annual International Symposium on Computer Architecture, pp. 289–300 (1993)

[16] Hosking, A.L., Nystrom, N., Whitlock, D., Cutts, Q.I., Diwan, A.: Partial redundancy elimination for access path expressions. In: Proceedings of the Workshop on Object-Oriented Technology, London, UK, pp. 138–141. Springer, Heidelberg (1999) ISBN 3-540-66954-X

[17] Kennedy, K., Allen, J.R.: Optimizing compilers for modern architectures: a dependence-based approach. Morgan Kaufmann Publishers Inc., San Francisco (2002) ISBN 1-55860-286-0

[18] Knoop, J., Rüthing, O., Steffen, B.: Lazy code motion. SIGPLAN Not. 27(7), 224–234 (1992) ISSN 0362-1340, doi http://doi.acm.org/10.1145/143103.143136

[19] Korland, G., Shavit, N., Felber, P.: Noninvasive concurrency with Java STM. In: MultiProg 2010: Programmability Issues for Heterogeneous Multicores (January 2010), http://www.deucestm.org/

[20] Lu, J., Cooper, K.D.: Register promotion in C programs. In: PLDI 1997: Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation, pp. 308–319. ACM, New York (1997) ISBN 0-89791-907-6, doi http://doi.acm.org/10.1145/258915.258943

[21] Martin, M., Blundell, C., Lewis, E.: Subtleties of transactional memory atomicity semantics. IEEE Comput. Archit. Lett. 5(2), 17 (2006) ISSN 1556-6056, doi http://dx.doi.org/10.1109/L-CA.2006.18

[22] Morel, E., Renvoise, C.: Global optimization by suppression of partial redundancies. Commun. ACM 22(2), 96–103 (1979) ISSN 0001-0782, doi http://doi.acm.org/10.1145/359060.359069

[23] Ni, Y., Welc, A., Adl-Tabatabai, A.-R., Bach, M., Berkowits, S., Cownie, J., Geva, R., Kozhukow, S., Narayanaswamy, R., Olivier, J., Preis, S., Saha, B., Tal, A., Tian, X.: Design and implementation of transactional constructs for C/C++. In: OOPSLA 2008: Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, pp. 195–212 (2008)

[24] Shavit, N., Touitou, D.: Software transactional memory. In: Proc. of the 12th Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 204–213 (1995)

[25] Shpeisman, T., Menon, V., Adl-Tabatabai, A.-R., Balensiefer, S., Grossman, D., Hudson, R.L., Moore, K.F., Saha, B.: Enforcing isolation and ordering in stm. SIGPLAN Not. 42(6), 78–88 (2007) ISSN 0362-1340, doi http://doi.acm.org/10.1145/1273442.1250744

[26] So, B., Hall, M.W.: Increasing the applicability of scalar replacement. In: CC, pp. 185–201 (2004)

[27] Spear, M.F., Michael, M.M., Scott, M.L., Wu, P.: Reducing memory ordering overheads in software transactional memory. In: CGO 2009: Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization, Washington, DC, USA, pp. 13–24. IEEE Computer Society, Los Alamitos (2009) ISBN 978-0-7695-3576-0, doi http://dx.doi.org/10.1109/CGO.2009.30

[28] von Praun, C., Schneider, F., Gross, T.R.: Load elimination in the presence of side effects, concurrency and precise exceptions. In: Rauchwerger, L. (ed.) LCPC 2003. LNCS, vol. 2958, pp. 390–405. Springer, Heidelberg (2004)

[29] Wang, C., Chen, W.-Y., Wu, Y., Saha, B., Adl-Tabatabai, A.-R.: Code generation and optimization for transactional memory constructs in an unmanaged language. In: CGO 2007: Proceedings of the International Symposium on Code Generation and Optimization, Washington, DC, USA, pp. 34–48. IEEE Computer Society, Los Alamitos (2007) ISBN 0-7695-2764-7, doi http://dx.doi.org/10.1109/CGO.2007.4

A Parallel Numerical Solver Using Hierarchically Tiled Arrays

James C. Brodman[1], G. Carl Evans[1], Murat Manguoglu[2], Ahmed Sameh[2], María J. Garzarán[1], and David Padua[1]

[1] University of Illinois at Urbana-Champaign, Dept. of Computer Science
{brodman2,gcevans,garzaran,padua}@illinois.edu
[2] Purdue University, Dept. of Computer Science
{mmanguog,sameh}@cs.purdue.edu

Abstract. Solving linear systems is an important problem for scientific computing. Exploiting parallelism is essential for solving complex systems, and this traditionally involves writing parallel algorithms on top of a library such as MPI. The SPIKE family of algorithms is one well-known example of a parallel solver for linear systems.

The Hierarchically Tiled Array data type extends traditional data-parallel array operations with explicit tiling and allows programmers to directly manipulate tiles. The tiles of the HTA data type map naturally to the block nature of many numeric computations, including the SPIKE family of algorithms. The higher level of abstraction of the HTA enables the same program to be portable across different platforms. Current implementations target both shared-memory and distributed-memory models.

In this paper we present a proof-of-concept for portable linear solvers. We implement two algorithms from the SPIKE family using the HTA library. We show that our implementations of SPIKE exploit the abstractions provided by the HTA to produce a compact, clean code that can run on both shared-memory and distributed-memory models without modification. We discuss how we map the algorithms to HTA programs as well as examine their performance. We compare the performance of our HTA codes to comparable codes written in MPI as well as current state-of-the-art linear algebra routines.

1 Introduction

Computer simulation has become an important tool for scientists and engineers to predict weather, forecast prices for financial markets, or test vehicle safety. Increasing the performance of these simulations is important to improve the accuracy of the prediction or to increase the number of tests that can be performed. One way to achieve this performance improvement is the parallelization of the kernels that lie at the core of many of these simulations and that solve systems of equations or perform signal transformations. Today many different types of computing platforms can be used to run these parallel codes, such as the new ubiquitous multicore, large clusters of machines where each node is typically a multicore, and the accelerators or clusters of accelerators like

K. Cooper, J. Mellor-Crummey, and V. Sarkar (Eds.): LCPC 2010, LNCS 6548, pp. 46–61, 2011.

the Cell Processor or GPUs. However, the many available options for parallel execution have increased the difficulty of the programmer's task as they usually must rewrite their computations with a different programming model for each different type of computing platform.

Programmer productivity can be improved with a programming model that produces one portable code that can target several different types of parallel platforms. We believe portable codes can be obtained by using high level abstractions that hide the details of the underlying architecture from the programmers and allow them to focus on the correct implementation of their algorithms. However, one does not want to raise the level of abstraction so high that programmers sacrifice control over performance. The Hierarchically Tiled Array (HTA) is a data type that uses abstractions to allow programmers to write portable numerical computations. HTA uses tiling as a high-level abstraction to facilitate the specification of the algorithms, while allowing the programmer to control the performance of their programs.

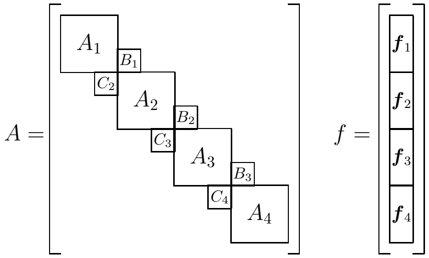
In this paper we show a proof-of-concept for high-performance computations that are portable across both shared-memory and message-passing. We present several versions of SPIKE, a parallel solver for linear banded systems, implemented using the HTA data type. We show that our implementations exploit the abstractions provided by the HTA to produce compact, clean code. Our experimental results show that the same code provides competitive performance when running on message-passing and shared-memory platforms.

The rest of this paper is organized as follows. Section 2 describes the SPIKE family of algorithms for solving banded systems of equations. Section 3 describes the Hierarchically Tiled Array data type used in our implementations. Section 4 describes how we actually implement several different SPIKE algorithms using HTAs. Section 5 presents the performance of our SPIKE implementations using both the shared-memory and distributed-memory runtimes and compares them to other libraries. Section 6 discusses related work and Section 7 summarizes our work.

2 SPIKE

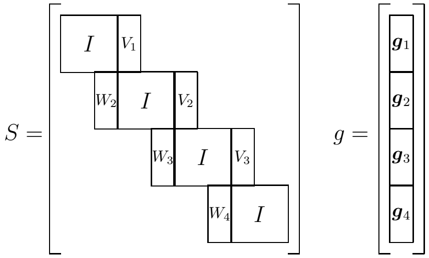
Linear solvers are an important class of numerical computation. Many important problems are sparse. It is well known that the desired data structure to represent sparse systems influences the performance of solvers for this type of linear system. These computations do not use dense arrays but rather only store the elements of a matrix that may be non-zero. Such storage mechanisms reduce not only the memory footprint but can also reduce the amount of computation needed by only performing computation on relevant elements. The SPIKE family of algorithms [15] is one such parallel solver for banded linear systems of equations.

Consider a linear system of the form $Ax = f$, where A is a banded matrix of order n with bandwidth much less than n. One can partition the system into p diagonal blocks. Given $p = 4$, the partitioned system is of the form,



where each A_i is a banded matrix of order n/p. The matrices B_i and C_i are of order m where the bandwidth of the original matrix A is $2m + 1$. Only the A, B, and C blocks need to be stored for this type of sparse matrix.

Let the block diagonal matrix $D = \text{diag}(A_1, ..., A_4)$. If one were to left-multiply each side of the above by D^{-1}, one would obtain a system of the form:



However, instead of computing D^{-1}, one can compute, as seen below, the blocks of V and W, or, the *spikes* by solving a system of equations. The spikes have the same width, m, as the B and C tiles in the original system.

$$A_i \left[V_i, W_i\right] = \begin{bmatrix} 0 & C_i \\ . & 0 \\ . & . \\ 0 & . \\ B_i & 0 \end{bmatrix} \tag{1}$$

Solving the original system $Ax = f$ now consists of three steps.

1. Solve (1)
2. Solve $Dg = f$
3. Solve $Sx = g$

The solution of the system $Dg = f$ yields the modified RHS for the system in the third step. Notice that each blocks of D are independent and thus can be computed in parallel. Solving the third system can be further reduced by solving the system $\hat{S}\hat{x} = \hat{g}$, which consists of the m rows of S directly above and below the boundaries between the I tiles. The spikes, f, and g can also be partitioned so that we have:

$$V_j = \begin{bmatrix} V_j^{(t)} \\ V_j' \\ V_j^{(b)} \end{bmatrix} \; W_j = \begin{bmatrix} W_j^{(t)} \\ W_j' \\ W_j^{(b)} \end{bmatrix} \; x_j = \begin{bmatrix} x_j^{(t)} \\ x_j' \\ x_j^{(b)} \end{bmatrix} \; g_j = \begin{bmatrix} g_j^{(t)} \\ g_j' \\ g_j^{(b)} \end{bmatrix} \tag{2}$$

The reduced system thus takes the following form:

$$\begin{bmatrix} I_m & 0 & V_1^{(t)} & & & & & & & \\ 0 & I_m & V_1^{(b)} & 0 & & & & & & \\ 0 & W_2^{(t)} & I_m & 0 & V_2^{(t)} & & & & & \\ & W_2^{(b)} & 0 & I_m & V_2^{(b)} & 0 & & & & \\ & & \ddots & \ddots & \ddots & \ddots & \ddots & & & \\ & & & & 0 & W_{p-1}^{(t)} & I_m & 0 & V_{p-1}^{(t)} & \\ & & & & & W_{p-1}^{(b)} & 0 & I_m & V_{p-1}^{(b)} & 0 \\ & & & & & & 0 & W_p^{(t)} & I_m & 0 \\ & & & & & & & W_p^{(b)} & 0 & I_m \end{bmatrix} \begin{bmatrix} \mathrm{x}_1^{(t)} \\ \mathrm{x}_1^{(b)} \\ \mathrm{x}_2^{(t)} \\ \mathrm{x}_2^{(b)} \\ \vdots \\ \mathrm{x}_{p-1}^{(t)} \\ \mathrm{x}_{p-1}^{(b)} \\ \mathrm{x}_p^{(t)} \\ \mathrm{x}_p^{(b)} \end{bmatrix} = \begin{bmatrix} \mathrm{g}_1^{(t)} \\ \mathrm{g}_1^{(b)} \\ \mathrm{g}_2^{(t)} \\ \mathrm{g}_2^{(b)} \\ \vdots \\ \mathrm{g}_{p-1}^{(t)} \\ \mathrm{g}_{p-1}^{(b)} \\ \mathrm{g}_p^{(t)} \\ \mathrm{g}_p^{(b)} \end{bmatrix}$$

Finally, once the solution to the reduced system has been directly computed sequentially, we will have the values of $x^{(b)}$s and $x^{(t)}$s. The rest of x can then be computed as follows:

$$\begin{cases} x_1' = g_1' - V_1' x_2^{(t)}, \\ x_j' = g_j' - V_j' x_{j+1}^{(t)} - W_j' x_{j-1}^{(b)}, j = 2, ..., p-1, \\ x_p' = g_p' - W_p x_{p-1}^{(b)}. \end{cases} \tag{3}$$

Thus the SPIKE algorithm can be broken down into the following steps:

1. Factorize the diagonal blocks of A.
2. Compute the spikes using the factorization obtained in the previous step and compute the right hand side. Solve (1) and $Dg = f$.
3. Form and solve the reduced system.
4. Compute the rest of x.

2.1 SPIKE Variants

The original SPIKE algorithm explained above has many variants. These variants target systems of equations with certain properties in order to reduce the amount of computation performed. They also increase the amount of parallelism available during different stages of the algorithm. In this paper we focus on two variants that use a truncated scheme to solve the reduced system. The truncated scheme is useful for systems that are diagonally dominant. In diagonally dominant systems, the values in the spikes far from the diagonal are likely to be very close to zero and therefore contribute little to the solution. Consequently, the truncated scheme treats these values as zero and only computes the $m \times m$ portion of the spikes close to the diagonal, specifically, $V^{(b)}$ and $W^{(t)}$.

This is accomplished by either using the LU or UL factorization computed for the blocks of the diagonal.

The two variants we present are called TU and TA, and both implement the truncated scheme. LU factorization of A_i is used to solve the bottom tips, $V_i^{(b)}$, of the spikes and the UL factorization of A_i is used to solve for the top tips, $W_i^{(t)}$, of the spikes. The difference between TU and TA lays in the decomposition of the work. In the TU scheme, the original matrix is partitioned into as many blocks as there are processors. Figure 1 shows this partitioning for the case with 4 processors. In this figure $\hat{B}_i$ and $\hat{C}_i$ are $\left[0 \ldots\ 0\ B_i\right]^T$ and $\left[C_i\ 0 \ldots\ 0\right]^T$ as in equation 1.

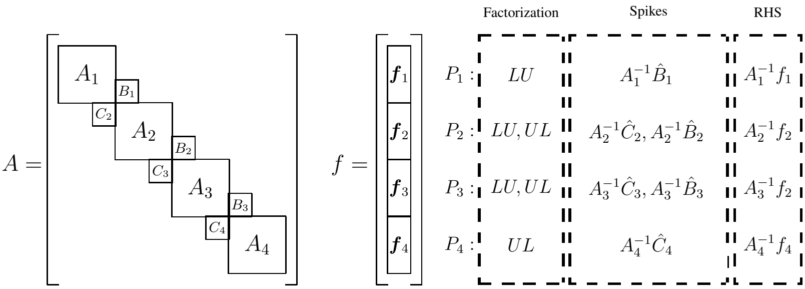

$A =$
A_1
B_1
C_2
A_2
B_2
C_3
A_3
B_3
C_4
A_4
$f =$
$\boldsymbol{f}_1$
$\boldsymbol{f}_2$
$\boldsymbol{f}_3$
$\boldsymbol{f}_4$
Factorization
Spikes
RHS
P_1 : LU $A_1^{-1}\hat{B}_1$ $A_1^{-1}f_1$
P_2 : LU, UL $A_2^{-1}\hat{C}_2, A_2^{-1}\hat{B}_2$ $A_2^{-1}f_2$
P_3 : LU, UL $A_3^{-1}\hat{C}_3, A_3^{-1}\hat{B}_3$ $A_3^{-1}f_2$
P_4 : UL $A_4^{-1}\hat{C}_4$ $A_4^{-1}f_4$

Fig. 1. Spike TU Partitioning

The TA scheme arises from the fact that the factorization step dominates execution time. TA is similar to TU with the exception that it partitions the matrix in a different fashion. Instead of each processor computing both LU and UL for a block since some blocks must compute two spikes, each processor now computes either LU or UL for a block but not both in order to compute a single spike. Note that this scheme partitions the matrix into fewer blocks than the TU scheme does, but results in better load balance for the computation of the spikes. Figure 2 shows this partitioning for 4 processors using $\hat{B}_i$ and $\hat{C}_i$ which are extended as above.

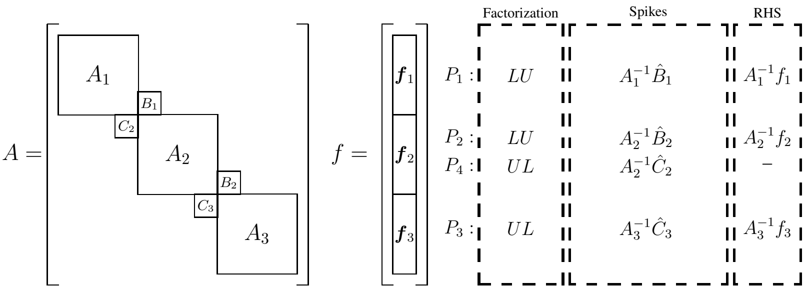

$A =$
A_1
B_1
C_2
A_2
B_2
C_3
A_3
$f =$
$\boldsymbol{f}_1$
$\boldsymbol{f}_2$
$\boldsymbol{f}_3$
Factorization
Spikes
RHS
P_1 : LU $A_1^{-1}\hat{B}_1$ $A_1^{-1}f_1$
P_2 : LU $A_2^{-1}\hat{B}_2$ $A_2^{-1}f_2$
P_4 : UL $A_2^{-1}\hat{C}_2$ –
P_3 : UL $A_3^{-1}\hat{C}_3$ $A_3^{-1}f_3$

Fig. 2. Spike TA Partitioning

Both versions of the algorithm compute the $W^{(t)}$, $V^{(b)}$, and g tips that are needed for the truncated reduced system, shown in Figure 3. This system will be block diagonal and has one less block than the original system. Thus when solving with p processors TU will have $p-1$ blocks in the reduced system while TA will have $(p+2)/2-1$ blocks in the reduced system. Thus the TU version will have more parallelism than the TA version in this stage of the computation. Unlike the original SPIKE algorithm, the reduced system for truncated schemes can be solved in parallel via a direct scheme where each block has the following form:

$$\begin{bmatrix} I_m & V_j^{(b)} \\ W_{j+1}^{(t)} & I_m \end{bmatrix} \begin{bmatrix} x_j^{(b)} \\ x_{j+1}^{(t)} \end{bmatrix} = \begin{bmatrix} g_j^{(b)} \\ g_{j+1}^{(t)} \end{bmatrix} \qquad (4)$$

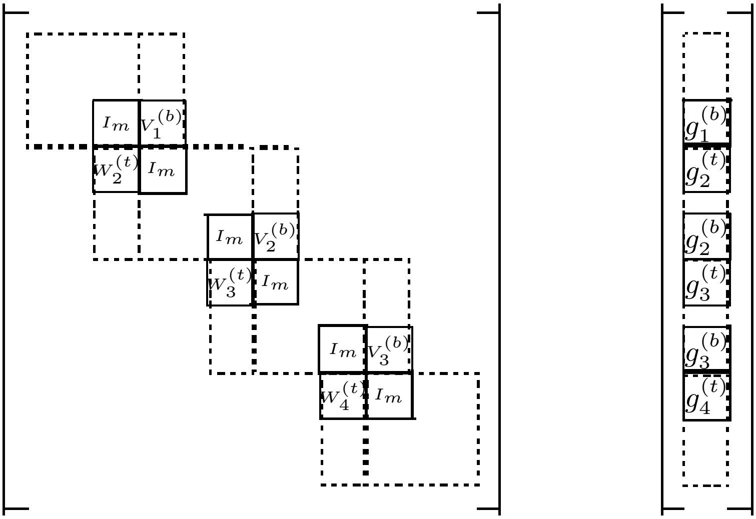


Fig. 3. Data sources for TU reduced with 4 blocks

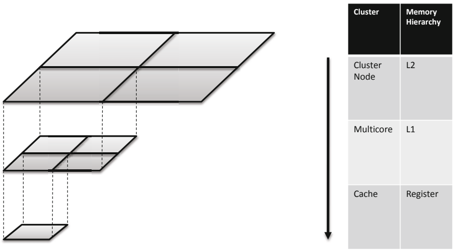
Finally the solution to the original system is recovered by solving:

$$A_j x_j = f_j - \begin{bmatrix} 0 \\ \vdots \\ 0 \\ B_j \end{bmatrix} x_{j+1}^{(t)} - \begin{bmatrix} C_j \\ 0 \\ \vdots \\ 0 \end{bmatrix} x_{j-1}^{(b)} \qquad (5)$$

This can be done in parallel with either the LU or UL factorization of A_j. Here again the TU version has more parallelism the the TA version.

3 Hierarchically Tiled Arrays

The Hierarchically Tiled Array [4,9,3], or HTA, data type extends earlier work on data parallel array languages with explicit tiling. An HTA object is a tiled array whose elements can be either scalars or tiles. HTAs can have several levels of tiling, allowing them to adapt to the hierarchical nature of modern machines. Figure 4 illustrates two examples of how HTAs can exploit hierarchical tiling. For example, tiles in the outermost level are distributed across the nodes in a cluster; then, the tile in each node can be further partitioned among the processors of the multicore node.



Cluster	Memory Hierarchy
Cluster Node	L2
Multicore	L1
Cache	Register

Fig. 4. Hierarchical Tiling

The HTA data type makes tiles first class objects that are explicitly referenced and extends traditional Fortran 90 style array operations to function on tiles. Figure 5 illustrates the ways in which HTAs can be indexed. HTAs permit indexing of both tiles and scalars. We use () to refer to tiles and [] to refer to elements. This way, A(0,0) refers to the top left tile of HTA A and A(0,1) [0,1] refers to the element [0,1] of the top right tile of HTA A. Also, HTAs support the triplet array notation in order to index multiple scalars and/or tiles, as shown in Figure 5 when accessing the two bottom tiles of A by using A(1, 0:1). Scalars can also be accessed in a flattened fashion that ignores the tiling structure of the HTA, as shown in the example when accessing the element A[0,3]. This flattened notation is useful for tasks such as initializing data.

HTAs provide several data parallel operators to programmers. One example is element-by-element operations such as adding or multiplying two arrays. HTAs also provide support for operations such as scans, reductions, matrix multiplication, and several types of transpositions of both tiles and data. Communication of data between tiles is usually expressed through array assignments, but can also be expressed with special operations such as transpositions or reductions.

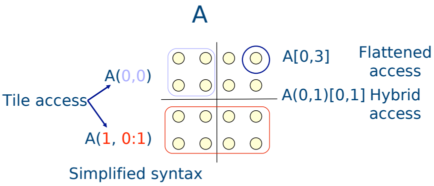


Fig. 5. HTA Indexing

The HTA also provides a map operator that applies a programmer-specified function to the corresponding tiles of a set of HTAS. On a parallel platform these functions may be executed in parallel. This way, the `A.hmap(func1())` will invoke `func1()` on all the tiles of HTA A. If another HTA B is passed as an argument of `hmap`, then `func1()` will execute on the corresponding tiles of both HTAs, A and B.

HTA Programs thus consist of a sequence of data parallel operators applied to HTAs that are implicitly separated by a barrier. These programs appear sequential to the programmer as all parallelism is encapsulated inside the operators. Numbers and sizes of tiles are chosen both to control the granularity of parallelism and to enhance locality.

The HTA data type has been implemented as libraries for both C++ and MATLAB. The C++ library currently supports two platforms: distributed-memory built on top of MPI and shared-memory built on top of Intel's Threading Building Blocks. These multiple backends allows programmers to write one code using HTAs that can run on either multicores or clusters.

4 Implementing SPIKE with HTAs

An implementation of the SPIKE family of algorithms is available in the Intel Adaptive Spike-based Solver[1], or SpikePACK. It is implemented using MPI and Fortran. We choose to implement several SPIKE algorithms using HTAs for two reasons. First, writing SPIKE using HTAs would allow programmers to write one portable code that can be run on both shared-memory and distributed-memory target platforms. Second, the HTA notation allows for an elegant, clean implementation of the algorithms. An HTA SPIKE would more closely resemble the high-level mathematical expression of the algorithms than Fortran+MPI. Communication takes the form of simple array assignments between tiles.

We have implemented the `TU` and `TA` variants of SPIKE with HTAs. The tiles of the HTAs map to the blocks of the banded linear system. The bands of the system are stored inside the tiles using the banded storage format used by LAPACK. Since the code makes extensive use of LAPACK routines such as `DGBTRF` and `DGBTRS` to factorize and solve banded systems, we modified the HTA library to support column-major data layout due to the Fortran origins of these routines. The HTA library is written in C++ and originally only supported row-major layout.

The blocks of the bands, spikes, and reduced system are all represented as HTA objects. Each of these collections of blocks can be viewed as an array of tiles. The storage for the blocks of the band is overwritten to store the LU or UL factorizations of each block. The storage for the `B` and `C` blocks is likewise overwritten to contain the tips of the spikes. The number of partitions used by the algorithm for a given number of processors directly determines the tiling of the HTA objects. The algorithm is represented as a sequence of data parallel operations. The semantics state that each data parallel operation is followed by an implicit barrier. This allows the programmer to reason about the algorithm sequentially as the parallelism is thus encapsulated inside of the data parallel operators. The data parallel operations often are represented as `hmap` operations.

This is the mechanism through which we apply LAPACK kernels in parallel across all the tiles of an HTA. Our implementations also use the array operations provided by the HTA library to construct the reduced system. When coupled with HTA's first class tile objects, array operations enable programmers to write simple, compact statements that can communicate a range of data from one set of tiles to another. This contrasts with a Fortran+MPI approach where it is difficult to separate the algorithm from the implementation.

Porting the programs from one platform to another is accomplished by simply changing the header file for the library. In order to target MPI, one includes `htalib_mpi.h`. In order to target TBB, one includes `htalib_shmem.h`.

4.1 TU

Figure 6 presents the core of our implementation. We use a simplified notation to represent triplets. Recall that `TU` partitions the matrix into as many blocks as processors. The HTAs `LUA` and `ULA` initially are identical and contain the diagonal blocks of the system. The LU and UL factorizations of these blocks are performed in-place and in parallel by the `hmap` operators used in lines 3-4. The off-diagonal blocks, `B` and `C`, that will contain the spike tips are stored in the HTA `BC`. Each tile of this HTA contains space for both the "left" ($W^{(t)}$) and "right" ($V^{(b)}$) spikes associated with each block. The spike tips are computed in line 7 using the LU and UL factorizations computed previously. The whole right-hand side (RHS) stored in `g` for the system is then updated in line 10 using the LU factorization of the diagonal blocks.

The reduced system, shown in Figure 3, can be formed now that the spikes and updated RHS have been computed. Lines 13-16 make use of HTA array assignments to construct the reduced system by copying the spike tips into the appropriate sections of each block of the reduced system. The HTAs `REDUCED` and `BC` are indexed using () and [] operators and triplet notation. The first () is shorthand for selecting every tile of the HTA `REDUCED`. For the HTA `BC`, we select different ranges of tiles for each statement. The [] operator is used to index a range of elements inside of a tile. The RHS of the reduced system is formed similarly in lines 19-20. Note that the array assignments used to form the reduced system imply communication. Once the reduced system has been formed, it may be solved in parallel as its blocks are independent. This is accomplished by calls to the `hmap` operator on lines 23 and 25.

Having solved the reduced system, the RHS of the original system is updated in lines 28-33. This is accomplished by array assignments and another call to `hmap` that performs matrix-vector multiplications in parallel. Once the RHS has been updated with the values computed from the reduced system, the rest of the solution is obtained in line 36.

Our implementation of the `TU` scheme slightly deviates from the SpikePACK implantation of the algorithm in two ways. First, the first and last partitions need only compute LU or UL, respectively. The inner partitions must compute both LU and UL in order to compute the tips of the left and right spikes. The first and last partitions only have either a right or a left spike and do not need to compute both. However, we chose to have the first and last partitions compute a fake spike in order to avoid special cases when computing the spikes. We compute both LU and UL for all partitions where as the SpikePACK only computes the LU for the first and the UL for the last as needed by the algorithm. Secondly the SpikePACK implementation uses a nonuniform distribution with larger partitions for the first and last partitions to balance the load since they are only computing one factorization. Since we compute two factorizations for every partition, our implementation uses a uniform size distribution.

```
...
// factorize blocks of A
LUA.hmap(factorize_lua());
ULA.hmap(factorize_ula());

// calculate the spike tips W(t) and V(b) from Bs and Cs
BC.hmap(solve_bc(),LUA,ULA);

// update right hand side
g.hmap(solve_lua(),LUA);

// form the reduced system
REDUCED()[0:m-1,m:2*m-1] =
      BC(0:num_blocks -2)[0:m-1,0:m-1];
REDUCED()[m:2*m-1,0:m-1] =
      BC(1:num_blocks -1)[0:m-1,m:2*m-1];

// form the reduced system RHS
greduced()[0:m-1] = g(0:num_blocks -2)[blocksize-m:blocksize -1];
greduced()[m:2*m-1] = g(1:num_blocks -1)[0:m-1];

// factorize the reduced system
REDUCED.hmap(factorize());
// solve the reduced system
greduced.hmap(solve(),REDUCED);

// Update RHS with the values from the spikes as r = r - Bz - Cz
fv = r(0:num_blocks -2); fr_half = greduced()[0:m-1];
B.hmap(dgemv(),fv, fr_half);
r(0:num_blocks -2) = fv;
fw = r(1:num_blocks -1); fr_half = greduced()[m:2*m-1];
C.hmap(dgemv(),fw,  fr_half);
r(1:num_blocks -1) = fw;

// Solve the updated system
r.hmap(solve_lua(),LUA);
...
```

Fig. 6. HTA SPIKE TU

4.2 TA

The implementation of the TA variant is structurally similar to our implementation of TU. Figure 7 presents the core of our implementation of TA. The algorithm consists of array assignments and calls to the hmap operator. The main difference from TU is that each processor now computes either the LU or the UL factorization for a block but not both. The TU variant partitions the matrix into one block per processor, and some processors must compute two spikes. TA has each processor compute only one spike. Consequently TA partitions the matrix into fewer blocks for a given number of processors than TU as shown in Figure 2. Whereas TU stored the diagonal blocks in the HTAs LUA and ULA, TA stores the appropriate blocks in the HTA DIAGS. Note that DIAGS can contain two copies of the same block of A since the same block is needed to compute two different spikes for the inner blocks. An additional HTA, DIAG_MAP, is used to set flags that indicate whether each tile needs to perform the LU or the UL factorization for its block. This can be seen in line 3 for the factorization and line 7 for the computation of the spike tips. The HTA TOSOLVERHS is used to refer to part of DIAGS as that HTA can contain multiple factorizations for each block. TOSOLVERHS, seen on line 4, contains only one factorization for each block of the matrix and is used to update the right hand side on lines 9 and 35. This is also matched with a map that indicates the type of factorization contained in the tile. Forming and solving the reduced system proceeds almost identically to the implementation of TU. Note that there is less parallelism available in this phase of TA than in TU due to partitioning the system into fewer blocks.

```
 1 ...
 2 // factorize the A blocks
 3 DIAGS.hmap(factorize_diag(),DIAG_MAP);
 4 TOSOLVERHS = DIAGS(0:num_blocks-1);

 6 // compute the spike tips from Bs and Cs
 7 BC.hmap(solve_bc(),DIAG_MAP,DIAGS);
 8 // generate modified right hand side
 9 g.hmap(solve_rhs(),TOSOLVERHS_MAP,TOSOLVERHS);

11 // form the reduced system
12 REDUCED()[0:m-1,m:2*m-1] =
13       BC(0:num_blocks-2)[0:m-1,0:m-1];
14 REDUCED()[m:2*m-1,0:m-1] =
15       BC(num_blocks-1:2*num_blocks-3)[0:m-1,0:m-1];

17 // form the reduced system right hand side
18 greduced()[0:m-1] = g(0:num_blocks-2)[blocksize-m:blocksize-1];
19 greduced()[m:2*m-1] = g(1:num_blocks-1)[0:m-1];

21 // factorize the reduced system
22 REDUCED.hmap(factorize());
23 // solve the reduced system
24 greduced.hmap(solve(),REDUCED);

26 // Update RHS with the values from the spikes as r = r - Bz - Cz
27 fv = r(0:num_blocks-2); fr_half = greduced()[0:m-1];
28 B.hmap(dgemv(),fv,fr_half);
29 r(0:num_blocks-2) = fv;
30 fw = r(1:num_blocks-1); fr_half = greduced()[m:2*m-1];
31 C.hmap(dgemv(),fw,fr_half);
32 r(1:num_blocks-1) = fw;

34 // Solve the updated system using the LU and UL as needed
35 r.hmap(solve_rhs(),TOSOLVERHS_MAP,TOSOLVERHS);
36 ...
```

Fig. 7. HTA SPIKE TA

5 Experimental Results

In order to evaluate the performance of our HTA implementations of the two spike variants, we conducted several experiments. We compare the performance of our implementations to both the SPIKE implementations in the Intel®Adaptive Spike-Based Solver version 1.0 and the sequential banded solvers found in the Intel®Math Kernel Library version 10.2 Update 5. The numbers reported are speedups over the sequential MKL routines, DGBTRF and DGBTRS. All code was compiled with the Intel®compilers icc and ifort version 11.1 Update 6, and all MPI programs were run using mpich2. The shared-memory HTA library runs on TBB version 2.2 Update 3.

In all cases several different systems of equations were tested and the results were similar. We present one for each algorithm. Tests were run on a four socket 32-core system using Intel®Xeon®L7555 processors running at 1.86 GHz. The system has 64 gigabytes of memory installed and on a cluster at University of Massachusetts with 8 compute nodes each with two Intel®Xeon®X5550 processors running at 2.66 GHz connected with InfiniBand. In testing we experienced large variations in the execution time of all programs due to the use of a shared system. To control for this all tests were run 8 times and the minimum execution time is reported.

5.1 TU

We present the test for a matrix of order 1048576 with a bandwidth of 513 here. This size was chosen in order to partition the matrix into blocks of uniform size. Figures 8a and 8c plot the speedups over sequential MKL for TU running on HTAs for shared-memory run on the 32-core shared memory system, HTAs for distributed-memory, and the Intel SpikePACK run on both the shared memory system and the cluster.

We believe that our performance advantage comes from implementation differences. SpikePACK uses larger blocks for the first and last partitions to attempt to minimize any load imbalance when computing factorizations and the spikes. However, this creates imbalance when retrieving the solution to the whole system after the reduced system has been solved since the retrieval for the outer blocks will require more time than the retrieval for inner blocks. As the number of processors increases, the retrieval becomes a larger portion of the total execution, and this imbalance is magnified.

It is also important to note that the performance of the HTA codes on shared-memory is almost identical with both the mpi and tbb backend. While at first this result surprised us, it is indeed what we should expect. The amount of computation is large, so the overheads of each runtime system are minimal. The ideal tiling structure may differ from one platform to the next, but a given tiling ought to perform similarly on the same system regardless of the backend.

5.2 TA

We present the test for a matrix of order 1093950 with a bandwidth of 513 here. This size was again chosen to partition the matrix into blocks of uniform size. Recall that the TA scheme partitions the matrix into fewer blocks than the TU scheme for a given number of processors. TU assigns one block of the matrix per processor while TA assigns one spike calculation per processor. The results of these tests are presented in Figures 8b and 8d which again shows speedup over sequential MKL for the three implementations. Each version tends to outperform TU and scales reasonably with increasing processors. However, SpikePACK begins to outperform the HTA implementations after 16 processors.

The performance difference seen in this case is due to the differences in the communication patterns between the HTA versions and the SpikePACK version. In the SpikePACK version of the algorithm, care is taken so that only one of the tips needs to be communicated to build the reduced system. This produces an irregular distribution of data. In cases where the number of partitions is small, distribution does not have a large impact but as the number of partitions grow the impact becomes more significant.

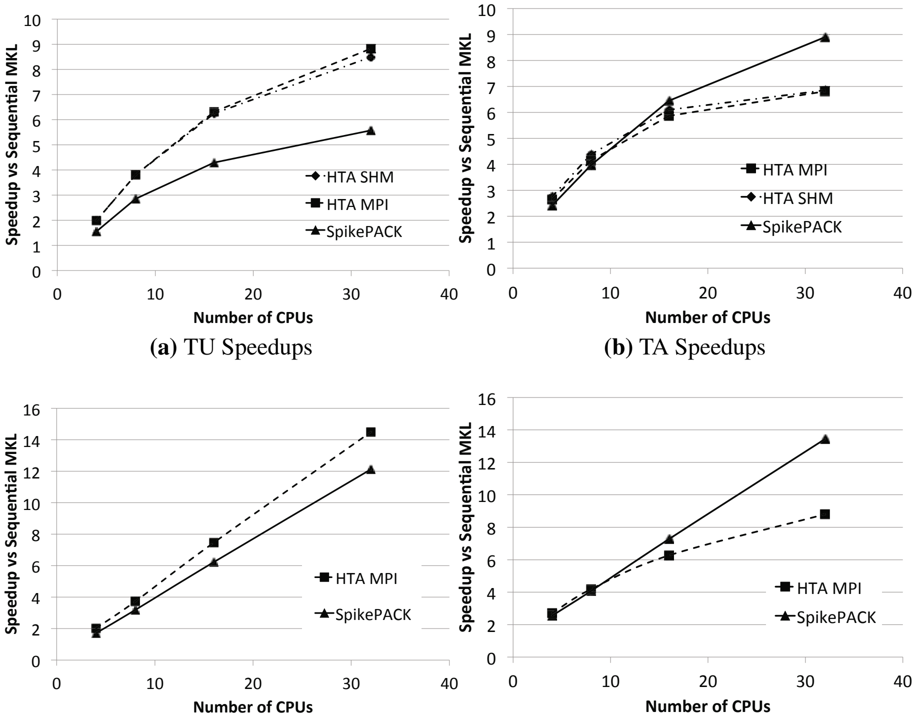
We believe that this behavior could implemented in the HTA versions of `TA` in two ways. First, the version of the library built on top of MPI provides support for user-defined distributions. These distributions could map the tiles of the spikes, RHS, and reduced system in such a way that minimizes communication between processors. The HTA library for shared-memory currently has no analog. This limitation is inherent in many libraries for shared-memory programming as they do not expose mechanisms to bind a thread to a particular core. The second way through which we could mimic SpikePACK's performance is through changing our implementation of the algorithm. By storing the blocks of the reduced system in a different order, we could more closely align the respective tiles of the spikes and RHS with the appropriate tiles of the reduced system. However, this complicates the implementation as the programmer becomes responsible for maintaining the mapping of the blocks of the reduced system to their locations in the HTA's tiling structure. We chose to initially focus on implementing a simple, elegant solution that closely maps to the algorithm.

6 Related Work

Implementing the SPIKE algorithms on top of the Hierarchically Tiled Array exploits both the portability and explicit tiling of the HTA programming model. Tiling has been extensively studied to improve performance of scientific and engineering codes [2,11,13,17] for parallel execution [16] and as a mechanism to improve locality [17]. However, most programming languages do not provide any support for tiles. In languages such as C or Fortran, either the programmer needs to write the code to support tiled computations or the programmer must rely on the compiler to generate them.

Languages such as HPF [10,12] or UPC [5] include support to specify how an array should be tiled and distributed among the processors, but the resulting tiles are only accessed directly by the compiler, and the programmer must use complex subscript expressions. Others like Co-Array Fortran [14] allow the programmer to refer to tiles and portions of them, but their co-arrays are subject to many limitations. Thus, the main difference of these languages with HTAs is that HTA Tiles are first class objects that are explicitly referenced, providing programmers with a mechanism for controlling locality, granularity, load balance, data distribution, as well as communication.

Sequoia [8] makes uses hierarchies of tasks to control locality and parallelism. Data is partitioned to create the parameters for the next level of tasks. In Sequoia, tasks communicate by passing parameters to children tasks and by accepting return values from them. HTA on the other hand, is data centric so that tiling is associated with each object and parallel computation follows the tiling. This, combined with the array notation of HTAs, simplifies the notation when programming algorithms that use tiled objects. Furthermore, the HTA semantics does not require insulation of the operation on tiles and therefore subsumes that of Sequoia.



(a) TU Speedups

(b) TA Speedups

(c) Cluster TU Speedups

(d) Cluster TA Speedups

Fig. 8. Speedups over Sequential MKL

Many Partitioned Global Address Space, or PGAS, languages aim to provide support for writing a single program that can run on many different platforms. Examples of these languages include X10 [7], UPC [5], Chapel [6], and Titanium [18]. These languages exploit locality by using distribution constructs or directives as hints to the compiler on how to partition or map the "global" array to the different threads. However, programmers cannot directly access these tiles and can only use flat element indexes to access the data (which is similar to our flattened notation). The explicit tiles of HTA programs increase programmability because they represent better the abstraction that the programmer has of how data are distributed. Programming using flat indexes forces the programmer to recover the implicit tiling structure of the data when data communication is required.

7 Conclusions

In this paper we have shown through the implementation of two variants from the SPIKE family of algorithms that the Hierarchically Tiled Array data type facilitates portable parallel programming and increases productivity. Tiles facilitate the mapping of block algorithms to code and result in programs that can run without modifications on both shared-memory and distributed-memory models.

Our experimental results show that the performance of the same HTA code when running on both shared-memory and distributed-memory models achieve almost identical performance, and are competitive to the reference Intel library implemented on top of MPI. In addition, our codes show that the features provided by the HTA result in programs that are both clean and compact and closely resemble the algorithm description of the problem.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Awards CCF 0702260 and by the Universal Parallel Computing Research Center at the University of Illinois at Urbana-Champaign, sponsored by Intel Corporation and Microsoft Corporation.

References

1. Intel adaptive spike-based solver, http://software.intel.com/en-us/articles/intel-adaptive-spike-based-solver/
2. Abu-Sufah, W., Kuck, D.J., Lawrie, D.H.: On the Performance Enhancement of Paging Systems Through Program Analysis and Transformations. IEEE Trans. Comput. 30(5), 341–356 (1981)
3. Andrade, D., Fraguela, B.B., Brodman, J., Padua, D.: Task-parallel versus data-parallel library-based programming in multicore systems. In: Euromicro Conference on Parallel, Distributed, and Network-Based Processing, pp. 101–110 (2009)
4. Bikshandi, G., Guo, J., Hoeflinger, D., Almasi, G., Fraguela, B.B., Garzarán, M.J., Padua, D., von Praun, C.: Programming for Parallelism and Locality with Hierarchically Tiled Arrays. In: Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, pp. 48–57 (2006)
5. Carlson, W., Draper, J., Culler, D., Yelick, K., Brooks, E., Warren, K.: Introduction to UPC and Language Specification. Tech. Rep. CCS-TR-99-157, IDA Center for Computing Sciences (1999)
6. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the chapel language. Int. J. High Perform. Comput. Appl. 21(3), 291–312 (2007)
7. Charles, P., Donawa, C., Ebcioglu, K., Grothoff, C., Kielstra, A., von Praun, C., Saraswat, V., Sarkar, V.: X10: An Object-oriented Approach to Non-uniform Cluster Computing. In: Procs. of the Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) – Onward! Track (October 2005)
8. Fatahalian, K., Horn, D.R., Knight, T.J., Leem, L., Houston, M., Park, J.Y., Erez, M., Ren, M., Aiken, A., Dally, W.J., Hanrahan, P.: Sequoia: programming the memory hierarchy. In: Supercomputing 2006: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, p. 83 (2006)
9. Guo, J., Bikshandi, G., Fraguela, B.B., Garzarán, M.J., Padua, D.: Programming with Tiles. In: Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, pp. 111–122 (February 2008)

10. High Performance Fortran Forum: High Performance Fortran specification version 2.0 (January 1997)
11. Irigoin, F., Triolet, R.: Supernode Partitioning. In: POPL 1988: Proc. of the 15th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, pp. 319–329 (1988)
12. Koelbel, C., Mehrotra, P.: An Overview of High Performance Fortran. SIGPLAN Fortran Forum 11(4), 9–16 (1992)
13. McKellar, A.C., Coffman Jr., E.G.: Organizing Matrices and Matrix Operations for Paged Memory Systems. Communications of the ACM 12(3), 153–165 (1969)
14. Numrich, R.W., Reid, J.: Co-array Fortran for Parallel Programming. SIGPLAN Fortran Forum 17(2), 1–31 (1998)
15. Polizzi, E., Sameh, A.H.: A parallel hybrid banded system solver: the spike algorithm. Parallel Computing 32(2), 177–194 (2006)
16. Ramanujam, J., Sadayappan, P.: Tiling Multidimensional Iteration Spaces for Nonshared Memory Machines. In: Supercomputing 1991: Proceedings of the 1991 ACM/IEEE conference on Supercomputing, pp. 111–120 (1991)
17. Wolf, M.E., Lam, M.S.: A Data Locality Optimizing Algorithm. In: Proc. of the Conf. on Programming Language Design and Implementation, pp. 30–44 (1991)
18. Yelick, K.A., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P.N., Graham, S.L., Gay, D., Colella, P., Aiken, A.: Titanium: A High-Performance Java Dialect. In: Workshop on Java for High-Performance Network Computing (February 1998)

Tackling Cache-Line Stealing Effects Using Run-Time Adaptation

Stéphane Zuckerman and William Jalby

University of Versailles Saint-Quentin-en-Yvelines, France
{stephane.zuckerman,william.jalby}@prism.uvsq.fr

Abstract. Modern multicore processors are now found in mainstream systems, as well as supercomputers. They usually embed prefetching facilities to hide memory stalls. While very useful in general, there are some cases where such mechanisms can actually hamper performance, as is the case with cache-line stealing. This paper characterizes and quantifies cache-line stealing, and shows it can induce huge slowdowns – down to almost 65%. Several solutions are examined, ranging from deactivation of hardware prefetching to array reshaping. Such solutions bring between 10% and 65% speedups in the best cases. In order to apply these transformations where they are relevant, we use run-time measurements and adaptive methods to generate code wrappers to be used only when prefetching hurts performance.

1 Introduction

Multicore processors are from now on the norm for almost any kind of computer-based system. They are used in high-performance computing, as well as domestic usage or even embedded systems. The increasing gap between how fast a CPU is and how fast data can be accessed from memory becomes even more problematic in this context: the old hardware and software techniques used to hide it must prove themselves useful in the wake of some kind of "multicore/manycore revolution". Indeed while the well-known gap between a unicore processor and memory is more or less solved thanks to well-known latency-hiding techniques (either in hardware or in software), the multiplication of the number of cores per chip tend to lengthen a given core's memory latencies, while reducing its effective bandwidth.

From the hardware side, the use of memory caches does a lot to make the memory wall [27] and the gap it causes become smaller and narrower. It effectively hides the latency caused by memory accesses. However, when multiprocessor or multicore systems are involved, it also means ensuring data coherence between the various (separate) caches. Hence the apparition of cache coherence protocols such as MSI, MESI, MOESI, MESIF, etc. [15]. With the advent of multiprocessor (and now multicore) systems a new kind of problem occurred: false-sharing, i.e. the fact that two processors (or cores) write to different values which are contained in the same cache-line. The negative impact of false-sharing on performance for multiprocessor and multicore systems has been extensively studied [9],

K. Cooper, J. Mellor-Crummey, and V. Sarkar (Eds.): LCPC 2010, LNCS 6548, pp. 62–76, 2011.

and multiple techniques have been devised to detect (via memory tracing for example), or even avoid false-sharing altogether (through data structure reshaping, loop transformations, etc.).

Another well-known technique which helped reduce the gap between memory and CPU is data prefetching. In a single-core context, data prefetching (whether performed automatically in hardware, or inserted by the compiler or the user in software) has shown to be a formidable ally to hide latencies, even though it induces a higher bandwidth usage in the case of hardware prefetching [8] or instruction overhead for software prefetching. By prefetching data into caches ahead of time, memory can be accessed much faster, thanks to well placed prefetch orders. However once again, the advent of multiprocessor and multicore systems tends to introduce difficulties. Among them is *Cache-Line Stealing*(CLS), which we present in Section 2.

To our great surprise, we could not find any paper detailing how too aggressive a data prefetching policy could indeed slow down an application by “stealing” another thread’s cache-line. The closest to our definition can be found in a paper by Hyde et al [9]. CLS is not so frequent that deactivating prefetch mechanisms altogether to solve cache-line stealing is considered an option, as hardware or software prefetching have many legitimate uses in the life of a running program. Hence it is necessary to remove prefetching only in those portions of code where it is known that no prefetching will make performance go up.

This paper’s contributions are twofold:

- It characterizes and quantifies cache-line stealing.
- It proposes to solve CLS through adaptive compilation methods, deactivating prefetching only when CLS actually hurts performance.

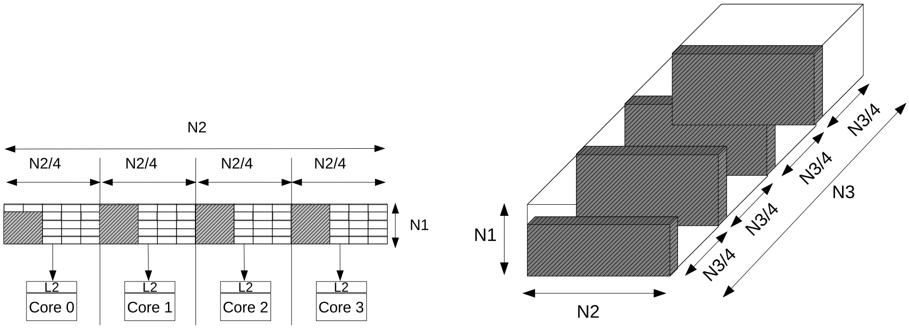
The remainder of this paper is as follows: Section 2 exposes an example that motivates this research and describes what cache-line stealing is, and it occurs, as well as experimental data; Section 3 presents several leads to solve cache-line stealing problems; Section 4 describes the use of adaptive methods to generate prefetch-less kernels when needed; Section 5 presents work related to this article; finally Section 6 presents our conclusions.

2 Motivation

2.1 What Cache-Line Stealing Is, and When It Occurs

Cache-line stealing (CLS) happens when a given core asks for a cache-line in advance, retrieves it and copies its data into its cache hierarchy while it does not need the data contained in this cache-line. At the same time, another core does need this cache-line and has already retrieved it or is about to. By prefetching an unnecessary cache-line, additional memory and coherency message traffic are incurred.

CLS happens only when certain conditions are met. First, accessing data in the “right” storage way usually does not provoke a significant cache-line stealing



(a) 2D array: "overprefetched" surface (b) 3D array: "overprefetched" volume

Fig. 1. Cache-line stealing: general idea. CLS on a 2D array (left) and a 3D one (right). Dark grayed areas are the ones which are prefetched by an adjacent core but are not needed by it. As the array is supposed to be one whole contiguous block, there is a wrap around where the first block of data (on the far left) is partially prefetched by a sibling thread, just like any other block.

event. CLS mostly happens when a multidimensional array is accessed in the "wrong" dimension. For example, accessing a 2D array in C through its columns could trigger CLS in a multithreaded environment. Hence partitioning the input data set along the columns of a 2D array in C for each thread to process could lead to additional memory traffic.

Figure 1 shows an example of what would occur in a 2D array as well as in a 3D one. Each thread prefetches data belonging to one of its siblings, copying said data into its cache hierarchy. This then triggers cache coherence messages, where cache-lines will be falsely tagged as shared, invalid, etc.. How negative an impact that kind of extra-traffic (in terms of data movement and coherence messages) will have on performance depends on the prefetching policy: if the hardware prefetcher systematically goes to prefetch the same stream, then this phenomenon will be amplified – and the more cores are used in the program, the more likely extra-traffic will occur.

Knowing how many extra cache-lines ($ExtraCL$) will be loaded in addition to everything else in an n-dimensional array sums up to solving

$$ExtraCL = NbOfStreams * \prod_{i=2}^{n} N_i * dist_{prefetch}(N_1 - 1)$$

n is the number of dimensions of the array, and N_i are the different dimensions of said array. N_n is the "innermost" dimension, i.e. if the array was traversed in the correct way (along the cache-lines), this dimension would be traversed in the innermost loop of the loop nest. $dist_{prefetch}$ is the prefetch distance. $NbOfStreams$ is the number of different memory streams which are concurrently being accessed. In a hardware-based prefetching mechanism, a limited amount of streams can be accessed (typically, from 2 to 8).

Computing the amount of data stolen can prove difficult: in a hardware-based prefetch mechanism, the prefetch distance is not necessarily given by the vendor. In a software context, it is much easier to do, provided the arrays are accessed directly.

2.2 Experimental Setup

Our experimental setup consists of a Xeon (2.0 Ghz) Core 2 system with 4 cores in total (2 × 2 cores), composed of two dual-core processors sharing 2 GB of main memory. The L1D caches(32 KB) are private to every core while the L2 caches (unified, holding both instructions and data) are shared by groups of 2 cores (called *adjacent cores*), their size being 4 MB. Core 2 cores are out-of-order, superscalar processors, embedding SIMD instructions (SSE). The two levels of caches embed hardware prefetchers capable of analyzing memory address streams and triggering prefetches called DPL. They also embed a prefetching mechanism called "adjacent cache-line fetch", which systematically fetches a cache-line situated right after the one being fetched [8].

Intel's C compiler (`icc` v11.1) is the reference compiler used in our experiments. Our main way of measurement was the `RDTSC` instruction, which gives an exact count of elapsed cycles. We also used Intel's Performance Tuning Utility (PTU) to measure cycle and instruction counts. Finally, `icc`'s OpenMP features were used to make the parallelizations used in the experiments.

With respect to experimental methodology, each kernel was run once (to put data into the cache hierarchy), then 2000 times and averaged (with respect to the cycles count), and each run is repeated 100 times. Among these runs, the lowest averaged result (in cycles) is then selected – i.e. the best performance is kept. This ensures that noise (due to the operating system for example) is kept minimal during the experiments. Moreover, the repetition loop has an interesting side effect on our study: if the arrays are small enough to fit in a given cache level (for example L2), the first iteration will initially fetch the arrays into the L2 cache, but the following iterations will already find them in L2. All the arrays are aligned on a page boundary basis. The fact that there are NUMA accesses is of no concern here, as data fit into caches, and there is sufficient repetitions performed.

Finally, the dimensions of the arrays have been chosen so that there is no inherent false-sharing when dividing work among threads.

2.3 Experimental Analysis of Cache-Line Stealing

To show evidence of cache-line stealing, two very simple kernels were used. As the ratio between loads and stores has an impact on performance, we started with a simple "store only" kernel (`memset2D`), then a "load-store" one (`memcpy2D`). Finally, we used a "one store, many loads" kernel, where the amount of loads varies to observe its impact on CLS. For the latter case, stencil operator kernels were used.

Table 1. Impact of hardware prefetching on performance for `memset2D` (left table) and `memcpy2D` (right table), with prefetching either turned ON or OFF. In each table columns labeled ON (resp. OFF) display the average performance in cycles per store for `memset2D` and per load-store for `memcpy2D`: lower is better. The last column labeled "Speedup" shows the impact of turning off prefetch: a speedup value greater than 1 shows that *turning off prefetch is beneficial.* i refers to the number of threads at runtime. For each kernel, the same amount of work was given to each thread to perform. In the case of `memset2D` and `memcpy2D`, a $k \times N$ subarray (or subarrays) was fed to each thread, $8 \leq k \leq 30$ and $800 \leq N \leq 3200$ per thread.

(a) Impact of hardware prefetching for `memset2D` for a $k \times 800i$ array

Number of threads	ON	OFF	Speedup: ON Vs OFF
1	1.27	1.14	1.12
2 (scatter)	2.09	1.37	1.52
2 (compact)	2.16	1.65	1.31
4	3.00	1.84	1.63

(b) Impact of hardware prefetching for `memcpy2D` for $k \times 800i$ arrays

Number of threads	ON	OFF	Speedup: ON Vs OFF
1	1.89	1.86	1.02
2 (scatter)	2.69	2.11	1.28
2 (compact)	2.63	2.43	1.08
4	3.42	2.63	1.30

`memset2D` and `memcpy2D` `memset2D` is the simplest kernel presented in this paper. It sweeps a 2D array of double precision cells, and sets each of them to a given value. While it may look like some artificial-made code, similar code does exist in current commercial libraries. For example, Intel's Math Kernel Library (MKL) decomposes $C = A \times B$ in two steps: it first performs $C = 0$, then $C = 1 \times C + 1 \times A \times B$. This is also an interesting kernel, as it only performs stores to memory. Finally, such a "double precision 2D memset" is used in the MKL's DGEMM.

As Table 1(a) illustrates, turning the hardware prefetchers off always improves performance (up to 63%), no matter which configuration was chosen to pin down the threads.

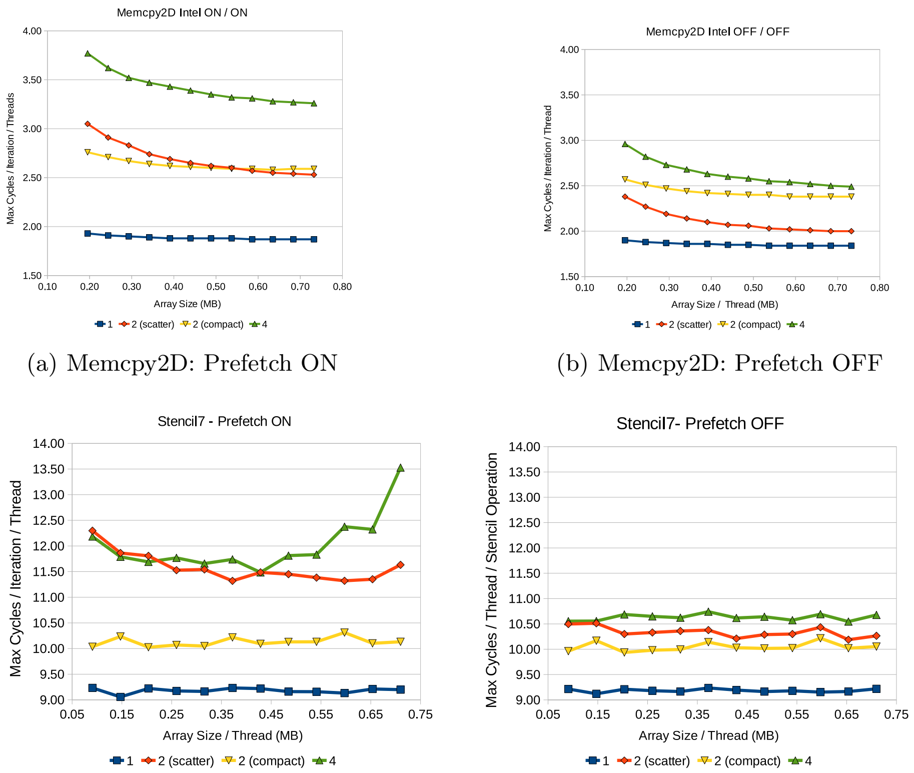
Because the array is partitioned along the second dimension (and not the first, see above), HW prefetchers load the next cacheline very aggressively *inside* an active thread's sibling dataset, effectively *stealing* useless cachelines for the sibling thread.

As for the `memcpy2D` kernel, a naive version of a 2D array copy is used [1].

As before, Table 1(b) illustrates the performance gains which are always brought by turning prefetching off, whatever the threads configuration (up to 30% on four cores).

7-Point Stencil Operator. The 7-point stencil operator yields a certain amount of 3D arrays to load from, and a single array to store to. The geometry of the arrays was carefully chosen: the last dimension is far bigger than the first two, thus making it somewhat "logical" to divide work between threads along the third dimension. Indeed, this stencil operator accesses data in all directions of the arrays, and "cutting" them vertically (as shown in figure 1(b)) reduces the surface where data are to be shared between threads.

[1] It is worth mentioning that `icc` recognizes that the innermost loop of the naive version is an array copy and thus calls its own version of `memcpy` on a per-line basis.



(a) Memcpy2D: Prefetch ON

(b) Memcpy2D: Prefetch OFF

(c) Stencil7: Prefetch ON

(d) Stencil7: Prefetch OFF

Fig. 2. The Impact of Cache-Line Stealing. On figure 2(a) (top left), where prefetching is ON, the amount of cycles per load-store is situated between 3.3 and 3.6 cycles on 4 cores, while on figure 2(b) (top right) where prefetching is OFF, the number of cycles continuously decreases to reach 2.5 cycles per load-store on 4 cores. On figure 2(c) (bottom left) prefetching is turned OFF, while on figure 2(d) prefetching is turned ON. "Scatter" and "compact" are two affinity policies defined in Intel's OpenMP runtime (see Section 2.2 for more details). Each thread processes the same amount of work. **Lower is better**.

Performance is very sensitive to the geometry of the data and the load-store ratio. For the 7-point stencil case, there is almost no advantage to turning off prefetching (except on 4-core) as there is a ratio of 14 loads for one store.

More "convenient" geometries were also used. 3D arrays would be shaped as $ik \times 10 \times 160$, with $3 \leq k \leq 32$, and i being the number of threads. Hence the size ratio between dimensions would become much less important. Data are distributed according to the storage policy, effectively making a thread access "its" subarray in a stride 1 manner. There is no CLS, but almost no advantage for using hardware prefetching either: speedups are flat (hence not shown here). Using such a geometry also shows that, contrary to the previous cases,

(c) Impact of hardware prefetching on the performance of `stencil7`. Shown are the average performance (in cycles per stencil computation step) and speedups. The baseline for speedups is the configuration where prefetching is turned ON. The dimensions of the 3D arrays are $k \times 10 \times 640i$, where $3 \leq k \leq 32$, and i is the number of threads running. 32-bits values are used in this example.

Threads #	ON	OFF	Speedup: OFF
1	9.22	9.23	1.00
2 (scatter)	11.60	10.38	1.12
2 (compact)	10.18	10.09	1.01
4	12.55	10.66	1.18

(d) Impact of hardware prefetching on the performance of `stencil7` in a favorable case. Shown are the average performance (in cycles per stencil computation step) and speedups. The baseline for speedups is the configuration where prefetching is turned ON.

Threads #	ON	OFF	Speedup: OFF
1	9.22	9.21	1.00
2 (scatter)	9.80	9.76	1.00
2 (compact)	9.78	9.75	1.00
4	10.14	10.23	0.99

whatever number of thread is used yields more or less the same performance on a per-thread basis. Thus contention is not the reason why in the previous cases performance decreases as the number of threads increases.

Table 1(d) sums up results for that geometry. Almost no difference appears, whatever configuration is chosen.

3 Possible Methods to Counter Cache-Line Stealing

CLS occurs mainly because data partitioning between threads was far from optimal. This is not necessarily trivial to avoid for the programmer, as such an ill-partitioned data set could happen when using external libraries on which the user has no control. In other cases there are inherent constrains to a given computation: for example when multiplying two matrices together, such as $C_{n,n} = A_{n,k} \times B_{k,n}$, where $k << n$ but both A and B still take a sizeable amount of space in memory, thus needing to block along not only on A, but also on B. It is first necessary to detect CLS. When it is clearly too important to ignore, then several countermeasures can be used against it.

Detecting Cache-Line Stealing. False-sharing can be detected using memory traces [9]. CLS, although different, might be tracked down the same way, tracking only stores.

Turning Off Prefetching. CLS can occur either in hardware or in software. Turning off prefetch is easy in a software context[2].

[2] For example, the `-no-prefetch` compiler option in Intel C Compiler (icc) enables the programmer to choose when not to activate software prefetching, which comes in handy for processors which support software prefetching only, such as the Itanium 2.

When talking about hardware prefetching, turning it off becomes more difficult, as it either requires physical access to the machine (which will need to be rebooted to access its BIOS or setup program), or because it requires superuser privileges to trigger it in software from the operating system. Used like this, prefetching can be turned off only when needed (i.e. potential cache-line stealing has been detected). The potential overhead of a per-process hardware prefetch trigger is not necessarily as high as the performance loss due to unnecessary data prefetching.

Loop Transformations. If an array is accessed along the wrong dimension (i.e. which leads to extensive CLS), it might simply be simpler to perform classical loop transformations on the loop nest which accesses it. For example, given a 2D array which was partitioned column-wise while storage policy is row-wise, loop interchange can totally void CLS, as is shown in figure 3.

On the x86 ISA, accessing the arrays column-wise hampers vector instruction generation, as there is no packed vector load (or store) memory instruction for non-unit strides. This effect could be worked around thanks to register blocking (i.e. accessing an array column-wise, but two cells by two cells). In this case the prefetcher will not be unnecessarily triggered.

Data Structure Reshaping. Finally, techniques similar to those used against false-sharing can be used (to a certain extent) against CLS. Array padding, while useful, will not prevent CLS as extra cache-lines are prefetched nonetheless. However array transposition, when possible, can help avoid CLS altogether, as it "reverses" the way cache-lines are packed together. For more complex data structures, techniques applied to avoid false-sharing can prove effective too.

This is more or less what was done in section 2.3, when arrays were reshaped to have a better way to parallelize the computation around them.

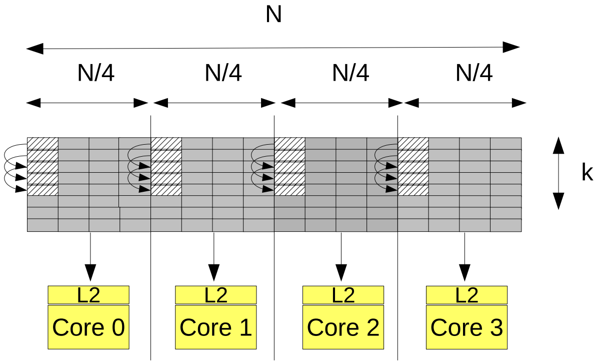


Fig. 3. A 2D array (stored row-wise) accessed column-wise by each thread. As data was distributed to threads by batches of columns rather than lines, accessing the array column-wise rather than row-wise does not prevent the IP-based prefetcher from fetching useful additional cache-lines (as there is a fixed stride), and avoids stealing adjacent cores' data.

4 Run-Time Adaptation to Solve Cache-Line Stealing Using a Hybrid Software/Hardware Framework

Cache-line stealing is not encountered very often, yet it can have a disastrous impact on performance, as section 2 shows. Finding out when CLS occurs is no easy task, as it both depends on the layout of the data structures, how data is partitioned between threads, and how many threads share arrays. Moreover, when using an external library, there is no certainty as to what a given library will do with the data types it takes as input or output. As the prefetch distance is seldom known in the case of hardware prefetchers (and as it can even be somewhat adaptive to the current workload of an application), the formula given in section 2 can only be applied with real success in the case of pure software prefetching. Other cases (i.e. hardware prefetching) must be dealt with differently.

We propose using adaptive methods such as the ones used by WEKA [6] to solve cache-line stealing, as very few different cases should occur in the kernels we are evaluating. Indeed, both `memset2D` and `memcpy2D`, as well as `stencil7` are "streaming" kernels, i.e. they are perfect loop nests with no control structure inside. Hence if performance is to vary, it is necessarily due to different input datasets. Moreover, adaptive methods can consider a given machine as some kind of "black box". Coupled with standard machine learning techniques, it is enough to spot problematic input datasets and try to generate a specific version of a kernel for similar cases.

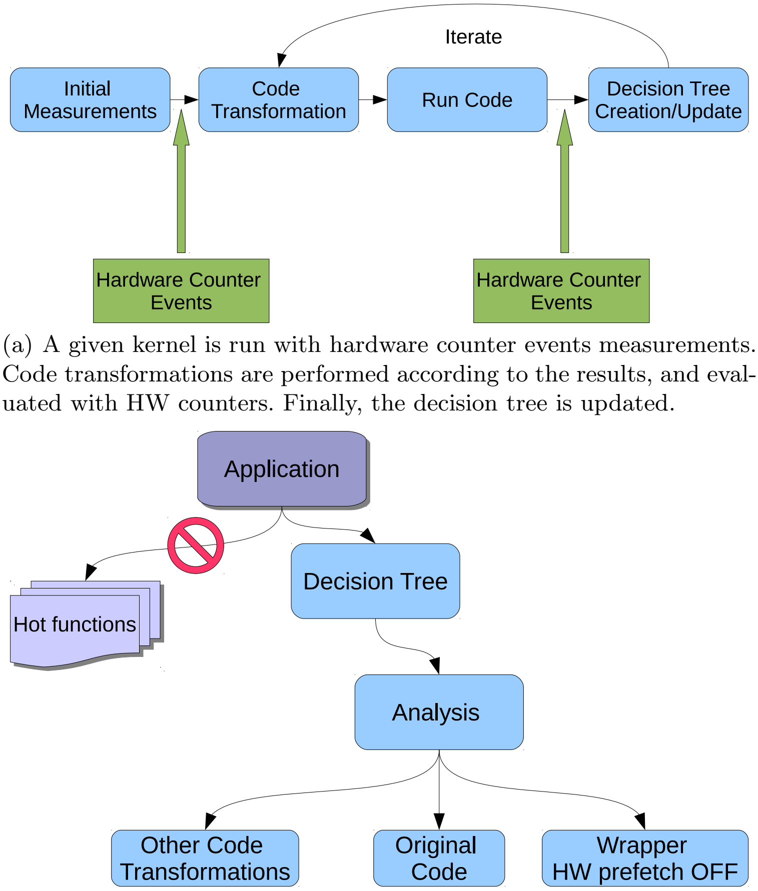
4.1 Description of the Adaptive Applications Framework

The framework used to counter CLS is explained in figure 4.

The hottest functions in a given application are identified during the initial measurements done on a given application. Then a driver is created to test such functions, to test its properties in a confined environment. In this specific case, only the shapes of the arrays given as input to the functions are of interest (be them 2D for `memset2D`/`memcpy2D` or 3D arrays). Each function is run to measure its IPC according to the various shapes the input arrays can have. IPCs are measured[3]. Usually, only a small number of shapes will incur a variation in the number of instructions per cycle. IPC is measured both for unicore and multicore executions, with various affinity policies. If an outstandingly bad IPC emerges from these tests, then a decision tree is generated or updated, which decides whether the original function or a wrapper (which turns off prefetching altogether for this specific input must) must be called. This wrapper function is also tested to make sure that its IPC is significantly better than the prefetch-enabled version. Figure 5 shows a C-like example code.

Once the good prefetch-less tuples {function, input dataset, partitioning scheme} are identified, a decision tree is generated or updated to switch on the right codelet to call at runtime. It must be noted that Figure 5 deals with

[3] Thanks to the `INSTRUCTION_RETIRED` and `CPU_CLK_UNHALTED` events.



(a) A given kernel is run with hardware counter events measurements. Code transformations are performed according to the results, and evaluated with HW counters. Finally, the decision tree is updated.

(b) Behavior at run-time. An application will first go through a decision tree which will quickly evaluate which kernel to use according to some evaluation of the input datasets.

Fig. 4. The adaptive compilation framework used to evaluate when to turn hardware prefetching off

```
return_type  wrapper_to_function_to_call(parameter1,  parameter2, ...)
{
    turn_off_prefetching();
    function_to_call(parameter1,parameter2, ...);
    turn_on_prefetching();
}
```

Fig. 5. Pseudo C-code for a wrapper to turn off prefetching for a given codelet

the case of hardware prefetchers. In the case of software prefetching (such as for the Itanium 2 processor), two possibilities exist:

1. All the source code is available. An exact copy of the function can thus be made, compiled with specific orders to turn prefetching off.
2. The function is located in a binary object. Then it must be directly modified, which is not currently possible. Additional features to manipulate binary executables must then be added to the framework.

On the Core 2 microarchitecture, there is no explicit user-space instruction to turn hardware prefetching on or off. Special files must be written to do so. Special rights must also be given to the user. In our different tests, trying to write too frequently into such a file[4] sometimes led to a system crash. Hence a more flexible mechanism to turn on or off hardware prefetching on different cores is a sorely missing feature in modern microprocessors.

Moreover, writing to such special files yields a severe overhead. There is a strong need for better hardware/software interactions. Such a hybrid approach with respect with data prefetching could actually perform better, as the resulting overhead would be much lower, as the few attempts at hybrid prefetching show [17,20,13].

4.2 Implementing the Adaptive Framework

Using IPC as an indicator, the framework was used on the `stencil7` kernel. Figure 6 illustrates the different shapes used for our experiments.

The decision tree thus built has two parameters: the shape of the 3D arrays, and the way data is partitioned. The latter can be inferred from the shape the arrays each thread has to process.

5 Related Work

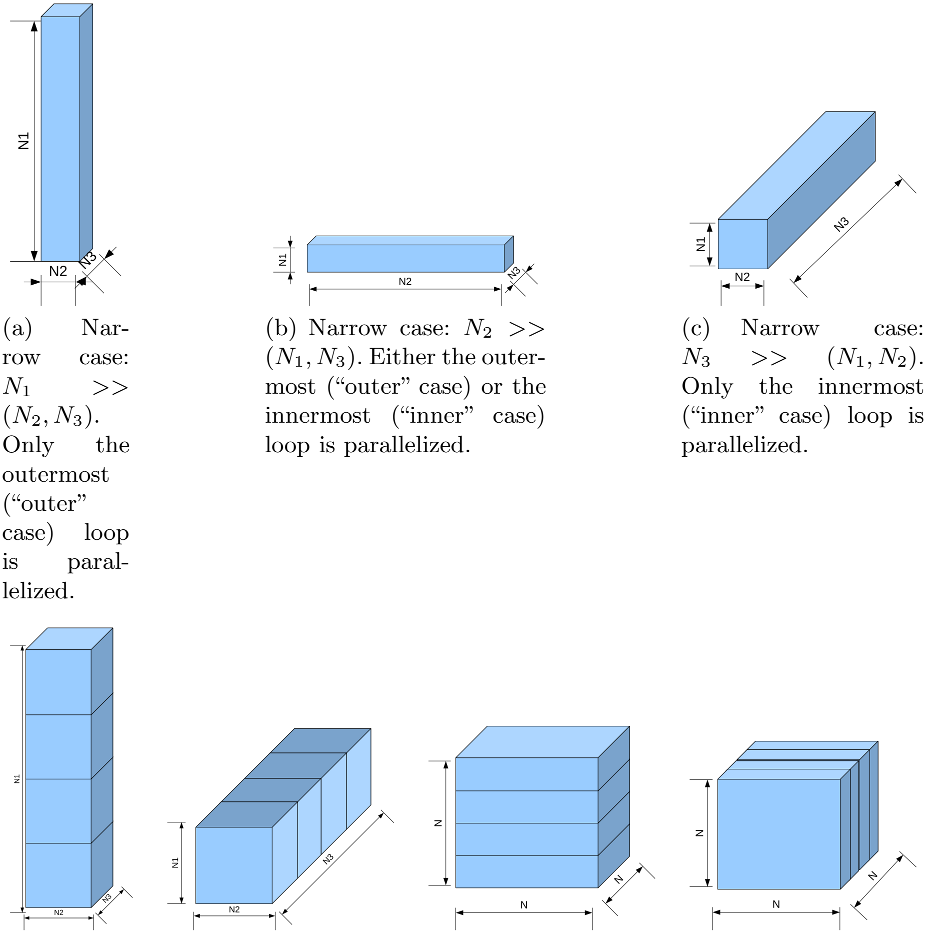
Hardware-based prefetching [5] eliminates the instruction overhead encountered in software-based prefetching [2,14] and may profit from runtime information, but tends to cause more unnecessary prefetches.

The problem of false-sharing has also been recognized for many years [4,21,10,3]. Data layout optimizations can help improve the memory performance of applications by controlling the way data is arranged in memory [16].

A benchmark enforcing false sharing with each write access to an array is introduced and analyzed for Intel Core Duo, Intel Xeon and AMD Opteron systems in [24]. The results show the impact of the cache architecture and of the coherency protocol on the performance.

Other benchmark results for various platforms on sparse matrix-vector product [26] show the impact of prefetching on such operations. Another example can be found in [25], not directly referring to cache-line stealing, but systematically benchmarking the impact of enabled/disabled hardware and software

[4] Under Linux, we are talking of the `/dev/msr/*` files.



(a) Narrow case: $N_1 >> (N_2, N_3)$. Only the outermost ("outer" case) loop is parallelized.

(b) Narrow case: $N_2 >> (N_1, N_3)$. Either the outermost ("outer" case) or the innermost ("inner" case) loop is parallelized.

(c) Narrow case: $N_3 >> (N_1, N_2)$. Only the innermost ("inner" case) loop is parallelized.

(d) Each thread processes a cube: "outer" case.

(e) Each thread processes a cube: "inner" case.

(f) The whole arrays are cubes: each thread processes a hyperplan ("outer" case).

(g) The whole arrays are cubes: each thread processes a hyperplan ("inner" case).

Fig. 6. Various 3D array shapes tested for their IPC in the case of the `stencil7` kernel

data prefetching on a medical imaging application running on Intel dual-core processors.

The notion of memory access intensity is introduced in [12] to facilitate quantitative analysis of a program's memory behavior on multicore systems with hardware prefetching and is demonstrated with three numerical solvers for large scale sparse linear systems.

Some research with different solutions have been proposed to solve the problem of prefetching in a multicore/multiprocessor context [11,22].

Finally, hybrid software/hardware data prefetching has been the interest of too few researchers. Wang et al [23] propose to improve on scheduled region prefetching (SRP) with guided region prefetching (GRP). Although the results are promising, the paper focuses on single-core execution only. Seung Woo Son et al [18] show how conventional data prefetching techniques usually don't scale on CMP systems. They propose compiler-driven prefetching techniques, which they validate on a simulator. To conclude, Gornish and Veidenbaum [7] propose a hybrid mechanism, which consists in using software prefetching before a given loop in order to "train" the hardware prefetchers, informing them of the stride to use, as well as the when to stop prefetching. This mechanism, if implemented, could really bring some leverage to the programmer and compiler writer to efficiently bring data back into the right cache, for the right core.

To our best knowledge, there has not been much published on investigating the impact of prefetching on "cache-line stealing" on contemporary multicore architectures, for either software- or hardware-based prefetching techniques. Close to that topic, the work of Song et al [19] tries to make an accurate model of direct-mapped caches to predict compulsory misses, capacity misses (both on private and shared data), and when cache hits become cache misses as well as the contrary.

6 Conclusion and Future Work

While very useful, data prefetching can severely hurt performance by triggering cacheline stealing. While the unicore case has been studied in depth, the multicore one features almost no study. Hardware prefetchers do not give the opportunity to the user to choose the prefetch distance according to some known pattern. Hence if cache-thrashing occurs, it is unpreventable. On the software-side, the shape of the arrays still conditions the efficiency of software prefetching, and this can only be corrected through iterative methods. Otherwise, cacheline stealing may occur.

Cache-line stealing induced by too aggressive data prefetching was studied in this paper, along with a way to compute the number of extra cache-lines brought into the cache-hierarchy of a given core for a given prefetch distance – which unfortunately can only be empirically guessed in the case of hardware prefetchers. Prefetching coupled with a certain way of partitioning data between threads provokes the apparition of cache-line stealing.

Some solutions were proposed to eliminate some or all of cache-line stealing, such as

- Turning off prefetching. A potential solution could come from microprocessor vendors to allow some kind of per-process way of activating or deactivating prefetching.
- Performing loop transformations on the incriminated code.
- Reshaping the data structures to better suit the prefetching policy.

Finally, an adaptive framework was proposed to counter cache-line stealing by deactivating hardware prefetching on Core 2 processors only when necessary.

Although this solution should provide a perfect hybrid way to solve CLS, the current way to turn HW prefetching on or of offsets the gains. We strongly advocate for a better hardware/software interaction in the case of prefetching, to reduce some of the negative impact hardware prefetching can have on performance in multicore.

Future work includes refining this adaptive framework to include other, more complex transformations. Among them, the ones described in section 3, which were performed by hand so far.

Acknowledgments

The research presented in this paper is supported by ITACA LAB (University of Versailles and CEA-DAM common lab) and the ITEA2 project "ParMA" [1] (June 2007 – May 2010).

References

1. ParMA: Parallel programming for multi-core architectures - ITEA2 project (06015), http://www.parma-itea2.org
2. Bacon, D.F., Graham, S.L., Sharp, O.J.: Compiler Transformations for High-Performance Computing. ACM Comput. Surv. 26(4), 345–420 (1994)
3. Bodin, F., Granston, E.D., Montaut, T.: Evaluating two loop transformations for reducing multiple writer false sharing. In: Pingali, K.K., Gelernter, D., Padua, D.A., Banerjee, U., Nicolau, A. (eds.) LCPC 1994. LNCS, vol. 892, pp. 421–439. Springer, Heidelberg (1995)
4. Bolosky, W.J., Scott, M.L.: False Sharing and its effect on shared memory performance. In: Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV), pp. 57–71 (1993)
5. Garzaran, M., Brit, J., Ibanez, P., Vinals, V.: Hardware Prefetching in Bus-Based Multiprocessors: Pattern Characterization and Cost-Effective Hardware. In: Ninth Euromicro Workshop on Parallel and Distributed Processing, pp. 345–354 (2001)
6. Holmes, G., Donkin, A., Witten, I.: WEKA: a machine learning workbench. In: Proceedings of the 1994 Second Australian and New Zealand Conference on Intelligent Information Systems, pp. 357–361 (1994)
7. Gornish, E.H., Veidenbaum, A.: An integrated hardware/software data prefetching scheme for shared-memory multiprocessors. Intl. Journal of Parallel Programming, 35–70 (1999)
8. Hedge, R.: Optimizing application performance on intel core microarchitecture using hardware-implemented prefetchers (2008), http://software.intel.com/en-us/articles/optimizing-application-performance-on-intel-coret-microarchitecture-using-hardware-implemented-prefetchers/
9. Hyde, R.L., Fleisch, B.D.: An analysis of degenerate sharing and false coherence. J. Parallel Distrib. Comput. 34(2), 183–195 (1996)
10. Jeremiassen, T.E., Eggers, S.J.: Reducing False Sharing on Shared Memory Multiprocessors through Compile Time Data Transformations. In: PPOPP, pp. 179–188 (1995)

11. Jerger, N., Hill, E., Lipasti, M.: Friendly fire: understanding the effects of multiprocessor prefetches. In: IEEE International Symmposium on Performance Analysis of Systems and Software, pp. 177–188 (2006)
12. Liu, L., Li, Z., Sameh, A.H.: Analyzing memory access intensity in parallel programs on multicore. In: ICS 2008, pp. 359–367. ACM, New York (2008)
13. Marathe, J., Mueller, F., de Supinski, B.R.: Analysis of cache-coherence bottlenecks with hybrid hardware/software techniques. ACM Trans. Archit. Code Optim. 3(4), 390–423 (2006)
14. Mowry, T.C.: Tolerating Latency in Multiprocessors Through Compiler-Inserted Prefetching. ACM Trans. Comput. Syst. 16(1), 55–92 (1998)
15. Papamarcos, M.S., Patel, J.H.: A low-overhead coherence solution for multiprocessors with private cache memories. In: ISCA 1984: Proceedings of the 11th Annual International Symposium on Computer Architecture, pp. 348–354. ACM, New York (1984)
16. Raman, E., Hundt, R., Mannarswamy, S.: Structure Layout Optimization for Multithreaded Programs. In: CGO, pp. 271–282. IEEE Computer Society, Los Alamitos (2007)
17. Skeppstedt, J., Dubois, M.: Hybrid compiler/hardware prefetching for multiprocessors using low-overhead cache miss traps. In: International Conference on Parallel Processing, p. 298 (1997)
18. Son, S.W., Kandemir, M., Karakoy, M., Chakrabarti, D.: A compiler-directed data prefetching scheme for chip multiprocessors. In: PPoPP 2009: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 209–218. ACM, New York (2009)
19. Song, F., Moore, S., Dongarra, J.: L2 cache modeling for scientific applications on chip multi-processors. In: ICPP 2007: Proceedings of the 2007 International Conference on Parallel Processing, Washington, DC, USA, p. 51. IEEE Computer Society, Los Alamitos (2007)
20. Struik, P., van der Wolf, P., Pimentel, A.D.: A combined hardware/software solution for stream prefetching in multimedia applications (1998)
21. Torrellas, J., Lam, M.S., Hennessy, J.L.: False sharing and spatial locality in multiprocessor caches. IEEE Transactions on Computers 43, 651–663 (1994)
22. Wallin, D., Hagersten, E.: Miss penalty reduction using bundled capacity prefetching in multiprocessors. In: International Parallel and Distributed Processing Symposium, p. 12a (2003)
23. Wang, Z., Burger, D., McKinley, K.S., Reinhardt, S.K., Weems, C.C.: Guided region prefetching: A cooperative hardware/software approach. In: Proceedings of the 30th International Symposium on Computer Architecture, pp. 388–398 (2003)
24. Weidendorfer, J., Ott, M., Klug, T., Trinitis, C.: Latencies of Conflicting Writes on Contemporary Multicore Architectures. In: Malyshkin, V.E. (ed.) PaCT 2007. LNCS, vol. 4671, pp. 318–327. Springer, Heidelberg (2007)
25. Whitepaper, I.: Optimizing Embedded System Performance - Impact of Data Prefetching on a Medical Imaging Application (2006), http://download.intel.com/technology/advanced_comm/315697.pdf
26. Williams, S., Oliker, L., Vuduc, R.W., Shalf, J., Yelick, K.A., Demmel, J.: Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In: SC 2007, p. 38 (2007)
27. Wulf, W.A., McKee, S.A.: Hitting the Memory Wall: Implications of the Obvious. Computer Architecture News 23, 20–24 (1995)

Locality Optimization of Stencil Applications Using Data Dependency Graphs

Daniel Orozco, Elkin Garcia, and Guang Gao

University of Delaware
Electrical and Computer Engineering Department
{orozco,egarcia,ggao}@capsl.udel.edu

Abstract. This paper proposes tiling techniques based on data dependencies and not in code structure.

The work presented here leverages and expands previous work by the authors in the domain of non traditional tiling for parallel applications.

The main contributions of this paper are: (1) A formal description of tiling from the point of view of the data produced and not from the source code. (2) A mathematical proof for an optimum tiling in terms of maximum reuse for stencil applications, addressing the disparity between computation power and memory bandwidth for many-core architectures. (3) A description and implementation of our tiling technique for well known stencil applications. (4) Experimental evidence that confirms the effectiveness of the tiling proposed to alleviate the disparity between computation power and memory bandwidth for many-core architectures. Our experiments, performed using one of the first Cyclops-64 many-core chips produced, confirm the effectiveness of our approach to reduce the total number of memory operations of stencil applications as well as the running time of the application.

1 Introduction

This paper addresses the problem of how to optimize a class of scientific computing programs called *stencil computations* running on parallel many-core processor architectures. This class of applications performs many read-modify-write operations on a few data arrays. The main challenge faced by stencil applications is the limitation of off-chip memory accesses, both in terms of bandwidth and memory latency.

Tiling, described in [10,11,3,14,15,6,5,4,8] and in many other publications, is a commonly used technique to optimize stencil applications. Tiling transformations attempt to reduce the number of off-chip memory accesses by exploiting locality in a program. To that effect, frequently used data is loaded to the local processor memory where increased bandwidth and better latency are available.

Previous tiling techniques, when applied to *stencil applications*, achieve suboptimal results because they either do not take advantage of the algorithm [5], they only work for one-dimensional loop structures [8], they require redundant computations [4], or, in general, a good tiling solution can not be found because the particular code written by the programmer does not fit the tiling technique.

K. Cooper, J. Mellor-Crummey, and V. Sarkar (Eds.): LCPC 2010, LNCS 6548, pp. 77–91, 2011.

Previous research by the authors looked at the problem of tiling stencil applications from the side of the data produced in the application regardless of the source code used to produce such data. The question posed was *"What program partitioning will result in maximum locality for stencil applications running on many-core processors?"*, and a partial answer was provided for a particular instance of a 1 Dimensional Stencil application. Evidence on the advantage of the technique proposed was produced using a simulator [8].

The limitation of memory bandwidth on many-core applications has been addressed to other levels in the memory hierarchy for some linear algebra applications [13,2]. They have proposed alternatives to find optimum tiling to the register level and mechanism for hiding memory latency.

The main contributions of this paper, outlined in the abstract, are influenced by a simple idea: Program tiling and/or partitioning should not be the result of code transformations upon the original code, they should be a natural consequence of the data dependencies between the computations of a program. The source code of the program should only be used to generate a full data dependency graph for the computations. The data dependency graph is then used to formulate an optimization problem whose solution is the tiling strategy that, in our case, would result in the least use of off-chip memory resources. This approach fundamentally departs from other tiling techniques where the program partition is heavily influenced by the original structure of the source code.

The tiling strategies resulting from a dependency-graph-based optimization for stencil computations can be collectively designated as *Diamond Tilings.* A very limited form of Diamond Tiling was successfully demonstrated in the past [8] without a detailed description of the optimization problem or a general solution. This paper fully develops the idea of how to generate Diamond Tilings based on the data dependencies of an application. We show guidelines to implement tiles that increase reuse of data in the case of parallel stencil applications.

The key idea behind the development of the Diamond Tiling class of tilings is simple: Maximum locality can be obtained if all local, *enabled* computations are executed. In here, the term *enabled* is borrowed from the dataflow field, and it refers to a computation whose input data is available, and whose input data resides in on-chip memory. This thought is translated into an optimization problem that is solved in the context of the implementation possibilities such as memory available (that limits the total size of a tile), synchronization primitives and so on. The main difference between Diamond Tiling and other tiling techniques is that Diamond Tiling is not concerned with the original structure of the code. The source code is only used to generate a complete data dependency graph of the application, but it is not required in itself to do the tiling transformation. Many programming models can benefit from the strategies used in the development of Diamond Tiling: Dataflow programs, serial programs, parallel programs and any other kind of program where a data dependency can be extracted.

The effectiveness of Diamond Tiling is presented in our results. Diamond Tiling was compared against other tiling techniques using an application that models the propagation of electromagnetic waves (using the FDTD algorithm) in multiple dimensions. The Cyclops-64 many-core processor developed by IBM [1] is used as our testbed architecture. When DRAM bandwidth is the bottleneck of the application, Diamond Tiling provides the shortest running time, the smallest total amount of off-chip memory operations and the best performance among all other tiling techniques, surpassing recent, state of the art tiling techniques by many times in some cases.

The rest of the paper is organized as follows: Section 2 provides relevant background on the field, including details on stencil applications and the Cyclops-64 processor. Section 3 formally describes the problem addressed by this paper. Section 4 provides a mathematical proof of the optimality of Diamond Tiling to reduce the amount of DRAM operations, and Section 5 shows how to apply the mathematical results of Section 4 to implement an application. The experiments used to test the effectiveness of Diamond Tiling and their results are described in Sections 6 and 7. The paper concludes with conclusions and future work on Sections 8 and 9.

2 Background

This section provides a brief introduction of the knowledge required to understand the paper. This section covers Stencil Applications, Cyclops-64 and many-core Architectures, DRAM memory limitations in many-core architectures, and traditional tiling approaches.

2.1 Stencil Applications

Stencil Applications are characterized by a kernel that repeatedly updates an array with a copy of a previous version of the array. Stencil applications typically solve partial differential equations useful in science and engineering. Common stencil applications include simulations of Heat Propagation, Particle Transport, Electromagnetic Propagation, Mass Transport and others.

The Finite Difference Time Domain (FDTD) [16] technique is a common algorithm to simulate the propagation of electromagnetic waves through direct solution of Maxwell's Equations. FDTD was chosen to illustrate the techniques presented here since it is easy to understand, it is widely used, and it can be easily written for multiple dimensions.

2.2 Cyclops-64 and Many-Core Architectures

Cyclops-64 [1] is a many-core processor chip recently developed by IBM. Cyclops-64 allows simultaneous execution of 160 hardware threads. Cyclops-64 shows a new direction in computer architecture: It has a user-addressable on-chip memory and no data cache, it has direct hardware support for synchronization (barriers as well as hardware sleep and awake) and a rich set of atomic operations. The main features of Cyclops-64 chip are shown in Table 1.

Cyclops-64 was chosen as the testbed for this application since it has a large amount of parallelism (160 threads simultaneously) and it is highly programmable (each thread runs its own code).

Table 1. Cyclops-64 Features

<table>
<tr><th colspan="4">Chip Features</th></tr>
<tr><td>Processor Cores</td><td>80</td><td>On-Chip Interconnect</td><td>Full Crossbar</td></tr>
<tr><td>Frequency</td><td>500 MHz</td><td>Off-Chip Interconnect</td><td>Proprietary</td></tr>
<tr><td>Hardware Barriers</td><td>Yes</td><td>Instruction Cache</td><td>320 KB</td></tr>
<tr><td>Hardware Sleep - Awake</td><td>Yes</td><td>Addressable Memory</td><td>1GB</td></tr>
<tr><td>On-Chip Memory</td><td>User Managed 5MB</td><td>Off-Chip Memory Banks</td><td>4 x 64 bits</td></tr>
<tr><th colspan="4">Processor Core Features</th></tr>
<tr><td>Thread Units</td><td>2, single issue</td><td>Local Memory</td><td>32KB</td></tr>
<tr><td>Floating Point Unit</td><td>1, pipelined</td><td>Registers</td><td>64 x 64 bit</td></tr>
</table>

2.3 DRAM Limitations for Many-Core Architectures

For many-core architectures, the disparity between the availability of DRAM and the availability of floating point units is perhaps the most limiting factor for performance. This disparity is only likely to increase with each new processor chip generation: it becomes increasingly easier to build more and more floating point units inside the chip, but the physical connections that supply memory from outside the chip improve only marginally.

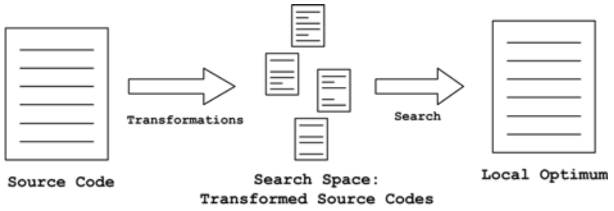
Current many-core architectures already suffer from this limitation. As an illustration, the ratio of bandwidth available to the Floating Point Units to off-chip bandwidth is already 60 to 1 in Cyclops (Table 1): There are 80 Floating Point Floating units that consume 2 and produce 1 64-bit values per cycle while there are only 4 memory banks that can supply 1 64-bit value per cycle each.

Floating point operations pay a low price in terms of latency, available resources and power, while, on the other hand, memory operations on DRAM have a long latency, use more power, and their total bandwidth is limited. For those reasons, memory operations to DRAM are the determinant factor in the cost of computations for scientific programs.

2.4 Tiling

The limitations of DRAM memory operations for many-core architectures drive program optimizations to regard memory operations as precious and it makes locality one of the most important optimization goals. An effective way to optimize for locality is to partition the computations of an application in groups called “Tiles”. Tiling, or partitioning the application into tiles, is an effective way to optimize for locality: The advantages of temporal locality can be exploited by reusing data.

Current tiling techniques follow the constraints of current paradigms. Traditional approaches are limited to search for possible partitions from a limited set of transformations on the starting source code. Newer approaches attempt to go beyond the limitations of source code, but their results are constrained to heuristics to transform the code or they are constrained by simplistic code generation approaches.



Traditional approaches find the best tiling technique from a fixed set of transformations. For example, they find the affine transformation that produces the best result on the source code.

Fig. 1. Traditional Approaches to Tiling

Some tiling techniques (Figure 1) apply a set of transformations to the source code provided and search for the best locality in the transformed programs. This approach was demonstrated with success in the past, but it is constrained by the fact that the search space is inherently limited by the programs that can be generated from the initial source code.

Tiling techniques that are not constrained by the structure of the original source code have been proposed [9]. The approach of those techniques is different to the approach presented in this paper and in some cases, they produce similar results.

3 Problem Formulation

The previous sections described the main techniques for tiling. Most of them are the result of applying clever transformations on the source code of the application.

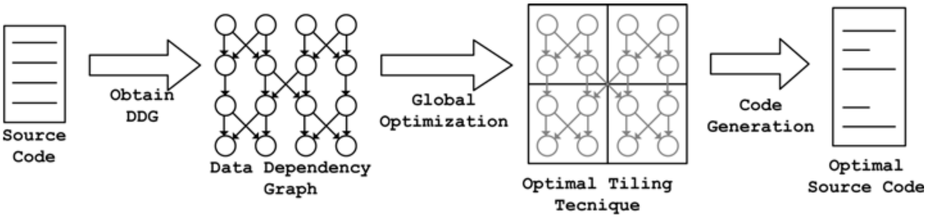
We propose instead focusing on the fundamental problem of tiling, regardless of code structure or expressiveness (Figure 2): *"What is the most effective tiling strategy to reduce off-chip bandwidth through the use of on-chip memory?"*

It is not claimed in this paper that it is always possible or convenient to find a solution to the tiling problem as presented in Figure 2, or that such a solution exists for all possible programs running all possible inputs on all possible architectures. For example, solution of the problem requires knowing the full dependency graph of the application, which can be practically or theoretically impossible for some applications.

The focus of this paper is specific: The problem of finding an optimum tiling is solved for stencil applications (See Section 2) running on parallel processors. Under those conditions the optimization problem can be approached because:

- The computations in the program are independent of the input data,
- it is possible to count all the computations in the program and
- it is possible to build a data dependency graph providing the relations between the computations on the program.

The rest of the paper presents the solution in detail.



Tiling, in this paper, is the result of an optimization done directly on the data dependency graph of the program. This is different to traditional tiling techniques where a set of proposed techniques is evaluated to find a local optimum.

Fig. 2. Tiling using Data Dependency Graphs

4 Optimal Tiling for Stencil Computations

The previous section formulated the problem of tiling optimality for a program in general. This section addresses the problem for the particular case of parallel stencil computations. Parallel in this context means that a tile can be fully computed when its memory has been loaded; no communication with off-chip memory or with other processors is required.

```
for t in 0 to NT-1

  for i in 1 to N-1
    E[i] = k1*E[i] +
        k2 * ( H[i] - H[i-1] )
  end for

  for i in 1 to N-1
    H[i]+=E[i]-E[i+1]
  end for

end for
```

Fig. 3. Kernel for FDTD 1D

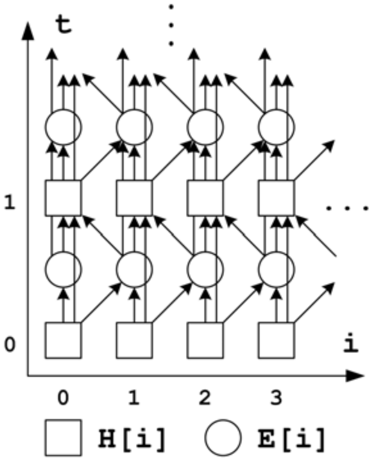


Fig. 4. DDG for FDTD 1D

The optimal tiling problem is approached as a formal optimization problem. The optimization problem maximizes the ratio of computations to memory operations for all possible tiling approaches.

Tiling of memory addresses and computations is an integer optimization problem on the computation space. Figure 4 shows the computation space for an implementation of FDTD (Figure 3). The problem is an integer optimization problem since all computations are associated with a spatial grid and a computation step. The problem is extended to allow continuous values for the memory addresses as well as the number of computations (Figure 5). The problem is then solved in the continuous space. As will be seen in the equations below, the optimum solution of the continuous problem is integer, which guarantees that it is also a solution of the integer optimization problem.

The rest of this section provides definitions of the variables used, some theorems and a final remark that ultimately results in the optimal tiling technique.

Definition 1. *D_T is the computation space. The elements of D_T are the values computed by the stencil application. Each one of the elements of D_T is associated with a vector $x_T = (x_{T,0}, x_{T,1}, ..., x_{T,M-1}, t)$ where M is the number of spatial dimensions in the stencil problem, and t represents time in the computation. All components in x_T are integers because they are associated with a spatial grid and a computation step.*

Example: The elements of D_T for the FDTD 1D implementation of Figure 3 are represented by circles and squares in Figure 4. In Figure 4, $x_{T,0}$ corresponds to the i direction and t, the last coordinate of the vector, corresponds to the t direction. For simplicity of the explanation, consider that the circle and the square with the same coordinates in space and time are merged and are referred to simply as one *element* of the computation space.

Definition 2. *Also, each element of D_T is associated with a set of vectors $d_i = (d_{i,0}, d_{i,1}, ..., d_{i,(M-1)}, d_{i,t})$, $i = 0, ..., p$ that represent the direction of the dependencies in the computation space. In Figure 4 each arrow represents each d_i These vectors are also illustrated in Figure 5.*

Definition 3. *D is the extended continuous space of D_T. In the same way, a point in D can be associated with a vector $x = (x_0, ..., x_{M-1}, t)$.*

Definition 4. *T is a region (e.g. a tile) of the continuous computation space D.*

Figure 5 shows the extended computation space D, and an arbitrary tile T.

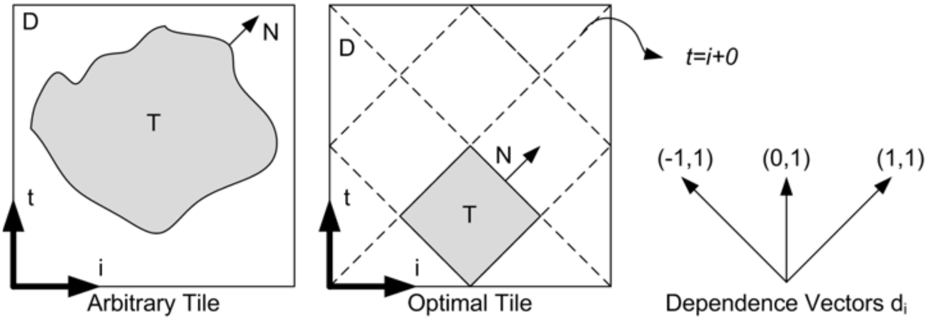


Fig. 5. Continuous computation space, tiles, and dependencies for FDTD 1D

Definition 5. *$S(T)$ represents the elements that have to be loaded to compute T and $V(T)$ represents the elements that can be computed in tile T. $S(T)$ and $V(T)$ can be interpreted as the surface and volume of T respectively.*

Definition 6. *N, or $N(x)$, $x \in S(T)$ is a unit vector perpendicular to the surface $S(T)$ at point x. The orientation of N is towards the inside of T if $S(T)$ is associated with loads or it is towards the outside of T if $S(T)$ is associated with stores.*

Definition 7. *Vertical bars $|\quad|$ are used to indicate the cardinality of a set in D_T. $|S(T)|$ and $|V(T)|$ can be interpreted as the area of the surface and volume of T respectively.*

Definition 8. *$m(T)$ is the amount of memory used by a particular tiling T and m_{max} is the maximum amount of on-chip memory available.*

The optimization problem for finding the tiling in the continuous space is:

$$\max_T f(T) = \frac{|V(T)|}{|S(T)|} \tag{1}$$

$$s.t. \quad m(T) \leq m_{max}, \quad T \text{ is a parallel tiling} \tag{2}$$

The solution can be found by the following theorems:

Theorem 1. *If T is a valid, parallel tiling, then $N \cdot d_i \geq 0$ for $i = 0, 1, ..., p$.*

Proof. This condition ensures the dependencies points inwards. It implies, computations inside T do not require loading data from other tiles or from memory and can be computed in parallel.

Theorem 2. *If T^* is an optimum solution of the optimization problem, then T^* is a convex tile.*

Proof. Considering a feasible non-convex tile T_{nc} and the smallest convex tile T_c that fully contains T_{nc}. T_c is also feasible. $f(T_{nc}) \leq f(T_c)$ because $|S(T_{nc})| \geq |S(T_c)|$ and $|V(T_{nc})| \leq |V(T_c)|$. Since for any feasible non-convex tile there exists a feasible convex tile with a better or equal value of $f(T)$, it follows that T^* is convex.

Theorem 3. *If T^* is an optimum solution of the optimization problem, then, at each point along the boundaries of T^*, N is perpendicular to at least one of the dependency vectors d_i at that point.*

Proof. Consider all the tiles T for which $|S(T)|$ is a particular constant value. For those tiles the optimization problem of Equation 1 is equivalent to maximization of $|V(T)|$ under the constraints of Equation 2.

Maximization of $|V(T)|$ is equivalent to maximization of a multidimensional integral[1] of $-N \cdot \hat{t}$ along the surface of T. Which is maximum when $N \cdot \hat{t}$ is minimum at all points along the surface of the tile T.

[1] The full details of the derivation of the integral are omitted due to space constraints.

Since theorem 1 implies that N lies in a cone extending in the direction of $+\hat{t}$. The minimum value of $N \cdot \hat{t}$ is when $N \cdot d_i = 0$, for some i and thus, N is perpendicular to at least one of the dependence vectors.

Theorem 4. *The Tiling technique that solves the optimization problem of Equations 1 and 2 are tiles where the computation space D_T is partitioned with planes that extend in the direction of the dependencies.*

Proof. Theorem 3 states that an optimum tile is formed by planes whose normal vectors are orthogonal to the dependence vectors. It follows that the planes extend in the direction of the dependence vectors. Because $N \cdot d_i = 0$ is required. The only shape that satisfies this requirement are diamonds. Thus the name given to the technique. This is also true in the computation space D_T where the optimum T^* is a diamond restricted to planes in a grid.

Theorem 5. *The optimum tile has a size that tries to use all the available memory.*

Proof. The function $f(T)$ is a ratio between volume and surface of the tile, and this ratio increases with the amount of memory used. The maximum tile size is bounded by the constraint described in Equation 2.

Theorems 4 and 5 are enough to uniquely describe the tiling partition (Figure 5). The tile partitions extend in the direction of the dependencies (Theorem 4) and the size of the tile should be as big as possible (Theorem 5).

Note that the results of this section find the global optimum of the described problem. The solution found here does not address other constraints not specified. For example, the result obtained in this section does not consider the effect of latency, only the effects of bandwidth because, as argued, bandwidth is likely to be the most important limiting factor, and other limitations such as latency can be hidden using techniques such as double buffering.

The following sections provide examples and results of using the theorems of this section to minimize the number of off-chip memory operations.

5 Implementation

The process to obtain the optimum tile size and shape for an application was described and solved for stencil applications in Section 4. This section provides and informal description of the results of previous sections that seeks clarity over formality. The complete details of the particular implementation used for the experiments has been prepared as a separate publication and it can be found in [7].

The main conclusion of Section 4 is that the optimum tiling technique is ultimately defined by the direction of the dependencies in the code. Those dependencies ultimately define how the application is tiled.

Consider the application of Figure 3, and its corresponding expanded data dependency graph (Figure 4). For the elements in the data dependency graph, the vectors that represent the dependencies are $(-1, 1)$, $(0, 1)$, and $(1, 1)$. Following the optimality result, the boundaries for the optimum tiling extend in the direction of the dependencies. The planes that form the tiles would be $t = i + c$, $i = c$ and $t = -i + c$, where c is a constant that is associated with the location of the tile, and t and i are the time and space indexes respectively. The normal vectors N, found using linear algebra on the planes, are $(-\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$, $(1, 0)$ and $(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$ respectively for the planes. The plane $i = c$ is not a valid plane because it violates the parallel constraint (Theorem 1) so $t = i + c$, and $t = -i + c$ are chosen as the partition planes.

The resulting planes partition the iteration space as shown in Figure 5. The planes are spaced so that each tile uses the maximum memory available.

Stencil applications in any number of dimensions can be similarly tiled.

6 Experiments

A number of experiments were conducted using the Cyclops-64 processor (described in Section 2) to test our claims of optimality for Diamond Tiling.

A simulation (using the FDTD technique) of an electromagnetic wave propagating both in 1 and 2 dimensions was used to compare Diamond Tiling to other traditional and state of the art tiling techniques.

The tiling techniques used were:

Naïve: The traditional rectangular tiling was used. Each one of the perfectly nested loops in the computation is fully tiled using all the available on-chip memory.

Overlapped: Tiling along the time dimension and the spatial dimensions is done at the expense of redundant computations and more memory operations [4].

Split: Two or more tile shapes are used to fully partition the iteration space such that time tiling with no redundant computations is achieved [4].

Diamond: Diamond Tiling, as described in this paper, was used.

Figure 6 shows a conceptual view of the tiling techniques when applied to a 1 dimensional problem. The experiments use tiling in both one and two dimensions.

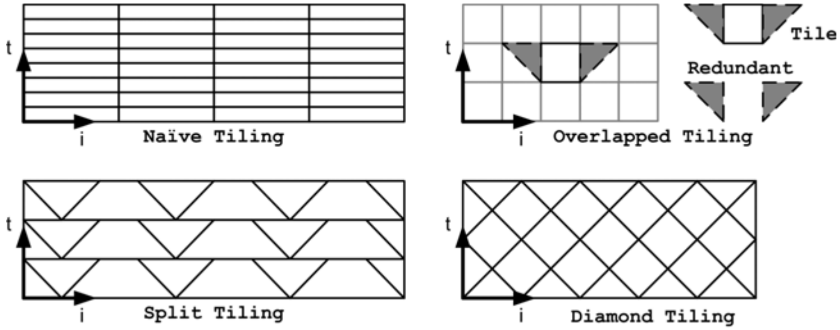


Fig. 6. Tiling techniques used in the experiments

Tiling was done at the on-chip memory level. Register tiling and tiling at other levels was not done because the focus of our study is the limitation in the memory operations between DRAM and the on-chip memory in many-core processors. Further levels of tiling are likely to accentuate the results, but such experiments fall outside of the scope of our work.

The code for each one of the tiling approaches was hand written in C. The programs were compiled with ET International's Compiler for Cyclops-64 version 4.8.17 with optimization flags -O3.

The required synchronization between the computational steps of each tiling approach was done using the hardware supported barriers available on Cyclops-64. The experiments were run using one of the first Cyclops-64 chips produced.

The 1 dimensional implementation computes 4000 timesteps of a problem of size 10000 while the 2 dimensional application computes 500 timesteps of a problem of size 1000 × 1000. The tile sizes used were 256 and 24 × 24 for the implementations in 1 and 2 dimensions respectively. These tile sizes were chosen so that they used all the available on-chip memory. Hardware counters on the Cyclops-64 chip where used to measure execution time, floating point operations and memory accesses.

All results are reported for the computational part of the program. The initialization and finalization stages of the program represent a minority of the execution time and code size and they do not participate in tiling. For that reason they are not included in the results.

7 Results

The result of executing the testbed application (FDTD) in 1 and 2 dimensions using several tiling techniques are presented in Figures 7 to 12.

The results confirm the hypothesis that Diamond Tiling is an effective way to tile stencil computations. For the two applications considered in section 6, the use of Diamond Tiling resulted in a lower total number of DRAM memory operations, and for the cases in which the bandwidth was the limiting factor of the application, Diamond Tiling resulted in the best performance.

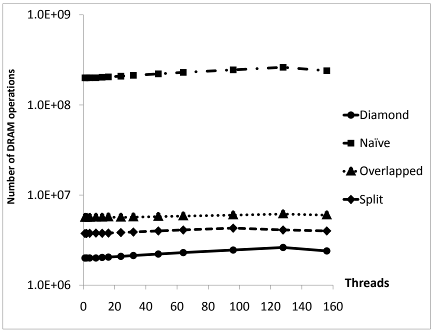


Fig. 7. FDTD 1D: Operations on DRAM

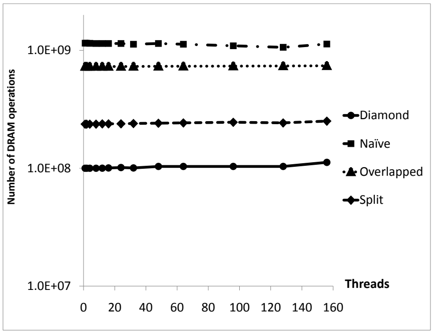


Fig. 8. FDTD 2D: Operations on DRAM

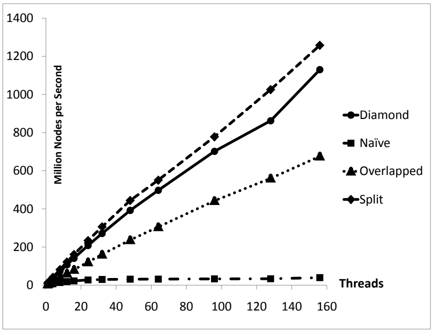


Fig. 9. FDTD 1D: Performance

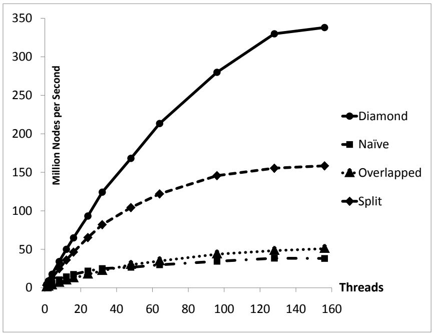


Fig. 10. FDTD 2D: Performance

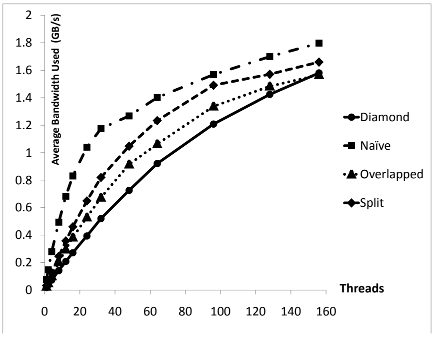


Fig. 11. FDTD 1D: Bandwidth used

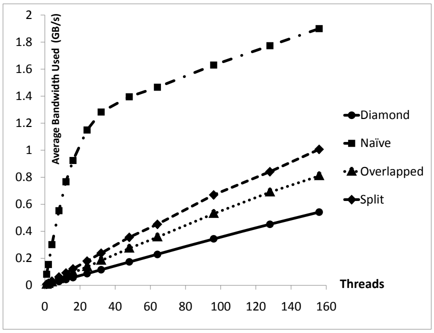


Fig. 12. FDTD 2D: Bandwidth used

Figures 7 and 8 show the total number of off-chip memory operations required to execute the program. They show how Diamond Tiling provides the lowest number of off-chip memory operations of all the tiling techniques in the experiments. As explained before, this decrease in the total number of memory operations greatly contributes to decrease the total running time of the application as can be detailed on Figures 9 and 10 using the number of nodes processed per second as a metric. As a result of the reduction in off-chip memory operations and the increasing of performance, the average bandwidth used is decreased, as can be seen on Figures 11 and 12. A better utilization of off-chip bandwidth is achieved particularly for 2D-FDTD using Diamond Tiling. In all cases, the maximum amount of bandwidth available was 2 GB/s.

Traditional tiling approaches such as the widely used rectangular tiling (our naïve implementation) have a performance that is far below the performance of Diamond Tiling. This is due to the fact that traditional tiling approaches do not always consider the advantages of tiling across several loops and are limited by the ability to generate code. Diamond Tiling has a much lower running time and it has far less off-chip memory operations.

The recent Overlapped Tiling [4] overcomes the limitation of low parallelism between tiles at the expense of more memory operations and redundant computations. The price paid for parallelism results in a lower execution speed and more off-chip memory operations. For large tiles, such as the ones used in the experiment, Diamond Tiles require less memory operations while still enabling full parallelism between tiles. Also, Diamond Tiling does not require redundant computations.

In summary, the experimental data presented here supports the idea that Diamond Tiling is an excellent technique to tile stencil applications in a parallel execution environment.

8 Conclusions

This paper presents a technique for data locality optimization that is based on the analysis of the data dependency graph of the application.

The most important contribution of this work is that it approaches the problem of tiling as a mathematical problem of optimality rather than as a transformation. Although the general approach of tiling a generic application through mathematical optimization of a performance function is not always feasible, it is nevertheless useful for Stencil Applications.

Section 4 showed a formal description of the problem and developed it to reach a simple conclusion: In stencil computations, the iteration space should be partitioned into Diamond-shaped tiles. This result was obtained for the case of parallel applications. If the parallel restriction is dropped, it may be possible to find other tiling techniques with better performance. For example if the execution is restricted to be serial, skewed tiling [15] may have a better performance than Diamond Tiling.

The resulting Diamond Tiling has excellent characteristics: (1) It produces fully parallel tiles, (2) it provides optimum utilization of the off-chip DRAM bandwidth, and (3) it is easy to generate code for it. Additionally, the positive impact of Diamond Tiling increases when combined with other optimizations and possible architecture changes such as better synchronization or more memory.

The results of this paper extend the results of previous research by the authors. In a previous publication, Diamond Tiling was conjectured to be optimal for FDTD in 1 dimension [8]. This paper formally confirms such claim and extends the result to general stencil applications in any number of dimensions.

This paper also opens the door for formal optimization of other applications where a full data dependency graph is possible to obtain. Or, at the very least, it provides an alternative to locality optimization from source code.

It is possible that the importance of optimization in a mathematical way will increase in future generations of many core architectures. If current trends in computer architecture continue, more and more parallelism will be available while data movement will become even more expensive.

9 Future Work

The main contributions of this paper are significant to stencil applications. Future work in locality using data dependency graphs can focus in extending its applicability to a larger number of applications, or to integrate other techniques into the formal framework proposed.

So far, the results presented here only address the difficulties of DRAM memory accesses of one many-core chip. The issue of how to partition a distributed memory application across multiple many-core chips using data dependency graphs is a future topic of research.

The tradeoff between barrier synchronizations and point to point synchronizations needs to be evaluated as well. Consumer-producer synchronization primitives such as phasers [12] can potentially reduce execution noise in stencil applications.

Future work will also focus on providing a generic mathematical framework for optimization. In the case of the stencil application optimized here, all the optimization steps were hand-designed by the authors. A more attractive approach would be to design a consistent methodology that could be applied to any application without being restricted to a particular class of applications. Whether or not that is possible is unknown to the authors at the time of publication.

Acknowledgements

This work was possible due to the generous support of the National Science Foundation through research grants CCF-0833122, CCF-0937907, CNS-0720531, CCF-0925863 and OCI-0904534. The authors thank the reviewers for their useful comments, and for helping improving the quality of the paper. The authors express their gratitude to all CAPSL members at the University of Delaware for their support during the course of this research.

References

1. del Cuvillo, J., Zhu, W., Hu, Z., Gao, G.R.: Toward a software infrastructure for the cyclops-64 cellular architecture. In: 20th International Symposium on High-Performance Computing in an Advanced Collaborative Environment, HPCS 2006, p. 9 (May 2006)
2. Garcia, E., Venetis, I.E., Khan, R., Gao, G.: Optimized dense matrix multiplication on a many-core architecture. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010. LNCS, vol. 6272, pp. 316–327. Springer, Heidelberg (2010)
3. Irigoin, F., Triolet, R.: Supernode partitioning. In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1988, pp. 319–329. ACM, New York (1988), http://doi.acm.org/10.1145/73560.73588
4. Krishnamoorthy, S., Baskaran, M., Bondhugula, U., Ramanujam, J., Rountev, A., Sadayappan, P.: Effective automatic parallelization of stencil computations. SIGPLAN Not. 42(6), 235–244 (2007)

5. Lam, M.S., Wolf, M.E.: A data locality optimizing algorithm. SIGPLAN Not. 39(4), 442–459 (2004)
6. Lim, A.W., Cheong, G.I., Lam, M.S.: An affine partitioning algorithm to maximize parallelism and minimize communication. In: ICS 1999: Proceedings of the 13th International Conference on Supercomputing, pp. 228–237. ACM, New York (1999)
7. Orozco, D., Gao, G.: Diamond Tiling: A Tiling Framework for Time-iterated Scientific Applications. In: CAPSL Technical Memo 91. University of Delaware (2009)
8. Orozco, D., Gao, G.: Mapping the fdtd application for many core processor. In: International Conference on Parallel Processing ICPP (2009)
9. Rajopadhye, S.: Dependence analysis and parallelizing transformations. In: Srikant, Y.N.S., Shankar, P. (eds.) Handbook on Compiler Design, 1st edn. CRC Press, Boca Raton (2002) (in press)
10. Ramanujam, J., Sadayappan, P.: Tiling multidimensional iteration spaces for multicomputers. Journal of Parallel and Distributed Computing 16(2), 108–120 (1992)
11. Schreiber, R., Dongarra, J.: Automatic Blocking of Nested Loops (1990)
12. Shirako, J., Peixotto, D.M., Sarkar, V., Scherer, W.N.: Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In: ICS 2008, pp. 277–288. ACM, New York (2008)
13. Venetis, I.E., Gao, G.R.: Mapping the LU Decomposition on a Many-Core Architecture: Challenges and Solutions. In: Proceedings of the 6th ACM Conference on Computing Frontiers (CF 2009), Ischia, Italy, pp. 71–80 (May 2009)
14. Wolf, M.E., Lam, M.S.: A data locality optimizing algorithm. SIGPLAN Not. 26(6), 30–44 (1991)
15. Wolfe, M.: More iteration space tiling. In: Supercomputing 1989: Proceedings of the 1989 ACM/IEEE Conference on Supercomputing, pp. 655–664. ACM, New York (1989)
16. Yee, K.: Numerical solution of inital boundary value problems involving maxwell's equations in isotropic media. IEEE Transactions on Antennas and Propagation 14(3), 302–307 (1966)

Array Regrouping on CMP with Non-uniform Cache Sharing

Yunlian Jiang[1], Eddy Z. Zhang[1], Xipeng Shen[1],
Yaoqing Gao[2], and Roch Archambault[2]

[1] Computer Science Department
The College of William and Mary, Williamsburg, VA
{jiang,eddy,xshen}@cs.wm.edu
[2] IBM Toronto Software Lab, Toronto, Canada

Abstract. Array regrouping enhances program spatial locality by interleaving elements of multiple arrays that tend to be accessed closely. Its effectiveness has been systematically studied for sequential programs running on unicore processors, but not for multithreading programs on modern Chip Multiprocessor (CMP) machines.

On one hand, the processor-level parallelism on CMP intensifies memory bandwidth pressure, suggesting the potential benefits of array regrouping for CMP computing. On the other hand, CMP architectures exhibit extra complexities—especially the hierarchical, heterogeneous cache sharing among hyperthreads, cores, and processors—that impose new challenges to array regrouping.

In this work, we initiate an exploration to the new opportunities and challenges. We propose cache-sharing-aware reference affinity analysis for identifying data affinity in multithreading applications. The analysis consists of affinity-guided thread scheduling and hierarchical reference-vector merging, handles cache sharing among both hyperthreads and cores, and offers hints for array regrouping and the avoidance of false sharing. Preliminary experiments demonstrate the potential of the techniques in improving locality of multithreading applications on CMP with various pitfalls avoided.

1 Introduction

Modern processor industry relies on the continuous increase of processor-level parallelism. As a result, memory bandwidth is being shared by an increasing number of threads and processes. Its usage become more critical than before.

Array regrouping is one approach to reducing memory transactions by enhancing spatial locality and reducing cache conflicts. It has shown promising results on sequential programs in unicore processors. The basic idea of array regrouping is to merge arrays that are always accessed at the same time together to form a new big array. Figure 1 shows a simple example. Figure 1 (a) is the original program. Elements from arrays A and B with the same index are always accessed together. As shown in Figure 1 (b), to improve spatial locality, we can group arrays A and B together to form a new two-dimensional array F, in which, F[i][0] and F[i][1] represent A[i] and B[i] in the original program respectively. After the transformation, one iteration requires only at most one memory transaction, reducing memory bandwidth pressure significantly.

K. Cooper, J. Mellor-Crummey, and V. Sarkar (Eds.): LCPC 2010, LNCS 6548, pp. 92–105, 2011.

```
for(i=0;i<N;i++){
  A[C[i]] += B[C[i]];
 }
```

(a) Original

```
for(i=0;i<N;i++){
  F[C[i]][0] += F[C[i]][1];
 }
```

(b) Regrouped

Fig. 1. An example of array regrouping

As an effective method for spatial locality enhancement, array regrouping is potentially useful for the improvement of effective memory bandwidth of multithreading applications running on Chip Multiprocessors (CMP). However, some new features of CMP architecture, especially the heterogeneous sharing of memory systems, imposes additional complexities. On a CMP machine with SMT (Simultaneous Multithreading) enabled, multiple hyperthreads on a core share the whole memory hierarchy, multiple cores on a single chip share the hierarchy below a certain cache level, and the processors on different chips share the main memory only. The non-uniform sharing presents a spectrum of cache contention, hence both new challenges and new opportunities for array regrouping. Being oblivious to cache sharing, as existing techniques are, may lose optimization opportunities, as well as cause programs unnecessary cache thrashing.

This paper presents a three-fold exploration to the problem, with the focus on the following questions: What unique opportunities and challenges CMP brings to array regrouping, how to effectively apply the transformation to multithreading applications, and how to maximize the benefits and avoid various pitfalls.

First, through analytical and empirical investigations, we demonstrate that multithreading programs on CMP can not only benefit from array regrouping as sequential programs do, but also expose additional opportunities for array regrouping to exploit. These opportunities mainly come from the presence of cache sharing in CMP. Consider the example shown in Figure 2. If the two threads share a cache, grouping arrays A, B, and C may improve their spatial locality, forming a synergistic sharing between the two threads. This kind of regrouping is named *cache-sharing-aware array regrouping*. Such opportunities do not exist in the sequential version of the program or on traditional machines that have no shared cache.

```
for(i=0;i<N;i++){
  E[i] = A[P[i]] + B[P[i]];
 }
```

(a) Execution by thread 1

```
for(i=0;i<N;i++){
  F[i] = A[P[i]] + C[P[i]];
 }
```

(b) Execution by thread 2

Fig. 2. An example of cross thread affinity

Second, we show that existing data reference affinity analysis is insufficient for applying array regrouping in CMP settings. Reference affinity characterizes the access pattern of a group of arrays which are always accessed together in the whole program. Existing reference affinity analysis techniques [4, 12] are all built for sequential programs running on unicore processors. They are insufficient for exploiting the additional opportunities the new scenario offers. For the example in Figure 2, data reference analysis within a single thread is not able to recognize that A, B, C, are accessed at the same time and hence should be grouped together. Cross-thread data affinity analysis is indispensable. Moreover, existing analysis techniques are prone to various pitfalls, especially false sharing, when applied to systems with shared cache as Section 2.2 elaborates.

Finally, we propose a two-step cache-sharing-aware reference affinity analysis to compute the reference affinity among data objects both within the execution of a thread and among the executions of different threads. The algorithm first uses the hierarchical perfect matching algorithm to determine a good assignment of threads on CMP, and then computes the reference affinity among data objects with cache sharing taken into account. The computed affinities offer guidance to array regrouping and also help prevent false sharing from being introduced. Some preliminary experiments demonstrate the potential of the techniques for improving the performance of multithreading applications running in CMP.

The rest of this paper is organized as follows. Section 2 discusses the new opportunities and challenges that parallel computing on CMP imposes on array regrouping. It proposes a two-step cache-sharing-aware reference affinity analysis, and describes the use of the analysis for guiding array regrouping. Section 3 reports some preliminary results. Section 4 reviews the related work. Section 5 concludes this paper with a short summary.

2 Array Regrouping for Multithreading Applications on CMP

Reference affinity analysis—discovering the data objects in a program that have good reference affinity—is the key to effective array regrouping. To make array regrouping take advantage of the complex sharing in memory hierarchy of modern CMP, we develop a cache-sharing-aware reference affinity analysis for multithreading applications running on CMP. The approach is enlightened by a frequency-based affinity analysis proposed previously for sequential applications. This section first reviews the previous technique, then describes cache-sharing-aware reference affinity analysis, and finally discusses the use of the analysis for array regrouping.

2.1 Review of Basic Frequency-Based Affinity Analysis

Ding and Kennedy gave the first compiler technique for array regrouping [4]. They defined the concept reference affinity. Later, Zhong et al. redefined reference affinity at the trace level using reuse distance. They proposed a distance-based affinity analysis, which offers more accurate reference affinity information, but with a high profiling overhead.

Our current work is based on a frequency-based affinity analysis, proposed by Shen and others [12]. This analysis uses a frequency-based model, and employs interprocedural program analysis to measure the access frequency in the presence of array parameters and aliases, striking a good tradeoff between the overhead of analysis and the quality of produced results.

The analysis framework consists of five components.

- Building the control flow graph and the invocation graph with data flow analysis;
- Estimating the execution frequency through either static analysis or profiling;

– Building array access-frequency vectors using interprocedural analysis;
– Calculating the affinity between each array pair and constructing the affinity graph;
– Partitioning the graph to find affinity groups in linear time.

Our review concentrates on the frequency-based affinity model in the framework because of its close relevance to this current study. In the model, a program is regarded as a set of code units, in particular, loops. Suppose there are K code units. Let f_i represent the total occurrences of the ith unit in the program execution, and $r_i(A)$ represent the number of references to array A in an execution of the ith unit. The frequency vector of array A is defined as follows:

$$V(A) = (v_1, v_2, \cdots, v_k)$$

where,

$$\begin{aligned} v_i &= 0 \quad if \;\; r_i(A) = 0; \\ v_i &= f_i \quad if \;\; r_i(A) > 0. \end{aligned}$$

A code unit i may have branches inside and may call other functions. The authors of the previous work conservatively assume that a branch goes both directions when collecting the data access. They use interprocedural analysis to find the side effects of function calls. To save space, a bit vector substitutes the access vector of each array and a separate vector records the frequency of code units.

The affinity between two arrays is the Manhattan distance between their access-frequency vectors, as shown below. It is a number between zero and one. Zero means that two arrays are never used together, while one means that both are accessed whenever one is.

$$affinity(A,B) = 1 - \frac{\sum_{i=1}^{K} |v_i(A) - v_i(B)|}{\sum_{i=1}^{K}((v_i(A) + v_i(B))}.$$

The calculated affinities compose an affinity graph, from which, affinity groups are derived. In the graph, each node represents an array, and the weight of an edge between two nodes is the calculated affinity between them. There are additional constraints. To be regrouped, two arrays must be compatible in that they should have the same number of elements and they should be accessed in the same order. The data access order is not always possible to analyze at compile time. However, when the information is available to show that two arrays are not accessed in the same order in a code unit, the weight of their affinity edge will be reset to zero. The same is true if two arrays differ in size.

Graph partitioning is done through a graph traversal. It merges two nodes into a group if the affinity weight is over a threshold. After partitioning, each remaining node is a set of arrays to be grouped together. The threshold determines the minimal amount of affinity for array regrouping.

This frequency-based reference affinity shows good results on some sequential scientific benchmarks. Next, we describe how to extend it to handle complex cache sharing on CMP.

2.2 Cache-Sharing-Aware Reference Affinity Analysis

The goal of cache-sharing-aware reference affinity analysis is to recognize the appropriate arrays in a multithreading application to regroup so that the usage of the cache in the CMP can be maximized. We first outline the main challenges, and then describe our solutions.

Complexities to Consider. Compared to reference affinity analyses for sequential applications, there are three-fold distinctive challenges for multithreading applications running in CMP.

First, the non-uniform cache sharing in CMP suggests that different analyses are necessary for threads sharing different levels of cache. Figure 3 shows an example. Loop I reads array A and loop II updates array B. Because the two loops run in parallel, A, B and C have good affinity, and can be regrouped together. If the two threads run on the same core with hyperthreads enabled, the regrouping works fine. But if they run on different cores or processors, the regrouping causes unnecessary false sharing. The updates by thread II invalidate the cache lines that contain data to be read by thread I. This example also demonstrate the needs for distinguishing benign and malicious data sharing in different scenarios during the affinity analysis.

```
for(i=0;i<N;i++){
 temp += A[D[i]] + C[D[i]]
}
```

(a) Loop I executed by thread 1.

```
for(i=0;i<N;i++){
 B[D[i]] += A[D[i]]
}
```

(b) Loop II executed by thread 2.

Fig. 3. An example showing different analyses are needed for threads sharing different levels of cache

The second challenge to reference affinity analysis on CMP relates to the first: Reference affinity must couple with thread binding and scheduling. Explicitly binding threads to computing units reveals the cache sharing relation among threads and hence prepares for the computing of inter-thread data affinity. On the other hand, the data sharing relation among threads determines the best way to bind threads to computing units. The two form a chicken-egg problem.

The third challenge facing reference affinity analysis in CMP is the timing issue. It is important to match the code segments that are executed in parallel by different threads. However, it is sometimes difficult to accurately determine the matching during compile time given the pointers, aliases, and synchronization complexities; profiling may help, but is subject to input changes.

Overview of Cache-Sharing-Aware Reference Affinity Analysis. To overcome the various challenges, we develop a two-step reference affinity analysis. During the first step, we employ coarse-grained thread affinity to determine the appropriate bindings of threads to computing units. During this step, we construct a thread-affinity graph and formulate the thread scheduling problem to a minimum-weight perfect matching problem. In the second step, we build some reference frequency vectors for each thread and employ a hierarchical merging algorithm to compute the reference affinity and partition program data into appropriate affinity groups.

Before proceeding to the details of the analysis, we emphasize the distinction between two terms, *thread affinity* and *reference affinity*. The former refers to the relation between two **threads**—whether they have lots of data sharing; whereas, the latter refers to the relation between two **data objects**—whether they are used together during the execution of the program. Thread affinity is used during the first step of cache-sharing-aware reference affinity analysis, and reference affinity is the key concept for the second step.

Step 1: Scheduling Guided by Thread Affinity. We employ coarse-grained thread affinity to facilitate the assignment of threads to computing units. The basic idea is to arrange threads with a large volume of shared data close to each other to shorten their data-sharing path.

The thread affinity between two threads i and j is defined as follows:

$$Affinity_{ij} = \frac{|Data_i \bigcap Data_j|}{|Data_i \bigcup Data_j|}$$

where, $Data_i$ is the set of data accessed by thread i. Thread affinity shows how much data is shared between two threads. It is coarse-grained, considering the entire execution of a thread rather than each individual program construct executed by the thread. The large granularity may loose some accuracy but gain simplicity and efficiency.

With the affinities computed for every two threads, we use a thread-affinity graph to formulate the thread scheduling problem into a minimum-weight perfect matching problem. To simplify the explanation, the following description first assumes that the underlying architecture contains C dual-core processors without hyperthreads, and there are T threads, where $T = 2 * C$. The scheduling problem is to assign each thread to one core, forming a one to one matching between the threads and cores.

A thread-affinity graph is a fully connected graph, with each node representing a thread. Each edge carries a weight, equaling $(1 - affinity_{ij})$ for the edge between the nodes of threads i and j, as Figure 4 illustrates.

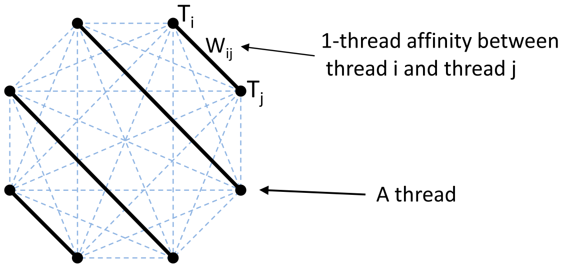


Fig. 4. A thread-affinity graph. The thick lines compose a perfect matching for the graph, corresponding to one schedule for the corresponding threads on a dual-core architecture.

After the construction of an affinity graph, the thread-core assignment problem becomes similar to the optimal job co-scheduling problem tackled in previous work [6]. The difference is that in the previous problem, the jobs are independent and share no data.

To explain the solution, we introduce two concepts in graph theory. A *perfect matching* in a graph is a subset of edges that cover all vertices but only once. A *minimum-weight perfect matching* problem is to find a perfect matching that has the minimum sum of edge weights in a graph.

It is obvious that a minimum-weight perfect matching in an affinity graph corresponds to an optimal thread-core assignment if the criterion is to maximize the sum of the affinity values of all co-running threads. We employ a well-known polynomial-time algorithm named *blossom* [5,2] to find minimum-weight perfect matchings. The time complexity of the algorithm is $O(T^4)$, where T is the number of nodes.

For a CMP architecture with either hyperthreads or more than two cores per chip, the thread scheduling problem can be solved through a hierarchical minimum-weight perfect matching algorithm [6]. The idea is to start from the smallest computing units, and apply the minimum-weight perfect matching algorithm iteratively.

As an example, consider a system with C quad-core chips and each core has two hyperthreads. We have $T = 8 * C$ threads to run and the goal is to assign each thread to a hyperthread so that the total thread affinity among co-running threads is maximized. In the first iteration, the thread-affinity graph has T nodes. After applying the minimum-weight perfect matching algorithm, we obtain $T/2$ pairs of threads, which tell us which two threads should co-run on a core. Next, we treat each pair of threads as one single large thread and construct a new thread-affinity graph with $T/2$ nodes. The minimum-weight perfect matching algorithm then produces $T/4$ groups of threads. We then treat each group of threads as one single large thread and apply the operations once again. All original threads are now partitioned into $T/8$ groups, and each group consists of four pairs of original threads. The thread scheduling becomes clear: We just need to assign each group of threads to one processor, with each pair of threads in the group assigned to one core in the processor.

Using this hierarchical algorithm, we can approximate the optimal solution for K-core H-hyperthread CMP by applying the minimum perfect matching algorithm $log(KH)$ times. The result will be a hierarchy of thread groups, corresponding to a reasonable schedule of threads to computing units.

Step 2: Reference Affinity Analysis through Hierarchical Merging. With threads bound to computing units, we can now analyze the reference affinity among threads that are assigned to the same processor. Recall that the goal of reference affinity analysis is to measure the reference affinity between major data objects (we concentrate on arrays in this work) in a program and find data objects with good affinity to guide array regrouping. Our approach consists of two stages.

In the first stage, we attempt to find array affinity groups within each single thread. We employ the basic frequency-based affinity analysis described in Section 2.1 to do so. By estimating the access number and the access stride of every array in each inner-most loop, we build a reference frequency vector for each array. A reference vector of an array A for a thread is defined as $V_A = (c_1, c_2, ..., c_n)$, where c_i is the number of references that thread conducts to array A in the i^{th} inner-most loop of the program. Based on these vectors, we can build a reference-affinity graph of all the arrays for a thread and find the affinity groups using the algorithm described in Section 2.1. After obtaining reference-affinity groups for each thread, we need to verify that an affinity group in one thread is indeed an affinity group of the whole program. The condition to check for an affinity group of a thread is that for each of the other threads, that group of arrays are either an affinity group as well or not accessed at all.

In the second stage, we deal with array affinity across the threads that run on the same core or chip. We build two access frequency vectors for each array: One for reads, the other for writes. The separation is important for false sharing and other reasons mentioned in Section 2.2.

With the read and write vectors built for every thread, the next step is vector merging. If the loop corresponding to an element in V_A of thread I happens to run in parallel with the loop corresponding to an element in V_A of thread II, the two elements are summed together to form one element in the merged vector. Figure 5 shows an example.

```
L1{                       L4{
   ... = A[i]                ... = B[i]
}                         }
L2{                       L5{
   ... = A[i] + B[i]         ... = A[i] - B[i]
}                         }
L3{                       L6{
   ... = B[i]                ... = A[i]
}                         }
   thread 1                  thread 2
```

(a) Code executed

(array A)	(array B)
a_1	b_4
$a_2 + a_5$	$b_2 + b_5$
a_6	b_3

(b) Elements of merged read vectors

Fig. 5. An example illustrating basic merging of frequency vectors. Assume thread 1 and thread 2 are assigned on the same processor, with each executing 3 loops. The read vector of array A on thread 1 and thread 2 are $< a_1, a_2, 0 >$ and $< 0, a_5, a_6 >$ respectively, where a_i is the references to A in loop i. After merging, the vector of array A becomes $< a_1, a_2 + a_5, a_6 >$.

To match with the hierarchical cache sharing on CMP, we apply the vector merging in a hierarchical fashion. The hierarchy includes two levels: the *hyperthread level*, and *CPU core level*. If the system has hyperthreads enabled, the merging is first applied to the threads bound to the hyperthreads of a single core. This step does not distinguish reads from writes: The read and write vectors of an array for a thread are consolidated into one reference vector before the merging algorithm is applied. This consolidation is because hyperthreads on one core share the entire memory hierarchy including the cache closest to the computing units; as a result, writes by one thread to a cache line do not invalidate the cache line read by another thread in the same core. After this first step, we construct the reference-affinity graphs using the merged vectors and apply the graph algorithm mentioned in section 2.1 to compute the reference affinity groups for the data referenced by the threads on each core.

In the next step, we apply the merging algorithm to the threads running on the same chip (assuming they share a single last-level cache). But this time, we distinguish writes from reads and merge the two types of vectors separately. The reference affinity is computed using only merged *read* vectors because of the risks of false sharing that writes may cause. Section 2.3 will explain the uses of the affinity analysis results and roles of write frequency vectors for array regrouping.

The reference affinity analysis requires the knowledge of the time matching between threads. Obtaining an accurate matching for complex programs with various synchronizations may be difficult. Some previous work (e.g., [11]) on parallel program analysis can help; detailed explorations to this issue is out of the scope of this paper. In our experiments, we use the OpenMP pragmas and pthread functions in the applications as heuristics for the matching.

The final output from the reference affinity analysis is two levels of affinity groups. Each affinity group contains a number of arrays whose reference affinity is greater than a predefined threshold (0.95 in our experiments). The groups in the first level correspond to the execution of the threads bound to one core, and the groups in the second level correspond to the execution of the threads bound to all cores on a single processor.

2.3 Array Regrouping

A straightforward way to use the affinity groups for array regrouping is just to group the arrays that belong to a single affinity group into one. However, this simple approach cannot work for several problems.

First, the affinity groups may conflict with one another. The conflict may exist in two dimensions. The first dimension is among the affinity groups in the same level. For example, two arrays may belong to the same affinity group for threads on processor 1, but not for threads on processor 2. The second dimension is across levels. Two arrays may belong to the same affinity group on the hyperthread level, but not on the processor level.

How to solve the conflicts depends on the types of transformations that can be applied to the program. Sophisticated transformations may create different versions of the relevant code segments to accommodate the different affinity groups. For instance, threads on processor 1 may invoke the version which has A and B regrouped, while, threads on processor 2 may run the version with A and B separated.

Our manual array regrouping produces only one version for a program. Therefore, before the regrouping is applied, we validate the affinity groups in a whole-program, entire-system prospective. Only the groups that show affinity on all cores are regrouped. More sophisticated treatment will be explored in the future.

The second issue for array regrouping is to prevent regrouping from introducing false sharing. For a reference-affinity group $P = a_1, a_2, ..., a_n$, where a_i is an array, if a_i is written on core c_p and $\exists c_q \neq c_p, a_j \neq a_i$, a_j has read accesses at core c_q, the compiler checks whether its data reference pattern analysis [12] can ensure that the read-write sharing of an array causes no false sharing. If it finds risks of false sharing or cannot determine the risks, a_i is removed from the affinity group.

3 Evaluation

This section reports some preliminary evaluation results. We use the IBM optimizer TPO (Toronto Portable Optimizer) as the framework for implementing a prototype of the techniques. TPO is the core optimization component in IBM XL C/C++ and FORTRAN compilers. It implements both compile-time and link-time methods for intra- and interprocedural optimizations. After getting the affinity groups, we manually conduct regrouping transformations. To examine the benefits on different CMP architectures, our experiments run on three types of machines, which will be described along with the following experimental results.

We use three programs for evaluation, demonstrating three-fold benefits of the shared-cache-aware affinity analysis and transformations respectively.

3.1 Affinity-Guided Scheduling for *Streamcluster*

We use the first program to examine the usefulness of the thread-affinity analysis for guiding thread scheduling. The program is an online clustering program, named "streamcluster". It originally comes from the PARSEC suite [1]. The version we use goes through the shared-cache-aware optimizations described in a previous work [14]. Every two sibling threads work on a chunk of data points. Each thread computes the distance from every point in the chunk to a number of centers. In our experiments, there are eight threads.

The computed thread affinities fall into two categories. A pair of sibling threads share a chunk of data points. In the native inputs coming with the benchmark, the chunk size is 1000; the thread affinity among sibling threads is 96.2%. Non-sibling threads share nothing except a center; their affinity values are about 0. With the direction of the affinity values, threads are scheduled such that sibling threads are bound to the same chip.

The performance gain of the affinity-guided scheduling depends on the architecture. On a Westmere machine (a dual-socket quad-core workstation with Westmere processors clocked at 2.53 GHz; each core has a 32 KB private L1 D-cache, 256 KB private L2 cache; the four cores on a chip share a 12 MB L3 cache), the performance gain is negligible. Whereas, on a a Dell PowerEdge 2950 server equipped with 2 quad-core Intel Xeon E5310 processors, the scheduling yields a speedup as much as a factor of 1.4 compared to the case where sibling threads are scheduled to separate chips. The main reason for the negligible benefits on the Westmere machine is that the performance bottleneck exists in the references to the remote memory in the NUMA architecture. The scheduling benefits are overshadowed by the effects of the bottleneck. The previous intra-thread affinity analysis is obviously insufficient to discover the cross-thread data affinity and guide the scheduling.

3.2 Spatial Locality Enhancement for *Summation*

In our second experiment, we use the "summation" kernel shown in Figure 6 to demonstrate the benefits of shared-cache-aware array regrouping in enhancing the spatial locality. We select this kernel because it is amenable for experimenting different size of footprints and analyzing cache behaviors. Because only the first elements in a row of the two-dimensional arrays are used in the computation, the original program has poor spatial locality, especially when N is considerably large. This scenario may seem artificial, but in many programs where only one field of a structure or class is used in a phase, such poor spatial locality is not rare.

For this program, the prior intra-thread reference affinity cannot recognize that arrays A, B, and C are used at the same time. But the cross-thread reference affinity analysis

```
double A[M][N], B[M][N], C[M][N];
double V[M][N], W[M][N];

// Thread 1:
  for (i=0; i< IT; i++)
    for (j=0; j<M; j++)
      V[j][0] += (A[j][0]+B[j][0]);

// Thread 2:
  for (i=0; i< IT; i++)
    for (j=0; j<M; j++)
      W[j][0] += (A[j][0]+C[j][0]);
```

Fig. 6. A kernel named “summation”

successfully recognizes such a reference affinity group. We measure the benefits of the corresponding array regrouping on two architectures. One is an IBM Power 5 server (1.7 GHz; 32 KB private L1 D-cache; 1.5 MB L2 D-cache shared by two cores), the other is the Intel Westmere machine as described in the previous sub-section.

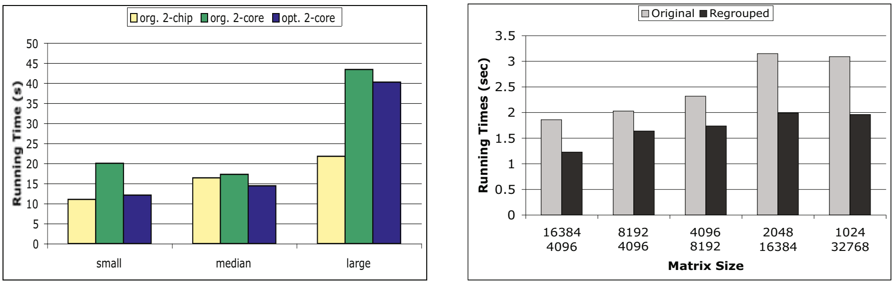
Figure 7 (a) shows the performance improvement on the IBM machine. The small run (M=128, N=8) has footprint smaller than the size of L1 cache, the median run (M=1024, N=8) larger than L1 but smaller than L2 and the large run (M=2048, N=32) larger than L2. In each run, we control the value of “IT” to ensure that each thread add exactly 1000 million array elements. The “org. 2-core” bars show that because of cache contention, the original program exhibits 5–100% performance degradation. The “opt. 2-core” bars show that array regrouping cuts the degradation by 15–70%. It is worth noting that the “org. 2-core” of the small run is slower than that of the median run; having more cache conflicts is the major cause—we will come back to this point in the next sub-section.

Figure 7 (b) summarizes the speedups brought by the array regrouping on the Intel Westmere architecture. We still use two threads and arrange them to run on the same chip (different cores). But because of the much larger cache and higher clock frequency than the Power 5, we use larger data sizes than before. The results show that the regrouping improves the performance across all the inputs with an average speedup of 1.3. The improvement mainly comes from the synergistic memory loading between threads and the spatial locality improvement.

3.3 Cache Conflict Reduction for *Swim*

The third experiment demonstrates that array regrouping can help a program even if it already has good spatial locality. The main benefits are the reduction of cache conflicts. The program we chose is “swim”, a SPEC OpenMP 2001 benchmark. Its memory references are regular, showing good spatial locality. However, it references 14 arrays, which tend to cause cache conflicts when multiple threads co-run together.

Our analysis finds four affinity groups ($< unew, vnew, pnew >, < u, v >, < uold, vold, pold >, < cu, cv, z, h >$); we regroup each of them into one array. This regrouping reduces the number of arrays to be accessed from 14 to 5, hence reducing the chance



(a) On IBM Power5.

(b) On Intel Westmere.

Fig. 7. Running times of the summation kernel on two CMP architectures

of cache conflicts. Figure 8 reports the speedups on the Intel Westmere machine with different numbers of threads assigned to various numbers of cores and hyperthreads. They demonstrate the gains from the reduced cache conflicts. We note that even though the reduction of cache conflicts through array regrouping may exist on unicore processors as well, it is more prominent for multithreading applications running on shared cache, where the chance for cache conflicts is greater than in unicore scenarios. In Figure 8, the benefits become modest for the eight-thread case because as the working set becomes small, cache conflicts and contention in the original program become less intensive than in the 4-thread or 2-thread cases.

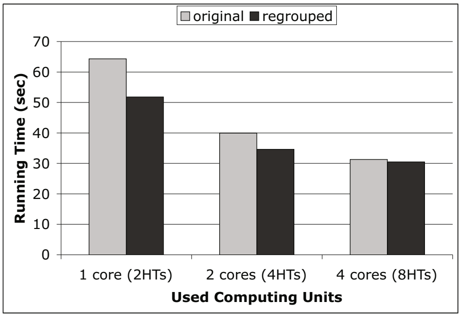


Fig. 8. Speedups of Swim

The benefits for cache conflicts reduction are sensitive to cache configurations. In various architectures we experiment with, the transformed *swim* shows different speedups. But in no cases, the transformation slows down the program executions.

4 Related Work

There have been a large number of studies on program data locality improvement. We concentrate our discussion on some recent work closely related to data reorganizations and software explorations to take advantage of cache sharing on CMP.

Ding and Kennedy used array regrouping to improve program locality [4]. Their technique interleaves the elements of two arrays if they are always accessed together. Zhong and others used trace-level reference affinity and profiling-based method for array regrouping and structure splitting [16]. A series of studies have explored automatic, safe data transformations for general purpose programs [8, 15, 3]. These studies concentrate on sequential programs and the enhancement of locality on unicore processors with no cache sharing.

Cache sharing exists in both SMT and CMP architectures. Its presence has drawn some recent work in compiler research. Tullsen and others [10, 7] have proposed compiler techniques to change data and instructions placement to reduce cache conflicts among independent programs. Nikolopoulos [9] has examined a set of manual code and data transformations for improving shared cache performance on SMT processors. In a recent study, Zhang and others [14] systematically examine the influence of cache sharing on the performance of modern multithreading applications running on CMP, concluding that program-level cache-sharing-aware transformation is one of the most important approaches for maximizing the usage of shared cache. Our current work is a step in that direction.

There are some previous work on cache-sharing-aware scheduling. Besides the optimal co-scheduling of independent jobs by Jiang and others mentioned in Section 2.2, Tam and others [13] propose thread clustering to group threads that share many data to the same processor through runtime hardware performance monitoring. As a runtime approach, their scheduling method is heuristics-based; the affinity-based thread scheduling presented in this paper maximizes the total affinity value.

5 Conclusion

This paper explores the special opportunities and challenges facing the use of array regrouping for locality enhancement of multithreading applications to improve effective bandwidth on modern CMP architectures. It presents a cache-sharing-aware reference affinity analysis to take cache sharing into account for analyzing memory reference patterns. The analysis consists of affinity-guided thread scheduling and hierarchical vector merging, handles cache sharing both within a core (i.e., among hyperthreads) and across cores, and offers hints to both array regrouping and the avoidance of false sharing. Preliminary experiments demonstrate the potential of the analysis for benefiting locality improvement of multithreading applications running on CMP.

Acknowledgment

We owe the anonymous reviewers our gratitude for their helpful comments on the paper. This material is based upon work supported by the National Science Foundation under Grant No. 0720499, 0811791 and 0954015, along with IBM CAS Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or IBM Corporation.

References

1. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The PARSEC benchmark suite: characterization and architectural implications. In: Proceedings of International Conference on Parallel Architectures and Compilation Techniques, pp. 72–81 (2008)
2. Cook, W., Rohe, A.: Computing minimum-weight perfect matchings. INFORMS Journal on Computing 11, 138–148 (1999)
3. Curial, S., Zhao, P., Amaral, J.N., Gao, Y., Cui, S., Silvera, R., Archambault, R.: Memory pooling assisted data splitting (mpads). In: Proceedings of the International Symposium on Memory Management (2008)
4. Ding, C., Kennedy, K.: Improving effective bandwidth through compiler enhancement of global cache reuse. Journal of Parallel and Distributed Computing 64(1), 108–134 (2004)
5. Edmonds, J.: Maximum matching and a polyhedron with 0,1-vertices. Journal of Research of the National Bureau of Standards B 69B, 125–130 (1965)
6. Jiang, Y., Shen, X., Chen, J., Tripathi, R.: Analysis and approximation of optimal co-scheduling on chip multiprocessors. In: Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT), pp. 220–229 (October 2008)
7. Kumar, R., Tullsen, D.: Compiling for instruction cache performance on a multithreaded architecture. In: Proceedings of the International Symposium on Microarchitecture, pp. 419–429 (2002)
8. Lattner, C., Adve, V.: Automatic pool allocation: improving perfor- mance by controlling data structure layout in the heap. In: Proceedings of the ACM SIGPLAN Conference On Programming Language Design and Implementation (2005)
9. Nikolopoulos, D.: Code and data transformations for improving shared cache performance on SMT processors. In: Proceedings of the International Symposium on High Performance Computing, pp. 54–69 (2003)
10. Sarkar, S., Tullsen, D.: Compiler techniques for reducing data cache miss rate on a multithreaded architecture. In: Proceedings of The HiPEAC International Conference on High Performance Embedded Architectures and Compilation, pp. 353–368 (2008)
11. Sarkar, V.: Analysis and optimization of explicitly parallel programs using the parallel program graph representation (1997)
12. Shen, X., Gao, Y., Ding, C., Archambault, R.: Lightweight reference affinity analysis. In: Proceedings of the 19th ACM International Conference on Supercomputing, Cambridge, MA (June 2005)
13. Tam, D., Azimi, R., Stumm, M.: Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. SIGOPS Oper. Syst. Rev. 41(3), 47–58 (2007)
14. Zhang, E.Z., Jiang, Y., Shen, X.: Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs? In: PPoPP 2010: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 203–212 (2010)
15. Zhao, P., Cui, S., Gao, Y., Silvera, R., Amaral, J.N.: Forma: A framework for safe automatic array regrouping. ACM Transactions on Programming Languages and Systems (2) (2007)
16. Zhong, Y., Orlovich, M., Shen, X., Ding, C.: Array regrouping and structure splitting using whole-program reference affinity. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 255–266 (June 2004)

Sublimation: Expanding Data Structures to Enable Data Instance Specific Optimizations

Harmen L.A. van der Spek and Harry A.G. Wijshoff

Leiden University, LIACS, Niels Bohrweg 1, 2333 CA Leiden, The Netherlands

Abstract. Knowledge of specific properties of input data should influence the compilation process, as different characteristics of input data might have different optimal solutions. However, in many applications, it is far from obvious how this can be achieved, as irregularity in program code prevents many optimizations from being applied. Therefore, we propose a two-phase compilation system, which in the first phase analyzes the code and transforms it into a regular intermediate code using a technique we call sublimation. Sublimation is a process in which different access functions in code are remapped using a common, injective access function. The resulting, regular intermediate code is compiled in a second phase, when the actual data input set is known. This allows for optimizations that compile the regular intermediate into a new code that uses data structures especially tailored to the input data provided. We evaluate this compilation chain using three sparse matrix kernels and show that our data instance specific optimization can provide considerable speedups.

1 Introduction

For practical reasons, data structure selection and its actual mapping onto hardware do not necessarily match with the logical structure of a problem. Often, a single solution is implemented that is supposed to fit a wide range of problem instances. This, however, is very unlikely to yield optimal performance on the entire range of input sets. Each problem instance has its own characteristics, which are not taken into account by the "one solution fits all" implementation.

An obvious example is sparse matrix representation. Many different storage mechanisms have been proposed, each of which has its applications for specific instances of a more general problem, for instance, solving a sparse system of linear equations. This diversity in implementations, each of which is tailored to specific properties of a problem at hand, is a nightmare from a code management point of view. For each different class of problems, new code should be written. In practice, sub-optimal performance will be the result of this trade-off between implementation effort and efficiency.

Many proposals to optimize irregular and sparse applications have been made recently. Most, if not all, of these proposal rely heavily on extensive run-time analysis [10,5,9]. Although these techniques themselves might work very well,

K. Cooper, J. Mellor-Crummey, and V. Sarkar (Eds.): LCPC 2010, LNCS 6548, pp. 106–120, 2011.

they suffer from the fact that the overhead incurred by these run-time mechanisms must be amortized over multiple runs or iterations. To optimize specifically for each input set, a code using these run-time mechanisms is generally not feasible. So in order to allow data instance specific optimizations, other mechanisms are needed.

In this paper, we propose a very ambitious and aggressive compilation trajectory to overcome the issues described above and to allow for data instance specific optimization. Basically, our approach relies on the fact that for large scale simulation codes, like circuit simulation, structural mechanics, computational fluid dynamics, etc., the reference codes which are used have a long life cycle. Although these codes employ libraries to exploit specific architectural features, in general, it is a major and error-prone task to rewrite these codes and optimize them for a specific problem at hand.

In our approach, these codes will be aggressively analyzed, both at compile and run-time, and they will be automatically expanded into a form which allows compiler optimizations to be much more effective. Also, other compiler optimizations are enabled, which optimize these codes for specific problem instances. We envision that this whole transformation chain is split into two parts. The first part consists of compile and run-time analysis, combined with iteration space expansion and will happen at the vendor/code owner's site. So even before the code is shipped to the customer, the code is prepared, instrumented and expanded. Because this is done on a per customer basis, the amount of time it takes to prepare this code can be considerable. In this phase, representative data sets can be used to identify access patterns that are likely to be useful for restructuring later. In the second phase, after the code is installed at the customer's site, "back-end" compiler optimizations take place, which not only optimize for specific architectural features of the computing platform, but also take the specific characteristics of the problem to be solved into account. In the second phase, the overhead incurred by these compiler optimizations should be minimized.

One might wonder why the code resulting from the first phase cannot be directly provided by the code owner. However, if this approach would be taken, the code owner must maintain multiple versions of the code, as different problem domains show different data usage patterns. A major advantage of this approach is that the code owner only has to maintain one reference code and not multiple versions of his codes for the different customers. Note that the different versions of the code, which otherwise have to be maintained, differ in an essential way, in the sense that they are based on different data structure choices. Therefore, this effort would go far beyond common practice, in which the code owners just have to maintain differently configured versions of its code base for the different customers.

In this paper, we mainly describe the first phase of this compilation chain, where we specifically describe the way the code could potentially be expanded. This expansion is based on the notion of *sublimation*, in which data structures used are being embedded into *enveloping data structures*, such that proper data dependence analysis can be performed in the second phase. In fact, sublimation converts indirect addressed based loops into loop structures that do not contain indirect addressing and therefore can be analyzed by classical methods. In order to enable this sublimation as a preparatory phase, the pointer-based codes are being transformed into (indirect addressed) array-based codes using pool allocation and structure splitting. Although this in itself is a challenging problem, recent results show that this can be solved using compile-time techniques [6,3,13,14]. This phase will be extended by automatically generated compiler instrumentation (for tracing memory pool accesses [13]), which uses run-time information to identify regions in which indirections are referring to distinct objects. This information is used in the expansion phase to further eliminate indirect addressing, yielding indirection free, array-based code which can therefore can be analyzed by standard compiler optimizations. This representation of the code is then analyzed at the customer's site together with the characteristics of the specific problem instance to be solved to obtain a problem/architectural optimized implementation of the code. This last phase is based on compilation techniques that optimize dense codes together with a non-zero pattern specification into an optimized sparse implementation [2].

Within the scope of this paper, we will not be able to describe all the specifics of each of these phases, but we will mainly concentrate on the sublimation/ expansion process. As already described, the overall compilation chain is very ambitious, but it is our belief that such an approach is needed in the future to tackle the problems faced when implementing data specific optimizations.

This paper is organized as follows. Using an array-based representation of a pointer-based application, which can be obtained by applying the transformations described by Van der Spek et al. [13,14], *sublimation*, (the technique to embed data structures into an enveloping data structure) is explained in Section 2. We have applied sublimation to three sparse matrix kernels, which are *sparse matrix vector multiplication, Jacobi iteration* and a *direct solver* using an LU-factorized matrix. The derivation using sublimation is done in Section 3. The kernels have been optimized using a variety of different input matrices and the results are presented in Section 4, which also evaluates and describes the overhead of the compilation chain in the second phase. Section 5 concludes our paper.

2 Sublimation

The basic idea behind sublimation is to transform a code using indirect addressing into a dense code, thereby eliminating the occurrence of indirect addressing entirely. So for example, we want to transform a code using compressed row format-based to represent sparse matrices into a dense code using compiler transformations. In this section, we describe how this could be achieved by first restructuring access to arrays to conform to a common access pattern. By transferring the indirection from the loop body to the header, followed by an expansion of the iteration space, a regular intermediate code is obtained.

Prototypes of these techniques have been implemented [15,11]. These address either simple indirect addressing based loops or nicely nested pointer traversal loops. These implementations have not yet been extended to use pool-allocated data structures. For the sake of a concise description, we will describe these transformations using generic code samples.

Note that sublimation is part of the first phase of the compilation process. Therefore, the overhead needed when implementing the transformations as described in this section is only incurred once. In the second phase, the data instance specific optimizations will have to be much more efficient.

The example codes all use an array-based representation. However, Van der Spek et al. have recently shown that type-homogeneous data structures that are pool-allocated can be rewritten using an indirection array-based style [14]. This paper also describes a method to detect invariant loop traversal patterns which can be replaced by counted loop structures. This invariance is used to obtain a countable iteration space for data dependent loop structures whose conditions are proved to be constant with respect to a specific program region. In the remainder of this paper, we express our transformations using the indirection array-based version of the pointer-based codes, and thus assume that pointers and data dependent *while* loops have been eliminated using the techniques in the paper cited above.

The analysis and code generation presented in this section which eliminate indirection consists of three (sub)phases. First, data access functions are determined, which ensures that data access patterns in the loop body are aligned. In the second phase, the indirection is eliminated in the loop body and transferred to the loop header. Subsequently, in the third phase, the irregular iteration space is embedded into a containing iteration space, with known loop bounds.

2.1 Data Access Restructuring

Consider the following loop structure:

```
for (i=0; i<n; i++ ) {
   ...A[i]...
   ...B[C[i]]...
}
```

In this loop, two arrays are accessed using the iteration counter i and the index array C, which basically is a function of the iteration counters. An obvious choice to restructure data in this loop is to restructure array B in order to follow the same access pattern as A.

```
for (i=0; i<n; i++ ) {
   ...A[i]...
   ...B'[i]...
}
```

Where B' is defined as: $\forall i, 0 \leq i \leq n : B'[i] = B[C[i]]$. While this *gather* operation is the most obvious choice for restructuring, we will focus on another choice, which is sublimation of array A, by changing its regular access pattern into the same irregular access pattern of B. In this case, the code is transformed into:

```
for (i=0; i<n; i++ ) {
   ...A'[C[i]]...
   ...B[C[i]]...
}
```

In the resulting code, both the arrays are accessed using the same access pattern. Of course, the restructuring of the arrays must meet certain criteria in order to be valid. All values used in the original array must be preserved in the target array and in case an array is written to, data should not be duplicated.

Generally speaking, let $f(I)$ and $g(I)$ denote the new and original access functions, respectively. An access function is defined as a function that maps a point from the iteration space into an integer offset. In the example above, $f(i) = C[i]$ (access using index array) and $g(i) = i$ (the iteration counter). The following condition must hold if $f(I)$ is used to access data that is being read: $\forall I, J : I \neq J \rightarrow f(I) \neq f(J)$ (thus, f is an injective function). If $f(I)$ is used to index an array that is written to, then the injectivity of $f(I)$ is not sufficient. In that case, the original access function must also be injective, in order to avoid duplication of data originating from the same location in memory. For writes, the following condition must hold: $\forall I, J : I \neq J \rightarrow (f(I) \neq f(J) \wedge g(I) \neq g(J))$ (both f and g are injective functions).

2.2 Identifying Injective Functions in Code

The notion of sublimation that was explained in the previous section was based on a simple regular access pattern combined with an indirect access pattern. In general, the simple regular access pattern can be extended to any injective access pattern derived from loop counters. Such patterns naturally arise in counted loop structures. From such loop structures, the iteration space and a total ordering of its traversal can be determined at compile-time. On this total order T, a *bijective* mapping $h : T \rightarrow \mathcal{Z}^n$ to the iteration space can be defined. Using this bijective mapping, all of the techniques described above are generalized to multi-dimensional iteration spaces.

Within a loop, multiple access patterns can be identified, each of which is a potential candidate to be used as the access function to which other access functions adapt. This adaptation is what the notion of sublimation refers to. While not all access functions are injective, all access functions can be made injective with respect to the containing loop structure by expanding their dimensionality. We will clarify this using the example shown in Figure 1a. This code computes a sparse matrix vector product. The addressing expressions that can be derived from this code sample are: i, k and $ColIdx[k]$. Of these, $colIdx[k]$ is injective with respect to every single iteration of the inner loop and k is injective across the entire iteration space. This can be specified in a directive (as done in our example) or we can choose to speculate on this property and check injectiveness at run-time.

```
#pragma INJECTIVE(k)
for (i=0; i<n; i++ ) {
#pragma INJECTIVE(colIdx[k])
   for (k=start[i]; k<start[i+1]; k++ ) {
     result[i] += M[k] * right[ colIdx[k] ];
   }
}
```

(a)

```
for (i=0; i<n; i++ ) {
   for (k=start[i]; k<start[i+1]; k++ ) {
     result[i] += M'[i,colIdx[k]] *
       right[colIdx[k]];
   }
}
```

(b)

Fig. 1. Example of injective access functions

In this example, $colIdx[k]$ is not injective across the entire iteration space. By extending the access function using one of the dimensions of the iteration space, a new, injective function is obtained. In our example, the new injective function is $f(i, k) = (i, colIdx[k])$. The resulting loop structure that is obtained (the *#pragma* statements are left out) is shown in Figure 1b. Note that this resulting code is *never* directly executed. It only serves as an intermediate code. In this code, `M'[i, colIdx[k]] = M[k]`.

2.3 Eliminating Indirect Addressing in the Loop Body

Irregularity can be present both in the loop header as well as in the loop body. With irregularity, we mean any property that cannot be statically determined. This includes, data dependent loop conditions and unpredictable memory reference patterns [12]. In our example, the lower and upper bound of the inner loop are data dependent and thus irregular. The inner loop is a counted loop, which in itself defines an injective function. This injective property can be used to transfer the irregular access that still exists within the inner loops to the loop header. Let the original iteration space be Δ and let h be an irregular access function within the loop. Then the irregular access function can be transferred to the loop header as follows:

```
for ( I ∈ Δ ) {
   ...A'[h(I)]...
}
```

is transformed to:

```
for ( I' ∈ h(Δ) ) {
   ...A'[I']...
}
```

Applied to our example, the irregular access function defined by the index array expression $colIdx[k]$ is dependent on the inner loop counter, and the access function can be transferred to the loop bounds of the inner loop:

```
for (i=0; i<n; i++ ) {
   for ( q ∈ { colIndex[start[i]], colIndex[start[i]+1],
            ..., colIndex[start[i+1]-1] } ) {
     result[i] += M'[i, q] * right[q] ];
   }
}
```

The irregularity has now been transferred from the loop body to the loop header.

2.4 Expanding the Iteration Space

The resulting loop header is still data dependent and thus irregular. This form of irregularity can be eliminated by expanding the iteration space to a space that encompasses the entire iteration space using a fixed interval that is large enough to contain all of the elements of the original iteration space. The fact that this can be done, relies on the property that statements of the following form do not have any effect:

(1) $A = A + 0$
(2) $A = A * C$, if A is zero.

Therefore, any extraneously executed statements will not change the semantics of programs. Such statements naturally occur in numerically intensive applications, and therefore this method is suitable for large scale simulation codes as mentioned in the introduction.

In general, let Δ be the iteration space after transferring the indirect access to the loop header as described in the previous section. Let Ω be the new iteration space that extends Δ ($\Omega \supseteq \Delta$). The injective function g is the extended function of the original access function f. If A is the result of sublimation on an array, the array A' used in the expanded iteration space is defined as follows: $\forall I \in \Omega$: $A'[g(I)] = A[f(I)]$ if $I \in \Delta, 0$ if $I \in \Omega \backslash \Delta$.

Applied to our example code, the iteration space can be extended by transforming the inner loop to a counted loop with a range that covers all possible values. The new access function is still (i, q), but q covers a larger range of values.

```
for (i=0; i<n; i++ ) {
   for ( q=0; q<MAX_INT; q++ ) {
     result[i] += M’’[i, q] * right[q] ];
   }
}
```

Note that this code is an intermediate code, and therefore is *never executed.* The loop bounds here are taken very conservatively. Using directives or results from other analyses, a smaller iteration space could be used. Eventually, this intermediate is recompiled and proper loop bounds are generated. Note that the definition of M' needs to be changed, in order to specify the zero elements:

```
$\forall(i,q)$ :  if $q \in \{colIndex[start[i]], \ldots, colIndex[start[i+1]-1]\}$
                    $M''[i,q] = M'[i,q]$
                  else
                    $M''[i,q] = 0$
```

2.5 Restructuring in the Application Context

To summarize, the iteration space and data redefinitions are depicted in pseudocode sample in Figure 2. It should be noted again that this code should not be viewed as code that will be executed. The extended iteration space Ω, for example, might be huge (read: unfeasible to traverse) and the code samples above either serve as *semantic definitions* or are compiled to a data-instance specific optimized code.

```
                       ⎧...                                   | /* Loop kernel */
                       ⎪/* Initialization of zero space */    | for (I ∈ Ω) {
...                    ⎪for (I ∈ Ω) {                         |    ...A'[g(I)]...
/* Loop kernel */      ⎪   A'[g(I)] = 0;                      | }
for (I ∈ Δ) {          ⎪}                                     | /* Write-back stage */
   ...A[f(I)]...     → ⎨/* Sublimation of array A */          | for (I ∈ Δ) {
}                      ⎪for (I ∈ Δ) {                         |    A[f(I)] = A'[g(I)];
...                    ⎪   A'[g(I)] = A[f(I)];                | }
                       ⎪}                                     | ...
                       ⎩
```

Fig. 2. Summary of sublimation and the embedding in a regular, expanded iteration space

3 Application of Sublimation to Pointer-Based Matrix Kernels

We have applied sublimation on three sparse matrix kernels, that use orthogonally linked lists to store the matrix data. The three kernels considered are *sparse matrix multiplication, sparse Jacobi iteration* and a *direct sparse solver* using an LU-factorized matrix, and are taken from SPARK00 [12]. These kernels have been transformed to an array-based representation using the transformations described by Van der Spek et al. [14]. On each of these kernels, sublimation is applied, causing the array representation of the matrix values to be extended. Although all programs use the orthogonally linked list representation for sparse matrices, the difference in traversal patterns may lead to completely different implementations for different kernels. In this section, the transformation of each of the three kernels is described and characteristic features of each of these kernels are explained.

3.1 Sparse Matrix Vector Multiply

Sparse matrix vector multiplication is an important kernel in many applications. Figure 3 shows the code samples while the code is being transformed. We start with the array-based code resulting from the pointer to array conversion. In this code, the variable k is increased in the inner loop body. Therefore, k defines an injective access pattern. The values that k takes for each inner loop can be determined during the pointer to array conversion and results in a loop structure where the bounds of k are defined in the loop header.

We pick the access pattern $colIdx[k]$ to be used for sublimation and redefine M accordingly. As explained in Section 2.2, access functions that are not injective can be made injective by extending them using a dimension from the iteration space. In this case, it is extended using the iteration counter i. Injectivity must be determined either by using directives or by run-time analysis.

Subsequently, the indirection is moved to the loop header, after which the iteration space is expanded, using the property that the access function is injective with respect to the inner loop. The resulting dense intermediate code will be optimized later when the actual data set has been loaded at run-time.

```
k=0;
for (i=0; i<n; i++ ) {
   for (j=0; j<limj[i]; j++ ) {
     result[i] += M[k] * right[colIdx[k]];
     k++;
   }
}
```

(a) Pointers transformed to arrays

```
for (i=0; i<n; i++) {
   for (k=start[i]; k<start[i+1]; k++) {
     result[i] += M[k] * right[colIdx[k]];
   }
}
```

(b) Injective inner loop

```
for (i=0; i<n; i++) {
   for (k=start[i]; k<start[i+1]; k++) {
     result[i] += M'[i,colIdx[k]] *
             right[colIdx[k]];
   }
}
```

(c) Sublimation of M to M'

```
for (i=0; i<n; i++) {
   for (q=0; q<INT_MAX; q++) {
     result[i] += M''[i,q] * right[q];
   }
}
```

(d) Transferred and expanded loop bounds

Fig. 3. Starting point for code analysis and sublimation of sparse matrix vector multiply

3.2 Jacobi Iteration

Figure 4 shows the code for Jacobi iteration while it is being transformed. Jacobi iteration has the interesting property that it consists of two inner loops. These loops originate from the pointer-based code, where elements before and after the diagonal are traversed in separate loops, while storing the diagonal entry to a temporary variable. In the array-based version shown here, the diagonal entries are stored in the array *diag*. Therefore, the loops can now be merged into a single loop. Similar as in the sparse matrix multiplication code, k is injective and can be transferred to the inner loop by storing the lower and upper bounds per execution of the inner loop.

The sublimation process and iteration space expansion follow the same pattern as previously. It should be noted, though, that the definition of M'' will not include

```
k = 0;
for (h=0; h<hlim; h++) {
   for (i=0; i<ilim[h]; i++) {
     x_2[h] -= M[k] * x_1[colIdx[k]];
     k++;
   }

   for (j=0; j<jlim[h]; j++ ) {
     x_2[h] -= M[k] * x_1[colIdx[k]];
     k++;
   }
   x_2[h] = x_2[h] / diag[h];
}
```

(a) Pointers transformed to arrays

```
for (h=0; h<hlim; h++) {
   for (k=start[h]; k<start[h+1]; k++) {
     x_2[h] -= M[k] * x_1[colIdx[k]];
   }
   x_2[h] = x_2[h] / diag[h];
}
```

(b) Fused injective inner loop

```
for (h=0; h<hlim; h++) {
   for (k=start[h]; k<start[h+1]; k++) {
     x_2[h] -= M'[h,colIdx[k]]*x_1[colIdx[k]];
   }
   x_2[h] = x_2[h] / diag[h];
}
```

(c) Sublimation of M to M'

```
for (h=0; h<hlim; h++) {
   for (q=0; q<INT_MAX; q++ ) {
     x_2[h] -= M''[h,q]*x_1[q];
   }
   x_2[h] = x_2[h] / diag[h];
}
```

(d) Transferred and expanded loop bounds

Fig. 4. Starting point for code analysis and sublimation of Jacobi iteration

entries from the main diagonal, as these are separately stored in the array *diag*. While looking similar to sparse matrix multiplication, the missing diagonal might lead to different optimization choices in the optimization back-end.

3.3 Direct Solver

The results of the transformation steps on the direct solver are shown in Figure 5. Characteristic for the kernel is the use of two different index arrays, one using row indices (*rowIdx*) upon column-wise traversal of the matrix and one using column indices (*colIdx*) upon row-wise traversal. The lower triangle of the matrix is traversed column-wise (first loop), the upper triangle is traversed row-wise (second loop). Indirect access is both present in reads of arrays as well as in writes.

```
I = 1; k = 0;
for (g=0; g<glim; g++) {
 Temp = Intermediate[I];
 Temp = Temp / pivot[g];
 Intermediate[I] = Temp;
 for (m=0; m<mlim[i]; m++) {
   Intermediate[rowIdx[k]] -= Temp * M[k];
   k++;
 }
 I++;
}

I2 = Size; k2 = 0;
for (h=0; h<hlim; h++) {
 Temp = Intermediate[I2];
 for (n=0; n<nlim[h]; n++) {
   Temp -= M2[k2]*Intermediate[colIdx[k2]];
   k2++;
 }
 Intermediate[I2] = Temp;
 I2--;
}
```

(a) Pointers transformed to arrays

```
for (g=0; g<glim; g++) {
 Temp = Intermediate[g+1];
 Temp = Temp / pivot[g];
 Intermediate[g+1] = Temp;
 for( k=start[g]; k<start[g+1]; k++ ) {
   Intermediate[rowIdx[k]] -= Temp * M[k];
 }
}

I2 = Size;
for (h=0; h<hlim; h++) {
 Temp = Intermediate[I2];
 for( k2=start2[h]; k2<start2[h+1]; k2++) {
   Temp -= M2[k2]*Intermediate[colIdx[k2]];
 }
 Intermediate[I2] = Temp;
 I2--;
}
```

(b) Injective inner loops

```
for (g=0; g<glim; g++) {
 Temp = Intermediate[g+1];
 Temp = Temp / pivot[g];
 Intermediate[g+1] = Temp;
 for( k=start[g]; k<start[g+1]; k++ ) {
   Intermediate[rowIdx[k]] -= Temp *
           M’[g,rowIdx[k]];
 }
}

I2 = Size;
for (h=0; h<hlim; h++) {
 Temp = Intermediate[I2];
 for( k2=start2[h]; k2<start2[h+1]; k2++) {
   Temp -= M2’[h,colIdx[k2]] *
           Intermediate[colIdx[k2]];
 }
 Intermediate[I2] = Temp;
 I2--;
}
```

(c) Sublimation of arrays M and M2

```
for (g=0; g<glim; g++) {
 Temp = Intermediate[g+1];
 Temp = Temp / pivot[g];
 Intermediate[g+1] = Temp;
 for( q=0; q<INT_MAX; q++ ) {
   Intermediate[q] -= Temp * M’[g,q];
 }
}

I2 = Size;
for (h=0; h<hlim; h++) {
 Temp = Intermediate[I2];
 for( r=0; r<INT_MAX; r++) {
   Temp -= M2’’[h,r] * Intermediate[r];
 }
 Intermediate[I2] = Temp;
 I2--;
}
```

(d) Transferred and expanded loop bounds

Fig. 5. Starting point for code analysis and sublimation of a direct solver

In the code, the data has been segmented in two different arrays, M and $M2$, representing the lower triangle and upper triangle, respectively. k and $k2$ define an injective access pattern, and are put in the inner loops with corresponding lower and upper loop bounds. In the first loop, the variable i is an induction variable and is replaced by $g + 1$. In the first loop, M is sublimated using the access function $rowIdx[k]$, while in the second loop $M2$ is sublimated using the access pattern $colIdx[k2]$. After transferring indirect addressing and expanding the iteration space, the intermediate dense code specifies two dense matrices, each representing the lower and upper triangle of the input matrix. Similar as in Jacobi iteration, the main diagonal is not part of the M'' or $M2''$, but is stored separately in the array *pivot*.

4 Experiments

The sparse matrix kernels, on which sublimation has been applied, have been optimized and executed using a variety of matrices from Davis's *University of Florida Sparse Matrix Collection* [4]. Not all data sets that are used in the sparse matrix multiplication are used in Jacobi iteration because input matrices for Jacobi iteration cannot contain zero entries on the diagonal. For the direct solver, the matrices have been LU-factorized prior to running the program. Matrices taking excessive time to factorize have not been used.

The dense intermediate codes derived in the previous section are compiled to a data set-specific implementation of the kernel. In our experiments, we use the MT1 compiler as our back-end [2]. MT1 compiles a dense specification together with the specification of the non-zero patterns and produces a data set-specific optimized implementation. The access patterns can be obtained by instrumenting the code obtained by the methods proposed by Van der Spek et al. [14]. All experiments have been run on Intel Core 2 Duo 2.33GHz system with 2GiB of main memory, running Mac OS X 10.6.2. The programs are compiled using GCC 4.2.1 using the options '-O3 -ftree-vectorize'.

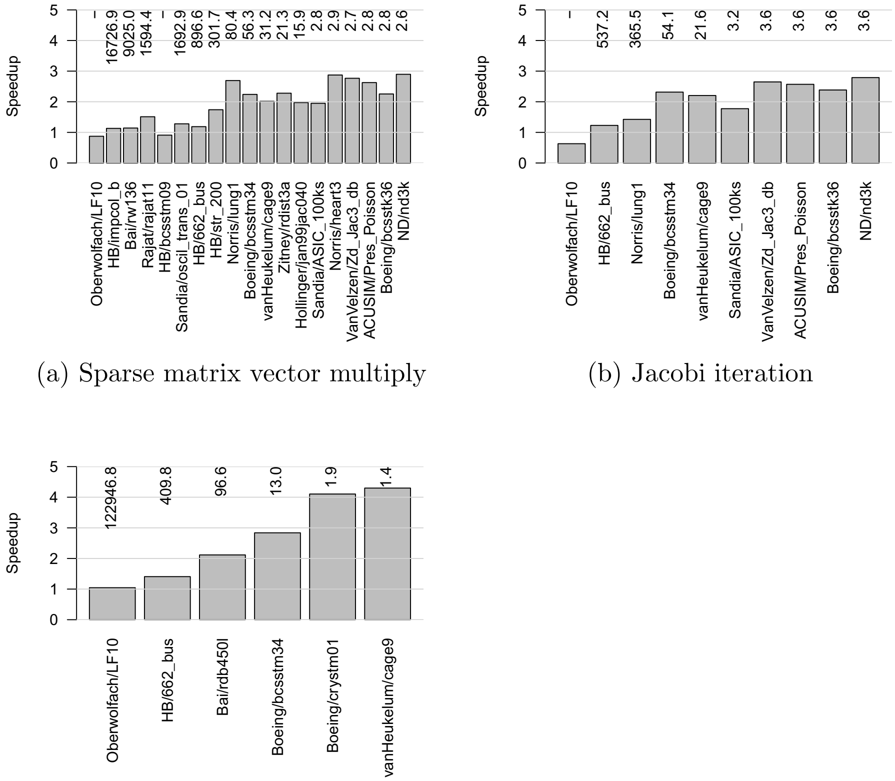
First, we will present the results of the recompiled kernel that uses the dense code as input and the access patterns as defined by the specific input matrix. Next, the overhead of the dynamic code-generation is assessed. Note that in the first phase of our overall compilation process, the overhead can be substantial. However, when transforming the intermediate (dense) code into a data instance specific code, which belongs to the second phase of the overall compilation process, the overhead should be minimized.

4.1 Results on Sparse Matrix Kernels

As soon as the data is loaded from the input file and it is determined that the access pattern of the kernel will remain static (as described by Van der Spek et al. [14]), the code resulting from sublimation together with a definition of the access patterns is compiled using the MT1 compiler. The overhead is measured separately, as this is constant for each kernel and data set combination. Including the overhead in the kernel execution time yields arbitrary results. In many cases, iterative numerical solution methods require multiple executions of a sparse matrix kernel with a right hand side vector which converges to the correct solution. In each of these subsequent iterations, the underlying structure of the matrix does not change. So, by increasing the number of iterations, the overhead per iteration can be made as small as desired. Therefore, we will show more details about the overhead in the next section, and here we will focus on the performance of the kernel itself.

Figure 6 shows the speedups obtained by running the optimized kernels. In these figures, the data sets are sorted by increasing size from left to right. For each of these kernels, we can observe that the restructuring methods are most suitable for the larger data sets, while the optimizations do not negatively effect performance for the small data sets, in general. The bars are annotated with the break-even points (×1000 iterations).

As we can see, the speedups can be substantial. Although the data structure layout for the original kernels have been optimized and take into account the

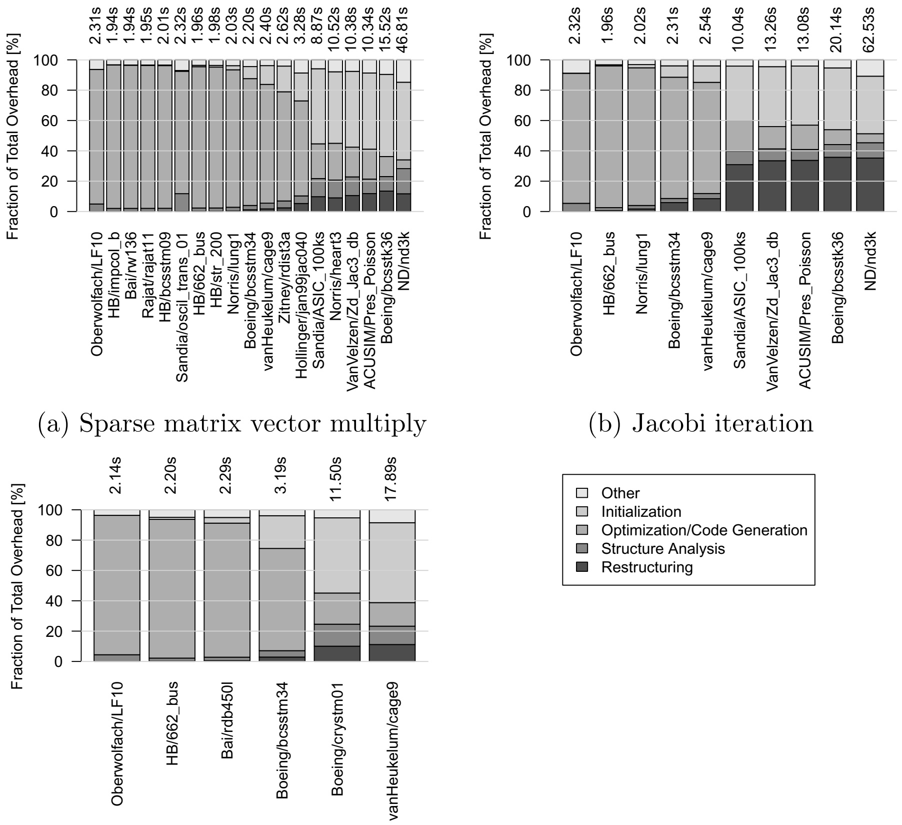


(a) Sparse matrix vector multiply

(b) Jacobi iteration

(c) Direct solver

Fig. 6. Speedups obtained on the data set specific optimized kernels. The numbers show the break-even point (×1000 iterations).



(a) Sparse matrix vector multiply

(b) Jacobi iteration

(c) Direct solver

(d) Legend

Fig. 7. Contribution of different phases to the overhead of run-time compilation of the dense intermediate with data set dependent access pattern to a data set specific implementation. The time plotted at the top of each bar is the total time spent in run-time compilation.

sparsity of the data, they have not been optimized for specific non-zero structure information. As a result of our transformations, these original codes are compared to kernels which are specifically optimized for a particular input matrix.

4.2 Overhead

While the application of sublimation is a compile-time analysis and transformation, which generates the dense intermediate code, the data set specific compilation is deferred to run-time. At run-time, the final re-targeting of the code using input data set dependent access patterns is performed before the kernel is executed.

Figure 7 shows the overhead for the run-time compilation for the kernels with their different input data sets. For the smaller data sets, it is clear that the majority of the time is spent in optimization and code generation (this is the generation of a shared library). This is caused by the fact that this is a fairly constant factor, as this consists of compiling the code emitted by MT1 (using for example GCC) into a binary. More complicated code could be emitted for larger data sets, but this only results in a relatively small increase in compilation time (observed times for this phase range from $0.27s - 4.37s$). For the larger data sets, restructuring, structure analysis and initialization (loading data using the restructured data layout) are the dominating factors.

5 Conclusions

In this paper, we proposed a two-phase approach to the optimization of applications. The first phase consists of compile-time analysis in which data used in the application is embedded in enveloping data structures. This is driven by a technique we call *sublimation*, which forces data to be laid out in memory using a common and appropriate access function. The resulting code is an intermediate code which is not directly executed. In the second phase this intermediate is compiled together with actual data and an instance specific optimized code is generated.

Other methods often target the definition of sparse algorithms. Mateev et al. employ generic programming in C++ to provide separate APIs for algorithm specification and data access [7], A restructuring compiler is responsible for the translation from the algorithm API to the data structure API. Reordering strategies to improve the overlap in computation and communication have been proposed by Basumallik and Eigenmann in the context of OpenMP to MPI translation for irregular applications [1]. A key difference with our work is that we derive a dense intermediate code and preserve the nesting structure of loops. An intermediate code for the generation of sparse code from dense code, SIPR, has been developed by Pugh and Shpeisman [8]. Our transformations work in the opposite direction, as we transform pointer-based and sparse code into a dense intermediate.

We have described how sublimation can potentially be applied to codes including pointer-based data structures. The subsequent optimizations in the first phase are all based on array-based codes. Using three sparse matrix kernels, we have evaluated the potential of the transformation to an enveloping data structure (a dense matrix in this case) and show that considerable speedups can be achieved. The overhead involved in the restructuring during the second phase has been evaluated. For smaller data sets, the final code generation time is dominating. For large sets, restructuring and initialization of the new data structures after compilation are dominating.

While the experiments show the potential of our two-phase compilation system using sublimation, these experiments are only a limited application of these techniques. We envision applications in which computational intensive parts are compiled into an intermediate code using sublimation, which is compiled together with specific input data to enable even more aggressive optimizations to be applied.

References

1. Basumallik, A., Eigenmann, R.: Optimizing irregular shared-memory applications for distributed-memory systems. In: PPoPP 2006: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 119–128. ACM, New York (2006)
2. Bik, A.J.C., Wijshoff, H.A.G.: Automatic data structure selection and transformation for sparse matrix computations. IEEE Trans. Parallel Distrib. Syst. 7(2), 109–126 (1996)
3. Curial, S., Zhao, P., Amaral, J.N., Gao, Y., Cui, S., Silvera, R., Archambault, R.: MPADS: memory-pooling-assisted data splitting. In: ISMM 2008: Proceedings of the 7th International Symposium on Memory Management, pp. 101–110 (2008)
4. Davis, T.A.: The University of Florida sparse matrix collection. submitted to ACM Trans. on Mathematical Software, http://www.cise.ufl.edu/research/sparse/matrices
5. Kodukula, I., Pingali, K.: Data-centric transformations for locality enhancement. Int. J. Parallel Program. 29(3), 319–364 (2001)
6. Lattner, C., Adve, V.: Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005) (June 2005)
7. Mateev, N., Pingali, K., Stodghill, P., Kotlyar, V.: Next-generation generic programming and its application to sparse matrix computations. In: ICS 2000: Proc of the 14th Int. Conference on Supercomputing, pp. 88–99. ACM, New York (2000)
8. Pugh, B., Shpeisman, T.: SIPR: A new framework for generating efficient code for sparse matrix computations. In: Carter, L., Ferrante, J., Sehr, D., Chatterjee, S., Prins, J.F., Li, Z., Yew, P.-C. (eds.) LCPC 1998. LNCS, vol. 1656, pp. 213–229. Springer, Heidelberg (1999)
9. Rus, S., Rauchwerger, L., Hoeflinger, J.: Hybrid analysis: static & dynamic memory reference analysis. Int. J. Parallel Program. 31(4), 251–283 (2003)
10. Saltz, J.H., Mirchandaney, R., Crowley, K.: Run-time parallelization and scheduling of loops. IEEE Trans. Comput. 40(5), 603–612 (1991)
11. van der Spek, H.L.A., Bakker, E.M., Wijshoff, H.A.G.: A compile/run-time environment for the automatic transformation of linked list data structures. International Journal of Parallel Programming 36(6), 592–623 (2008)
12. van der Spek, H.L.A., Bakker, E.M., Wijshoff, H.A.G.: Characterizing the performance penalties induced by irregular code using pointer structures and indirection arrays on the Intel Core 2 architecture. In: CF 2009: Proceedings of the 6th ACM Conference on Computing Frontiers, pp. 221–224. ACM, New York (2009)
13. van der Spek, H.L.A., Holm, C.W.M., Wijshoff, H.A.G.: Automatic restructuring of linked data structures. In: Gao, G.R., Pollock, L.L., Cavazos, J., Li, X. (eds.) LCPC 2009. LNCS, vol. 5898, pp. 263–277. Springer, Heidelberg (2010)
14. van der Spek, H.L.A., Holm, C.W.M., Wijshoff, H.A.G.: How to unleash array optimizations on code using recursive data structures. In: ICS 2010: Proc. of the 24th ACM Int. Conf on Supercomputing, pp. 275–284. ACM, New York (2010)
15. Zhao, L., Wijshoff, H.A.G.: A case study in automatic data structure selection for optimizing sparse matrix computations. In: Proceedings of the IEEE International Workshop on Advanced Compiler Technology for High Performance and Embedded Systems (IWACT), pp. 22–55 (July 2001)

Optimizing and Auto-tuning Belief Propagation on the GPU

Scott Grauer-Gray and John Cavazos

Computer and Information Sciences
University of Delaware
Newark, DE 19716
{grauerg,cavazos}@cis.udel.edu

Abstract. A CUDA kernel will utilize high-latency local memory for storage when there are not enough registers to hold the required data or if the data is an array that is accessed using a variable index within a loop. However, accesses from local memory take longer than accesses from registers and shared memory, so it is desirable to minimize the use of local memory. This paper contains an analysis of strategies used to reduce the use of local memory in a CUDA implementation of belief propagation for stereo processing. We perform experiments using registers as well as shared memory as alternate locations for data initially placed in local memory, and then develop a hybrid implementation that allows the programmer to store an adjustable amount of data in shared, register, and local memory. We show results of running our optimized implementations on two different stereo sets and across three generations of nVidia GPUs, and introduce an auto-tuning implementation that generates an optimized belief propagation implementation on any input stereo set on any CUDA-capable GPU.

1 Introduction

Belief propagation is a general-purpose iterative algorithm used for inference on problems that utilize Bayesian networks, Markov random fields, and other graphical representations. Applications of the algorithm include free energy estimation in proteins, turbo code decoding, and satisfiability.

Sun [16] introduced belief propagation as applied to the stereo vision problem, and implementations that incorporate the algorithm generally output desirable results according to the Middlebury stereo evaluation [14]. The input to the problem consists of two images of the same scene, a reference image and a test image, with each image displaying the scene from a different perspective along the x-axis. The goal is to accurately retrieve the difference, or disparity, in location along the x-axis of the object shown in each pixel of the reference image to the same scene object in the test image. The disparity space refers to the set of possible disparity values at each pixel. The output is given as a disparity map with a disparity estimate at each pixel, and it is often visualized as a 8-bit grayscale image by multiplying the disparity estimates by an appropriate value

K. Cooper, J. Mellor-Crummey, and V. Sarkar (Eds.): LCPC 2010, LNCS 6548, pp. 121–135, 2011.

to cover the 8-bit range. The results are often used to estimate distance to an object in the scene, as objects corresponding to pixels of greater disparity are closer than objects of corresponding to pixels of lesser disparity.

For this problem, the belief propagation implementation consists of each node in a 2-D grid computing and sending messages to each of its four-connected neighbors in each iteration, where each message can be viewed as a vector containing a value corresponding to each possible disparity. The algorithm must iterate enough times for the message values to converge in order to obtain an accurate output.

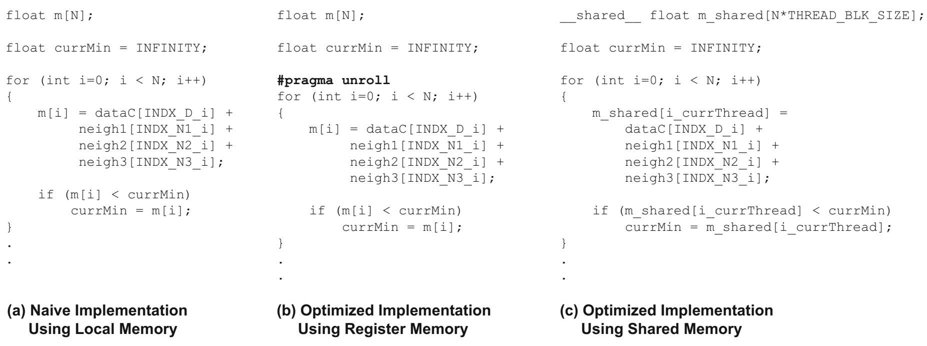
Unfortunately, the algorithm has some shortcomings. Many iterations are necessary for the message values to converge, ample storage is needed to hold the message value vectors at each pixel, and the naive computation time of each message vector is of order $\mathrm{O(n^2)}$, where n refers to the number of values in the disparity space. Some applications of stereo vision require real-time processing, and a naive belief propagation implementation on the CPU suffers from slow run-time and high storage requirements.

Felzenszwalb [6] presents methods to mitigate these downsides, presenting a hierarchical scheme to reduce the number of iterations necessary for message value convergence, a checkerboard scheme that reduces storage requirements by allowing updates to be performed 'in place' and that halves the number of messages computed and sent in each iteration, and an algorithm that generates the message vectors in O(n) time using certain discontinuity models. These methods are regularly applied in belief propagation implementations for stereo processing.

The running time of a belief propagation implementation can be reduced further by taking advantage of the parallel processing capabilities of the graphics processing unit (GPU), as each step of the algorithm can be performed in parallel on each pixel. Brunton [2], Yang [18], Grauer-Gray ([8] and [7]), Xu [19], Liang [10], and Ivanchenko [17] present implementations of belief propagation for stereo processing on the GPU. Brunton and Yang map their implementations to the graphics API, while Grauer-Gray, Xu, Liang, and Ivanchenko take advantage of the CUDA architecture.

There are challenges to optimizing a program on the CUDA architecture, as noted by Ryoo [13] and Datta [5]. An optimization that decreases the running time of one program may increase the running time of another program, or of the same program with a change of parameters, and the impact of an optimization may vary across GPUs with non-uniform architectures.

In this paper, we focus on optimizations that can be applied to a CUDA implementation of belief propagation as applied to stereo vision. We explore ways to minimize the number of accesses to high-latency local memory on the GPU. Data in local memory is frequently accessed in our initial implementation, and a decrease in the number of local memory accesses is likely to lead to a faster runtime. We present three optimized CUDA implementations of belief propagation, and go on to present an auto-tuning implementation that can optimize CUDA belief propagation across different stereo sets and GPUs.



```
float m[N];

float currMin = INFINITY;

for (int i=0; i < N; i++)
{
    m[i] = dataC[INDX_D_i] +
          neigh1[INDX_N1_i] +
          neigh2[INDX_N2_i] +
          neigh3[INDX_N3_i];

    if (m[i] < currMin)
        currMin = m[i];
}
.
.
```

(a) Naive Implementation Using Local Memory

```
float m[N];

float currMin = INFINITY;

#pragma unroll
for (int i=0; i < N; i++)
{
    m[i] = dataC[INDX_D_i] +
          neigh1[INDX_N1_i] +
          neigh2[INDX_N2_i] +
          neigh3[INDX_N3_i];

    if (m[i] < currMin)
        currMin = m[i];
}
.
.
```

(b) Optimized Implementation Using Register Memory

```
__shared__ float m_shared[N*THREAD_BLK_SIZE];

float currMin = INFINITY;

for (int i=0; i < N; i++)
{
    m_shared[i_currThread] =
        dataC[INDX_D_i] +
        neigh1[INDX_N1_i] +
        neigh2[INDX_N2_i] +
        neigh3[INDX_N3_i];

    if (m_shared[i_currThread] < currMin)
        currMin = m_shared[i_currThread];
}
.
.
```

(c) Optimized Implementation Using Shared Memory

Fig. 1. Code analogous to a portion of the dominant kernel of our CUDA belief propagation implementation; shows the naive and optimized register / shared memory implementations

2 Optimization Overview

As described in the programming guide [1], CUDA allows the GPU to be utilized as a co-processor to the CPU for general-purpose programming, processing a kernel function on multiple threads simultaneously. Each thread contains a unique ID within a 1D, 2D, or 3D thread block structure, and each thread block is part of a 1D or 2D grid structure of thread blocks. Each thread within a block is executed on the same multiprocessor and is processed as part of a 32-thread chunk known as a warp. The number of active warps per multiprocessor is bounded by a GPU-specific maximum, placing a ceiling on the parallelism in the program execution. However, the number of active warps is often limited by the number of registers or the shared memory available on each multiprocessor. The ratio of the actual number of active warps to the maximum number of active warps during the kernel execution represents the multiprocessor occupancy.

To illustrate potential optimizations, we describe the following algorithm which is analogous to a portion of the most heavily-used/dominant kernel of our belief propagation implementation:

1. Set a float variable 'currMin' to infinity.
2. For i = 1 to some N do the following:
   (a) Compute m$_i$, the sum of the values accessed from specific indices in four separate arrays (the data cost array and the message arrays from three neighbors, using the indices INDX_D_i, and INDX_N1_i, INDX_N2_i, and INDX_N3_i, respectively) in global memory, and store the value to a specific form of storage (local, shared, or register memory).
   (b) Check if m$_i$ is less than the value of 'currMin'; if so, set 'currMin' to m$_i$.

The set of m$_i$ values are accessed, manipulated, and compared with each other in future operations, so it makes sense to store them in a structure such as an array that allows for easy access via index value.

Our initial implementation causes the m_i values to be stored in an array which is local to the thread, resulting in the utilization of local memory on the GPU. Unfortunately, accesses to data in local memory are slow compared to accesses to data in registers or shared memory. We go on to generate optimized implementations where local memory accesses are converted to register or shared memory accesses. Code corresponding to each implementation is displayed in Figure 1. In the register memory implementation, the loop is completely unrolled via the '#pragma unroll' directive; this changes the implementation such that the array is no longer indexed via a variable value within a loop and is no longer automatically stored in local memory. In the shared memory implementation, the array m_shared is shared across every thread in a thread block of size THREAD_BLK_SIZE; the index i_currThread in the code is unique to each thread in the block; it corresponds to the location of the m_i value stored in the array m_shared for the thread.

Intuitively, one would expect the optimized implementations to be faster than the initial implementation since accesses to high-latency local memory in the initial implementation are replaced with accesses to low-latency registers or shared memory in the optimized implementations. However, the utilization of a greater number of registers per thread in the register implementation and of shared memory in the shared memory implementation limits the number of thread warps which can be processed in parallel, decreasing the multiprocessor occupancy and possibly adversely affecting the running time. The only way to truly determine the effect of these optimizations is to perform experiments which involve comparing the results of the different implementations; we perform such experiments using these optimizations and describe the results in Section 5.

3 CUDA Belief Propagation

In this section, we present a initial CUDA implementation of belief propagation; the implementation utilizes the speed-ups described by Felzenszwalb [6] and consists of the following steps:

1. Calculate the data cost for each pixel at each disparity in the disparity space at the bottom level of the hierarchy.
2. Iteratively calculate the data costs at each succeeding level of the hierarchy.
3. For each level in the hierarchy (starting from top):
   (a) For each pixel in the current 'checkerboard' set, compute the message to send to its four-connected neighbors in the alternate set using the current message values and data cost. Repeat for i iterations, alternating between the two checkerboard sets.
   (b) If not at the bottom level of the hierarchy, copy the message values at each pixel to a 2 X 2 block of corresponding pixels in the succeeding level of the hierarchy.
4. Retrieve the disparity estimate at each pixel using the current message values and data costs, with the output corresponding to the disparity that minimizes the sum of the current message values and data cost at the pixel. The disparity estimates across every pixel represent the output disparity map.

Grauer-Gray [8] showed that each of the steps of the algorithm can be performed in parallel using the CUDA architecture, and the resulting disparity map is obtained more quickly using a CUDA implementation as compared to a sequential CPU implementation. However, that work does not discuss optimizations which can be applied to decrease the running time of the CUDA implementation. In this paper, we present optimizations that can be utilized to reduce the running time of a CUDA implementation without affecting the output disparity map.

In our experiments, we first smooth the images using a Gaussian filter of sigma value 1.0, then run belief propagation using 5 hierarchical levels, 10 BP iterations per level, and a disparity space ranging from 0 to 14 in increments of 1. The truncated linear model is used for the data and discontinuity costs, with a maximum value of 15.0 and 1.7 for the data and discontinuity cost, respectively. The relative weight of the data cost as compared to the discontinuity cost is held at .07.

Our initial CUDA experiments utilize the nVidia GTX 285 GPU and are compiled using CUDA toolkit 3.1. The GTX 285 contains 30 multiprocessors, each containing 8 processors for a total of 240 parallel processors, with 16392 registers and 16 KB shared memory available in each multiprocessor. The thread block dimensions are set to 32 X 4, with 32 corresponding to the width and 4 corresponding to the height.

We begin with a naive CUDA belief propagation implementation, using a separate kernel for each step/sub-step of the algorithm, and run our implementation on the 384 X 288 Tsukuba stereo set shown in Figure 2. This implementation runs in 47.0 ms; the resulting disparity map is shown to the right of the stereo set in the figure.

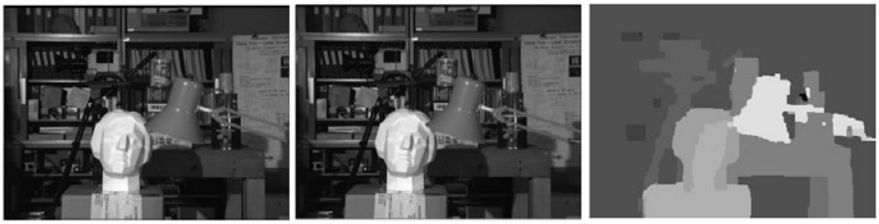


Fig. 2. Left/Middle: Images of Tsukuba stereo set; Right: Computed disparity map using our implementation

We utilize the CUDA profiler to analyze the running time of each kernel and discover that almost 70% of the running time is spent in the kernel which computes four arrays of message values, each to be received by one of the current pixel's four-connected neighbors. As a result, we focus our optimizations on this kernel, which corresponds to step 3a of the aforementioned belief propagation algorithm. Each array of message values is computed in the kernel using the following O(n) algorithm introduced by Felzenszwalb [6]:

1. For each disparity d in the disparity space, initialize the message value m$_d$ to the sum of the data cost and the current message value of each non-recipient neighbor, where the data costs and message values are retrieved from global memory, and retrieve the minimum m$_d$ value; this step corresponds to the code described in Section 2.
2. Set m$_{max}$ to the sum of the minimum m$_d$ value and T$_{data}$, the truncation value that corresponds to the maximum possible discontinuity cost.
3. Loop from d = 1 to d = max$_{disp}$, setting each m$_d$ = min(m$_{d-1}$ + 1, m$_d$), assuming that the values in the disparity space differ by 1.
4. Loop from d = max$_{disp-1}$ to d = 0, setting each m$_d$ = min(m$_{d+1}$ + 1, m$_d$), assuming that the values in the disparity space differ by 1.
5. Loop from d = 0 to d = max$_{disp}$, setting each m$_d$ = min(m$_d$, m$_{max}$), and compute the summation of the output m$_d$ values.
6. Retrieve the average message value by dividing the summation of the output m$_d$ values by the number of message values.
7. Loop from d = 0 to d = max$_{disp}$, setting m$_d$ = m$_d$ - (average message value).
8. Store the resulting message values m$_d$ for each disparity d in the appropriate location in global memory for use in the following iteration or final disparity estimation.

Inspection of the resulting PTX code and the profiling output reveals that local memory is utilized to hold the array that contains the message values that are currently being computed, causing a large number of accesses to the high-latency storage.

In the following sections, we discuss strategies to reduce or eliminate the use of local memory in this array, looking at ways to utilize registers and shared memory rather than local memory.

The number of local loads/stores given in the results correspond to totals across all invocations of this kernel, while the number of registers and the occupancy refers to the resource use in a single invocation of the kernel.

4 Experimental Methodology

We first run experiments using optimized implementations that utilize either registers or shared memory in the array that contains the message values that are currently being computed, go on to introduce a hybrid implementation that combines the usage of register, shared, and local memory in the array, and finally develop an auto-tuning implementation to generate the optimal parameters for the hybrid implementation that works across different stereo sets and GPUs.

We initially perform our experiments using the GTX 285 GPU, and then go on perform some of the same experiments using the Tesla C870 and the GTX 470, CUDA-capable GPUs with architectures that differ from the GTX 285.

5 Optimization Results: Register and Shared Memory Implementations

In this section, we describe the results of applying optimizations to the most heavily used kernel of our CUDA belief propagation implementation; these optimizations are intended to eliminate the use of local memory in the computation of the message values corresponding to each disparity, placing the data in registers or shared memory rather than local memory.

The first optimized implementation utilizes registers to store message values via the method shown in the sample code in Figure 1; this can be viewed as an application of the register promotion/scalar replacement optimization described by Cooper [4] and Callahan [3], where a value in memory is placed in a register for quick access. However, this optimization increases register pressure, which decreases the number of thread blocks which can run in parallel and may cause register spilling into local memory.

The second optimized implementation takes advantage of the shared memory present on each multiprocessor to store message values via the method shown in the right of Figure 1; the utilization of shared memory as a user-managed cache is a common CUDA optimization as described in the programming guide [1]. It is often profitable to load values from global memory into shared memory and then access/update the values from there to take advantage of the low latency associated with shared memory.

The results of running these optimizations are displayed in Table 1. The multiprocessor occupancies did decrease due to increased use of registers/shared memory in the optimized implementations. Still, the total running time on the GTX 285 decreased from 47.0 ms using the initial CUDA implementation to 24.3 ms using the register implementation and to 25.4 ms using the shared memory implementation, corresponding to speed-ups of 1.93 and 1.85, respectively, over the initial CUDA implementation.

Table 1. Running times and resource use of the initial and the optimized register and shared memory CUDA belief propagation implementations on Tsukuba stereo set on the GTX 285. The resource use data corresponds to the most heavily used kernel described in Section 3.

Storage/Loop Unroll Setting	Num local loads	Num local stores	Num regs	Occup.	Total running time	Speedup from init. CUDA imp.
Initial CUDA imp. on GTX 285	1307529	7360296	37	0.375	47.0 ms	—
Register memory imp. on GTX 285	0	0	110	0.125	24.3 ms	1.93
Shared memory imp. on GTX 285	0	0	53	0.25	25.4 ms	1.85

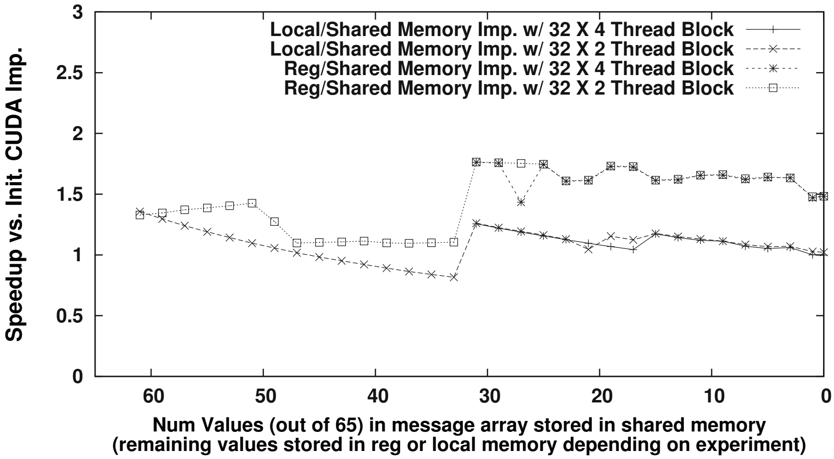


Fig. 3. Speedup of running the hybrid CUDA imp. (vs the initial CUDA imp.) on the 'Cones' stereo set with the GTX 285 using varying amounts of shared and register/local memory

6 Hybrid Implementation: Multiple Memory Modes in a Single Implementation

Next, we create a hybrid implementation which can utilize shared, register, and local memory in the array used for computation of the message values on the GPU. This allows more values to be placed in low-latency shared/register memory without resorting to high-latency local memory and allows the programmer to direct storage to local memory without dealing with the unpredictable effects of register spilling.

Our implementation allows the programmer to store x values of the array in register memory, y values in shared memory, and z values in local memory. The values in register memory may be spilled into local memory if there are not enough registers allocated to hold them or if any loops accessing them becomes too large to be unrolled, while the number of values which can be stored in shared memory is limited by the shared memory available on the multiprocessor.

We run our hybrid implementation on the GTX 285 using the quarter-sized version of the 'Cones' stereo set included as part of the 2003 Middlebury stereo sets [15]. These images measure 450 by 375 and have a disparity range from 0 to 64; the remaining parameters are the same when processing this image set as the Tsukuba set described in Section 3.

Our initial CUDA implementation described in Section 3 processes the images in 420 ms on the GTX 285; we use this result as the basis for the speed-up of our optimizations. The optimized register memory implementation described in Section 5 processes the stereo set in 300 ms, while the size of the input disparity range precludes running our optimized shared memory implementation (also described in Section 5); there is not enough shared memory available on the multiprocessor to store the entire array used for message value computation when using a disparity increment of 1 and thread block dimensions of 32 X 4.

We go on to perform four sets of experiments using the our hybrid implementation; the results of each set of experiments on the GTX 285 in terms of speed-up over the initial CUDA implementation are shown in Figure 3, while the data corresponding the best-performing configuration is displayed in Table 2.

In the first set of experiments, we hold the thread block dimensions constant at 32 X 4 and adjust the register/shared memory usage when placing the values in the array used for the computation of message values. In the second set of experiments, we set the thread block dimensions to 32 X 2 to allow greater use of shared memory when updating the message values; these dimensions allow up to 61 message values in each thread to be stored in the 16 KB shared memory available on each multiprocessor on the GTX 285 compared to 31 message values when using 32 X 4 thread blocks. In the third and fourth sets of experiments, we use the same 32 X 4 and 32 X 2 thread block dimensions and utilize shared and local memory rather than shared and register memory.

The best-performing configuration on the GTX 285 gives a speedup of 1.76 times over the initial CUDA implementation; this shows that with the right choice of parameters, our hybrid implementation can be utilized to reduce the running time of CUDA belief propagation.

Table 2. Optimal configuration and corresponding results obtained when running our CUDA hybrid implementation on the 'Cones' stereo set using the GTX 285, Tesla C870, and GTX 470 GPUs. The resource use data corresponds to the most heavily used kernel described in Section 3.

	Optimal Configuration and run-time data/results							
GPU (config.)	Thread Block Dims	Num vals in reg.-shared-loc. mem. in mess. array	Num local loads	Num local stores	Num regs used	Occup.	Total run-time	Speedup over Init. CUDA imp.
GTX 285	32 X 2	34-31-0	2209184	4418368	124	0.125	238.0 ms	1.76
Tesla C870	32 X 2	14-51-0	0	0	120	0.083	840.2 ms	1.43
GTX 470 (fav L1 cache)	32 X 2	0-7-58	4523184	4339010	61	0.333	231.2 ms	0.94
GTX 470 (fav shared mem)	32 X 2	0-63-2	87327	133654	59	0.125	189.6 ms	1.19

6.1 Hybrid Results Discussion

While investigating the results of the hybrid implementation experiments, we discovered that the running time often decreases when moving to a configuration which allows for an increase in occupancy from the previous configuration, and then the running time increases with increased usage of local memory until the parameter set allows for another increase in multiprocessor occupancy.

Based on this observation, we believe that it is possible to run experiments using a single well-generated configuration at each occupancy to retrieve the optimal configuration, rather than searching the entire optimization space. In the next section, we introduce such a auto-tuning system that uses this observation to optimize belief propagation on any CUDA-capable GPU.

7 Auto-tuning Implementation

In this section, we present our auto-tuning implementation introduced in Section 6.1. At each occupancy, we produce the configuration that maximizes the usage of shared/register memory, with the goal to minimize the number of accesses to local memory. Then, we compare the results across occupancies to retrieve the optimal configuration.

The steps of our auto-tuning implementation are as follows:

1. For the input max occupancy, determine the thread block dimensions. The width is set at 32, while the height is retrieved as follows:
   (a) Retrieve the whole number of rows of length 32 which are to be processed in parallel on each multiprocessor using the input max occupancy and the GPU-specific max number of threads which can be processed in parallel on each multiprocessor.
   (b) Calculate the maximum thread block height (1-16) that allows for this number of rows to be processed concurrently on each multiprocessor, operating under the GPU-imposed constraint that no more than 8 thread blocks can be processed concurrently on each multiprocessor.
   (c) If no thread block height meets the above criteria, decrement the number of rows by one and return to the previous step. This process continues until the criteria is met.
2. Determine the number of registers which can be allocated to each thread based on the thread dimensions, max occupancy, and the number of registers available on each multiprocessor.
3. Set a NUM_REG_VALUES_INITIAL number of values to be stored in register memory in the array used for the computation of message values (note that this data may be spilled to local memory).
4. If there are still values left to be stored in the array, determine the number of values that can be stored in shared memory with the given occupancy, and set the minimum of that value and the number of values that still need to be stored in shared memory.
5. If there are still values to be stored, place the remaining values in register memory until MAX_REG_VALUES are placed in register memory. Then place the remaining values in local memory.

This implementation can be viewed as an application of tiling, as we perform experiments using data partitions of varying sizes via changing the maximum

Table 3. Optimal Results obtained from auto-tuning at varying maximum occupancies

	Optimal Configuration from auto-tuning			
GPU (config.)	Max occupancy	Running Time (ms)	Speed-up over Init. CUDA imp.	% Difference in speedup from opt. imp. from Section 6
GTX 285	0.14	238.3 ms	1.76	0.0%
Tesla C870	0.10	797.7 ms	1.51	+5.06%
GTX 470 (favor L1 cache)	0.58	227.6 ms	0.95	+1.56%
GTX 470 (favor shared mem)	0.16	188.0 ms	1.20	+1.01%

occupancy. A greater portion of the data can be stored in low-latency register and shared memory with a lower multiprocessor occupancy, but at the cost of less parallelism.

In our experiments, we set the NUM_REG_VALUES_INITIAL to 12 and MAX_REG_VALUES to 50 data elements, which leads to register spillover but which led to a faster run-time than specifically placing more of the data in local memory. Then, we test our implementation at each occupancy from 0.04 to 1.00 in increments of 0.02. The results are given in terms of speed-up over the initial CUDA implementation in Figure 4, with the maximum speed-up and the difference from the optimal implementation in Section 6 shown in Table 3. The maximum speedup over the initial CUDA implementation on the GTX 285 using this implementation is 1.76, which is the same as the speed-up found in Section 6 and is generated using fewer trials. In the future, we plan to look into improvements to the framework in order to generate better results in fewer trials.

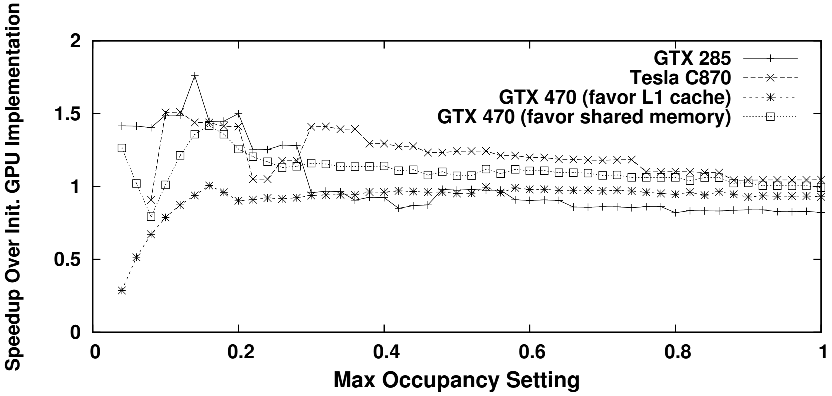


Fig. 4. Results of auto-tuning at different occupancy levels

8 Experiments Using Different GPUs

To compare the results across multiple generations of GPUs, we perform the hybrid implementation and auto-tuning experiments on the Tesla C870 GPU, which uses a GPU architecture which proceeded the GTX 285, and the GTX 470 GPU, which utilizes the Fermi architecture that succeeded the GTX 285. The results are shown with the GTX 285 results in Figure 4 and Table 2.

The Tesla C870 GPU utilizes the G80 architecture that proceeded the GTX 285; it contains 16 multiprocessors, each with 8 processors for a total of 128 processors [1]. Each multiprocessor contains 16 KB shared memory and 8192 registers, representing half the number of registers per multiprocessor compared to the GTX 285. Meanwhile, the GTX 470 utilizes the GF100 (Fermi) architecture that succeeded the GTX 285; this GPU contains 14 multiprocessors with 32 processors each for a total of 448 processors. Each multiprocessor contains 32768 registers and 64 KB which is shared between shared memory and L1 cache. The L1 cache on each multiprocessor along with the global L2 cache reduces the latency associated with local memory. The programmer can either allocate 16 KB shared memory / 48 KB L1 cache per multiprocessor or 48 KB shared memory / 16 KB L1 cache per multiprocessor [1]; we perform experiments using each configuration. Our experiments on the GTX 285 and Tesla C870 are compiled and run using CUDA 3.1, while the experiments using the GTX 470 utilize a beta version of CUDA 3.2.

Our experiments using the GTX 470 reveal that unrolling the loops to place the data in registers does not decrease the running time as it does on the GTX 285 and the Tesla C870; one possible reason for this is the L1 cache present on the GTX 470, as this decreases the latency associated with local memory. As a result, we set NUM_REG_VALUES_INITIAL and MAX_REG_VALUES to 0 when performing auto-tuning on this GPU; this places as much data as possible in shared memory given the occupancy, and then places the remaining data in local memory.

The results show that our implementations are flexible across GPUs; our system is able to generate an optimal configuration for each architecture. Interestingly, the optimal implementation retrieved on the GTX 470 when favoring L1 cache does not give a speed-up over the initial implementation; this is likely because the presence of the larger L1 cache results in lower-latency accesses to local memory. Nevertheless, our framework is able to obtain a significantly faster implementation on the GTX 470 when favoring shared memory; we obtain a speedup of 1.20 times over the initial CUDA implementation using this option.

9 Splitting Up the Image

Now, we modify the implementation to allow for image splitting in order to increase the flexibility of our implementation across GPUs with varying amounts of DRAM and to relax the constraint on image size and the number of disparity levels. This implementation splits the input images into multiple partitions, runs belief propagation on each partition, and then combines the results.

To prevent inaccurate measurements on the edge of each partition, our implementation allows padding to be applied on each image partition, making the partition size larger than the section included in the output disparity map.

We perform our experiments using the half-sized and full-sized images of the 'Cones' stereo set described in Section 6; these stereo sets measure 900 by 750 with a disparity range from 0 to 128 and 1800 by 1500 with a disparity range from 0 to 255, respectively. We set the padding to 20 pixels in each experiment. On the half-sized 'Cones' stereo set, we divide the image into three rows, and on the full-sized set, we divide the image into 25 partitions (5 ways vertical and 5 ways horizontal). The remaining parameters remain the same as in the previous experiments.

We benchmark our implementations on the GPU using the initial CUDA implementation as described in Section 3. Then, we utilize our auto-tuning implementation to retrieve an optimal configuration on each GPU; the resulting running time and speed-up over the initial CUDA implementation on the image sets using this optimal configuration are shown in Table 4.

As the number of disparity values increases, a greater portion of the data is placed into local memory due to the limited amount of registers and shared memory on each processor. As a result, the optimized results are very similar to the initial CUDA results in these experiments. Future work includes research into methods intended to optimize the running time when there is a larger disparity space.

Table 4. Optimal Results on half-sized (full-sized) 'Cones' stereo set obtained from auto-tuning

	Optimal Configuration from auto-tuning		
GPU (config.)	Max occupancy	Running Time	Speed-up over initial CUDA imp.
GTX 285	0.14 (0.14)	3120 ms (31500 ms)	1.18 (0.97)
Tesla C870	0.36 (0.36)	8260 ms (74900 ms)	1.01 (0.94)
GTX 470 (favor L1 cache)	0.54 (0.64)	1980 ms (18600 ms)	0.98 (0.98)
GTX 470 (favor shared mem)	0.58 (0.54)	1940 ms (18500 ms)	1.02 (0.99)

10 Related Work

We discussed related work in GPU belief propagation in Section 3. In addition, there is a body of work related to optimizing/auto-tuning on the GPU. Ryoo [13] looked at optimizations targeted to hide the stalling associated with long-latency operations, methods to be distribute the workload, reducing the number of dynamic instructions, and maximizing intra-thread parallelism and resource use on the GTX 8800 GPU. Datta [5] looked at optimizing and auto-tuning the stencil computation on a variety of multi-core architectures, including the GTX 280 GPU. Nukada [12] presented a method of auto-tuning the 3D FFT library on CUDA, while Li [9] looked at optimizing and auto-tuning a CUDA implementation of the GEMM algorithm. Meanwhile, Liu [11] introduced a more general adaptive framework that takes the input parameters and uses the framework to generate the optimal CUDA implementation for the given input.

Our work differs from this body of related work because of our focus on optimizing and developing an auto-tuning framework for the belief propagation algorithm that has not been optimized for CUDA across a large range of possible inputs and GPUs.

11 Conclusions and Future Work

In this paper, we explored methods to optimize a CUDA belief propagation implementation for stereo vision processing. Our results provide insights and results which can be used to optimize a real-life implementation of belief propagation for stereo; such an implementation could be utilized as part of a real-time computer vision system, among other real-world applications.

In the process, we explored the optimization space of using local, shared, and register memory options for data storage on the GPU. It is clear that high-latency local memory accesses should be kept to a minimum, and we explored various options to achieve that goal. We looked at the results of optimizations on the GTX 285, Tesla C870, and GTX 470 GPUs, and discovered that the properties of the target GPU(s) must be taken into account when optimizing a CUDA program. We showed that our optimizations work on two distinct stereo sets with different properties.

In the future, we intend to explore various properties of the CUDA compiler, such as when the compiler will automatically unroll a loop inside a kernel and how the compiler handles register spilling. We plan to run our optimized belief propagation implementations on stereo sets with varying characteristics and with a variety of input parameters, as well as explore how to optimize a belief propagation implementation for 2D motion estimation from a set of sequential images.

References

1. NVIDIA CUDA Programming Guide: Version 3.0. NVIDIA Corporation (2010), http://developer.nvidia.com/cuda
2. Brunton, A., Shu, C., Roth, G.: Belief propagation on the gpu for stereo vision, June 2006, pp. 76–76 (2006)
3. Callahan, D., Carr, S., Kennedy, K.: Improving register allocation for subscripted variables. SIGPLAN Not. 25(6), 53–65 (1990)
4. Cooper, K.D., Lu, J.: Register promotion in c programs. In: Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI 1997), pp. 308–319. ACM Press, New York (1997)
5. Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., Yelick, K.: Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: SC 2008: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, pp. 1–12. IEEE Press, Piscataway (2008)

6. Felzenszwalb, P.F., Huttenlocher, D.P.: Efficient belief propagation for early vision. In: IEEE Computer Society Conference on Computer Vision and Pattern Recognition, vol. 1, pp. 261–268 (2004)
7. Grauer-Gray, S., Kambhamettu, C.: Hierarchical belief propagation to reduce search space using cuda for stereo and motion estimation. In: 2009 IEEE Workshop on Applications of Computer Vision, WACV 2009 (2009)
8. Grauer-Gray, S., Kambhamettu, C., Palaniappan, K.: Gpu implementation of belief propagation using cuda for cloud tracking and reconstruction. pp. 1–4 (2008)
9. Li, Y., Dongarra, J., Tomov, S.: A note on auto-tuning GEMM for gPUs. In: Allen, G., Nabrzyski, J., Seidel, E., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2009. LNCS, vol. 5544, pp. 884–892. Springer, Heidelberg (2009)
10. Liang, C.K., Cheng, C.C., Lai, Y.C., Chen, L.G., Chen, H.H.: Hardware-efficient belief propagation. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 80–87 (2009)
11. Liu, Y., Zhang, E.Z., Shen, X.: A cross-input adaptive framework for gpu program optimizations. In: International Parallel and Distributed Processing Symposium, pp. 1–10 (2009)
12. Nukada, A., Matsuoka, S.: Auto-tuning 3-d fft library for cuda gpus. In: SC 2009: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, pp. 1–10. ACM, New York (2009)
13. Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D.B., Hwu, W.m.W.: Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In: PPoPP 2008: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 73–82. ACM Press, New York (2008)
14. Scharstein, D., Szeliski, R., Zabih, R.: A taxonomy and evaluation of dense two-frame stereo correspondence algorithms, pp. 131–140 (2001)
15. Scharstein, D., Szeliski, R.: High-accuracy stereo depth maps using structured light. In: IEEE Computer Society Conference on Computer Vision and Pattern Recognition, vol. 1, p. 195 (2003)
16. Sun, J., Zheng, N.N., Shum, H.Y.: Stereo matching using belief propagation. IEEE Trans. Pattern Anal. Mach. Intell. 25(7), 787–800 (2003)
17. Ivanchenko, V., Shen, H., Coughlan, J.: Elevation-based stereo implemented in real-time on a gpu. In: 2009 IEEE Workshop on Applications of Computer Vision, WACV 2009 (2009)
18. Yang, Q., Wang, L., Yang, R., Wang, S., Liao, M., Nistér, D.: Real-time global stereo matching using hierarchical belief propagation. In: British Machine Vision Conf., pp. 989–998 (2006)
19. Xu, Y., Chen, H., Klette, R., Liu, J., Vaudrey, T.: Belief propagation implementation using cuda on an nvidia gtx 280, pp. 180–189 (2009)

A Programming Language Interface to Describe Transformations and Code Generation

Gabe Rudy[1], Malik Murtaza Khan[2], Mary Hall[1],
Chun Chen[1], and Jacqueline Chame[2]

[1] School of Computing, University of Utah, Salt Lake City, UT
[2] USC/Information Sciences Institute, Marina del Rey CA

Abstract. This paper presents a programming language interface, a complete scripting language, to describe composable compiler transformations. These transformation programs can be written, shared and reused by non-expert application and library developers. From a compiler writer's perspective, a scripting language interface permits rapid prototyping of compiler algorithms that can mix levels and compose different sequences of transformations, producing readable code as output. From a library or application developer's perspective, the use of transformation programs permits expression of clean high-level code, and a separate description of how to map that code to architectural features, easing maintenance and porting to new architectures.

We illustrate this interface in the context of CUDA-CHiLL, a source-to-source compiler transformation and code generation framework that transforms sequential loop nests to high-performance GPU code. We show how this high-level transformation and code generation language can be used to express: (1) complex transformation sequences, exemplified by a single loop restructuring construct used to generate a series of tiling and permute commands; and, (2) complex code generation sequences to produce CUDA code from a high-level specification. We demonstrate that the automatically-generated code either performs closely or outperforms two hand-tuned GPU library kernels from Nvidia's CUBLAS 2.2 and 3.2 libraries.

1 Introduction

Programmers of petascale and high-performance desktop platforms alike demand high levels of performance on their library and application code. Despite a large body of research on compiler techniques for increasing parallelism or better managing the memory hierarchy [34,31,32,25,28,4,20,16,23,21,1,18,19,12], the complexity of optimizing for today's diverse architectural features leaves a significant gap between performance produced by a compiler and that achievable by a savvy programmer. For this reason, many such programmers continue to manually apply the very same code transformations that their compiler can apply automatically, not only increasing programming time but also producing low-level architecture-specific code that is difficult to port and maintain. The performance differences between automatic and manual optimization stem from many sources:

K. Cooper, J. Mellor-Crummey, and V. Sarkar (Eds.): LCPC 2010, LNCS 6548, pp. 136–150, 2011.

- Compilers must be conservative to avoid generating incorrect code and are therefore limited to optimizations that are provably safe.
- Compiler decision algorithms are usually based on static compile-time analysis which cannot always accurately predict dynamic behavior.
- Programmers can fundamentally rewrite their code (*e.g.,* use a new algorithm), while compilers must optimize the computation as written.

Even with these limitations, compilers nevertheless provide powerful and robust techniques for transformation and code generation, which if harnessed, could greatly accelerate the human effort involved in performance tuning and facilitate clean, portable code for high-performance architectures. Our research has been addressing this performance gap in two ways: (1) we use *auto-tuning* to generate a *search space* of alternative implementations of key computations and then explore this space with empirical measurement to identify the best-performing solution; and, (2) we collaborate with application and library developers in describing how to generate code through a composition of code transformations. These two separate concepts have motivated a rethinking of the compiler structure and how both application/library programmers and compiler developers should interact with it.

A previous paper described *transformation recipes*, which are descriptions of a composition of program transformations to be applied to a piece of code containing loop nests [9]. These recipes can be written by a programmer or generated automatically by the compiler using the same structure. Similar concepts appear in the literature, some recent, although primarily other work focuses on annotations on the source code [22,7,10,36]. The recipes in [9] are in a separate script, permitting greater flexibility and portability, since a collection of different scripts, possibly targeting different architectures, can be associated with the same high-level code.

This paper takes the notion of transformation recipes a significant step further by providing a *programming language interface* for describing transformation and code generation. We have developed an embedded scripting language interface to express high-level transformation recipes that are translated to a lower-level interface, in the same way that high-level programming languages are translated to lower level ones to improve programmer productivity. Using a programming language offers several advantages: (1) the system can integrate *queries* of the program state and other *control flow* to guide optimization decisions; (2) the programmer can create variables to refer to output objects from code transformations to which additional transformations can be applied, and which are used to produce *readable code*; and, (3) the programmer of the system can express *encapsulation* into functions to prototype or implement high-level transformation algorithms, which can be reused by other, possibly less-skilled developers or for other pieces of similar code.

To illustrate the power of this programming language interface, we have used it to implement a core set of transformations to generate CUDA code, a parallel programming language interface for Nvidia GPUs [15]. Each CUDA-specific transformation generates a composition of a sequence standard compiler

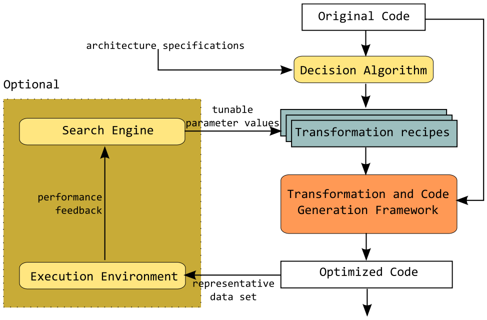


Fig. 1. Compiler Developer Workflow

transformations. The programming language interface is also being used for other data-parallel and locality optimization and code generation tasks.

This paper makes the following contributions over our previous work [9]. We introduce the notion of a programming language interface for implementing sequences of transformation and code generation tasks, with compiler workflow and structure and interface constructs described in Section 2. We use this framework to develop CUDA-CHiLL, which generates high-performance CUDA code for sequential loop nest computations. Section 3 provides an example of implementing a complex transformation sequence involving a series of tiling and permute commands that can be expressed with a single line, high-level construct. Section 4 presents a complex code generation sequence for CUDA targeting a GPU, which is also expressed compactly with a single line. We demonstrate in Section 5 that this system generates high-performance code that can perform comparably or even outperform manually-tuned library code from Nvidia's CUBLAS library.

2 Compiler Structure and Motivation

This section provides an overview of the auto-tuning compiler organization we have been developing, which combines a polyhedral code generation and transformation framework called CHiLL, a loop transformation recipe interface, and auto-tuning framework for empirically searching a space of possible implementations of a key computation, as shown in Figure 1. The loop transformation recipe, which could either be generated by the compiler or written by an application developer, describes a sequence of code transformations that, when applied to the original code, corresponds to an optimized version of the code. An auto-tuning search engine can explore the space of multiple code variants described by different recipes and parameter values within each recipe to find the best-performing implementation. More details of the system organization and use in optimizing library and application code can be found elsewhere [2,29,37,27,26].

In this paper, we illustrate the power of this compiler structure with an example use of the language interface called CUDA-CHiLL [24]. The goal of

CUDA-CHiLL is to automate many of the difficult programming tasks in generating high-performing, equivalent CUDA code targeting an Nvidia GPU, starting from a sequential computation [15]. The automatically generated CUDA code is then compiled by the Nvidia backend to exploit its ability to perform low-level architecture-specific optimization. While such GPUs represent a powerful and programmable parallel architecture, with hundreds of parallel cores, it is still the case that generating high-performance CUDA code is challenging. A loop transformation framework such as ours can be extremely valuable for CUDA optimization and code generation. Known compiler transformations can be adapted and applied both in the decomposition and mapping process, and in subsequently optimizing the kernel code to manage the memory hierarchy and parallelism tradeoffs. Since there is significant performance variation on GPUs for very subtle differences in code, the integration with an auto-tuning framework allows us to explore a space of different implementations, and different values of parameters associated with the mapping. Compiler automation is also useful for error-prone code generation tasks such as array indexing and bounds checking, and mundane code generation tasks such as allocating memory for GPU input and output, copying data to and from the GPU, and performing block and thread decomposition.

2.1 Requirements for Translating Sequential Loop Nests to CUDA

In CUDA, a thread program (called a *kernel*) is executed on each point of a multidimensional space, or *grid*. A grid defines a two-level parallelism hierarchy of *thread blocks*, where blocks can be indexed into one or two dimensions, and each block is further decomposed into *threads*, where threads can be indexed into one, two or three dimensions. A computation mapping of a loop nest to a grid is a mapping from the iteration space of the loop nest onto grid points (with portions of the iteration space possibly executed within each kernel).

To take advantage of these two levels of parallelism, the iteration space of a loop nest can be decomposed into loops such that each loop is mapped to a level of blocks or threads. Partitioning the iteration space of a loop into blocks or

```
void seqMV(float c[N][N], float a[N],
                          float b[N]) {
  int i, j;
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      a[i] = a[i] + c[j][i] * b[j];
}
```

(a) Matrix-vector multiply source code.

```
TI=32
tile_by_index({"i"}, {TI},
   {l1_control="ii"},
   {"ii", "i", "j"})
```

(b) Tiling for computation partitioning.

```
/* map ii to block, i to threads */
for(ii = 0; ii < 32; ii++)
 for(i = 32 * ii; i < 32 * ii + 32; i++)
    for(j = 0; jj < N; jj++)
        a[i] = a[i] + c[j][i] * b[j];
```

(c) After tile command in (b) is applied to example in (a), with N=1024.

Fig. 2. Example of tile transformations to deconstruct iteration space

tiles with a fixed maximum size has been widely used when constructing parallel computations [33,14]. The shape and size of the tile can be chosen to take advantage of the target parallel hardware and memory architecture, maximizing reuse while maintaining a data footprint that meets memory capacity constraints.

Tiling involves re-structuring an iteration space into a control loop and tile loop. Given an iteration space of size N and a tile size of TX, the tile loop iterates over a space defined by the tile size (TX), while the constructed control loop has N/TX iterations (when TX divides N evenly) [1]. By using tiling with appropriate tile sizes it is possible to derive a transformed loop nest where each loop level (and its iteration space) corresponds to a specific block or thread dimension, normalized to start at 0 and have a unit step size.

Figure 2 shows an example of using loop tiling to re-structure the iteration space of the matrix-vector multiplication code in Figure 2 (a). The transformation recipe in Figure 2 (b) contains a tile command specifying that loop "i" should be tiled with a tile size of 32. This generates a control loop "ii" which is placed just outside "i". The tile command has a parameter (not shown in the script) that specifies the stride of the new control loop as either 1 (*counted*) or the tile size (*strided*). The resulting code, given N=1024, TI=32, and control loop with stride 1, is shown in Figure 2 (c). The outermost *ii* loop is then mapped to a single GPU block dimension and the *i* loop is mapped to a single thread dimension.

Explicit data movement. The GPU memory hierarchy is heterogeneous and must be explicitly managed by software. Data must be copied from the host into global device memory, with latencies of hundreds of cycles per access and no caching (on the NVIDIA GTX280.) All data are stored in global memory by default, unless explicitly copied into low-latency shared memory which can be accessed across threads running concurrently on a streaming multiprocessor (SM). Using a *datacopy* transformation, the compiler introduces a new, usually smaller data structure from the footprint of array accesses within a given loop nest. Although not an explicitly controlled memory structure, the large register file can be exploited by copying data into fixed sized thread-local arrays and scalars. A variant of *datacopy*, which we refer to as *datacopy_privatized*, targets registers by copying local data that are touched only by the subset of iterations assigned to a thread to thread-local arrays and scalars kept in registers.

2.2 Foundation from CHiLL Loop Transformation Recipes

Our previous work describes providing the transformation recipe interface to library and application developers, so that they can control the compiler's optimization strategy [9]. We developed a script interface with a flat sequence of transformations that were unconditionally applied, and a collection of standard compiler transformations with parameters that fully specified the application

[1] The simplicity of an evenly divisible problem size is not a requirement, but is used to clarify the example.

Table 1. Description of common CHiLL commands, used by CUDA-CHiLL

Command	Description
tile	Tile a loop with a given tile size. Specify an index variable for the new control loop and a second index variable to optionally rename the original index variable of the tiled loop.
permute	For a specified statement, reorder loops in a loop nest (the iteration space of the statement.) The permutation order is specified by a list of loop index variables.
datacopy	For a specified statement, starting loop level and array variable, copy array data that is accessed within the starting loop level to a smaller dimensional structure. Optionally annotate the new data structure with `__shared__` to specify a copy to shared memory.
datacopy_privatized	Similar to `datacopy` but used to copy data private to a thread and thus does not have an option to flag for shared memory.

of each transformation. Table 1 shows a few of the previously available CHiLL commands, ones that are used by CUDA-CHiLL.

2.3 Using a Lua Programming Language Interface in CUDA-CHiLL

CUDA-CHiLL maps sequential code to CUDA primarily through a sequence of tile, permute and datacopy transformations. Other than the specific CUDA constructs discussed in Section 4, a sequence of the CHiLL transformations in Table 1 can produce the parallel code structure for the CUDA program. However, the transformation recipes to realize these implementations can be fairly complex, and for different programs the recipes share a similar standard structure. Therefore, as part of this work, we have developed a higher-level interface that encapsulates these standard mapping strategies.

Using the programming language interface, transformation recipes are expressed in Lua [11], a lightweight, embeddable scripting language with extensible semantics and easy integration with a host program. Through Lua, we are able to create scripts that describe transformation "programs", ones that can query the internal data structures of the compiler to make optimization decisions, create output variables that represent the results of transformations, incorporate control flow to tailor optimization to context, and encapsulate commonly used constructs into "subroutines" that can be part of an optimization library.

Table 2 illustrates how the programming language interface implements these higher-level commands and complex sequences by composing basic commands from Table 1. Each table entry lists the command and its description, with example parameters shown in Table 2. We will describe a few of these commands in more detail in subsequent sections. Note that the commands in Table 1 and Table 2 can be mixed together to fully utilize the expressive power of CUDA-CHiLL's programming language interface, and further control the mapping if the high-level constructs are not producing the desired results. As a measure of power of Lua as a scripting language, the entire Phase I CUDA-CHiLL extensions represented in Table 2 are implemented in just over 300 lines of Lua code. The tiling-related code and the copy-related code are comprised of roughly 100 lines of Lua each, the unroll algorithm is roughly 50 lines of code, and the remainder is utility functions.

Table 2. Description of prominent commands in CUDA-CHiLL scripts

Command	Example Parameter	Description
`tile_by_index`	`{"i","j"}`	The index variables of the loops to be tiled
	`{TI,TJ}`	The respective tile sizes for each index variable

	`{l1_control="ii",`
	`l2_control="jj"}`
		A mapping that specifies control loop variable names and optionally renames tile loop index variables.

	`{"ii", "jj", "i", "j"}`	Final order of nested loops with updated loop index names
`cudaize`	`"gpuMV"`	The name of the kernel function
	`{a=N, b=N, c=N*N}`	The data sizes of the arrays if not statically determinable

	`{block={"ii"},`
	`thread={"jj"}}`
		Block and thread indices for mapping. The bounds for these loops are used to define the grid dimensions.

`copy_to_registers`	`"kk"`	The loop level, given as an index variable, that is the target of register structure
	`"c"`	The name of the array variable to be copied
`copy_to_shared`	`"tx"`	The loop level, given as an index variable, that is the target of the copied data
	`"b"`	The name of the array variable to be copied
	`-16`	Ensure the last dimension of the temporary array are coprime with 16
`unroll_to_level`	1	Unrolls all statements up to one level from innermost loops outwards. This construct will stop unrolling if it encounters a CUDA thread mapped index.

3 Computation Decomposition of a Loop Nest: A Complex Transformation Sequence

In this section we discuss the semantics and implementation of key CUDA-CHiLL constructs, focusing on the loop restructuring.

Loop restructuring using tile_by_index. As described in Section 2, a computation partition that takes advantage of the multiple levels of parallelism on a GPU grid can be achieved by decomposing the iteration space of a loop nest into a set of loops such that each loop is mapped to a block or thread dimension, or is within a thread loop. This decomposition requires a complex sequence of tile and permute transformations when expressed using CHiLL's lower-level transformation interface. In the CUDA-CHiLL interface, this sequence of transformations can be expressed by a single command, `tile_by_index`, which directs the computation space decomposition of the original loop nest as well as laying the foundation for control of subsequent memory hierarchy optimizations. Table 2 provides descriptions of each parameter of the `tile_by_index` command.

Figure 3(a) shows a CUDA-CHiLL script for the matrix-vector multiply code in Figure 2(a) that uses `tile_by_index` to specify multiple tile transformations. We can see the lower-level nature of the corresponding sequence of CHiLL transformations is shown in Figure 3(b). Source code and a more complex script for matrix-matrix multiply is shown in Figure 4.

Figure 5 presents the details of the algorithm used in the `tile_by_index` function. The algorithm builds a list of `tile` transformations based on the desired

```
N=1024
TI=32
tile_by_index({"i","j"}, {TI,TI},
    {l1_control="ii", l2_control="jj"},
    {"ii", "jj", "i", "j"})
normalize_index("i")
cudaize("mv_GPU", a=N, b=N, c=N*N,
    block="ii", thread="i")
copy_to_shared("tx", "b", 1)
copy_to_registers("jj", "a")
unroll_to_depth(1)
```

(a) CUDA-CHiLL script for matrix-vector multiply.

```
N=1024
TI=32
original()
tile(0, 1, TI, 1, i, ii, 1)
tile(0, 3, TI, 2, j, k, 1)
tile(0, 3, 3)
datacopy(0, 3, b, "tmp1", "tmp2", false,
         0, 1, 1, true)
tile(1, 3, 3)
datacopy_privatized(0, k, a, tx)
unroll(0, 5, 0)
```

(b) CHiLL script generated by Lua interface.

```
void seqMV(float (*c)[1024], float *a,
          float *b)
{
float *devO1Ptr, *devI1Ptr, *devI2Ptr;
cudaMalloc(&devO1Ptr, 1024 * 4);
cudaMemcpy(devO1Ptr, a, 1024 * 4,
    cudaMemcpyHostToDevice);
cudaMalloc(&devI1Ptr, 1048576 * 4);
cudaMemcpy(devI1Ptr, c, 1048576 * 4,
    cudaMemcpyHostToDevice);
cudaMalloc(&devI2Ptr, 1024 * 4);
cudaMemcpy(devI2Ptr, b, 1024 * 4,
    cudaMemcpyHostToDevice);
dim3 dimGrid(64, 1);
dim3 dimBlock(64, 1);
gpuMV<<<dimGrid,dimBlock>>>(devO1Ptr,
    (float(*) [1024])devI1Ptr, devI2Ptr);
cudaMemcpy(a, devO1Ptr, 1024 * 4,
    cudaMemcpyDeviceToHost);
cudaFree(devO1Ptr);
cudaFree(devI1Ptr);
cudaFree(devI2Ptr);
}
```

(c) CUDA host code.

```
__global__ void gpuMV(float *a, float **c, float *b)
{
  int bx = blockIdx.x; int tx = threadIdx.x;
  __shared__ float bcpy[32];
  float acpy = a[tx + 32 * bx];
  for (jj = 0; jj < 32; jj++) {
    bcpy[tx] = b[32 * jj + tx];
    __syncthreads();
    //this loop is actually fully unrolled
    for (j = 32 * jj; j <= 32 * jj + 32; j++)
      acpy = acpy + c[j][32 * bx + tx] * bcpy[j];
    __syncthreads();
  }
  a[tx + 32 * bx] = acpy;
}
```

(d) CUDA kernel code.

Fig. 3. Scripts for matrix-vector multiply and generated code

loop order and tile sizes. At each step it calls *BuildOrder* to construct an order that reflects the number of tile operations that have already been performed. Given this order it can choose a couple of different ways of tiling to ensure the proper placement of the control loop and tile loop. After each tile transformation, it performs a *permutation* to handle further changes to the loop order. This algorithm and its auxiliary functions is implemented in roughly 300 lines of Lua code. In the figure, the use of control flow and queries to CHiLL enable the algorithm to determine the current loop structure and the sequence of permute and tile functions needed to achieve the desired goal.

Unrolling and Copy commands. Due to space limitations, we omit a detailed discussion of other CUDA-CHiLL commands, but summarize their functionality here. The unrolling function applies aggressive loop unrolling to fully unroll a specific loop nesting level in the GPU code, and is implemented in about 50 lines of Lua code. While the use of aggressive unrolling can create register pressure and fill up the instruction cache, the size of the iteration space of inner loops on the GPUs is typically small due to managing limited memory resources, and the

```
void seqMM(float c[N][N], float a[N][N],
                          float b[N][N])
{
  int i, j, k;
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      for (k=0; k < N; k++)
         c[i][j] = c[i][j] +
                   a[i][k] * b[k][j];
}
```

(a) Matrix-matrix multiply source code.

```
1.  tile_by_index({"i","j"},{TI,TJ},
                 {l1_control="ii",l2_control="jj"}
                 {"ii","jj","i","j"})
2.  tile_by_index({"k"},{TK},
                 {l1_control="kk"},
                 {"ii","jj","kk","i","j","k"},
                 strided)
3.  tile_by_index({"i"},{TJ},
                 {l1_control="tt",l1_tile="t"},
                 {"ii","jj","kk","t","tt","j","k"})
4.  cudaize("mm_GPU",{a=N*N,b=N*N,c=N*N},
           {block={"ii","jj"},thread={"t","tt"}})
5.  copy_to_registers("kk","c")
6.  copy_to_shared("tx","b",-16)
7.  unroll_to_depth(2)
```

(b) CUDA-CHiLL script.

Fig. 4. Source and CUDA-CHiLL script for matrix-matrix multiply

register file is extremely large, so aggressive unrolling usually is profitable and can be avoided when it is not. It not only reduces control overhead and improves instruction-level parallelism, but it also simplifies the code and encourages the `nvcc` backend compiler to allocate array variables to registers. The implementation of the `unroll_to_depth` function is the application of unrolling to all statements in the transformed code. Importantly, after tiling for problem sizes that are not divisible by the tile size, the compiler generates additional clean-up code, so this function applies unrolling to the clean-up loops as well.

The `copy_to_registers` and `copy_to_shared` are the most complex functions, implemented in about 300 lines of Lua code. These functions take as input the level of the loop nest to place the copy function, and then the compiler implicitly generates the temporaries and derives their size according to the footprint of the accesses to the original array within the portion of the loop nest specified by the parameter. For registers, the temporary is declared as a local array in the thread program, while for shared memory, it is declared with a `_shared_` attribute. An additional parameter for copying to shared memory is used to specify padding to avoid shared memory bank conflicts.

4 CUDA Code Generation

After CHiLL's internal abstraction of the code is modified through a sequence of these transformations, the resulting changes to the code are applied to the compiler's intermediate representation to produce the desired GPU code.

As an example, consider the CUDA code shown in Figure 3(c). This code is automatically generated to provide the proper scaffolding to handle data movement between host and GPU and make the GPU kernel call. The array references in the sequential loop nest are analyzed to determine their read and write properties. Appropriate `cudaMalloc` and `cudaMemcpy` calls are then generated for each array to transfer data to and from GPU *global memory*. If size attributes were specified in the `cudaize` call in the CUDA-CHiLL transformation script, they will be used in these memory copy operations.

```
TileByIndexCommands(s,I,S,M,O)
Input: $s$: Statement number; $I$: Indices to tile; $S$: Tile sizes;
       $M$: Map of names for indices; $O$: Final loop nest order
Output: $F$: Set of transformation operations
begin
  $F$ := $\emptyset$
  $C$ := extract control loop name list from $M$
  $I'$ := extract renamed tile loop name map from $M \cup I$
  $order$ := $BuildOrder(O, C, I', 0)$
  $F$ := $F + [\![$permute($s$,$order$)$]\!]$
  for $i$ in $1..|I|$ do
    $level$ := $FindLevel(I_i)$
    $order$ := $BuildOrder(O, C, I', i)$
    $offset$ := offset between $I'_i$ and $C_i$ in $order$
    if $offset < 0$ then
      $F$ := $F + [\![$tile($s$,$level$,$S_i$,$level + offset$,$I'_i$,$C_i$)$]\!]$
    then
      $F$ := $F + [\![$tile($s$,$level$,$S_i$,$level$,$I'_i$,$C_i$)$]\!]$
    end
    $order$ := $BuildOrder(O, C, I',i)$
    $F$ := $F + [\![$permute($s$,$order$)$]\!]$
  end
  return $F$
end
```

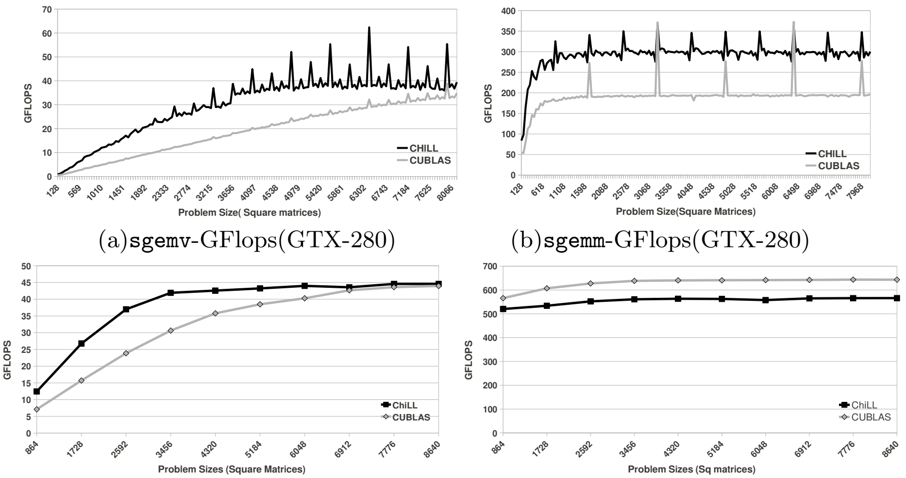
```
BuildOrder(O,C,I,n)
Input: $O$: Final loop nest order;
       $C$: Control loop list;
       $I$: Index loop mapping;
       $n$: Current index;
Output: $B$: Built order
begin
  $B$ := $\{\}$
  for $o$ in $O$ do
    Skip if $o \in C$ after $n$
    if $o \in I$ then
      $o$ := $I(o)$
    end
    $B$ := $B :: o$
  end
  return $B$
end
```

Fig. 5. Algorithm used by `tile_by_index` where *FindLevel* finds the loop level of a index variable in the current internal representation and *BuildOrder* builds the snapshot of what the order should be between its current state and its final state given that n tile operations were already processed.

The kernel call requires parameters to define the CUDA grid for the computation. In this case, there is only one grid dimension and thread dimension (all other dimensions are set to 1). The call to the generated GPU kernel, `gpuMV`, is made with the `dim3` variables that define the execution grid space as extra parameters. CUDA uses the `<<< >>>` syntax in a C++-template manner to prefix the parameter list at the call site of the kernel function with these dimension variables. The kernel code is shown in Figure 3(d).

Finally, as an effect of the `cudaize` call in the transformation script, for each statement group there should be loop levels with specially renamed index variables from the set `bx,by,tx,ty,tz`. During the transformation to CUDA code, loops with these indices are removed and references to these variables are replaced with the CUDA provided index variables from the set `blockIdx.x, blockIdx.y, threadIdx.x, threadIdx.y, threadIdx.z`. If there was a compound upper bound for a removed loop it is replaced with a bounds check to ensure correctness. For example, if the upper bound for the `tx` loop was `min(-(58 * bx) + 116, 128)` and the `thread.x` grid dimension was 128, the loop construct would be replaced with the condition `if(tx < -(58 * block.x) + 116)`.

Interestingly, the generated code is fairly complex even for this simple kernel, and can be quite different for different problem sizes. The code for problem size 4096, which includes the CUDA scaffolding code of Figure 3(c) and the CUDA kernel of Figure 3(d), has 97 lines of code, partially due to the aggressive loop unrolling. The code for size 1975 is longer, with 142 lines of code. Because 1975 is not divisible by the tile size and thread partitioning size of 32, the compiler must generate "clean up" code for the portion of the computation that goes beyond an



(a)`sgemv`-GFlops(GTX-280)

(b)`sgemm`-GFlops(GTX-280)

(c)`sgemv`-GFlops(Tesla C2050)

(d)`sgemm`-GFlops(Tesla C2050)

Fig. 6. Performance comparison of automatically-generated CUDA-CHiLL code with for Single Precision Matrix-Vector and Matrix-Matrix Multiplication kernels, on a GTX-280 vs. CUBLAS 2.2 (a,b) and on a Tesla C2050 vs. CUBLAS 3.2 (c,d).

even multiple of 32, and also test to ensure that a particular thread participates in a portion of the computation. This clean up code falls out naturally from the compiler's polyhedral framework for code generation, using the same set of tile commands for both problem sizes.

5 Performance Results

We applied the transformation scripts in Figures 3 and 4 to single precision matrix-vector (`sgemv`) and matrix-matrix (`sgemm`) multiplication (with an additional script not shown to optimize `sgemm` for the C2050). We examined the performance of the generated code for a set of square matrix sizes in the range 128-8192. We illustrate the power of our approach with a performance comparison between our automatically-generated code and the CUBLAS 2.2 and 3.2 library versions released by Nvidia as shown in Figure 6. All results were obtained on an Nvidia GTX280 and Tesla C2050, and show the average performance over three runs, with a standard deviation of less than 0.1 msec. We show a large collection of points in the matrix size range to demonstrate the success of the approach across different matrix sizes. We step by 49 when comparing to CUBLAS 2.2 in Figure 6(a,b) to show a diversity of problem sizes and highlight the unstable behavior of CUBLAS for different problem sizes. As performance of CUBLAS 3.2 is much more stable, we we used a smaller set of matrix sizes for the comparison in Figure 6(c,d).

We used auto-tuning to explore a small set of tile sizes (multiples of 16 that are smaller than 128) to achieve these performance results. As tile sizes govern the number of threads in a block, and thus affect the memory coalescing behavior, we chose tile sizes that are multiples of a warp (*i.e.,* 32, the scheduling unit) or a half-warp (16, the memory scheduling unit on the GTX-280).

Comparing against CUBLAS 2.2 on a GTX 280 in 6(a,b), the CUDA-CHiLL code outperforms CUBLAS 2.2 in `sgemv` over the whole range of problem sizes considered. Similarly, for almost all problem sizes used in our experiment CHiLL outperforms CUBLAS 2.2 for `sgemm`. CHiLL achieves a 1.78x average speed up over CUBLAS 2.2 for `sgemv` and 1.5x for `sgemm`. The maximum GFlops achieved by CHiLL generated code is 366GF. Figure 6(c,d) shows similar comparisons with CUBLAS 3.2 on TeslaC2050, an architecture with more cores and larger shared memory. Even for this newer library implementation, our automatically-generated `sgemv` always outperforms CUBLAS 3.2 and achieves up to 44GFlops. We are within 11-13% the CUBLAS 3.2 `sgemm` code, reaching up to 565 GFlops. Some of the performance gap in the C2050 `sgemm` performance comes from the explicit use of texture memory with dedicated hardware which CUDA-CHiLL does not currently use, but is planned for future work.

6 Related Work

Polyhedral loop transformation frameworks (*e.g.,* [13,8]) are known to support composition of multiple transformations. Internally, these frameworks manipulate mathematical representations of iteration spaces and loop bounds, and expose interfaces that allow users (or compilers) to manipulate these low-level mathematical representations or individual loop or statement manipulations. Such interfaces are still too cumbersome to use when implementing a complex optimization strategy because descriptions of transformation sequences tend to be lengthy. Our own compiler uses a polyhedral transformation framework, but the transformation recipes specify high-level transformations that operate on a complete loop nest; transformation algorithms translate from the recipes to the iteration space manipulations for all statements enclosed in the loop nest [5,6]. As compared to other polyhedral frameworks, the transformation recipes described in this paper target a higher level description of a composition of transformations that is suitable for savvy application developers in addition to compiler developers; further, the interface can be broadly applicable to non-polyhedral frameworks and polyhedral frameworks alike.

A number of interfaces to code transformation exist that are targeting a similar level to the transformation recipes presented here. These include pragma-oriented transformation specifications such as LoopTool [22], X language [7], and Orio [10]. A related tool POET uses an XML-based description of code transformation behavior to produce portable code transformation implementations [36], These tools all provide a general and flexible way to express a set of transformations on a specific code fragment and several of these generate a set of alternative implementations to support auto-tuning. Looking across the tools, the set of supported transformations is different, and ours is neither a subset or superset of other systems. Some distinguishing characteristics of our supported transformations include the OpenMP and CUDA parallel code generation, specialization and index set splitting. Further, these tools support a core set of transformations such as loop unrolling and loop tiling, and some support extension to add new transformations. In summary, each tool has its unique strengths and most suitable applications. Thus, we expect that the existence of other interfaces at this level gives promise to potential for interoperability between tools.

Compilers for generating GPU code include work in Pluto that automatically parallelizes a sequential code [3],and generation of CUDA from OpenMP [17]. GPU code optimizers include generation of optimized code from a naiive gpu kernel in a CUDA to CUDA code optimizer [35] and CUDA-lite [30], which optimize CUDA code to improve the coalescing of global memory accesses and the bandwidth to global memory.

7 Summary and Future Work

This paper has described a programming language interface to transformation and code generation that permits high-level description and encapsulation of complex transformation sequences. In this paper, we have developed this interface in the context of CUDA-CHiLL, which provides an interface for generating CUDA code for GPUs from sequential code and a transformation recipe. We demonstrate that the automatically-generated code either performs closely or outperforms two hand-tuned GPU library kernels from Nvidia's CUBLAS 2.2 and 3.2 libraries, matrix-vector and matrix-matrix multiply.

We envision this approach as a step towards building compilers in a different way, where these "programs" become libraries of transformation strategies that could be made available to users of different expertise levels. Over time, this library could grow and be customized for specific domains, applications or user communities.

Acknowledgements. This work has been funded in part by Department of Energy grants DE-FC02-06ER25765 and DE-FC02-10ER25898, National Science Foundation grants SHF-1018881, SHF-0916436, and OCI-0749360, and DARPA contract HR0011-10-9-000.

References

1. Ahmed, N., Mateev, N., Pingali, K.: Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In: Proceedings of the 2000 ACM International Conference on Supercomputing (May 2000)
2. Bailey, D.H., Chame, J., Chen, C., Dongarra, J., Hall, M., Hollingsworth, J.K., Hovland, P., Moore, S., Seymour, K., Shin, J., Tiwari, A., Williams, S., You, H.: PERI auto-tuning. Journal of Physics: Conference Series 125(1) (2008)
3. Baskaran, M.M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: A compiler framework for optimization of affine loop nests for GPGPUs. In: Proceedings of the 22nd Annual International Conference on Supercomputing, ICS 2008, pp. 225–234. ACM, New York (2008)

4. Carr, S., Kennedy, K.: Improving the ratio of memory operations to floating-point operations in loops. ACM Transactions on Programming Languages and Systems 16(6), 1768–1810 (1994)
5. Chen, C.: Model-Guided Empirical Optimization for Memory Hierarchy. PhD thesis, University of Southern California (May 2007)
6. Chen, C., Chame, J., Hall, M.: CHiLL: A framework for composing high-level loop transformations. Technical Report 08-897, University of Southern California (June 2008)
7. Donadio, S., Brodman, J., Roeder, T., Yotov, K., Barthou, D., Cohen, A., Garzarán, M.J., Padua, D., Pingali, K.: A language for the compact representation of multiple program versions. In: Ayguadé, E., Baumgartner, G., Ramanujam, J., Sadayappan, P. (eds.) LCPC 2005. LNCS, vol. 4339, pp. 136–151. Springer, Heidelberg (2006)
8. Girbal, S., Vasilache, N., Bastoul, C., Cohen, A., Parello, D., Sigler, M., Temam, O.: Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. International Journal of Parallel Programming 34(3), 261–317 (2006)
9. Hall, M., Chame, J., Chen, C., Shin, J., Rudy, G., Khan, M.M.: Loop transformation recipes for code generation and auto-tuning. In: Gao, G.R., Pollock, L.L., Cavazos, J., Li, X. (eds.) LCPC 2009. LNCS, vol. 5898, pp. 50–64. Springer, Heidelberg (2010)
10. Hartono, A., Norris, B., Sadayappan, P.: Annotation-based empirical performance tuning using Orio. In: Proceedings of the 23rd International Parallel and Distributed Processing Symposium (May 2009)
11. Ierusalimschy, R., de Figueiredo, L.H., Filho, W.C.: Lua an extensible extension language. Softw. Pract. Exper. 26, 635–652 (1996)
12. Jiménez, M., Llabería, J.M., Fernández, A.: Register tiling in nonrectangular iteration spaces. ACM Transactions on Programming Languages and Systems 24(4), 409–453 (2002)
13. Kelly, W., Pugh, W.: A framework for unifying reordering transformations. Technical Report CS-TR-3193, Department of Computer Science, University of Maryland (1993)
14. Kennedy, K., McKinley, K.: Optimizing for parallelism and data locality. In: ACM International Conference on Supercomputing (July 1992)
15. Kirk, D., Hwu, W.: Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann Publishers, San Francisco (2010)
16. Kodukula, I., Ahmed, N., Pingali, K.: Data-centric multi-level blocking. In: Proceedings of ACM SIGPLAN Conference on Programming Languages Design and Implementation (June 1997)
17. Lee, S., Min, S.-J., Eigenmann, R.: OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (February 2009)
18. Lim, A.W., Lam, M.S.: Maximizing parallelism and minimizing synchronization with affine partitioning. In: Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1997) (January 1997)
19. Lim, A.W., Liao, S.-W., Lam, M.S.: Blocking and array contraction across arbitrarily nested loops using affine partitioning. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (June 2001)
20. McKinley, K.S., Carr, S., Tseng, C.-W.: Improving data locality with loop transformations. ACM Transactions on Programming Languages and Systems 18(4), 424–453 (1996)

21. Pugh, B., Rosser, E.: Iteration space slicing for locality. In: Carter, L., Ferrante, J. (eds.) LCPC 1999. LNCS, vol. 1863, p. 164. Springer, Heidelberg (2000)
22. Qasem, A., Jin, G., Mellor-Crummey, J.: Improving performance with integrated program transformations. Technical Report TR03-419, Rice University (October 2003)
23. Rivera, G., Tseng, C.-W.: Data transformations for eliminating conflict misses. In: Proceedings of ACM SIGPLAN Conference on Programming Languages Design and Implementation (June 1998)
24. Rudy, G.: CUDA-CHiLL: A programming language interface for GPGPU optimizations and code generation. Master's thesis, University of Utah (May 2010)
25. Sarkar, V., Thekkath, R.: A general framework for iteration-reordering loop transformations. In: Proceedings of ACM SIGPLAN Conference on Programming Languages Design and Implementation (June 1992)
26. Shin, J., Hall, M., Chame, J., Chen, C., Fischer, P.F., Hovland, P.D.: Speeding up nek5000 with autotuning and specialization. In: Proceedings of the 2010 ACM International Conference on Supercomputing (June 2010)
27. Shin, J., Hall, M.W., Chame, J., Chen, C., Hovland, P.D.: Autotuning and specialization: Speeding up matrix multiply for small matrices with compiler technology. In: Proceedings of the 4th International Workshop on Automatic Performance Tuning (October 2009)
28. Temam, O., Granston, E.D., Jalby, W.: To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In: Proceedings of Supercomputing 1993 (November 1993)
29. Tiwari, A., Chen, C., Chame, J., Hall, M., Hollingsworth, J.K.: A scalable autotuning framework for compiler optimization. In: Proceedings of the 24th International Parallel and Distributed Processing Symposium (April 2009)
30. Ueng, S.-Z., Lathara, M., Baghsorkhi, S.S., Hwu, W.-m.W.: CUDA-Lite: Reducing GPU Programming Complexity. In: Amaral, J.N. (ed.) LCPC 2008. LNCS, vol. 5335, pp. 1–15. Springer, Heidelberg (2008)
31. Wolf, M.E., Lam, M.S.: A data locality optimizing algorithm. In: Proceedings of ACM SIGPLAN Conference on Programming Languages Design and Implementation (June 1991)
32. Wolf, M.E., Lam, M.S.: A loop transformation theory and an algorithm to maximize parallelism. IEEE Transactions on Parallel and Distributed Systems 2(4), 452–471 (1991)
33. Wolfe, M.: More iteration space tiling. In: Proceedings of the 1989 ACM/IEEE Conference on Supercomputing, pp. 655–664. ACM, New York (1989)
34. Wolfe, M.: Data dependence and program restructuring. The Journal of Supercomputing 4(4), 321–344 (1991)
35. Yang, Y., Xiang, P., Kong, J., Zhou, H.: A gpgpu compiler for memory optimization and parallelism management. SIGPLAN Not. 45(6), 86–97 (2010)
36. Yi, Q., Seymour, K., You, H., Vuduc, R., Quinlan, D.: POET: parameterized optimizations for empirical tuning. In: Proceedings of the 21st International Parallel and Distributed Processing Symposium (March 2007)
37. Zima, H., Hall, M., Chen, C., Chame, J.: Model-guided autotuning of high-productivity languages for petascale computing. In: Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing (HPDC 2009) (June 2009)

Unified Parallel C for GPU Clusters: Language Extensions and Compiler Implementation

Li Chen[1], Lei Liu[1], Shenglin Tang[1], Lei Huang[2],
Zheng Jing[1], Shixiong Xu[1], Dingfei Zhang[1], and Baojiang Shou[1]

[1] Key Laboratory of Computer System and Architecture,
Institute of Computing Technology, Chinese Academy of Sciences, China
{lchen,liulei2007,tangshenglin,jingzheng,xushixiong,
zhangdingfei,shoubaojiang}@ict.ac.cn
[2] Department of Computer Science, University of Houston; Houston, TX, USA
lei.huang@mail.uh.edu

Abstract. Unified Parallel C (UPC), a parallel extension to ANSI C, is designed for high performance computing on large-scale parallel machines. With General-purpose graphics processing units (GPUs) becoming an increasingly important high performance computing platform, we propose new language extensions to UPC to take advantage of GPU clusters. We extend UPC with hierarchical data distribution, revise the execution model of UPC to mix SPMD with fork-join execution model, and modify the semantics of upc_forall to reflect the data-thread affinity on a thread hierarchy. We implement the compiling system, including affinity-aware loop tiling, GPU code generation, and several memory optimizations targeting NVIDIA CUDA. We also put forward unified data management for each UPC thread to optimize data transfer and memory layout for separate memory modules of CPUs and GPUs. The experimental results show that the UPC extension has better programmability than the mixed MPI/CUDA approach. We also demonstrate that the integrated compile-time and runtime optimization is effective to achieve good performance on GPU clusters.

1 Introduction

Following closely behind the industry-wide move from uniprocessor to multi-core and many-core systems, HPC computing platforms are undergoing another major change: from homogeneous to heterogeneous platform. This heterogeneity - exhibited in compute capability, speed, memory size and organization, and power consumption - may increase in the near future, along with the increasing number of cores configured. General-purpose graphics processor units (GPGPUs) based clusters are attractive in obtaining orders of magnitude performance improvement with relatively low cost.

Current prevailing programming models, like MPI and OpenMP, have not supported heterogeneous platforms yet. Programming for the current dominant heterogeneous systems equipped with general-purpose multi-core processors and NVIDIA GPUs, programmers typically first program in OpenMP, Pthreads or other suitable programming interfaces, then manually identify code regions for GPU

K. Cooper, J. Mellor-Crummey, and V. Sarkar (Eds.): LCPC 2010, LNCS 6548, pp. 151–165, 2011.

acceleration, partition computation to GPU threads, and generate optimal accelerated code in OpenCL (or CUDA which still dominates). Besides, users have to manipulate data transfers and specify architecture-specific parameters in code optimizations. For GPGPU clusters, hybrid MPI and OpenCL is a suggested programming method, but it is not easy since both of them are low level programming interfaces.

Partitioned Global Address (PGAS) languages are successful in capturing non-uniform memory access feature of clusters and provide programmability very close to shared memory programming. There are a number of PGAS language extensions available. UPC, Co-array Fortran and Titanium are the dialects for C, Fortran and Java, respectively. For UPC, there are open source implementations as Berkeley UPC, GCC-based Intrepid UPC and MTU UPC, and commercial ones such as Cray UPC, HP UPC, IBM XL UPC and etc. Previous work [15,16] demonstrated that UPC delivers flexible programmability, high performance, and portability across different platforms. UPC encourages global view programming with upc_forall, and it lays the basis for hierarchical computation partitioning.

We make three contributions in this paper. Firstly, we extend UPC with hierarchical data distribution and advance the semantics of *upc_forall* to support multi-level work distribution. Affinity expression with hierarchical data distribution indicates how to map the corresponding iterations to an implicit thread hierarchy. Code portability can be gained across traditional HPC clusters and GPU clusters. Secondly, we investigate important compiler analysis and runtime supports for these language extensions. Affinity-aware loop tiling is put forward for computation partitioning, array region analysis is used for inter-UPC communication and explicit data transfer for GPUs, and the unified data management is introduced to the runtime system to optimize data transfer and memory layout on different memory modules of CPUs and GPUs. Thirdly, we implement several memory-related optimizations for GPUs. Experimental results show that the UPC extensions have better programmability than MPI+CUDA approach, and the integrated compile-time and runtime optimization is effective to achieve good performance.

The rest of the paper is organized as follows. Section 2 presents our language extension. Section 3 outlines the compiling and runtime framework for GPU clusters, including affinity-aware loop tiling, memory layout optimization and the unified data management system. Section 4 presents our experimental methodology and the results. Section 5 discusses related work, and Section 6 contains our conclusion and future work.

2 Extending UPC with Hierarchical Parallelism

In this section, we describe our revised execution model to UPC and the extension on hierarchical data distribution. We also explain how to exploit massive parallelism upon the upc_forall construct.

2.1 UPC's Execution Model on GPGPU Clusters

Standard UPC only has flat SPMD parallelism among UPC threads. In order to match the hierarchical thread organization of GPU, we introduce implicit threads. Implicit

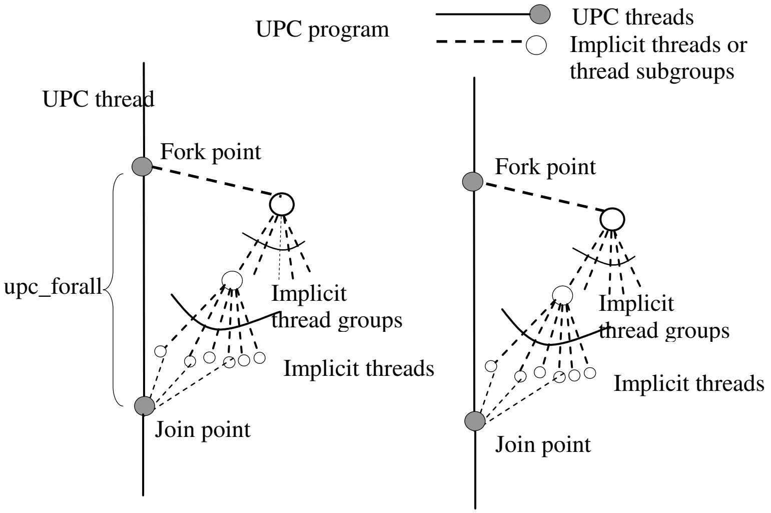


Fig. 1. Extension on UPC's execution model

threads are created in the fork-join style by each UPC thread at *upc_forall*, and are organized in groups similar to the CUDA thread blocks. Implicit threads are created and managed by runtime, and programmers do not have much control over them, except their granularity and organization.

Figure 1 shows the extension on UPC's execution model on GPU cluster. Solid lines represent the original UPC threads that exist from the very beginning of a UPC program. UPC threads have their thread identities and are synchronized using barriers and locks. Implicit threads represented as dotted lines, are created at the entry of upc_forall loops, then synchronize with the UPC thread that forks them at the join point, and then disappear. So, it is a mix between SPMD and fork-join model. Thread group is the form of organizing implicit threads. Implicit threads run completely on GPUs, and the original UPC threads do not participate in the computation on GPU.

In our UPC execution model, the organization of implicit threads is determined by the affinity expression of *upc_forall*. Since there may be multiple *upc_forall*s in a UPC program, the number and the organization of implicit threads varies during the program execution.

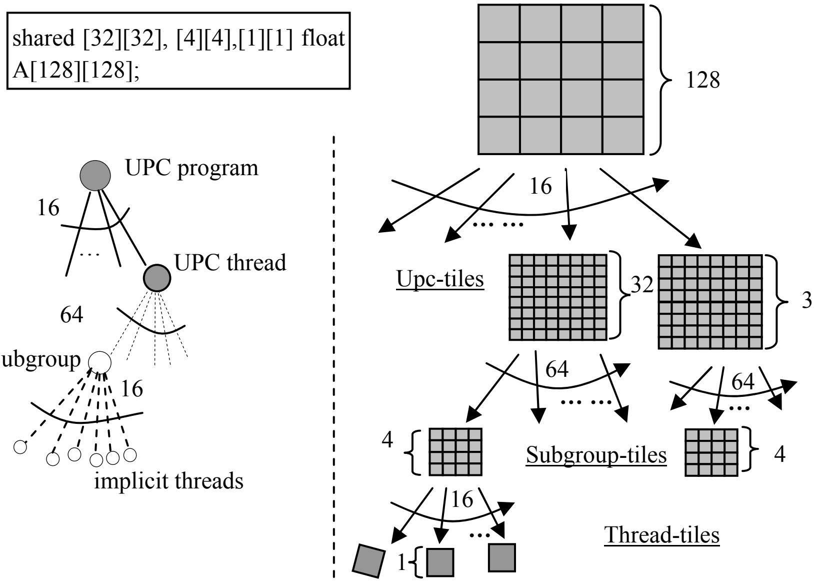
2.2 Hierarchical Data Distribution

UPC maintains the partitioned global address space for different UPC threads in the clusters. We introduce hierarchical data distribution into UPC allowing users to map data to a hierarchical machine abstraction of GPU clusters.

We adopt the multi-dimensional shared array data distribution extension as in [7]. Upon this, a shared array is decomposed into a tree of data tiles and mapped to a thread hierarchy. The syntax is as follows.

shared $[ub_1][ub_2]...[ub_n]$, $[sb_1][sb_2]...[sb_n]$, $[tb_1][tb_2]...[tb_n]$ ***<type>*** $A[d_1][d_2] ... [d_n]$;

Here, the sequence of $[ub_1][ub_2]...[ub_n]$ is the layout qualifier, describing the shape of a data tile. This means that array A is decomposed into data tiles sizing $\prod_{1\le i\le n}(ub_i)$ at the first level, and these data tiles are distributed to UPC threads in lexicographic order and in block-cyclic manner. We call these data tiles, *upc-tiles*. The second and third level layout qualifiers are separated using commas. Each upc-tile is further split into smaller tiles, whose number is $\prod_{1\le i\le n}\lceil ub_i/sb_i\rceil$. These smaller data-tiles are called *subgroup-tiles*, and are also arranged in lexicographic order. Similarly, each *subgroup-tile* is again split into data tiles sizing $\prod_{1\le i\le n}(tb_i)$, which are called *thread-tiles*, and this further set up father-child relationship between subgroup-tiles and thread-tiles. From this father-child relationship, each data distribution implies a tree of data-tiles.



```
shared [32][32], [4][4],[1][1] float
A[128][128];
```

(a) Implicit thread tree

(b) Tree of data-tiles

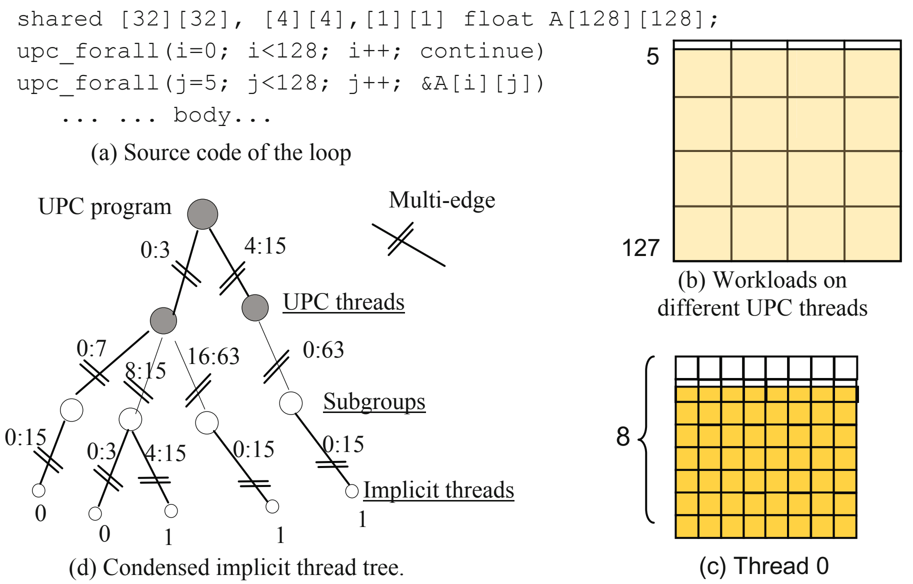
Fig. 2. Two hierarchies implied by the same data distribution

Figure 2 illustrates the two hierarchies implied by a three-level data distribution. The tree of data-tiles is given in Figure 2b, while Figure 2a illustrates the related thread tree. In this example, there are 16 sub-trees of implicit threads, and four such sub-trees will map to the same UPC thread if THREADS=4.

The meaning of data affinity is different on the three thread levels. Level-1 data tiles are physically distributed to UPC threads. The data tiles on the other levels do not indicate physical data distribution. The computation granularity of implicit threads is decided by the leaf thread-tiles, while subgroup-tiles decide the grouping of implicit threads, and indicate that each subgroup can share a logical shared memory.

2.3 Loop Partitioning to Implicit Thread Tree

We leverage the global view loop construct *upc_forall* to express hierarchical parallelism. Its semantics are extended in several ways. Firstly, *upc_forall* is further restricted to genuine independent loops, while the UPC standard only requires that there should be no data dependences between UPC threads in this loop. Secondly, for affinity expression with pointer-to-shared type, the semantics of workload partitioning is extended to a thread hierarchy. Affinity expression actually has many instances during the execution of the loop nest. For a parallel loop with pointer-to-shared type affinity expression, a loop iteration will be mapped to a certain (implicit) thread if the corresponding instance of the affinity expression has affinity to that (implicit) thread. Since each distributed array decides an implicit thread tree, there is definitive mapping between the loop iterations of a parallel loop and the related implicit thread tree.



```
shared [32][32], [4][4],[1][1] float A[128][128];
upc_forall(i=0; i<128; i++; continue)
upc_forall(j=5; j<128; j++; &A[i][j])
    ... ... body...
```

(a) Source code of the loop

(b) Workloads on different UPC threads

(d) Condensed implicit thread tree.

(c) Thread 0

Fig. 3. Workload distribution of an upc_forall

Figure 3a presents the source code of a parallel loop nest. In figure 3b, the shadowed area indicates the whole iteration space, and this shows the workload distribution on all UPC threads, so if there are 16 UPC threads, each gets one block, Figure 3c shows the workload distribution on thread 0 where each “box” is a subgroup. And figure 3d describes workload distribution on the condensed implicit thread tree. Here, weight is given to each leaf node to indicate if it contains real workload. To make the tree concise, two nodes are merged when they are isomorphic and have the same father. Two nodes are isomorphic if they have the same amount of work. Node merging brings about multi-edges, labeled with integer intervals to represent those children merged. The children of each node have the serial numbers ranging from 0 to n-1.

Of course, when the affinity expression is of integer type or the corresponding shared array has only one level of data distribution, the workload partitioning is compatible to standard UPC.

According to a machine configuration file, compiler maps the implicit thread tree to concrete GPU platform. Basically, each implicit thread maps to one CUDA thread, and each subgroup maps to one thread block. When the size of the implicit thread tree surpasses that of the architectural constraints, compiler can choose between reporting an error and applying another level of loop tiling.

3 Implementation on GPU Clusters

We implement the compiling system in Berkeley UPC compiler v2.8.0. The source-to-source translator is extended to support hierarchical data distribution, affinity-aware loop tiling, DSM consistency maintenance and memory optimization for CUDA, the output source code is CUDA C source code. UPC runtime is extended with unified memory management between CPU and GPU, and data regrouping supports for GPU.

The translated CUDA C source code is compiled using nvcc compiler and linked with CUDA runtime, CUDA core library and UPC runtime. The object code runs on a GPGPU cluster in the same way as a traditional UPC application.

3.1 Overview of the Compiling System

A upc_forall loop nest is called *well-formed hierarchical parallel loop nest*, if it meets the following criteria, 1) its effective affinity expression is of pointer-to-shared type. 2) the related array to the affinity expression has three-level data distribution.3) each subscript of the affinity expression is unary affine function.

For well-formed hierarchical parallel loop nests, legality analysis rules out loops which are invalid to apply loop tiling or include operations that platform does not support, such as I/O operations etc. There is a definite mapping between the iteration space and the CUDA thread topology. From inside out, the loop index of upc_forall loops corresponds to the x, y, and z dimensions of the grid topology respectively. In general, compiler applies three-level loop tiling, level one is used for affinity expression promotion, the second and the third level tiling map the loop iteration space to implicit thread tree. If the memory is insufficient on GPU, there needs another level of loop tiling between the first and the second level loop tiling.

The compiler has three tasks, 1) Try affinity-aware multiple-level loop titling. 2) Apply array region analysis for GPU's heap allocation, and apply exposed array region analysis for communication generation which is launched in the runtime. 3) Data layout analysis for GPUs, including inter-thread data reuse analysis for shared memory, and data layout transformation analysis for memory coalescing. Data movement of these data transformations is carried out in the runtime system.

Runtime system is comprised of three parts, unified data management, inter-UPC communication and data regrouping for GPUs.

Unified data management takes care of the local memory space of each UPC thread. It is a simple DSM system, scheduling data transfer between CPUs and GPUs.

The data granularity of shared data is upc-tile, while the granularity of private data is the whole array. Shared heap is initialized on both GPU and CPU by the runtime system at startup time. Further, the runtime system analyzes the exposed array region that compiler provides to see if inter-UPC communication is needed, and it will launch inter-UPC communication if necessary and shuffle data on the GPU to get a contiguous memory region. Here, inter-UPC messages are private data, and will be disposed of after use.

Data regrouping includes array transposition and structure splitting, and is realized in CUDA code as a runtime library routine.

3.2 Affinity-Aware Loop Tiling Transformation

In this section, we introduce affinity-aware loop tiling to map well-formed hierarchical parallel loop nests to the implicit thread tree. Affinity-aware loop tiling assures that each tile-element-loop always has a unique thread affinity.

In a well-formed hierarchical parallel loop nest, an *upc_forall* loop is called an *effective parallel loop*, if its loop index appears in a subscript of the affinity expression. Otherwise, this upc_forall loop will not be partitioned among different UPC threads, and basically it is a serial loop. The subscript of an affinity expression related to the effective parallel loops are called *effective subscripts*.

Effective *upc_forall* loops will be partitioned and distributed to an implicit thread tree according to the data distribution of the related affinity expression. For each effective *upc_forall* loop, an affine loop transformation *T* can be introduced to make the effective subscript of the affinity expression have the form of unit coefficient and zero offset, thus forming a new iteration space. If represented as a bounding box, the new iteration space is consistent with the data space decided by the effective subscripts of the affinity expression. Traditional loop tiling then can be applied regularly on the new iteration space such that iterations inside each tile-element-loop have the same affinity attribute. Finally, guard conditions are sank into the innermost loop body. And this is called *affinity-aware loop tiling transformation*. An example is given in Figure 4.

3.3 Memory Optimizations for CUDA

GPUs have complex memory hierarchy. Shared memory is suitable to store data with high temporary locality, in order to reduce the number of global memory access.

For each array, an *average, temporary reuse degree* is computed. The problem is formulated as a 0/1 binpack problem, if regarding reuse degree and the size of referenced regions as the benefit and the cost respectively. In our implementation, we use an exhaustive method.

Dynamic data transformation is introduced to change array's access pattern. Array transposition and structure splitting are two kinds of data transformations that we consider. Dynamic data transformation is carried out in the runtime system using optimized GPU code. About profitability, coalescing references and non-coalescing references are counted respectively, and the extra cost on dynamic data transformation is included, so the profit of a new memory layout can be estimated.

```
shared [32][32], [4][4],[1][1] float A[128][128];
 … …
upc_forall(i=6; i<128; i++; continue)
upc_forall(j=0; j<128; j++; &A[i-1][ j])
  ... ... F[i][j]...
 Step1:  iteration space transformation, to make affinity expression consistent with data space
 upc_forall(i=5; i<127; i++; continue)
 upc_forall(j=0; j<128; j++; &A[i][j])
  ... ... F[i+1][j]...  //transformation
 Step2:  three level tiling, actually two level tiling here
 for (iu=0; iu<128; iu=iu+32)
 for (ju=0; ju<128; ju=ju+32)
   if (has_affinity(MYTHREAD, &A[iu][ju])) {//upc thread affinity
        for (ib=iu ; ib<min(128, iu+32); ib=ib+4)
        for (jb= ju; jb< min(128, ju+32); jb=jb+4)
            for (i=ib; i<min(128,ib+4); i=i+1)
            for (j=jb; j<min(128,jb+4); j=j+1)
                if(i>=5 && i<127)  //sink the guard condition here!
                    ... ... F[i+1][j]... ;
   }//of upc thread affnity
 Step 3:  spawn fine-grained threads
 dim3 threads (4,4);
 dim3 blocks (8,8);
 for (iu=0; iu<4; iu=iu+1)
 for (ju=0; ju<4; ju=ju+1)
   if (has_affinity(MYTHREAD, &A[iu][ ju])) {// upc thread affinity
         …dsm_read… F[iu+1:min(128, iu+32)][ ju: min(127,ju+31)] // for exposed region
        $F_{offset}$= compute_device_address(F, iu+1, ju, … …)
        comput<<<blocks,threads>>>( $F_{offset}$,…);
 }

 __global__ void comput(float* F,… …){
 … …
x = blockIdx.x * blockDim.x + threadIdx.x;
y = blockIdx.y * blockDim.y + threadIdx.y;
pitch= blockDim.y*gridDim.y; // for the linear address on device memory
   if(x>=5 && x<127)
      … F[(x+1)+y*pitch]…   ;
 }
```

Fig. 4. Illustration on how to apply affinity-aware loop tiling

It should be noted that layout transformation of an array which appears in the affinity expression of some upc_forall, actually changes the thread hierarchy that users specified by data distribution. If the compiler strictly complies with the user's intention, such data optimizations should be disabled.

3.4 Unified Data Management

The section focuses on data transfer scheduling in the runtime system. It manages UPC thread's local memory space, removing redundant data transfer and redundant data layout transformation.

A simple data transfer strategy is as follows. According to data flow analysis, a CPU-to-GPU data transfer is needed for any upward exposed use, and a GPU-to-CPU data transfer is needed for any downward exposed definition. But, this simple strategy may incur lots of redundant data transfer. In Figure 5 the data transfers between kernel1 and kernel2 are redundant. And the other two data transfers can be promoted outside of the *time* loop, because there is no data dependence carried by the *time* loop.

```
a = …
for (time=… … ){
    kernel1(/*in*/a, /*out*/ b);
    kernel2(/*in*/b, /*out*/c);
}
… = c;
```

Fig. 5. Example for redundant data transfer removal

This optimization can be realized at compiler-time or runtime. Compiler analysis needs to apply data flow analysis on array regions, and inter-procedural analysis is demanded in real applications. We adopt the runtime approach, maintaining the valid copies and deleting unnecessary data transfers between GPU and CPU. At each consistency maintenance point, status of the related memory regions is updated or a data transfer is launched when the current copy is stale. Currently we do not support concurrency between CPU and GPU. Consistency maintenance points are the boundaries between CPU codes and GPU codes.

For the example in Figure 5, the illustrative code of DSM-instrumentation is in Figure 6. Here *valid()* is used to check the validity of some data copies on a certain device, and *set_valid/invalid* will change the validity status of data copies. For each write operation, *set_exclusive_valid* will make the directory entry valid on a certain device, but invalid on all other devices. We can deduce that in this *time* loop there are only two data transfers, and the redundant data transfer is optimized by the runtime system.

It should be noted that there is a special argument TRANSPOSE in one *DSM* library call. It is used to declare an expected memory layout for the following kernel code. This data layout transformation also has similar optimization opportunity as redundant transfer removal. The naïve method is to change the layout of a certain array before the CUDA kernel call and restore back to the original one when CUDA kernel finish. We can delete the redundant data transformation using a demand-driven

```
a[…]= …
set_exclusive_valid(a, HOST);// multiple Devices
for (time=… … ) {
   if (!valid(a, this_dev)) {//
      device_a= upcr_dsm_input_shared
                (a,……,TRANSPOSE);
      set_valid(a, this_dev);
   }
   device_b=… …
   kernel1(in: device_a, out: device_b);
   set_exclusive_valid(b, this_dev);
   if (!valid(b, this_dev)) {
      upcr_dsm_input_private(b,device_b,… …);
      set_valid(b, this_dev);
   }
   kernel2(in: device_b, out: device_c);
   set_exclusive_valid(c, this_dev);
} //end recursive loop
if (!valid(c, HOST)) {
   cudaMemcpyD2H(c, device_c);
   set_valid(c, HOST);
}
……=c[2]
```

Fig. 6. Data transfer code inserted for the DSM system

approach. In each directory entry, an extra field is added to record the current memory layout. And at the entry point of each kernel call, data transformation will be triggered only when the current layout does not meet the expectation.

4 Experimental Results

The benchmarks we use are shown in table 1, four of them from Parboil, one from SPEC CPU 2006, and one from China CUDA Campus Programming Contest. We recode them into two UPC versions, one is in standard UPC, and the other is with hierarchical data distribution. The original version of Parboil and n-body can be regarded as highly tuned CUDA programs.

4.1 Programmability Evaluation

Conceptually, users only need to decide a proper hierarchical data distribution for certain shared arrays, when porting a UPC program to a GPU cluster. So, the porting process is easy.

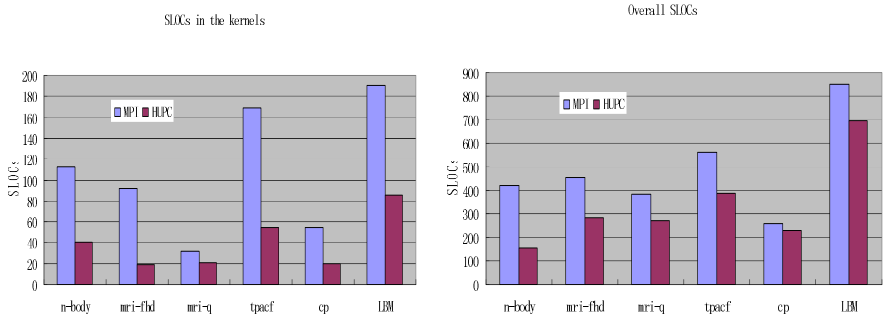
We use source lines of code (SLOCs) in this section to quantify the programmability of this UPC extension. The counterpart is MPI/CUDA, and is denoted as MPI in Figure 7.

Table 1. Applications used in the experiments

Benchmarks	Description	Original language	Application field	Source
N-body	n-body simulation	CUDA +MPI	Scientific computing	CUDA campus programming contest 2009[14]
LBM	Lattice Boltzmann method in computational fluid dynamics	C	Scientific computing	SPEC CPU 2006
CP	Coulombic Potential	CUDA	Scientific computing	UIUC Parboil Benchmark
MRI-FHD	Magnetic Resonance Imaging FHD	CUDA	Medical image analysis	UIUC Parboil Benchmark
MRI-Q	Magnetic Resonance Imaging Q	CUDA	Medical image analysis	UIUC Parboil Benchmark
TPACF	Two Point Angular Correlation Function	CUDA	Scientific computing	UIUC Parboil Benchmark

The source lines of code in the kernel part of each program are shown in Figure 7a. We can easily see that UPC versions have remarkably less SLOCs than that of MPI/CUDA. Programmers do not need to transform the array subscripts in kernel functions, or to manage shared memory explicitly, or write code on data regrouping for memory coalescing, and all of the above optimizations are handled by the compiler.

Figure 7b compares the overall SLOCs between the two programming methods, and the same benchmarks are used. For the code outside of the kernel parts, UPC versions also save code lines on separated memory management, thread hierarchy declaration, kernel call invocation and inter-node MPI communications.



(a) SLOCs of the kernel parts (b) Overall source lines of code

Fig. 7. Source lines of codes compared with MPI/CUDA

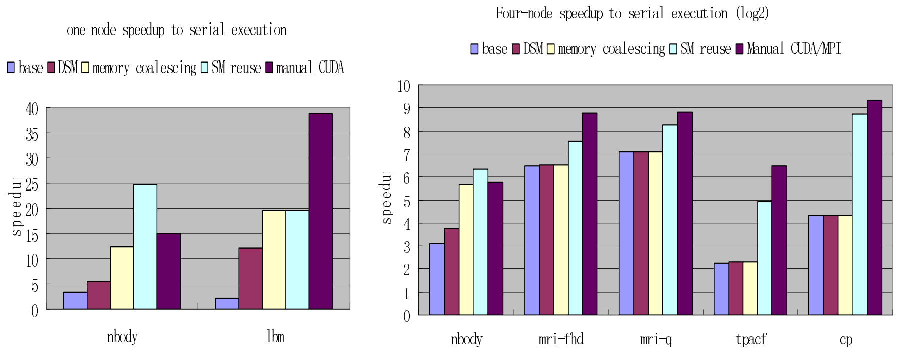
4.2 Performance Evaluation

In this section, we evaluate the performance of our UPC compiler. Our experimental platform is a 4 node GPU cluster connected through gigabyte Ethernet. Each node has two 2.5GHz dual-core AMD Opteron Processor 880 and NVIDIA GeForce 9800 GX2

(CUDA 1.1), having 2G main memory. The latter is a dual GPU card, each having 16 multiprocessors with a clock rate of 1.5GHz, 512M global memory, and 64K constant memory. Each multiprocessor is equipped with 8 SIMD processing units, 16K shared memory and 8192 registers. The compiler we use is GCC3.4.6 and NVCC2.2.

The serial performance is gained using standard UPC programs running with one UPC thread. We evaluate the speedups of UPC programs on our UPC compiler and compare them with hand-tuned CUDA versions. In this experiment, we always deploy one UPC thread. As shown in section 3, different optimizations are implemented in our UPC compiler, so we also illustrate their respective contribution to the overall performance.

Figure 8a shows the overall speedups on a single GPU node, and we choose two benchmarks, n-body and LBM. The serial performance is treated as 1. We consider four cases in the UPC implementation, basic code generation for GPU, unified data management (DSM), memory coalescing optimization and shared memory optimization. The optimizations are enabled one after another. Both the benchmarks benefit from DSM management, and lbm has higher speedup for referencing larger arrays and having large iteration count. Both n-body and lbm benefit from structure splitting, while array transpose contributes only to n-body. The compiler finds many inter-thread data reuses in n-body but none in lbm. For n-body, the three optimizations gain improvement of 64%, 121% and 102% respectively, and eventually surpass the hand-tuned version. The reason is that we exploit more data reuses through shared memory management, while the hand-tuned version misses the opportunity. For LBM, 50% of the hand-tuned performance is reached.



(a) On single GPU node (b) On a 4-node GPU cluster

Fig. 8. Overall performance speedups

We also test the benchmarks on the 4-node GPU cluster, and the results are shown in Figure 8b. All the performance data are compared to the serial CPU performance, and all the speedups are logarithm to base 2. Similarly to the single node experiments, we deploy one UPC thread on each node. The benchmarks include n-body and 4 kernels from parboil. The performance of n-body is similar to that on a single GPU node. In Parboil, the compiler finds no redundant data transfer or optimization opportunity for memory coalescing, but there exists large number of temporary data reuses. Shared memory management contributes an average speedup of 5.71, and the overall performance reaches 53% of the highly optimized versions on average. Hand tuned versions use constant memory other than shared memory to cache reused data, and it might lead to better performance. Further our simple shared memory strategy limits the concurrency for allowing less thread blocks sharing the same shared memory.

5 Related Work

CUDA targets general purpose computing, and provides relative good programmability. To ease programming on CUDA, many tools have been developed. CUDA-lite [5] performs code transformation of coalescing global memory access based on programmer's annotations. hiCUDA [6] also leverages annotations to partition parallel loops among CUDA blocks and threads. JCUDA [7] is a programming interface which makes it possible to invoke CUDA kernels in JAVA. The JCUDA compiler has a basic heterogeneous memory management strategy to schedule data transfer between CPU and GPU. In [12], compiler techniques were developed targeting naïve GPU kernel functions, it identifies the memory access patterns, and realizes different optimizations for memory bandwidth enhancement, data reuse and parallelism management.

In [8] a compiler framework is introduced to translate standard OpenMP shared-memory programs into CUDA-based GPGPU programs, and several key transformation techniques are exploited for efficient GPU global memory access. In industry, there have been some directive based programming efforts for GPGPU architectures. Some enterprise compilers, such as PGI Accelerator [9] and HMPP Workbench [10], provide directives indicating the code region that will be executed on the accelerator, or defining the relationship of input and output parameters between code regions, or related to specific code generation optimizations and etc.

For PGAS languages, there are other language extensions on hierarchical parallelism or on thread grouping. GWU [13] puts forward an extension on nested parallelism, and IBM suggests instant team [4] for collective operations in a data-centric approach. In order to allow optimizations on collective operations and provide adequate support for libraries, team is suggested to be added to Co-Array Fortran [14].

Sequoia [1] exposes data movement along the memory hierarchy and allows users to tune the data layout and data transfer. Sequoia supports uniprocessors, SMPs, Cell, clusters, and even GPUs[17]. Hierarchical place tree (HPT) [2] is introduced into X10 for both computation and data mapping in parallel systems with hierarchical memory hierarchy. In sequoia and X10, hierarchical parallelism is expressed through nested parallel constructs (or recursive methods). HTA [3] provides a new data type (hierarchical tiled array) to existing sequential languages to express multiple levels of tiling for locality and parallelism, and parallel operators are used to express hierarchical parallelism. Besides uniprocessors, SMPs and clusters, HTA plans to support GPUs too.

6 Conclusion and Future Work

In this paper, we extend UPC with hierarchical data distribution and introduce the implicit thread hierarchy in addition to UPC threads by leveraging the upc_forall construct. We investigate important compiler analysis as affinity-aware loop tiling, and implement unified data management on each UPC thread to optimize data transfer and data layout on different memory modules. We implement the compiler and the runtime system, and the experimental results show that UPC has better programmability than the mixed MPI/CUDA approach. We also demonstrate that the integrated compile-time and runtime optimization is effective to achieve good performance on GPU clusters.

There are several directions for further work. Firstly, UPC needs to be further extended to support collaboration between CPU and GPU to better utilize the computing resources. Secondly, both compiler and runtime support should be improved. Asynchronous data transfers can be used to realize communication overlap, and constant memory should be exploited along with shared memory optimization. At last, irregular applications should be supported. Currently we only support array-based applications with affine access patterns, but many real applications have irregular, dynamic data structures, and hierarchical data distribution should be expanded to non-array data structures.

Acknowledgements. This research was supported in part by the National Fundamental Research Program of China (2005CB321602), the National Natural Science Foundation of China (60970024, 60633040), the National Science and Technology Major Project of China (2009ZX01036-001-002), the Innovation Research Group of NSFC (60921002) and a gift grant from Intel.

We would like to thank the reviewers for valuable comments and suggestions. We are grateful to Professor Huiyang Zhou at University Central Florida for shepherding the paper.

References

1. Fatahalian, K., Knight, T., Houston, M., Erez, M., Horn, D., Leem, L., Park, H., Ren, M., Aiken, A., Dally, W., Hanrahan, P.: Sequoia: Programming the Memory Hierarchy. In: Proceedings of Supercomputing 2006 (November 2006)
2. Yan, Y., Zhao, J., Guo, Y., Sarkar, V.: Hierarchical place trees: A portable abstraction for task parallelism and data movement. In: Gao, G.R., Pollock, L.L., Cavazos, J., Li, X. (eds.) LCPC 2009. LNCS, vol. 5898, pp. 172–187. Springer, Heidelberg (2010)
3. Bikshandi, G., Guo, J., Hoeflinger, D., Almasi, G., Fraguela, B.B., Garzarán, M.J., Padua, D., von Praun, C.: Programming for parallelism and locality with hierarchically tiled arrays. In: PPoPP, New York, USA, March 29-31 (2006)
4. Nishtala, R., Almasi, G., Cascaval, C.: Performance without pain = productivity: data layout and collective communication in UPC. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2008 (2008)
5. Ueng, S., Lathara, M., Baghsorkhi, S.S., Hwu, W.W.: CUDA-lite: Reducing GPU programming complexity. In: Amaral, J.N. (ed.) LCPC 2008. LNCS, vol. 5335, pp. 1–15. Springer, Heidelberg (2008)

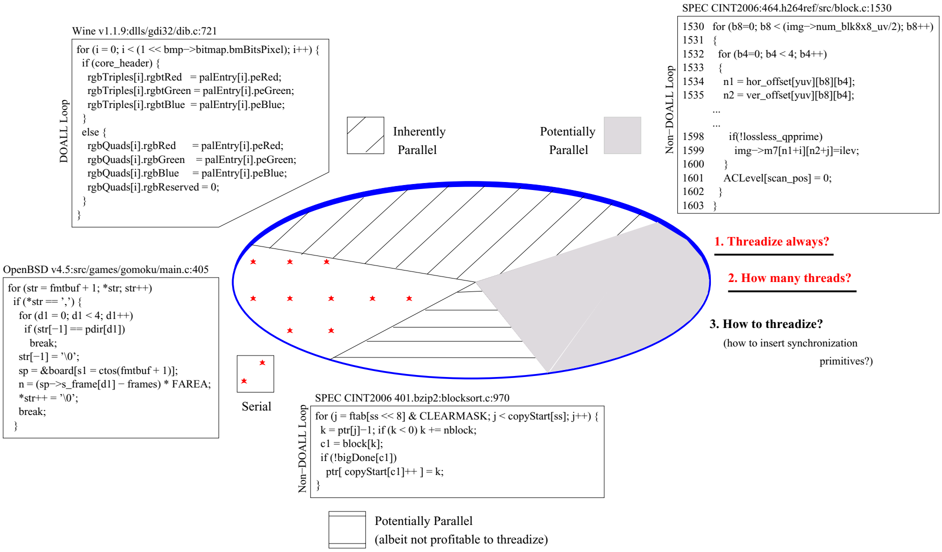
6. Han, T.D., Abdelrahman, T.S.: hiCUDA: a high-level directive-based language for GPU programming. In: Workshop on General Purpose Processing on Graphics Processing Units (GPGPU), pp. 52–61 (March 2009)
7. Yan, Y., et al.: JCUDA: a Programmer-Friendly Interface for Accelerating Java Programs with CUDA. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 887–899. Springer, Heidelberg (2009)
8. Lee, S., Min, S.-J., Eigenmann, R.: OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pp. 101–110 (February 2009)
9. The Portland Group. PGI Fortran & C Accelerator Programming Model (March 2010), http://grape.pgroup.com/lit/whitepapers/pgi_accel_prog_model_1.2.pdf
10. http://www.caps-entreprise.com/fr/page/index.php?id=49&p_p=36
11. NVIDIA CUDA, China campus programming contest (2009), http://cuda.csdn.net/contest/pro
12. Yang, Y., Xiang, P., Kong, J., Zhou, H.: A GPGPU Compiler for Memory Optimization and Parallelism Management. In: The ACM SIGNPLAN 2010 Conference on Programming Language Design and Implementation, PLDI 2010 (June 2010)
13. Serres, O., Kayi, A., Anbar, A., El-Ghazawi, T.: A UPC Specification Extension Proposal for Hierarchical Parallelism. In: The 3rd Conference on Partitioned Global Address Space Programming Models, Virginia, USA (October 2009)
14. Numrich, R.: Teams for Co-Array Fortran. In: The 3rd Conference on Partitioned Global Address Space Programming Models, Virginia, USA (October 2009)
15. Barton, C., Casçaval, C., Almási, G., Zheng, Y., Farreras, M., Chatterje, S., Amaral, J.N.: Shared memory programming for large scale machines. In: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14 (2006)
16. Husbands, P., Iancu, C., Yelick, K.: A performance analysis of the Berkeley UPC compiler. In: Proceedings of the 17th Annual International Conference on Supercomputing, San Francisco, CA, USA, June 23-26 (2003)
17. Bauer, M., Clark, J., Schkufza, E., Aiken, A.: Sequoia++ User Manual, http://sequoia.stanford.edu/

How Many Threads to Spawn during Program Multithreading?

Alexandru Nicolau[1] and Arun Kejariwal[2]

[1] University of California, Irvine
Irvine, CA 98612, USA
[2] Yahoo! Inc.
Sunnyvale, CA 94089, USA

Abstract. Thread-level program parallelization is key for exploiting the hardware parallelism of the emerging multi-core systems. Several techniques have been proposed for program multithreading. However, the existing techniques do not address the following key issues associated with multithread execution of a given program: (a) Whether multithreaded execution is faster than sequential execution; (b) How many threads to spawn during program multithreading. In this paper, we address the above limitations. Specifically, we propose a novel approach – **T-OPT** – to determine how many threads to spawn during multithreaded execution of a given program region. The latter helps to check under-subscribing and over-subscribing of the hardware resources. This in turn facilitates exploitation on higher level of thread-level parallelism (TLP) than what can be achieved using the state-of-the-art. We show that, from program dependence standpoint, use of larger number of threads than advocated by the proposed approach does not yield higher degree of TLP. We present a couple of case studies and results using kernels, extracted from open source codes, to demonstrate the efficacy of our techniques on a real machine.



Wine v1.1.9:dlls/gdi32/dib.c:721

```
for (i = 0; i < (1 << bmp−>bitmap.bmBitsPixel); i++) {
 if (core_header) {
  rgbTriples[i].rgbtRed   = palEntry[i].peRed;
  rgbTriples[i].rgbtGreen = palEntry[i].peGreen;
  rgbTriples[i].rgbtBlue  = palEntry[i].peBlue;
 }
 else {
  rgbQuads[i].rgbRed      = palEntry[i].peRed;
  rgbQuads[i].rgbGreen    = palEntry[i].peGreen;
  rgbQuads[i].rgbBlue     = palEntry[i].peBlue;
  rgbQuads[i].rgbReserved = 0;
 }
}
```

SPEC CINT2006:464.h264ref/src/block.c:1530

```
1530  for (b8=0; b8 < (img−>num_blk8x8_uv/2); b8++)
1531  {
1532    for (b4=0; b4 < 4; b4++)
1533    {
1534      n1 = hor_offset[yuv][b8][b4];
1535      n2 = ver_offset[yuv][b8][b4];
        ...
        ...
1598        if(!lossless_qpprime)
1599          img−>m7[n1+i][n2+j]=ilev;
1600      }
1601      ACLevel[scan_pos] = 0;
1602    }
1603  }
```

OpenBSD v4.5:src/games/gomoku/main.c:405

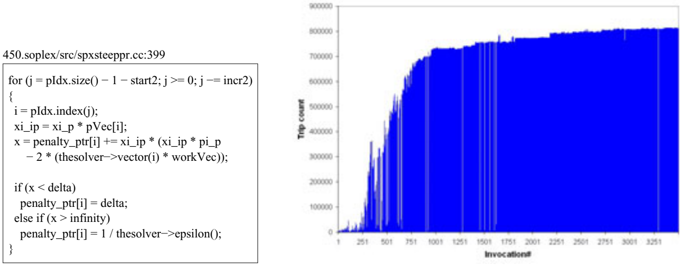
```
for (str = fmtbuf + 1; *str; str++)
 if (*str == ',') {
  for (d1 = 0; d1 < 4; d1++)
   if (str[−1] == pdir[d1])
    break;
  str[−1] = '\0';
  sp = &board[s1 = ctos(fmtbuf + 1)];
  n = (sp−>s_frame[d1] − frames) * FAREA;
  *str++ = '\0';
  break;
 }
```

SPEC CINT2006 401.bzip2:blocksort.c:970

```
for (j = ftab[ss << 8] & CLEARMASK; j < copyStart[ss]; j++) {
 k = ptr[j]−1; if (k < 0) k += nblock;
 c1 = block[k];
 if (!bigDone[c1])
  ptr[ copyStart[c1]++ ] = k;
}
```

Fig. 1. Loop Threadization Space

K. Cooper, J. Mellor-Crummey, and V. Sarkar (Eds.): LCPC 2010, LNCS 6548, pp. 166–183, 2011.



450.soplex/src/spxsteeppr.cc:399

```
for (j = pIdx.size() − 1 − start2; j >= 0; j −= incr2)
{
  i = pIdx.index(j);
  xi_ip = xi_p * pVec[i];
  x = penalty_ptr[i] += xi_ip * (xi_ip * pi_p
      − 2 * (thesolver−>vector(i) * workVec));

  if (x < delta)
    penalty_ptr[i] = delta;
  else if (x > infinity)
    penalty_ptr[i] = 1 / thesolver−>epsilon();
}
```

(a) (b)

Fig. 2. An example Non-DOALL loop with non-uniform trip count distribution

1 Introduction

Multi-core processors are becoming ubiquitous as exemplified by Intel's Core i7 [4] processor. The number of cores per chip is expected to rise in foreseeable future, as evidenced by the recently announced AMD's 6-core Istanbul, 8-core, 12-core Magny-Cours and the 16-core Interlagos processors [1] and Intel's 80-core Teraflops Research Chip. Harnessing the hardware parallelism of multi-core systems calls for program multithreading. Given that loops account for a large percentage of program runtime in general [25], we focus on loop threadization in the rest of this paper. Figure 1 presents an overview of loop threadization space (kernels shown in the figure are taken from industry standard SPEC CPU benchmarks [9] and other open source applications such as Wine v1.1.9 [11] and OpenBSD v4.5 [5]). We partition the loop threadization space into the following three classes:

- *Parallel loops*: A parallel loop is a loop in which there does not exist any loop-carried dependence [16] (parallel loops also referred to as DOALL loops [29]). Hence, different iterations of a DOALL loop can be executed on concurrent threads, where each thread in software is mapped on to a separate core, without any explicit thread synchronization. An example DOALL loop is shown on the top left of Figure 1. Note that write to rgbTriples and rgbQuads in the different iterations are independent of each other.
- *Serial loops*: A serial loop is a loop in which there exists one or more early exits in the loop body. An example serial loop is shown on the left hand side of Figure 1. Note that there are multiple early exits in the loop body.
- *Potentially parallel loops*: A potentially parallel loop is a loop in which there may *may* exist one or more loop-carried dependence(s). From hereon, we refer to such loops as non-DOALL loops. An example non-DOALL loop is shown on the top right of Figure 1. Note that there is a potential loop-carried

dependence based on the writes to `img->m7` in different iterations of the loop. Multithreaded execution of non-`DOALL` loops requires support for explicit thread synchronization in order to preserve the dependences between the different concurrent threads.

We sub-classify non-`DOALL` loops, akin to Wu et al. classification [44], into the following two categories:

- *Unprofitable to threadize*: The efficacy of multithreaded execution of a non-`DOALL` loop is dependent on a wide variety of parameters such as (but not limited to) threading overhead, trip count and amount of computation in the loop body. Multithreaded execution of loops belonging to this category result in performance slowdown w.r.t. serial execution. An example non-`DOALL` loop not suitable for threadization, owing to small amount of computation in the loop body, is shown at the bottom of Figure 1; in particular, multithreaded execution of the loop resulted in a slowdown w.r.t. sequential execution.
- *Profitable to threadize*: The remaining set of non-`DOALL` loops fall under this category. An example non-`DOALL` loop suitable for threadization is shown on the left hand side of Figure 1.

In the rest of the paper, we address threadization of non-`DOALL` loops unless stated otherwise explicitly.

Three questions arise in the context of threadization of non-`DOALL` loops, as enumerated on the right hand side of Figure 1.

1. **When to threadize?**
   As mentioned above, it is not profitable to threadize every non-`DOALL` loop. Furthermore, as illustrated in Figure 2, it may not be practical – owing to, for example, threading overhead, memory bandwidth pressure, to threadize every invocation of a given loop. Note that the trip count of the loop across different invocations is not uniform. The trip count profile was obtained using the `ref.mps` input data set and the `-m3500` parameter.
2. **How many threads to spawn?**
   The current industrial trend has been towards increasing number of cores per processor. However, in the context of a single application, using as many threads as the maximum number of cores will often not yield the best performance owing to, say, thread creation and switching overhead, thread synchronization and memory bandwidth issues. This is exemplified in Figure 3, wherein use of two threads on a quad-core processor yields best performance. Detailed experimental setup is given in Table 1. The application was executed with the reference data set. Furthermore, in the current case, the pattern in variation in performance with increasing number of threads is *consistent* across different compilers – we used the state-of-the-art Intel C++ compiler and the widely used gcc [3] compiler.

Table 1. Experimental setup corresponding to Figure 3

System	Quad Core Processor	
Processor	Intel®Xeon®CPU 1.86GHz	
L1 Cache	32 KB	
L2 Cache	4096 KB	
Memory	4038908 KB	
Compilers	gcc 4.4.0	Intel C++ Compiler v11.1
OS	Linux perflab139 2.6.9-34.ELsmp #1 SMP	

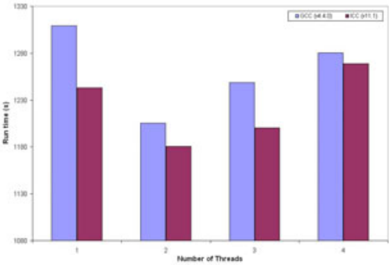


Fig. 3. Variation in run time of 470.lbm [8] with increasing number of threads

3. **How to threadize?**
   This question pertains to how to efficiently insert synchronization primitives such as `post` and `wait`.[1] In this regard, recently techniques based on code motion were proposed [33,34]. Existing techniques can be used to orchestrate upward and downward code motion of `post` and `wait` primitives respectively. Further, libraries such as Intel®Threading Building Blocks (TBB) [38] or directives such as OpenMP [6] can be used for threadization. We applied the optimizations proposed in [34] prior to the application of the techniques proposed in this paper.

In this paper we address the first two questions. In particular, the main contributions of the paper are:

❒ We propose a novel technique, referred to as **T-OPT**, to determine how many threads to spawn during threadization of an innermost loop. Further, we show that, from program dependence standpoint, use of larger number of threads than advocated by the proposed approach does not yield higher degree of TLP.

  Specifically, the **T-OPT** algorithm determines a kernel (discussed in the next section) such that the multithreaded execution of a given instance of the kernel does not require any synchronization between the concurrent threads. To the best of our knowledge, **T-OPT** is the first algorithm that can detect a synchronization-free kernel. The *span* of the **T-OPT** kernel – number of iterations the kernel spreads across – corresponds to the number of threads to spawn. We formally prove that a **T-OPT** kernel always emerges.

❒ We present a couple of case studies and results using kernels, extracted from open source codes, to illustrate the efficacy of the proposed techniques on a real machine.

The rest of the paper is organized as follows: Section 2 introduces the terminology used in the rest of the paper and background. Section 3 presents novel

[1] There exists a vast amount of literature, spanning over four decades, on the design and use of synchronization primitives, refer to [S1]-[S43] in [26].

techniques addressing the aforementioned questions. Case studies and results are presented in Section 4. An overview of related work is presented in Section 5. Finally, Section 6 concludes with directions for future work.

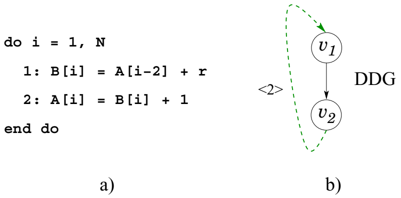
2 Terminology and Background

In this section, we introduce the terminology used in the rest of the paper and present a brief background. Given two operations u, v with a loop-carried dependence between them $u \rightarrow v$ [27], we say that u is the *source* and v is the *sink*. Next, we introduce a couple of definitions.

Definition 1. *Given a dependence graph $G(V, E)$, a path from an operation v_1 to an operation v_k is a sequence of operations $\langle v_1, \ldots, v_k \rangle$, such that $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \ldots, k$.*

Definition 2. *A* cycle *in a dependence graph is a path $\langle v_1, \ldots, v_k \rangle$ such that $v_1 = v_k$ and the path contains at least one edge. A cycle is* simple *if, in addition, $v_1, \ldots, v_{k-1}$ are distinct.*

A loop carried dependence in conjunction with intra-iteration dependences may form a simple cycle in the dependence graph. For example, in Figure 4, the intra-iteration dependence and the loop carried dependence, denoted by dashed arrow, between the operations v_1 and v_2 form a simple cycle. A loop has a *recurrence* if an operation in one iteration of the loop has a direct or indirect dependence upon the same operation from a previous iteration, e.g., in Figure 4, operations v_i^k and v_i^{k-2} constitute a recurrence, where v_i^k represents the i-th operation of the k-th iteration. The dependence distance (in number of iterations) [42,16] between v_2 and v_1 is shown within <>. In general, a recurrence may span several iterations, i.e., an operation v_i^k may be depend on an operation v_i^{k-j}, where $j > 1$. The existence of a recurrence manifests itself as a simple cycle in the dependence graph. Subsequently, a cycle refers to a simple cycle in a dependence graph.

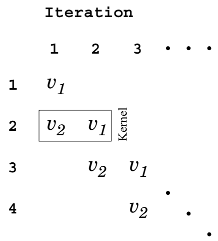


```
do i = 1, N
  1: B[i] = A[i-2] + r
  2: A[i] = B[i] + 1
end do
```

Fig. 4. Loop recurrences

Given that, in most cases, loops account for most of the run time of programs [25], several techniques for loop optimization have been proposed [43]. In the context of VLIW processors [24], modulo scheduling [37] and software pipelining [32,39,22,28,36] have been proposed. Similar techniques have been proposed for superscalar processors [23]. Software pipelined schedule corresponding to the loop shown in Figure 5. Note that the *span* of the kernel is 2.

In [12], Aiken and Nicolau proposed a software pipelining technique, referred to as **OPT**, for optimal parallelization of innermost loops. Arguably, in the context of loop threadization, one could threadize the kernel obtained via **OPT**.



	Iteration			
	1	2	3	. . .
1	v_1			
2	v_2	v_1	Kernel	
3		v_2	v_1	
4			v_2	. . .

Fig. 5. Software pipelined schedule corresponding to the loop shown in Figure 4 (a)

However, this does not necessary yield the best multithreaded performance (this is illustrated in the next section). We propose a technique to alleviate the above. Specifically, using the **OPT** kernel as the base,[2] the proposed technique, **T-OPT**, gears advanced TLP-centric code motion to eliminate the need the thread synchronization in the kernel. Additionally, the proposed technique helps to determine how many threads to use.

3 The "What" and "How"

In this section, we illustrate, with the help of a running example, the two problems we address in this paper and walk through the techniques we propose to address the same.

Let us consider the data dependence graph (DDG) shown in Figure 6, taken from [21]. We shall use this DDG as our running example in the rest of this section. The dashed arrows represent loop-carried dependences and the solid arrows represent intra-iteration data dependences. The data dependence distances are shown in angle brackets. Note that the following simple cycles [20] exist in the DDG shown in Figure 6:

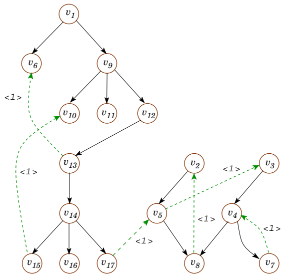


Fig. 6. A data dependence graph

$v_2 \rightarrow v_5 \rightarrow v_8 \rightarrow v_2$
$v_4 \rightarrow v_7 \rightarrow v_4$
$v_2 \rightarrow v_5 \rightarrow v_3 \rightarrow v_4 \rightarrow v_8 \rightarrow v_2$

3.1 Greedy Schedule

One of the straightforward ways to schedule the loop corresponding to DDG shown in Figure 6 is ASAP (As Soon As Possible) scheduling (also referred to as

[2] The rationale behind selecting **OPT** kernel as the base stems from the fact that, for innermost loops, **OPT** achieves an optimal schedule, subject to dependences and given enough cores. **OPT** does *not* account for synchronization cost in the case of asynchronous multithreaded execution.

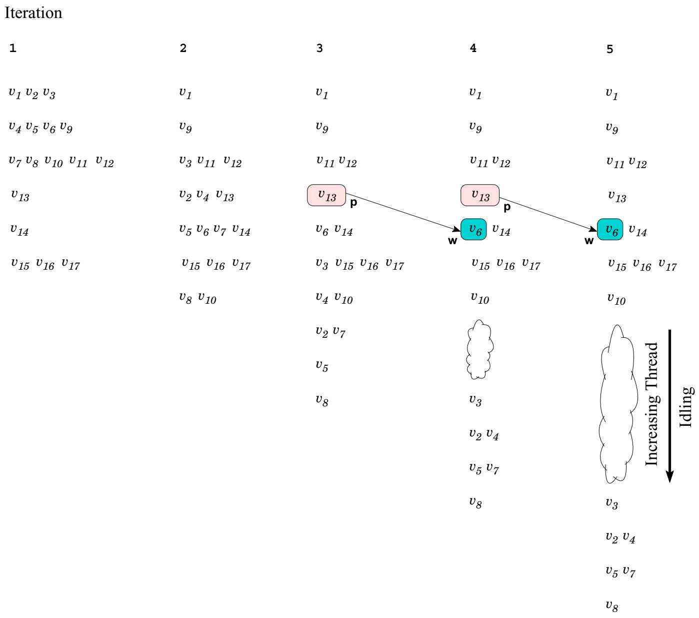


Fig. 7. Greedy schedule

greedy scheduling). Greedy schedule for the DDG in Figure 6 is shown in Figure 7 wherein each iteration is mapped on to a different thread. Given an iteration, operations placed at the same horizontal level can be executed in parallel on different functional units, subject to their availability. For example, in iteration 3, operations v_{11} and v_{12} can be executed in parallel. An iteration is mapped on to a thread as and when a thread become available. Data dependences between iterations are preserved with the help of synchronization primitives such as `post`, `wait` . For clarity purposes, only one pair of `post, wait` primitives are shown between consecutive iterations. The other data dependences between iterations i and $i+1$ are the following:

$v_{15} \longrightarrow v_{10}$
$v_{17} \longrightarrow v_5$
$v_8 \longrightarrow v_2$
$v_5 \longrightarrow v_3$
$v_7 \longrightarrow v_4$

Exploitation of TLP is limited by the need for thread synchronization. From Figure 7 we note that as the greedy schedule progresses, a pipeline bubble develops. This is due to the loop-carried dependence $v_5 \rightarrow v_3$. More importantly, we observe that the pipeline bubble lengthens as the schedule progresses. Over time, the threads get "aligned" as governed by loop-carried dependences; in other words, *thread pipelining* takes effect. The enlarging bubble in turn induces increased thread idling which hampers multithreaded performance. In such a

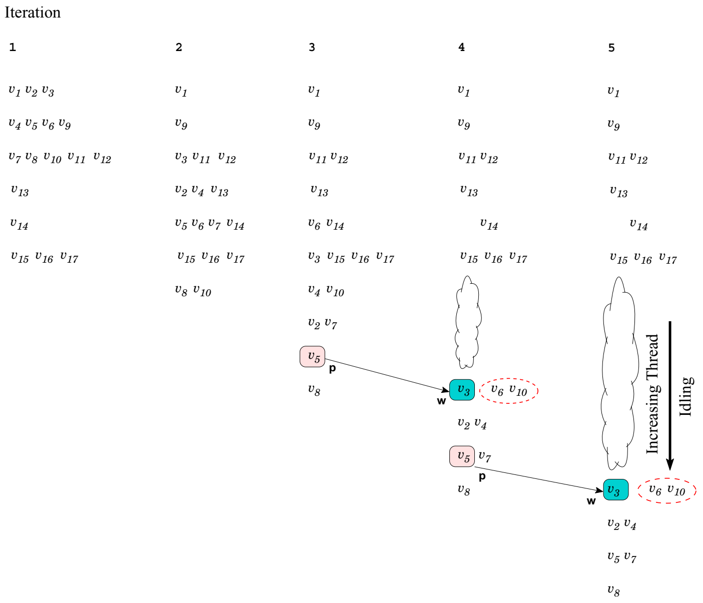


Fig. 8. Scheduling with delayed `waits` (using *MoveOpDown* [33])

scenario, spawning additional threads does not boost TLP. The current discussion is strictly limited to data dependences and does not account for run time effects such as memory subsystem performance, context switching et cetera.

3.2 Scheduling with Delayed `waits`

In [33], Nicolau et al. showed that multithreaded performance can be enhanced via downward percolation of `waits`. To this end, they proposed a transformation, referred to as *MoveOpDown*, to delay the execution of the `wait` primitive, subject to data dependences. In the current context, from Figure 7 we note that, from iteration 4 onwards, there are 2 sources of loop-carried dependences above the pipeline bubble, viz.,

$v_{15} \longrightarrow v_{10}$
$v_{13} \longrightarrow v_{6}$

These dependences further (i.e., beyond the pipeline bubble) limit extraction of TLP. However, this can be alleviated by downward percolation, via the *MoveOpDown* transformation [33], of the sinks (and the associated `waits`) of the corresponding dependences. The schedule obtained after applying the transformation is shown in Figure 8 – the operations percolated down are embedded in a dashed circle in the figure. Note that the downward percolation does *not* lengthen the critical path and eliminates the need for synchronization corresponding to operations v_6 and v_{10}, thereby improving the efficiency of the code. Unlike the schedule shown in Figure 7, in Figure 8, operations $\{v_{13}, v_{14}, v_{15}, v_{16}, v_{17}\}$ of the different iterations can be executed asynchronously on different threads. This boosts the TLP achieved.

From Figure 8 we observe that the set of operations $V = \{v_1, v_9, v_{11}, v_{12}, v_{13}, v_{14}, v_{15}, v_{16}, v_{17}\}$ of each iteration can be executed in parallel without any need of thread synchronization. Therefore, conceivably, the loop corresponding to the DDG shown in Figure 6 can be distributed such that the operations in V form one loop and the remaining operations form another loop. The former is a `DOALL` loop. However, threadization of the distributed loops may not be profitable owing to the small amount of computation in their respective loop bodies and the high threading overhead. The trade-off between loop transformations such as loop distribution and thread synchronization is beyond the scope of the paper.

3.3 OPT-Driven Scheduling

In the previous subsection we saw that downward percolation of `wait`s boosts TLP. However, the transformation neither eliminated the pipeline bubble nor did it shorten the pipeline bubble. On the contrary, the bubble in the schedule shown in Figure 8 is longer than the bubble shown in Figure 7. This raises the question whether one can eliminate the pipeline bubble(s) induced by data dependences.[3]

As mentioned earlier, in the context of VLIW compilation, there has been a lot of work on compaction-based parallelization. In [13], Aiken proposed a technique, referred to as **OPT**, for optimal parallelization of innermost loops. They showed that eliminating "gaps" (pipeline bubble in Figures 7 and 8) yields a recurring kernel. Most importantly, they show that no operation on the critical path is delayed as result of "gap" elimination. The **OPT** schedule corresponding to the greedy schedule shown in Figure 7 is shown in Figure 9. Although **OPT** was proposed for innermost loops without conditionals, it should be noted that **OPT** can also be applied to innermost loops with conditionals by applying hierarchical reduction [28] or by employing *if-conversion* [14].

Multithreaded execution of the **OPT** kernel does not incur any pipeline bubbles induced by data dependences. However, we note that there exists a loop-carried dependence between iterations 1 and 2 of the **OPT** kernel. This necessitates insertion of `post, wait` synchronization primitives (see Case I in Figure 10). This adversely affects performance owing to the synchronization overhead and the partial ordering imposed by the `post, wait` primitives.

One way to eliminate the need for thread synchronization in the **OPT** kernel is clustering. Under clustering, iterations corresponding to the source and sink of a given loop-carried dependence are mapped on to the same thread. This obviates the need for thread synchronization in the kernel. A clustered version of the **OPT** kernel is shown in Case II of Figure 10, wherein each cluster is mapped on to a different thread. Note that there is at least one dependence between consecutive instances of the **OPT** kernel. This is highlighted with a dashed arrow in Figure 9. Consequently, barrier synchronization would be required to

[3] Again, we do *not* address the pipeline bubbles which arise due to run time issues such as L2 cache misses, DTLB misses and resource stalls.

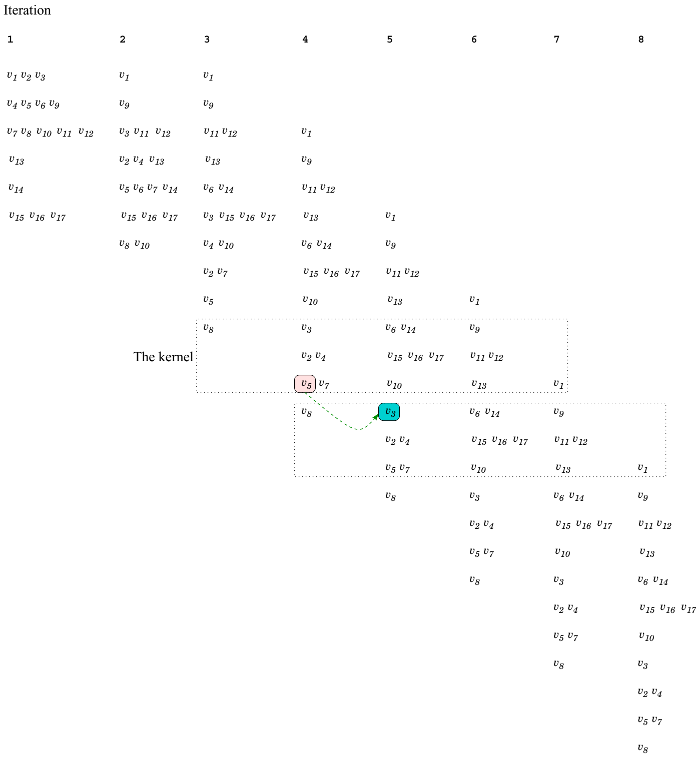

Iteration
1 2 3 4 5 6 7 8
v_1 v_2 v_3 | v_1 | v_1
v_4 v_5 v_6 v_9 | v_9 | v_9
v_7 v_8 v_{10} v_{11} v_{12} | v_3 v_{11} v_{12} | v_{11} v_{12} | v_1
v_{13} | v_2 v_4 v_{13} | v_{13} | v_9
v_{14} | v_5 v_6 v_7 v_{14} | v_6 v_{14} | v_{11} v_{12}
v_{15} v_{16} v_{17} | v_{15} v_{16} v_{17} | v_3 v_{15} v_{16} v_{17} | v_{13} | v_1
v_8 v_{10} | v_4 v_{10} | v_6 v_{14} | v_9
v_2 v_7 | v_{15} v_{16} v_{17} | v_{11} v_{12}
v_5 | v_{10} | v_{13} | v_1
The kernel
v_8 | v_3 | v_6 v_{14} | v_9
v_2 v_4 | v_{15} v_{16} v_{17} | v_{11} v_{12}
v_5 v_7 | v_{10} | v_{13} | v_1
v_8 | v_3 | v_6 v_{14} | v_9
v_2 v_4 | v_{15} v_{16} v_{17} | v_{11} v_{12}
v_5 v_7 | v_{10} | v_{13} | v_1
v_8 | v_3 | v_6 v_{14} | v_9
v_2 v_4 | v_{15} v_{16} v_{17} | v_{11} v_{12}
v_5 v_7 | v_{10} | v_{13}
v_8 | v_3 | v_6 v_{14}
v_2 v_4 | v_{15} v_{16} v_{17}
v_5 v_7 | v_{10}
v_8 | v_3
v_2 v_4
v_5 v_7
v_8

Fig. 9. Optimal ILP-centric schedule using **OPT**

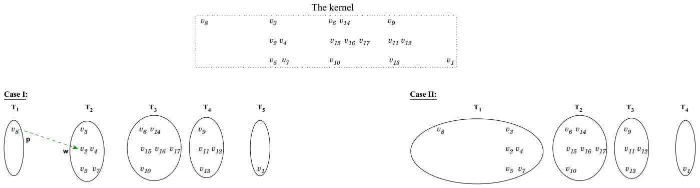

The kernel
v_8 | v_3 | v_6 v_{14} | v_9
v_2 v_4 | v_{15} v_{16} v_{17} | v_{11} v_{12}
v_5 v_7 | v_{10} | v_{13} | v_1
Case I:
T_1 T_2 T_3 T_4 T_5
p
w
Case II:
T_1 T_2 T_3 T_4

Fig. 10. Threadizing the **OPT** kernel

preserve dependences between consecutive instances of the **OPT** kernel. The cost of the barrier can be amortized by unrolling the kernel.

3.4 T-OPT Algorithm

Although clustering of the **OPT** kernel obviates the need for thread synchronization, it reduces the effective span of the **OPT** kernel. This in turn limits the exploitation of the hardware parallelism. To alleviate this, we propose a novel approach, based on code motion. Broadly speaking, the code motion orchestrated by Algorithm 1 must conform to the following:

- There is one instance of each operation in the new kernel (akin to the **OPT**-kernel.[4]
- There is no data dependences across iterations in the new kernel.

Prior to describing the **T-OPT** algorithm, we introduce the following definition.

Definition 3. *A* dependence chain C *is a sequence of operations* $\{x_{h_1}, \ldots, x_{h_n}\}$ *such that* $(x_{h_i}, x_{h_{i+1}})$ *is a loop-carried dependence edge in the DDG. Also,* start$(C) = x_{h_1}$ *and* $V(C)$ *denote the set of operations constituting* C*.*

Algorithm 1. T-OPT Algorithm

```
Input: An OPT kernel $\mathcal{K}$
Output: T-OPT kernel

Set counter ← 1
repeat
  Determine the set of dependence chains $\mathbf{C}$ in $\mathcal{K}$
  for each $c_i \in \mathbf{C}$ do
    Let $V^{'}(c_i) = V(c_i) - \{start(c_i)\}$
    $S$ = Set of operations dependent on the operations in $V^{'}(c_i)$
    $M = V^{'}(c_i) \cup S$
    for each instance $\ell$ (> counter) of $\mathcal{K}$ do
      Percolate all operations in $M$ from instance $\ell$ to $\ell + 1$
    end for
    counter ← counter + 1
  end for
until a recurring pattern with no dependence chains is found
```

Algorithm 1 iterates over multiple instances of the input **OPT** kernel. Specifically, the algorithm percolates operations downwards as to obviate the need for thread synchronization during multithreaded execution of the **OPT** kernel. The code motion is targeted towards operations that are part of a dependence chain. This results in "breaking" the dependence chains in the instance of the **OPT** kernel under consideration. On the other hand, as a result of the downward code motion, "new" dependence chains may arise in the subsequent instances of the **OPT** kernel. The algorithm applies code motion iteratively until there is no dependence chain in the instance of the **OPT** kernel – which constitutes the **T-OPT** kernel – under consideration. The span of the **T-OPT** kernel is the number of threads that should be spawned during threadization of innermost loops. Spawning more threads will not boost TLP owing to pipeline bubbles induced by data dependences. Theorem 1 proves that application of Algorithm 1 always yields the **T-OPT** kernel.

[4] It is assumed that transformations such as loop unrolling are applied (if at all) prior to code motion.

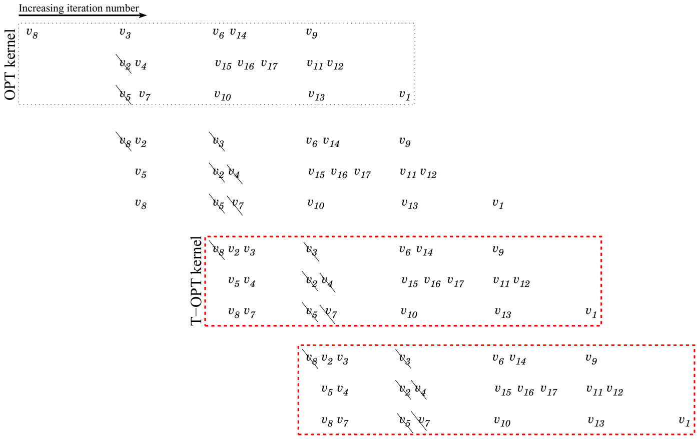

Increasing iteration number
OPT kernel
T–OPT kernel
v_8 v_3 v_6 v_{14} v_9
v_2 v_4 v_{15} v_{16} v_{17} v_{11} v_{12}
v_5 v_7 v_{10} v_{13} v_1
v_8 v_2 v_3 v_6 v_{14} v_9
v_5 v_2 v_4 v_{15} v_{16} v_{17} v_{11} v_{12}
v_8 v_5 v_7 v_{10} v_{13} v_1
v_8 v_2 v_3 v_3 v_6 v_{14} v_9
v_5 v_4 v_2 v_4 v_{15} v_{16} v_{17} v_{11} v_{12}
v_8 v_7 v_5 v_7 v_{10} v_{13} v_1
v_8 v_2 v_3 v_3 v_6 v_{14} v_9
v_5 v_4 v_2 v_4 v_{15} v_{16} v_{17} v_{11} v_{12}
v_8 v_7 v_5 v_7 v_{10} v_{13} v_1

Fig. 11. Obtaining the **T-OPT** kernel

Next, we illustrate Algorithm 1 by applying it on the **OPT** kernel shown in Figure 9.

a) Consider the first instance of $\mathcal{K}$. The dependence chains are the following:
$v_8 \longrightarrow v_2$
Note that v_5 is dependent on v_2. Therefore, we percolate $\{v_2, v_5\}$ down from instance ℓ to instance $\ell + 1$.

b) Now let us consider the second instance of $\mathcal{K}$. The downward percolation of v_2 from instance 1 to instance 2 (in step a) above) induces the following dependence chain: $v_2 \longrightarrow v_3$.
Note that operations $\{v_4, v_7\}$ are dependent on v_3. Therefore, we percolate $\{v_3, v_4, v_7\}$ down from instance ℓ to instance $\ell + 1$.

c) In the third instance of $\mathcal{K}$, we observe that there do not exist any dependence chains. Hence, the third instance corresponds to the **T-OPT** kernel.

The new kernel, referred to as **T-OPT**, corresponding to the **OPT** kernel shown in Figure 9 is shown in Figure 11. On comparing Figures 10 and 11, we note that Algorithm 1 obviates the need for thread synchronization during threadization of the software pipelined kernel. Furthermore, note that there are no pipeline bubbles, owing to data dependences, in the **T-OPT** kernel.

Theorem 1. *Algorithm 1 always yields the* **T-OPT** *kernel.*

Proof. We break down the proof into the following three cases:

❐ First, if there are no dependence chains in the **OPT** kernel, then the **OPT** kernel is the **T-OPT** kernel. In [13], Aiken proved that the **OPT** kernel can always exists.

❐ Second, if downward percolation of operations from one instance of the **OPT** kernel does not give rise to "new" dependence chain(s) in the **OPT** kernel, then the resulting kernel constitutes the **T-OPT** kernel.

❒ Third, the downward percolation of operations from one instance of the **OPT** kernel gives rise to "new" dependence chain(s) in the **OPT** kernel. Recall that the downward percolation breaks the existing dependence chains. Given this and the fact that there are finite number of loop-carried dependences in the DDG, a recurring pattern – the **T-OPT** kernel – will always emerge as a consequence of downward code motion (as described in Algorithm 1).

3.5 Remark

Let us consider the example shown in Figure 12(a). There exist two loop-carried dependences, viz., $v_2 \rightarrow v_1, v_3 \rightarrow v_2$. The **OPT** kernel of the loop is shown in Figure 12(b). Note that there exists a dependence chain – marked with an arrow – in the **OPT** kernel. Breaking the chain via Algorithm 1 results in the **T-OPT** kernel shown in Figure 12(c), which is the original loop body. Thus, there does not exist a synchronization free **OPT** kernel for this particular loop.

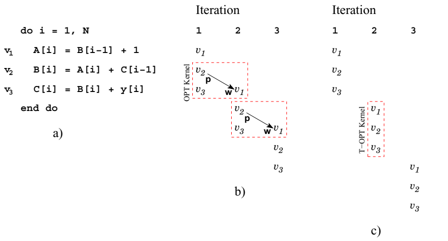


Fig. 12. Illustration of serialization induced by Algorithm 1

The serialization induced the Algorithm 1 exemplifies the trade-off between elimination thread synchronization and loss of TLP.

4 Case Studies and Results

In this section, we present a couple of case studies and results using kernels, extracted from open source codes, to demonstrate the efficacy of the techniques proposed in Section 3. We use multithreaded execution of the **OPT** kernel as our baseline. Evaluation of the proposed techniques on overall benchmarks is beyond the scope of this paper. The primary focus herein is to showcase a previously unexplored optimization opportunity, and provide evidence of its practical applicability in real codes using real hardware.

The two case studies correspond to the DDG shown in Figure 6 and a loop taken from `189.lucas`, a benchmark in the SPEC CFP2000 benchmark suite [7]. The kernels were optimized using the technique proposed in Section 3. We compiled the optimized kernels using the Intel C++ compiler and ran the kernels on a real machine. The detailed experimental setup is given in Table 2.

Table 2. Experimental Setup

System	2 Quad-Core Processor
Processor	Intel®Xeon®CPU 1.86GHz
L1 Cache	32 KB
L2 Cache	4096 KB
Memory	4038908 KB
Compiler	Intel C++ Compiler v11.1
Compiler Flags	-parallel -openmp -O3
OS	Linux 2.6.9-34.ELsmp #1 SMP

We first analyze the multithreaded performance, refer to Figure 13, of the loop corresponding to the DDG shown in Figure 6. From Figure 13 we note that spawning as many threads as the number of cores available does not necessarily yield best performance (the execution times are normalized with respect to **OPT**) . On the contrary, in the current context, spawning 3 threads, as guided by Algorithm 1 yields best performance. Observe that the multithreaded performance with 5 threads (equal to the span of the **OPT** kernel) is worse than **T-OPT**. Likewise, performance with 2 threads (equal to the span of the kernel obtained using modulo scheduling) is worse than **T-OPT**. Lastly, multithreaded performance with 8 threads (equal to the number of cores) is worse than **T-OPT**.

The second case study corresponds to the loop taken from `189.lucas`, a benchmark in the SPEC CFP2000 benchmark suite [7]. The span of the **T-OPT** for this loop is 8. As per Algorithm 1, best performance corresponds to 8-way multithreaded execution. The same is observed from Figure 14. From the figure we note that performance improves with increasing number of threads and best performance is achieved with 8 threads.

Note that **T-OPT** performs better than **OPT** even when both yield kernels of the same width. This can be attributed to the fact that synchronization is not needed during multithreaded execution of the **T-OPT** kernel.

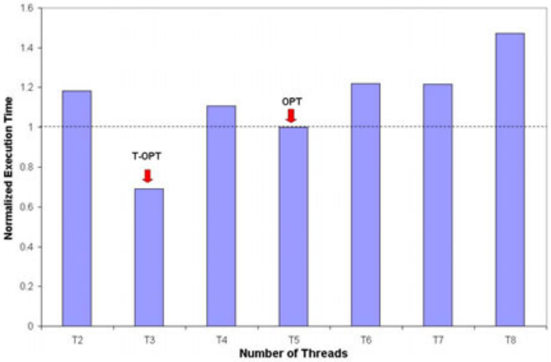


Fig. 13. Performance variation with varying number of threads for the loop corresponding to the DDG shown in Figure 6

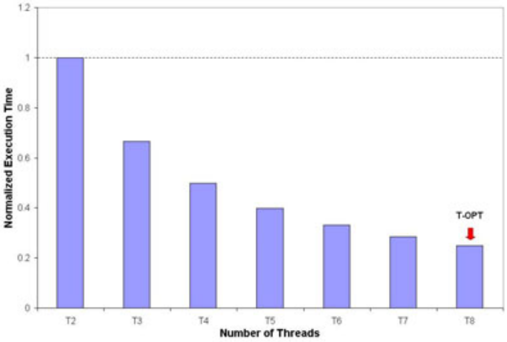


Fig. 14. Performance variation with varying number of threads for the loop taken from `189.lucas`

Next, we present results using kernels, extracted from open source codes. The kernels were extracted from industry-standard SPEC CPU2006 benchmark suite [10] and open source FreeBSD, version 7.0 [2], see Table 3. We applied **T-OPT** to the kernels listed in Table 3.

Table 3. Kernels

Kernel	Benchmark	Suite
K(calculix)	454.calculix	CFP2006
K(bzip2)	401.bzip2	CINT2006
K(hmmer)	456.hmmer	CINT2006
K(h264ref)	464.h264ref	CINT2006
K(soplex)	450.soplex	CFP2006
K1(bzip2)	bzip2	FreeBSD
K(gcc)	gcc	FreeBSD

Table 4. Experimental results

Kernel	# Threads	Speedup
K(calculix)	1	NA
K(bzip2)	3	12%
K(hmmer)	5	22%
K(h264ref)	2	26%
K(soplex)	1	NA
K1(bzip2)	3	14%
K(gcc)	5	18%

The results are presented in Table 4. The second column in Table 4 corresponds to the number of threads spawned (which was determined as discussed in Section 3). The speedups reported in the table are with respect to multithreaded **OPT** kernel (recall that, unlike the **T-OPT** kernel, multithreaded execution of an instance the **OPT** kernel requires thread synchronization).

From Table 4 we note that a couple of kernels (with "NA" in the third column) were executed in a sequential fashion. This is due to the fact that application of Algorithm 1 resulted in serialization of the loop body (refer to the discussion in subsection 3.5). Furthermore, the two kernels were deemed to be unprofitable for multithreaded execution owing to small loop bodies. Multithreaded execution of the **OPT** kernel resulted in a performance slowdown! Lastly, from the table we observe that speedup up to 26% was obtained.

5 Previous Work

In this section we overview related work in three different subsections.

5.1 Compaction-Based Parallelization

Compaction-based parallelization has received considerable attention in the past. The primary focus has been on extraction of instruction-level parallelism (ILP) [13,30]. As mentioned earlier, modulo scheduling [37] and several techniques for software pipelining [32,39,22,28,36] have been proposed for loops. Similar techniques have been proposed for superscalar processors [23]. None of the techniques proposed in the aforementioned works can be used to address the problems addressed in this paper. This can be ascribed to the fact that these techniques are instruction-level based, whereas the current focus is at the thread-level.

5.2 Multithreaded Performance

There exists a large amount of prior work in the context of multithreaded performance. Owing to space limitations, we present a brief overview herein.

Bokhari proposed techniques for task mapping and reassignment for distributed systems [18,19]. An early survey of strategies for task allocation for distributed systems is presented in [35]. In [17], Billionnet et al. presented a cost-model for task assignment in an heterogeneous multiple processors systems. Ernst et al. presented exact solutions for the task allocation problem in the same context. Arguably, one could leverage the techniques proposed in the works mentioned above in the current context. However, this is not feasible owing to the fact that multi-core systems are shared-memory based.

In [15], Anderson et al. examine the performance implications of several data structure and algorithm alternatives for thread management in shared-memory multiprocessors. For applications with fine-grained parallelism, they show that small differences in thread management can have significant performance impact. In addition, they show that the method used by processors to queue for locks can affect performance significantly. In [41], Weissman proposed an approach to improve thread locality via the use of hardware performance monitors of modern processors and to use program-centric code annotations to guide thread scheduling on SMPs. Later, Narlikar proposed an approach for scheduling threads for low space requirement and good locality [31]. None of the above works address the issue of how many threads to use and when to threadize. We believe the aforementioned works are complimentary to the work presented in this paper. Recently, Suleman et al. proposed a technique – Feedback-Driven Threading (FDT) – to dynamically control the number of threads at run-time [40]. As a next step, FDT is used to implement Synchronization-Aware Threading (SAT), which predicts the optimal number of threads depending on the amount of time spent in critical section and parallel part of a given loop. Unlike **T-OPT**, SAT is oblivious of the dependence graph of the loop. Given the runtime nature of SAT, we believe that **T-OPT** and SAT are complementary to each other.

6 Conclusion

We proposed a novel technique for determining how many threads to spawn during threadization of innermost loops. We formally proved that a **T-OPT** kernel always emerges on orchestrating code motion outlined in Algorithm 1. Spawning additional – more than the span of the **T-OPT** kernel – does not yield higher TLP w.r.t. pipeline bubbles induced by data dependences. We illustrated the efficacy of our techniques using a couple of case studies.

An **OPT** kernel is an input to the **T-OPT** algorithm. Recall that the **OPT** algorithm [12] determines the first repeating pattern while closing the gaps, irrespective of the existence of dependence chains within the pattern. As future work, we intend to develop a parameterized algorithm which factors in the synchronization cost[5] while determining a repeating pattern. Also, we plan to extend our current work to N-dimensional loops.

References

1. AMD's 16-core Interlagos, processor,
   `http://www.tgdaily.com/content/view/42125/135/`
2. FreeBSD,
   `http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/anoncvs.html`

[5] Note that the synchronization cost is a function of the design of the synchronization primitives and the target hardware.

3. GCC, the GNU Compiler Collection, `http://gcc.gnu.org/`
4. Intel® Core™ i7 Processor Datasheet, Vol. 1, `http://download.intel.com/design/processor/datashts/320834.pdf`
5. OpenBSD, `http://www.openbsd.org/`
6. OpenMP Specification, version 2.5, `http://www.openmp.org/drupal/mp-documents/spec25.pdf`
7. SPEC CFP2000, `http://www.spec.org/cpu2000/CFP2000`
8. SPEC CFP2006, `http://www.spec.org/cpu2006/CFP2006`
9. SPEC CPU Benchmarks, `http://www.spec.org/benchmarks.html`
10. SPEC CPU2006, `http://www.spec.org/cpu2006`
11. Wine, `http://sourceforge.net/project/showfiles.php?group_id=6241`
12. Aiken, A., Nicolau, A.: Optimal loop parallelization. In: Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation, Atlanta, GA (June 1988)
13. Aiken, A.S.: Compaction-based parallelization. PhD thesis, Dept. of Computer Science, Cornell University (August 1988)
14. Allen, J.R., Kennedy, K., Porterfield, C., Warren, J.: Conversion of control dependence to data dependence. In: Conference Record of the Tenth Annual ACM Symposium on the Principles of Programming Languages, Austin, TX (January 1983)
15. Anderson, T.E., Lazowska, D.D., Levy, H.M.: The performance implications of thread management alternatives for shared-memory multiprocessors. In: SIGMETRICS 1989: Proceedings of the 1989 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, Oakland, CA, pp. 49–60 (1989)
16. Banerjee, U.: Dependence Analysis. Kluwer Academic Publishers, Boston (1997)
17. Billionnet, A., Costa, M.C., Sutter, A.: An efficient algorithm for a task allocation problem. Journal of the ACM 39(3), 502–518 (1992)
18. Bokhari, S.: Dual processor scheduling with dynamic reassignment. IEEE Transactions on Software Engineering SE-5, 341–349 (1979)
19. Bokhari, S.: On the mapping problem. IEEE Transactions on Computers C-30, 207–214 (1981)
20. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. The MIT Press, Cambridge (1990)
21. Cytron, R.: Compile-time Scheduling and Optimization for Asynchronous Machines. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign (October 1984)
22. Ebcioğlu, K.: A compilation technique for software pipelining of loops with conditional jumps. In: Proceedings of the 20th Workshop on Microprogramming, Colarado Springs, CO (December 1987)
23. Ebcioğlu, K., Groves, R.D., Kim, K.C., Silberman, G.M., Ziv, I.: VLIW compilation techniques in a superscalar environment. In: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, pp. 36–48 (1994)
24. Fisher, J.A.: VLIW architectures: an inevitable standard for the future? Supercomputer 7(2), 29–36 (1990)
25. Kejariwal, A.: On the evaluation and extraction of thread-level parallelism in ordinary programs. PhD thesis, University of California, Irvine, CA (January 2008)
26. Kejariwal, A., Nicolau, A.: Reading list of mutual exclusion, locking, synchronization and concurrent objects, `http://www.ics.uci.edu/~akejariw/ConcurrentExecutionReadingList.pdf`

27. Kuck, D.: The Structure of Computers and Computations, vol. 1. John Wiley and Sons, New York (1978)
28. Lam, M.: Software pipelining: An effective scheduing technique for VLIW machines. In: Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation, Atlanta, GA (June 1988)
29. Lundstrom, S.F., Barnes, G.H.: A controllable MIMD architectures. In: Proceedings of the 1980 International Conference on Parallel Processing, St. Charles, IL, pp. 19–27 (August 1980)
30. Nakatani, T., Ebcioğlu, K.: Making compaction based parallelization affordable. IEEE Transactions on Parallel and Distributed Systems 4(9), 1014–1029 (1993)
31. Narlikar, G.J.: Scheduling threads for low space requirement and good locality. In: Proceedings of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures, Saint Malo, France, pp. 83–95 (1999)
32. Nicolau, A.: Parallelism, memory anti-aliasing and correctness for trace scheduling compilers (disambiguation, flow-analysis, compaction). PhD thesis, Dept. of Computer Science, Yale University (1984)
33. Nicolau, A., Li, G., Kejariwal, A.: Techniques for efficient placement of synchronization primitives. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Raleigh, NC, USA, pp. 199–208 (February 2009)
34. Nicolau, A., Li, G., Veidenbaum, A.V., Kejariwal, A.: Synchronization optimizations for efficient execution on multi-cores. In: Proceedings of the 23rd ACM International Conference on Supercomputing, New York, NY, pp. 169–180 (2009)
35. Price, C.C.: Task allocation in distributed systems: A survey of practical strategies. In: Proceedings of the ACM 1982 Conference, pp. 176–181 (1982)
36. Rau, B.R., Fisher, J.A.: Instruction level parallel processing: History, overview and perspective 7(1), 97 (January 1993)
37. Rau, B.R., Glaeser, C.D.: Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In: Proceedings of the 14th Annual Workshop on Microprogramming, Chatham, MA, pp. 183–198 (December 1981)
38. Reinders, J.: Intel threading building blocks. O'Reilly & Associates, Inc., Sebastopol (2007)
39. Su, B., Ding, S., Xia, J.: URPR - an extension of urcr for software pipelining. In: Proceedings of the 19th Workshop on Microprogramming, New York, NY (October 1986)
40. Suleman, M.A., Qureshi, M.K., Patt, Y.N.: Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on CMPs. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, Seattle, WA, pp. 277–286 (2008)
41. Weissman, B.: Performance counters and state sharing annotations: a unified approach to thread locality. In: Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII), San Jose, CA, pp. 127–138 (1998)
42. Wolfe, M.: The definition of dependence distance 16(4), 1114–1116 (1994)
43. Wolfe, M.J.: Optimizing Supercompilers for Supercomputers. The MIT Press, Cambridge (1989)
44. Wu, P., Kejariwal, A., Caşcaval, C.: Compiler-driven dependence profiling to guide program parallelization. In: Proceedings of the 21st International Workshop on Languages and Compilers for Parallel Computing, Alberta, Canada (2008)

Parallelizing Compiler Framework and API for Power Reduction and Software Productivity of Real-Time Heterogeneous Multicores

Akihiro Hayashi, Yasutaka Wada, Takeshi Watanabe, Takeshi Sekiguchi,
Masayoshi Mase, Jun Shirako, Keiji Kimura, and Hironori Kasahara

Department of Computer Science and Engineering, Waseda University,
3-4-1 Okubo, Shinjuku-ku, Tokyo, Japan
{ahayashi,yasutaka,watanabe,
takeshi,mase,shirako,kimura}@kasahara.cs.waseda.ac.jp,
kasahara@waseda.jp
http://www.kasahara.cs.waseda.ac.jp/

Abstract. Heterogeneous multicores have been attracting much attention to attain high performance keeping power consumption low in wide spread of areas. However, heterogeneous multicores force programmers very difficult programming. The long application program development period lowers product competitiveness. In order to overcome such a situation, this paper proposes a compilation framework which bridges a gap between programmers and heterogeneous multicores. In particular, this paper describes the compilation framework based on OSCAR compiler. It realizes coarse grain task parallel processing, data transfer using a DMA controller, power reduction control from user programs with DVFS and clock gating on various heterogeneous multicores from different vendors. This paper also evaluates processing performance and the power reduction by the proposed framework on a newly developed 15 core heterogeneous multicore chip named RP-X integrating 8 general purpose processor cores and 3 types of accelerator cores which was developed by Renesas Electronics, Hitachi, Tokyo Institute of Technology and Waseda University. The framework attains speedups up to 32x for an optical flow program with eight general purpose processor cores and four DRP(Dynamically Reconfigurable Processor) accelerator cores against sequential execution by a single processor core and 80% of power reduction for the real-time AAC encoding.

Keywords: Heterogeneous Multicore, Parallelizing Compiler, API.

1 Introduction

There has been a growing interest in heterogeneous multicores which integrate special purpose accelerator cores in addition to general purpose processor cores on a chip. One of the reason for this trend is because heterogeneous multicores allow us to attain high performance with low frequency and low power

K. Cooper, J. Mellor-Crummey, and V. Sarkar (Eds.): LCPC 2010, LNCS 6548, pp. 184–198, 2011.

consumption. Various semiconductor vendors have released heterogeneous multicores such as CELL BE[15], NaviEngine[11], Uniphier[13], GPGPU[9], RP1[20] and RP-X[21].

However, the softwares for heterogeneous multicores generally require large development efforts such as the decomposition of a program into tasks, the implementation of accelerator code, the scheduling of the tasks onto general purpose processors and accelerators, and the insertion of synchronization and data transfer codes. These software development periods are required even for expert programmers.

Recent many studies have tried to handle on this software development issue. For example, NVIDIA and Khronos Group introduced CUDA[3] and OpenCL[7]. Also, PGI accelerator compiler[19] and HMPP[2] provides a high-level programming model for accelerators. However, these works focus on facilitating the development for accelerators. Programmers need to distribute tasks among general purpose processors and accelerator cores by hand. In terms of workload distribution, Qilin[10] automatically decides which task should be executed on a general purpose processor or an accelerator at runtime. However, programmers still need to parallelize a program by hand. While these works rely on programmers' skills, CellSs[1] performs an automatic parallelization of a subset of sequential C program with data flow annotations on CELL BE. CellSs automatically schedules tasks onto processor elements at runtime. The task scheduler of CellSs, however, is implemented as a homogeneous task scheduler, namely the scheduler is executed on PPE and just distributes tasks among SPEs.

In the light of above facts, further explorations are needed since it is the responsibility of programmers to parallelize a program and to optimize a data transfer and a power consumption for heterogeneous multicores. One of our goals is to realize a fully automatic parallelization of a sequential C or Fortran77 program for heterogeneous multicores. We have been developing OSCAR paralleling compiler for homogeneous multicores such as SMP servers and real-time multicores[5,8,12]. These works realize automatic parallelization of programs written in Fortran77 or Parallelizable C, a kind of C programming style for parallelizing compiler, and power reduction with the support of both OSCAR compiler and OSCAR API(Application Program Interface)[6]. This paper describes an automatic parallelization for a real heterogeneous multicore chip. Though prior work demonstrates the performance of automatic parallelization of a Fortran program on a heterogeneous multicore simulator[18], this paper makes the following contributions:

– A proposal of an accelerator-independent and general purpose compilation framework including a compilation framework using OSCAR compiler and an extention of OSCAR API[8] for heterogeneous multicore
– An evaluation of a processing performance and a power efficiency using 3 Parallelizable C applications on the newly developed RP-X multicore chip[21].

In order to build an accelerator-independent and a general-purpose compilation framework, we take care of utilizing existing tool chains such as accelerator compilers and hand-tuned libraries for accelerators. Therefore, this paper firstly

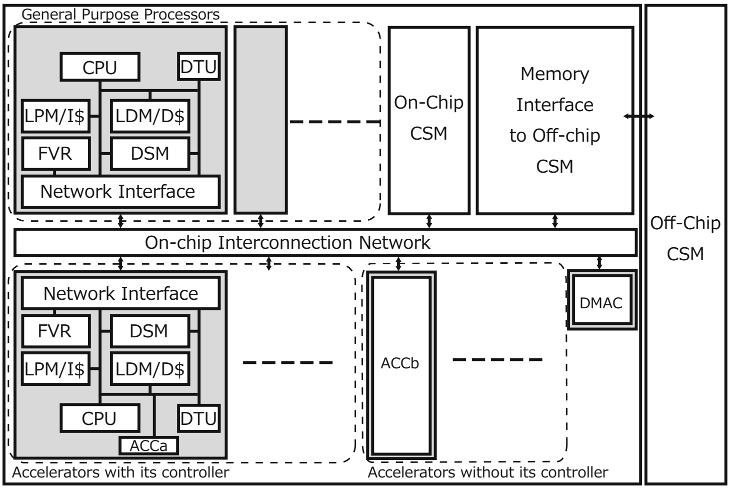


Fig. 1. OSCAR API Applicable heterogeneous multicore architecture

defines an general-purpose architecture and compilation flow in Section 2. Secondly, we defines distinct responsibilities among these tool chains and interface among them by extending OSCAR API in Section 3.

2 OSCAR API Applicable Heterogeneous Multicore Architecture and Overview of the Compilation Flow

This section defines both target architecture and compilation flow of the proposed framework. In this paper, define a term "controller" as a general purpose processor that controls an accelerator, that is to say, it performs part of coarse-grain task and data transfers from/to the accelerator and offload the task to the accelerator.

2.1 OSCAR API Applicable Heterogeneous Multicore Architecture

This section defines "OSCAR API Applicable Heterogeneous Multicore Architecture" shown in Fig.1.. The architecture is composed of general purpose processors, accelerators(ACCs), direct memory access controller(DMAC), on-chip centralized shared memory(CSM), and off-chip CSM. Some accelerators may have its own controller, or general purpose processor. Both general purpose processors and accelerators with controller may have a local data memory (LDM), a distributed shared memory (DSM), a data transfer unit (DTU), a frequency voltage control registers (FVR), an instruction cache memory and a data cache memory. The local data memory keeps private data. The distributed shared memory is a dual port memory, which enables point-to-point direct data transfer and low-latency synchronization among processors. Each existing heterogeneous multicore can be seen such as CELL BE[15], MP211[17] and RP1[20] as a subset

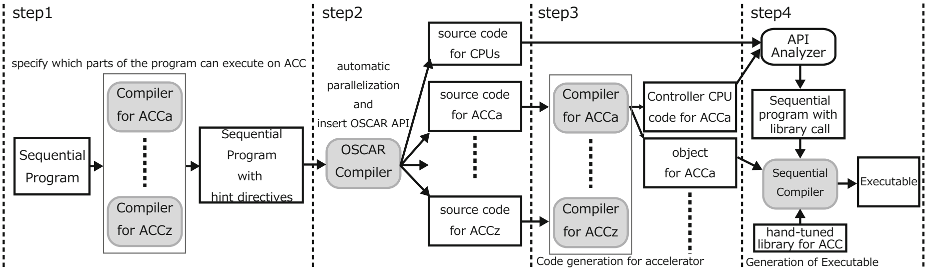


Fig. 2. Compilation flow of the proposed framework

of OSCAR API applicable architecture. Thus, OSCAR API can support such chips and a subset of OSCAR API applicable heterogeneous multicore.

2.2 Compilation Flow

Fig.2. shows the compilation flow of the proposed OSCAR heterogeneous compiler framework. The input is a sequential program written in Parallelizable C or Fortran77 and the output is an executable for a target heterogeneous multicore. The following describes each step in the proposed compilation flow.

Step 1: Accelerator compilers or programmers insert hint directives immediately before loops or function calls , which can be executed on the accelerator, in a sequential program.

Step 2: OSCAR compiler parallelizes the source program considering with hint directives: the compiler schedules coarse-grain tasks[18] to processor or accelerator cores and apply the low power control[8]. Then, the compiler generates a parallelized C or Fortran program for general purpose processors and accelerator cores by using OSCAR API. At that time, the compiler generates C source codes as separate files for accelerator cores. Each file includes functions to be executed on accelerators when a function is scheduled onto accelerator by the compiler.

Step 3: Each accelerator compiler generates objects for its own target accelerator. Note that each accelerator compiler also generates both data transfer code between controller and accelerator, and accelerator invocation code.

Step 4: An API analyzer prepared for each heterogeneous multicore translates OSCAR APIs into runtime library calls, such as pthread library. Afterwards, an ordinary sequential compiler for each processor from each vender generates an executable.

It is important that the framework also allows programmers to utilize existing hand-tuned libraries for the specific accelerator. This paper defines a term “hand-tuned library” as an accelerator library which includes computation body on the specific accelerator and both data transfer code between general purpose processors and accelerators and accelerator invocation code.

3 A Compiler Framework for Heterogeneous Multicores

This section describes the detail of OSCAR compiler and OSCAR API.

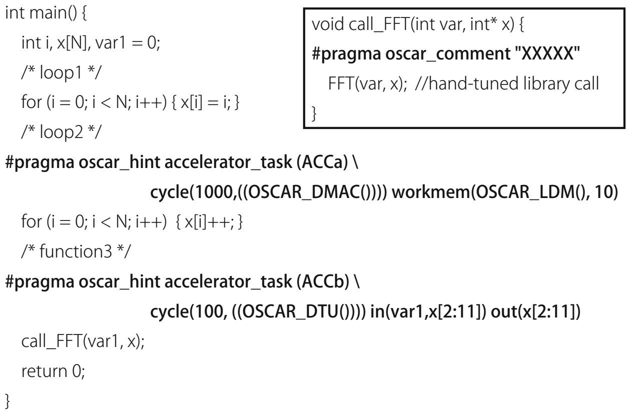
3.1 Hint Directives for OSCAR Compiler

This subsection explains the hint directives for OSCAR compiler that advice OSCAR compiler which parts of the program can be executed by which accelerator core.

Fig.3. shows an example code. As shown in Fig.3., there are two types of hint directives inserted to a sequential C program, namely "accelerator_task" and "oscar_comment". In this example, there are "#pragma oscar_hint accelerator_task (ACCa) cycle(1000, ((OSCAR_DMAC()))) workmem(OSCAR_LDM(), 10)" and "#pragma oscar_hint accelerator_task (ACCb) cycle(100, ((OSCAR_DTU()))) in(var1, x[2:11]) out(x[2:11])". In these directives, accelerators represented as "ACCa" and "ACCb" is able to execute a loop named "loop2" and a function named "function3", respectively. The hint directive for "loop2" specifies that "loop2" requires 1000 cycles including the cost of a data transfer performed by DMAC if the loop is processed by "ACCa". This directive also specifies that 10 bytes in local data memory are required in order to control "ACCa". Similarly, for "function3", it takes 100 cycles including the cost of a data transfer by DTU. Input variables are scalar variable "var1" and array variable "x" ranging 2 to 11. Also, output variable is array variable "x". "oscar_comment" directive is inserted so that either programmers or accelerator compilers give a comment to accelerator compiler through OSCAR compiler.

3.2 OSCAR Parallelizing Compiler

This subsection describes OSCAR compiler.



```
int main() {
  int i, x[N], var1 = 0;
  /* loop1 */
  for (i = 0; i < N; i++) { x[i] = i; }
  /* loop2 */
#pragma oscar_hint accelerator_task (ACCa) \
                  cycle(1000,((OSCAR_DMAC()))) workmem(OSCAR_LDM(), 10)
  for (i = 0; i < N; i++)  { x[i]++; }
  /* function3 */
#pragma oscar_hint accelerator_task (ACCb) \
                  cycle(100, ((OSCAR_DTU()))) in(var1,x[2:11]) out(x[2:11])
  call_FFT(var1, x);
  return 0;
}
```

```
void call_FFT(int var, int* x) {
#pragma oscar_comment "XXXXX"
  FFT(var, x);  //hand-tuned library call
}
```

Fig. 3. Example of source code with hint directives

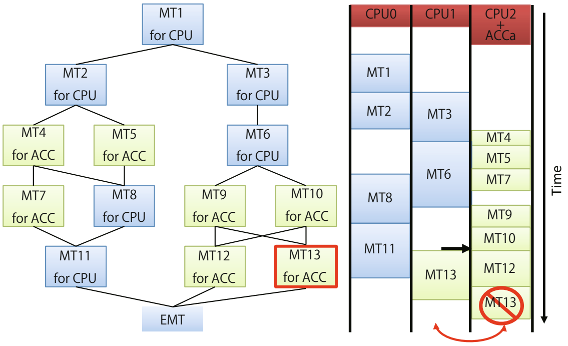


Fig. 4. An Example of Task Scheduling Result

First of all, the compiler decomposes a program into coarse grain tasks, namely macro-tasks (MTs), such as basic block (BPA), loop (RB), and function call or subroutine call (SB). Then, the compiler analyzes both the control flow and the data dependencies among MTs and represents them as a macro-flow-graph (MFG). Next, the compiler applies the earliest executable condition analysis, which can exploit parallelism among MTs associated with both the control dependencies and the data dependencies. The analysis result is represented as a hierarchically-defined macro-task-graph (MTG)[5]. When the compiler cannot analyze the input source for some reason, like hand-tuned accelerator library call, "in/out" clause of "accelerator_task" gives the data dependency information to OSCAR compiler. Then, the compiler calculates the cost of MT and finds the layer which is expected to apply coarse-grain parallel processing most effectively. "cycle" clause of "accelerator_task" tells the cost of accelerator execution to the compiler.

Secondly, the task scheduler of the compiler statically schedules macro-tasks to each core[18]. Fig.4. shows an example of heterogeneous task scheduling result. First the scheduler gets ready macro-tasks from MTG(MT1 in Fig.4 in initial state). Ready tasks satisfy earliest executable condition[4]. Then, the scheduler selects a macro-task to be scheduled from the ready macro-tasks and schedules the macro-task onto general purpose processor or accelerator considering data transfer overhead, according to the priorities, namely CP length. The scheduler performs above sequences until all macro-tasks are scheduled. Note that a task for an accelerator is not always assigned to the accelerator when the accelerator is busy. At this case, the task may be assigned to general purpose processor to minimize total execution time.

Thirdly, the compiler tries to minimize total power consumption by changing frequency and voltage(DVFS) or shutting power down the core during the idle time considering transition time[16]. The compiler determines suitable voltage and frequency for each macro-task based on the result of static task assignment in order to satisfy the deadline for real-time execution(Fig.5.). In Fig.5., FULL is 648MHz and MID is 324MHz, respectively. Each of which is used in RP-X described in Section4.

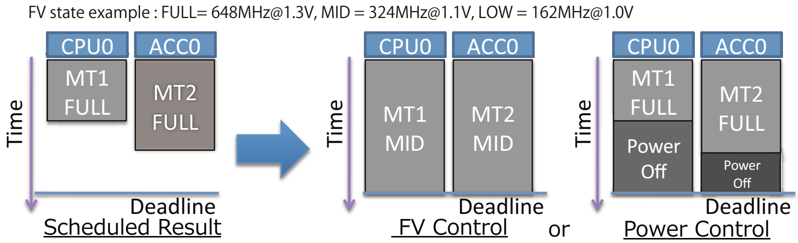


Fig. 5. Power control by compiler

Finally, the compiler generates parallelized C or Fortran program with OSCAR API. OSCAR compiler generates the function which includes original source for accelerator. Generation of data transfer codes and accelerator invocation code is responsible for accelerator compiler.

OSCAR compiler uses processor configurations, such as number of cores, cache or local memory size, available power control mechanisms, and so on. This information is provided by compiler options.

3.3 The Extension of OSCAR API for Heterogeneous Multicores

This subsection describes API extension for heterogeneous multicores to be the output of OSCAR compiler. Thee extension is very simple. Only one directive "accelerator_task_entry" is added to OSCAR homogeneous API. This directive specifies the function's name where general purpose processor invokes an accelerator.

Let us consider an example where the compiler parallelizes the program in Fig.3. We assume a target multicore includes two general purpose processors, one ACCa as an accelerator with its controller and one ACCb as an accelerator without its controller. One of general purpose processors, namely CPU1, is used as controller for ACCb in this case. Fig.6. shows as example of the parallelized C code with OSCAR heterogeneous directive generated by OSCAR compiler. As shown in Fig.6., functions named "MAIN_CPU0()", "MAIN_CPU1()" and "MAIN_CPU2()" are invoked in omp parallel sections. These functions are executed on general purpose processors. In addition, hand-tuned library "oscartask_CTRL1_call_FFT()" executed on ACCa is called by controller "MAIN_CPU1()". "MAIN_CPU2" also calls kernel function "oscartask_CTRL2_call_loop2()" executed on ACCb. "accelerator_task_entry" directive specifies these two functions. "controller" clause of the directive specifies id of general purpose CPU which controls the accelerator. Note that there exists "oscar_comment" directives at same place shown in Fig.3.. "oscar_comment" directives may be used to give accelerator specific directives, such as PGI accelerator directives, to accelerator compilers. Afterwards, accelerator compilers generates the source code for the controller and objects for the accelerator, interpreting these directives.

```
int main() {
#pragma omp parallel sections
  {
#pragma omp section
     { MAIN_CPU0(); }
#pragma omp section
     { MAIN_CPU1(); }
#pragma omp section
     { MAIN_CPU2(); }
  }
  return 0;
}
```

```
int MAIN_CPU1() {
  ...
  oscartask_CTRL1_call_FFT(var1, &x);
  ...
}
int MAIN_CPU2() {
  ...
  oscartask_CTRL2_call_loop2(&x);
  ...
}
```

Source Code for CPUs

```
#pragma oscar accelerator_task_entry controller(2) \
                                    oscartask_CTRL2_loop2
void oscartask_CTRL2_loop2(int *x) {
  int i;
  for (i = 0; i <= 9; i += 1) { x[i]++; }
}
```

Source Code for ACCa

```
#pragma oscar accelerator_task_entry controller(1) \
                                 oscartask_CTRL1_call_FFT
void oscartask_CTRL1_call_FFT(int var1, int *x) {
#pragma oscar_comment "XXXXX"
  oscarlib_CTRL1_ACCEL3_FFT(var1, x);
}
```

Source Code for ACCb

Fig. 6. Example of parallelized source code with OSCAR API

4 Performance Evaluations on RP-X

This section evaluates the performance of the proposed framework on 15 core heterogeneous multicore RP-X[21] using media applications.

4.1 Evaluation Environment

The RP-X processor is composed of eight 648MHz SH-4A general purpose processor cores and four 324MHz FE-GA accelerator cores, the other dedicated hardware IP such as matrix processor "MX-2" and video processing unit "VPU5", as shown in Fig.7. Each SH-4A core consists of a 32KB instruction cache, a 32KB data cache, a 16KB local instruction/data memory(ILM and DLM in Fig.7), a 64KB distributed shared memory(URAM in Fig.7) and a data transfer unit. Furthermore, FE-GA is used as an accelerator without controller because FE-GA is directly connected with on-chip interconnection network named "SHwy#1", a split transaction bus. With regard to the power reduction control mechanism of RP-X, DVFS and clock gating for each SH-4A core can be

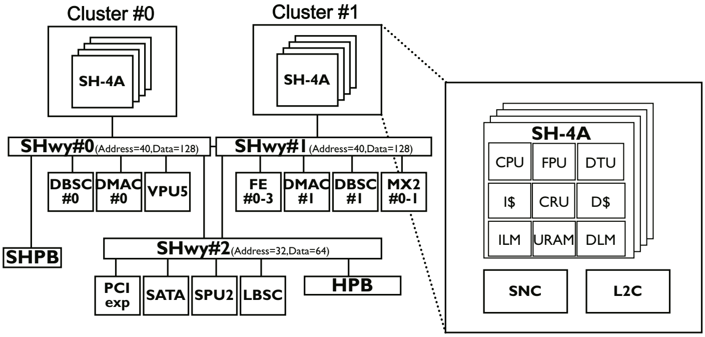


Fig. 7. RP-X heterogeneous multicore for consumer electronics

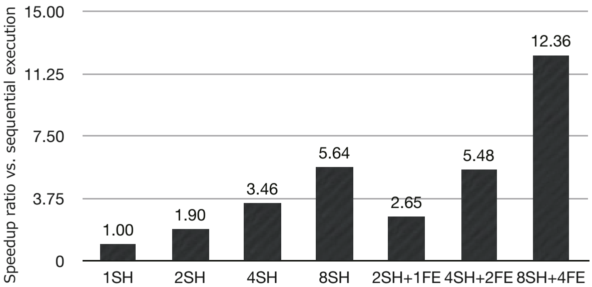


Fig. 8. Performance by OSCAR compiler and FE-GA Compiler(Optical Flow)

controlled independently using special power control register by a user. DVFS for FE-GAs can be controlled by a user. This hardware mechanism is low overhead, for example frequency change needs a few clocks. This paper evaluates both generating the object code by accelerator compiler and using the hand-tuned library on RP-X processor. We evaluate the processing performance and the power consumption of the proposed framework using upto eight SH-4A cores and four FE-GA cores.

4.2 Performance by OSCAR Compiler with Accelerator Compiler

An "optical flow" application from OpenCV[14] is used for this evaluation. The algorithm is a type of object tracking system, which calculates velocity field between two images. The program is modified in Parallelizable C[12] in this evaluation. This program consists of the following parts: dividing the image into 16x16 pixel blocks, searching a similar block in the next image for every block in the current image, shifting 16 pixels and generating the output. OSCAR compiler parallelizes the loop which searches a similar block in the next image. In addition, FE-GA compiler developed by Hitachi analyzed that the sum of absolute difference(SAD), which occupies a large part of the program execution time, is to be executed on FE-GA. FE-GA compiler also automatically inserts the hint directives to the C program. OSCAR compiler generates parallel C program with OSCAR heterogeneous API. The parallel program is translated into parallel executable binary by using API analyzer which translates the directives to library calls and sequential compiler and FE-GA compiler translates the program parts in the accelerator files to FE-GA binary. Input images are two 320x352 bitmap images. Data transfer between SH-4A and FE-GA is performed by SH-4A via data cache.

Fig.8. shows parallel processing performance of the optical flow on RP-X. The horizontal axis shows the processor configurations. For example, 8SH+4FE represents for the configuration with eight SH-4A general purpose cores and four FE-GA accelerator cores. The vertical axis shows the speedup against the sequential execution by a SH-4A core. As shown in Fig.8, the proposed compilation framework achieves speedups of up to 12.36x with 8SH+4FE.

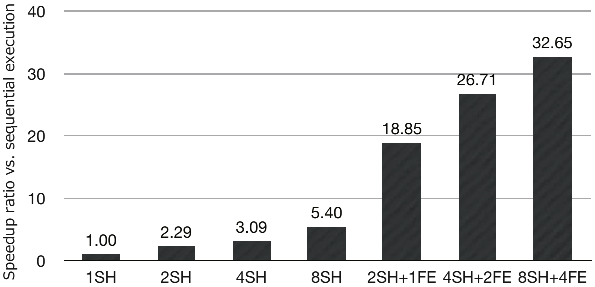


Fig. 9. Performance by OSCAR compiler and Hand-tuned Library(Optical Flow)

4.3 Performance by OSCAR Compiler and Hand-tuned Library

In this evaluation, we evaluate two applications written in Parallelizable C. The one is the optical flow from Hitachi Ltd. and Tohoku university, and the other is AAC encoder available on a market from Renesas Technology.

There are a few differences between the optical flow program used in this section and the program in Section4.2: In the optical flow program for this section, shift amount is 1 pixel, the input of the application is a sequence of images, and hand-tuned library for FE-GA is utilized. OSCAR compiler parallelizes the same loop, which is shown in the previous subsection. The hand-tuned library, which executes 81 SAD functions in parallel, is used for FE-GA. The hint directives are inserted to the parallelizable C program. OSCAR compiler generates parallel C program with OSCAR API or directives for these library function calls. The directives in the parallel program is translated to library calls by using API analyzer. Then, sequential compiler generates the executables linking with hand-tuned library for SAD. Input image size, number of frames and block size is 352x240, 450, 16x16, respectively. Data transfer between SH-4A and FE-GA is performed by SH-4A via data cache. AAC encoding program is based on the AAC-LC encode program provided by Renesas Technology and Hitachi Ltd. This program consists of filter bank, midside(MS) stereo, quantization and huffman coding. OSCAR compiler parallelizes the main loop which encodes a frame. The hand-tuned library for filter bank, MS stereo and quantization is used for FE-GA. Data transfer between SH-4A and FE-GA is performed by DTU via distributed shared memory.

Fig.9. shows parallel processing performance of the optical flow at RP-X. The horizontal axis shows the processor configurations. For example, 8SH+4FE represents for the configuration with eight SH-4A general purpose cores and four FE-GA accelerator cores. The vertical axis shows the speedup against the sequential execution by a SH-4A core. As shown in Fig.9, the proposed framework achieved speedups of up to 32.65x with 8SH+4FE.

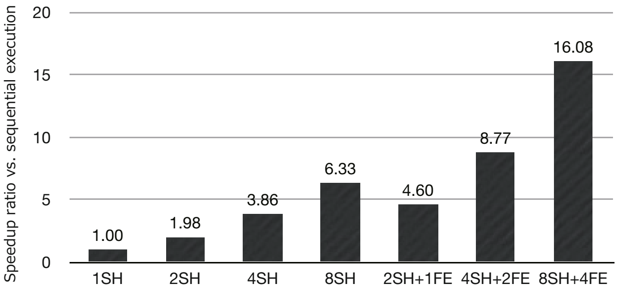


Fig. 10. Performance by OSCAR compiler and Hand-tuned Library(AAC)

Fig.10. shows parallel processing performance of the AAC at RP-X. As shown in Fig.10, the proposed framework achieved speedups of up to 16.08x with 8SH+4FE.

4.4 Evaluation of Power Consumption

This section evaluates a power consumption by using optical flow and AAC encoding for real-time execution on RP-X. Fig.11 shows the power reduction by OSCAR compiler's power control, under the condition satisfying the deadline. The deadline of the optical flow is set to 33ms for each frame processing so that standard 30 [frames/sec] for moving picture processing can be achieved. The minimum number of cores required for the deadline satisfaction of optical flow calculation is 2SH+1FE. As shown in Fig.11, OSCAR heterogeneous multicore compiler reduces from 65% to 75% of power consumption for each processor configuration. Although power consumption is increased by the augmentation of processor core, the proposed framework reduces the power consumption.

Fig.12 shows the waveforms of power consumption in the case of optical flow using 8SH+4FE. The horizontal axis and the vertical axis show elapsed time and a power consumption, respectively. In the Fig.12, the arrow shows a processing period for one frame, or 33ms. In the case of applying power control(shown in Fig.12. b), each core executes the calculation by changing the frequency and the voltage on a chip. As a result, the consumed power ranges 0.3 to 0.7[W] by OSCAR compiler's power control. On the contrary, in the case of applying no power control(shown in Fig.12. a), the consumed power ranges 2.25[W] to 1.75[W].

Fig.13 shows the summary of frequency and voltage status for optical flow calculation with 8SH+4FE. In this figure, FULL is 648MHz with 1.3V, MID is 324MHz with 1.1V, and LOW is 162MHz with 1.0V. Each box labeled "MID" and "timer" "Sleep" represents macro-task. As shown in Fig.13., four SAD tasks are assigned to each FE-GA, and the tasks are executed at MID. All SH-4A core except "CPU0" is shutdown until the deadline comes. "CPU0" executes "timer" task for satisfying the deadline. In other words, "CPU0" boot up other SH-4A cores when the program execution reaches the deadline. Note that FE-GA core is not shutdown after task execution because DVFS is only applicable.

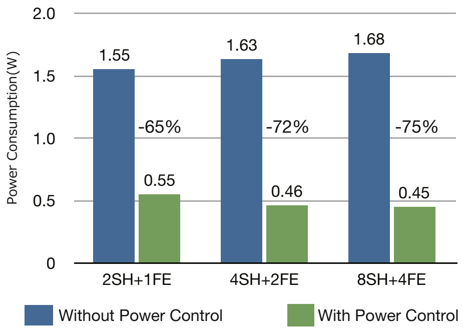
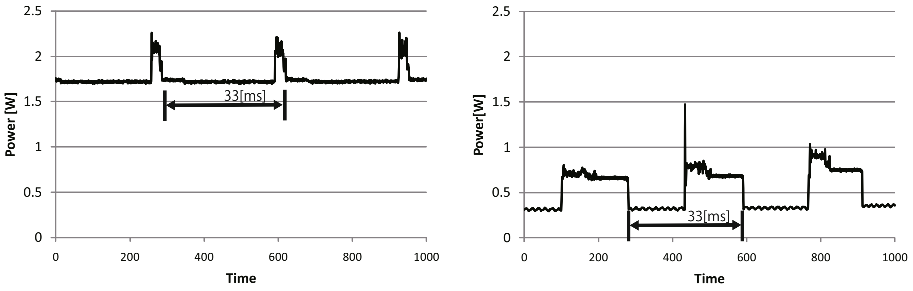


Fig. 11. Power reduction by OSCAR compiler's power control (Optical Flow)



a) Without Power Saving(Average:1.68W) b) With Power Saving(Average:0.45W)

Fig. 12. Waveforms of Power Consumption(Optical Flow)

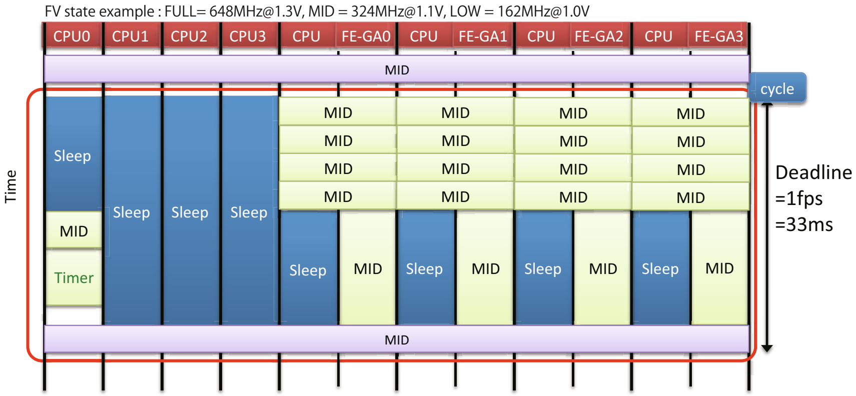


Fig. 13. Power Control for 8SH+4FE(Optical Flow)

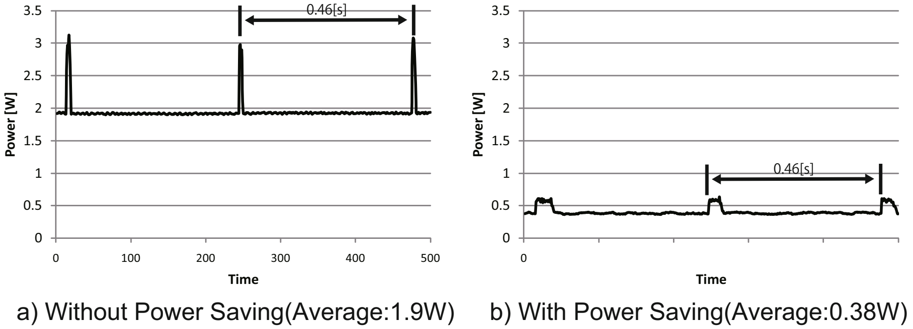


Fig. 14. Waveforms of Power Consumption(AAC)

For AAC program, an audio stream is processed per frame. The deadline of AAC is set to encode 1 [sec] audio data within 1 [sec]. Fig.14 shows the waveforms of power consumption in the case of AAC using 8SH+4FE. In the case of applying power control(shown in Fig.14. b)), each core execute the calculation by changing the frequency and the voltage on a chip. As a result, the consumed power ranges 0.4 to 0.55[W]. On the contrary, in the case of applying no power control(shown in Fig.14. a), the consumed power ranges 1.9[W] to 3.1[W]. In summary, the proposed framework realizes the automatically power reduction of heterogeneous multicore for several applications.

5 Conclusions

This paper has proposed OSCAR heterogeneous multicore compilation framework. In particular, this paper introduces (1)the general purpose and multiplatform automatic compilation flow using OSCAR compiler and various accelerator compilers or hand-tuned libraries and (2)the heterogeneous extension of OSCAR homogeneous API. In this paper, we have evaluated the processing performance and the power efficiency of the proposed framework using RP-X, 15 core heterogeneous multicore chip, as an example. The developed framework automatically gave us speedups of up to 32x for an optical flow program with eight general purpose processor cores and four accelerator cores against sequential execution. Also, it shows 80% of power reduction by automatic DVFS for the real-time AAC encoding execution mode with eight general purpose processor cores and four accelerator cores compared with no power control.

Acknowledgement

This work has been partly supported by the METI/NEDO project “Heterogeneous Multicore for Consumer Electronics” and MEXT project “Global COE Ambient Soc”. Specifications of OSCAR API[6] heterogeneous multicore extension are developed by NEDO Heterogeneous multicore architecture and API committee at Waseda university. The authors specially thanks to the members of the API committee from Fujitsu Laboratory, Hitachi, NEC, Panasonic, Renesas Technology, and Toshiba. The hand-tuned library for FE-GA is provided by Hariyama Lab. at Tohoku university and Hitachi.

References

1. Bellens, P., Perez, J.M., Badia, R.M., Labarta, J.: Cellss: a programming model for the cell be architecture. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC 2006 (2009)
2. Dolbeau, R., Bihan, S., Bodin, F.: Hmpp(tm):a hybrid multi-core parallel programmingg environment. In: GPGPU 2007: Proceedings of the 1st Workshop on General Purpose Processing on Graphics Processing Units (2007)
3. Garland, M., Grand, S.L., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y., Volkov, V.: Parallel computing experiences with cuda. IEEE Micro 28(4), 13–27 (2008)
4. Kasahara, H., Honda, H., Mogi, A., Ogura, A., Fujiwara, K., Narita, S.: A multigrain parallelizing compilation scheme for OSCAR (Optimally scheduled advanced multiprocessor). In: Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing, pp. 283–297 (August 1991)
5. Kasahara, H., Obata, M., Ishizaka, K.: Automatic coarse grain task parallel processing on SMP using openMP. In: Midkiff, S.P., Moreira, J.E., Gupta, M., Chatterjee, S., Ferrante, J., Prins, J.F., Pugh, B., Tseng, C.-W. (eds.) LCPC 2000. LNCS, vol. 2017, p. 189. Springer, Heidelberg (2001)
6. kasahara.cs.waseda.ac.jp: Oscar-api v1.0, `http://www.kasahara.cs.waseda.ac.jp/`
7. khronos.org: Opencl, `http://www.khronos.org/opencl/`
8. Kimura, K., Mase, M., Mikami, H., Miyamoto, T., Shirako, J., Kasahara, H.: OSCAR API for real-time low-power multicores and its performance on multicores and SMP servers. In: Gao, G.R., Pollock, L.L., Cavazos, J., Li, X. (eds.) LCPC 2009. LNCS, vol. 5898, pp. 188–202. Springer, Heidelberg (2010)
9. Luebke, D., Harris, M., Govindaraju, N., Lefohn, A., Houston, M., Owens, J., Segal, M., Papakipos, M., Buck, I.: Gpgpu: General-purpose computation on graphics hardware. In: 2006 ACM/IEEE Conference on Supercomputing, SC 2006 (11 November 2006 through 17 November 2006 2006)
10. Luk, C., Hong, S., Kim, H.: Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping, microarchitecture. In: Proceedings of42th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-42 (2009)
11. Masayasu, Y., Takeshi, S., Toshiaki, T., Yasuhiko, K., Toshinori, I.: Naviengine 1, system lsi for smp-based car navigation systems. NEC TECHNICAL JOURNAL 2(4) (2007)
12. Mase, M., Onozaki, Y., Kimuraa, K., Kasahara, H.: Parallelizable c and its performance on low power high performance multicore processors. In: Proc. of 15th Workshop on Compilers for Parallel Computing (July 2010)
13. Nakajima, M., Yamamoto, T., Yamasaki, M., Hosoki, T., Sumita, M.: Low power techniques for mobile application socs based on integrated platform "uniphier". In: ASP-DAC 2007: Proceedings of the 2007 Asia and South Pacific Design Automation Conference (2007)

14. opencv.org: Opencv, http://opencv.org/
15. Pham, D., Asano, S., Bolliger, M., Day, M.N., Hofstee, H.P., Johns, C., Kahle, J., Kameyama, A., Keaty, J., Masubuchi, Y., Riley, M., Shippy, D., Stasiak, D., Suzuoki, M., Wang, M., Warnock, J., Weitzel, S., Wendel, D., Yamazaki, T., Yazawa, K.: The design and implementation of a first-generation cell processor. In: 2005 IEEE International Solid-State Circuits Conference, ISSCC (6 February 2005 through 10 February 2005 2005)
16. Shirako, J., Oshiyama, N., Wada, Y., Shikano, H., Kimura, K., Kasahara, H.: Compiler control power saving scheme for multi core processors. In: Ayguadé, E., Baumgartner, G., Ramanujam, J., Sadayappan, P. (eds.) LCPC 2005. LNCS, vol. 4339, pp. 362–376. Springer, Heidelberg (2006)
17. Torii, S., Suzuki, S., Tomonaga, H., Tokue, T., Sakai, J., Suzuki, N., Murakami, K., Hiraga, T., Shigemoto, K., Tatebe, Y., Obuchi, E., Kayama, N., Edahiro, M., Kusano, T., Nishi, N.: A 600mips 120mw 70 μ a leakage triple-cpu mobile application processor chip. In: ISSCC (2005)
18. Wada, Y., Hayashi, A., Masuura, T., Shirako, J., Nakano, H., Shikano, H., Kimura, K., Kasahara, H.: Parallelizing compiler cooperative heterogeneous multicore. In: Proceedings of Workshop on Software and Hardware Challenges of Manycore Platforms, SHCMP 2008 (June 2008)
19. Wolfe, M.: Implementing the pgi accelerator model. In: GPGPU 2010: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (2010)
20. Yoshida, Y., Kamei, T., Hayase, K., Shibahara, S., Nishii, O., Hattori, T., Hasegawa, A., Takada, M., Irie, N., Uchiyama, K., Odaka, T., Takada, K., Kimura, K., Kasahara, H.: A 4320mips four-processor core smp/amp with individually managed clock frequency for low power consumption. In: IEEE International Solid-State Circuits Conference, ISSCC (February 2007)
21. Yuyama, Y., Ito, M., Kiyoshige, Y., Nitta, Y., Matsui, S., Nishii, O., Hasegawa, A., Ishikawa, M., Yamada, T., Miyakoshi, J., Terada, K., Nojiri, T., Satoh, M., Mizuno, H., Uchiyama, K., Wada, Y., Kimura, K., Kasahara, H., Maejima, H.: A 45nm 37.3gops/w heterogeneous multi-core soc. In: IEEE International Solid-State Circuits Conference, ISSCC (February 2010)

Debugging Large Scale Applications in a Virtualized Environment

Filippo Gioachin, Gengbin Zheng, and Laxmikant V. Kalé

Department of Computer Science
University of Illinois at Urbana-Champaign
gioachin@ieee.org, {gzheng,kale}@illinois.edu

Abstract. With the advent of petascale machines with hundreds of thousands of processors, debugging parallel applications is becoming an increasing challenge. Aside from the complicated debugging techniques required to debug applications at such scale, it is often difficult to gain access to these machines for a sufficient period of time, if at all. Some existing parallel debuggers are capable of handling these machines, but they still require the whole machine to be allocated. In this paper, we present an innovative approach to address debugging on such extreme scales. By leveraging the concept of object-based processor virtualization, our technique enables debugging of even a million processor execution under a simulated environment using only a relatively small cluster. We describe the obstacles we overcame to achieve this goal within two message passing programming models: CHARM++ and MPI. We demonstrate the results using real world applications such as Molecular Dynamics and Cosmological simulation programs.

1 Introduction

Debugging a parallel application requires numerous iterative steps. Initially, the application is tested on simple benchmarks on a few processors. At this point, many errors due to the communication exchanges between the processes in the parallel scenarios can be captured. Later, during production runs, the application will be deployed with larger input datasets, and on much bigger configurations. Oftentimes, the application will not behave as expected, and terminate abnormally. When this happens, the programmer is left to hunt the problem at the scale where it manifests, with possibly thousands of processors involved. If lucky, he may be able to recreate the problem on a smaller scale and debug it on a local cluster, but this is not always possible.

One example of a bug that may not be reproduced on a smaller scale is when the bug is located in an algorithm, and this algorithm depends on how the input data is partitioned among the processors. Reducing the problem size might be a solution to scale down the problem, but the inherent physics of the problem may not allow that. Another example is when the physics simulation output is incorrect. In this case, the problem can derive from rare conditions that only big datasets expose. Again, the problem size may not be reduced since otherwise the bug disappears. In all these examples, the only alternative left to the programmer is to use the whole machine, and debug with the full problem size on possibly thousands of processors.

K. Cooper, J. Mellor-Crummey, and V. Sarkar (Eds.): LCPC 2010, LNCS 6548, pp. 199–214, 2011.

Interactive sessions on large parallel machines are usually restricted to small allocations. For large allocations, batch scheduling is often required. To debug the application, the programmer will have to launch the job through the scheduler and be in front of the terminal when the job starts. Unless a specific allocation slot is pre-requested, this can happen at unpredictable, inconvenient times. Furthermore, the nature of debugging is such that it may require multiple executions of the code to track the bug, and to try different potential solutions. This exacerbates the problem and leads to highly inefficient debugging experience.

Moreover, debugging sessions on a large number of processors are likely to consume a lot of allocation time on supercomputers, and significantly waste precious computation time. During an interactive debugging session, the programmer usually lets the program execute for some time and then pauses it to inspect its data structures, then iteratively advances it step-by-step, while monitoring some data of interest. Therefore, processors are idle most of the time waiting for the user to make a decision on what to do next, which is a very inefficient use of supercomputers.

The innovative approach we describe in this paper is to enable programmers to perform the interactive debugging of their applications at full scale on a simulated target machine using much smaller clusters. We do this by making each processor in the application a *virtual processor*, and mapping multiple virtual processors to a single physical processor. This reduces the processor count needed for debugging. This mapping is transparent to the application, and only the underlying runtime system needs to be aware of the virtualization layer. A parallel debugger connected to the running application presents to the programmer the vision of the application running on thousands of processors, while hiding the fact that maybe only a few dozen were actually used.

Our idea transcends the programming model used for the virtualization and how the debugging infrastructure is implemented. However, to prove the feasibility of this approach, we implemented it within the CHARM++ runtime system [1,2], using the BigSim emulation environment, and the CHARMDEBUG debugger. Thus, applications written in CHARM++ will be the main target for our debugging examples. MPI applications are supported via a virtualized MPI implementation called AMPI [3].

In the remainder of this paper, we start by describing the infrastructure of the debugger, CHARMDEBUG, in Section 2 and BigSim Emulator in Section 3. We present the object-based virtualization approach we adopted to integrate the two systems into a virtualized debugger in Section 4. Section 5 further describes how we applied this method in the context of debugging MPI applications. Sections 6 and 7 analyze our system in terms of overhead and functionality with some examples. Related work is described in Section 8 followed by some comments on future work in the concluding section.

2 CharmDebug

CHARMDEBUG [4] is a graphical debugger designed for CHARM++ applications. It consists of two parts: a GUI with which a programmer interacts, and a plugin inside the CHARM++ runtime itself. The GUI is the main instrument that a programmer will see when debugging his application. It is written in Java, and is therefore portable to all operating systems. A typical debugging session is shown in Figure 1. The user will start the CHARMDEBUG GUI on his own workstation. He can then choose to start a new application to debug, or attach to a running application manually, using the appropriate commands available in the GUI. By default, every CHARM++ application contains a CHARMDEBUG plugin inside. This plugin is responsible to collect information from the running application, and to communicate with the CHARMDEBUG GUI. With this plugin integrated in the application itself, no external tool is necessary on every compute node. Thanks to the tight coupling between these two components of CHARMDEBUG, the user can visualize several kinds of information regarding his application. Such information includes, but is not limited to, the CHARM++ objects present on any processor and the state of any such objects, the messages queued in the system, and the memory distribution on any processor.

The communication between the CHARMDEBUG GUI and the CHARMDEBUG plugin happens through a high-level communication protocol called Converse Client-Server, or CCS [4]. This protocol has become a standard for CHARM++, and is built into all CHARM++ applications. It can be used both by the user directly into his own application, for example to enable live streaming of images to remote clients, or internally by the system, as in this case by CHARMDEBUG to collect status information. The CCS server, which is the parallel application in the case of CHARMDEBUG, opens a single socket connection and listens to it for incoming connections. Later, the CHARMDEBUG CCS client initiates the communication by connecting to this socket and sending a request. This request is translated by the server (i.e. the application), into a CHARM++ level message which is then delivered to a pre-registered routine. This routine can perform any operation it deems necessary, including operations involving parallel computations. This leverages the message-driven scheduler running on each processor in CHARM++: in addition to dealing with application messages, the scheduler also naturally handles messages meant for debugging handlers. Finally, the server can return an answer to the waiting client, if appropriate. Note that since only one single connection is needed between the debugger and the application under examination, we avoid the scalability bottleneck of having the debugger connect directly to each process of the parallel application. This allows CHARMDEBUG to scale to as large a configuration as CHARM++ does.

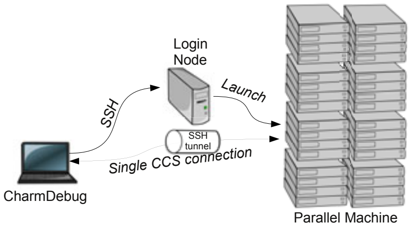


Fig. 1. Diagram of CHARMDEBUG's system

In CHARM++, every parallel application is integrated with debugging support in the form of a CHARMDEBUG plugin. When a program starts, this plugin registers inspection functions that the CHARMDEBUG GUI will send requests to. This initialization happens by default during CHARM++'s startup without the user intervention. Therefore, any program is predisposed for analysis with CHARMDEBUG. Although lacking direct connection to each processor, the user can request the debugger to open a

GDB [5] session for any particular processor. This gives the user flexibility to descend to a lower level and perform operations that are currently not directly supported by CHARMDEBUG.

3 BigSim Emulator

Although CHARMDEBUG as described in the previous section is implemented to be scalable and efficient for debugging very large scale applications, in practice, its usefulness is greatly impaired by the constraint of large amount of resource needed for debugging. This motivated the work in this paper to exploit a virtualized environment called BigSim to reduce the need for the whole machine.

BigSim [6,7] is a simulation framework that provides fast and accurate performance evaluation of current and future large parallel systems using much smaller machines, while supporting different levels of fidelity. It targets petascale systems composed of hundreds of thousands of multi-core nodes. BigSim consists of two components. The first component is a parallel *emulator* that provides a virtualized execution environment for parallel applications. This emulator generates a set of event logs during execution. The second component is a post-mortem trace-driven parallel *simulator* that predicts parallel performance using the event logs as input, and supports multiple resolutions for prediction of sequential and network performance. For example, the simulator can (optionally) predict communication performance accurately by simulating packets of each message flowing through the switches in detail, using a parallel discrete event simulation technique. Since the simulator only considers the trace logs and does not re-execute the application at the code level, it is not suitable for debugging purpose. However, the BigSim Emulator, which supports emulation of a very large application using only a fraction of the target machine, is useful for debugging. In the remainder of this section, we shall focus our attention on the emulator component.

Since multiple target processors are emulated on one physical processor, the memory usage on a given physical processor may increase dramatically. It may thus become impossible to fit the whole application into the physical memory available. Interestingly, our studies show that many real world scientific and engineering applications, such as molecular dynamics simulation, do not require a large amount of memory. For example, in one experiment, we were able to emulate NAMD [8] running on a 262,144-core Blue Waters machine [9] using just 512 nodes of the Ranger cluster, a Sun Constellation Linux Cluster at the Texas Advanced Computing Center (TACC).

For applications with large memory footprint, the physical amount of memory available per processor indeed poses a constraint. However, even in this scenario, we can still emulate these applications by using an efficient out-of-core technique [10,11] optimized for the BigSim Emulator. Clearly, out-of-core execution, even with optimization, incurs a much higher overhead than the pure in-memory execution, mainly due to the constraint imposed by disk I/O bandwidth. For example, we observed a slowdown of about 18 times in terms of the total execution time of a Jacobi application in [10].

For interactive debugging, the degraded performance due to the out-of-core execution may impact the user experience with slow responsiveness especially when the user requests involve all the virtual processors on disk. Increasing the number of emulating processors, and hence memory, helps reducing the need for extensive disk I/O in the out-of-core execution. Even though inefficient, this is a viable debugging solution when there is no other workaround.

4 Debugging CHARM++ Applications on BigSim

In order to combine the BigSim emulation system with the CHARMDEBUG debugging framework, several new problems had to be solved. Most arose from the fact that CHARMDEBUG needs to deal with the virtualized CHARM++ and other virtualized layers in the emulation environment.

Normally, CHARM++ is implemented directly on top of CONVERSE, which is responsible for low-level machine-dependent capabilities such as messaging, user-level threads, in addition to message-driven scheduling. This is shown on the left branch of Figure 2. When CHARM++ is re-targeted to the BigSim Emulator, there are multiple target CHARM++ virtual processors running on one physical processor, as explained in the previous section. Therefore, all layers underneath CHARM++ must be virtualized. This new software stack is shown in the same Figure 2, on the right branch. Specifically, the virtualized CONVERSE layer becomes BigSim CONVERSE, which is the CONVERSE system implemented using the BigSim Emulator as communication infrastructure. This is equivalent to treating the BigSim Emulator as a communication sub-system.

4.1 Communicating with Virtual Processors

One problem we had to overcome was the integration of the CCS framework into BigSim. CCS connects CHARMDEBUG and a running application considering each operating system process as an individual CHARM++ processor. However, in the BigSim Emulation environment, CCS is unaware of the emulated target processors because it is implemented directly on CONVERSE. Therefore, it needs to be adapted to the emulation system so that the CHARMDEBUG client can connect to the emulated virtual processors. To achieve this, we created a middle layer for CCS (virtualized CCS) so that messages can reach the destination virtual processor. The target of a CCS message becomes now the rank in the virtual processor space. Figure 3 depicts the new control flow.

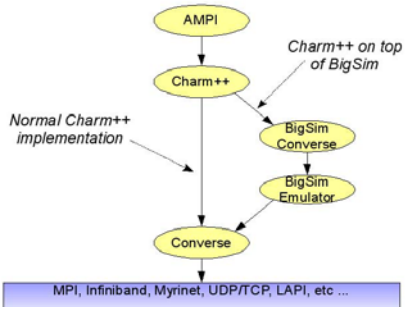

AMPI
Charm++
Charm++ on top of BigSim
Normal Charm++ implementation
BigSim Converse
BigSim Emulator
Converse
MPI, Infiniband, Myrinet, UDP/TCP, LAPI, etc ...

Fig. 2. BigSim Charm++ Software Stack

When a CCS request message is sent from CHARMDEBUG to a virtual processor, the message first reaches the CCS host (1). From here, it is routed to the real processor where the destination virtual processor resides (2). The processor level scheduler in CONVERSE will pick up the request message, but not execute the message immediately. Instead, it enqueues the message to the corresponding virtual node, and activates it (3).

The scheduler on the virtual node will serve the CCS request by invoking the function associated with the request message (4), and return a response message. Notice that the response does not need intervention from Converse since the virtual processor has direct access to the data structures stored in the common address space. Multicast and broadcast requests are treated in the virtualized environment. While this can add some overhead to the execution of a CCS request, it greatly simplifies the system, and the code reuse between the emulated and non-emulated mode.

Some CCS request messages are not bound to any specific virtual processor. For example, CharmDebug may send CCS requests to physical processors to query processor-wide information such as those related to the system architecture or the memory system. However, since all virtual processors on the same physical processor have access to the processor information including the whole memory, any of these virtual processors can, in fact, serve the CCS requests. Therefore, our approach is to have CharmDebug client always send such CCS requests to a virtual processor on a physical processor. This approach greatly simplifies the design and implementation of the CCS protocol, since we eliminate the need of having to specify if the request needs to be treated at the physical processor level, or at the virtual processor level.

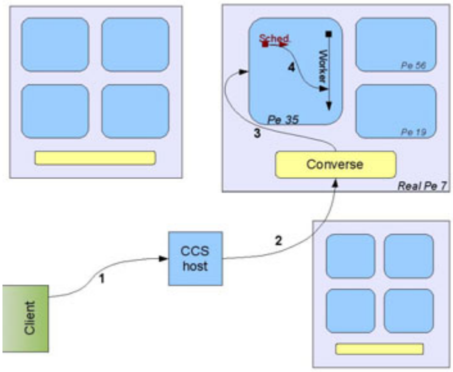


Fig. 3. Diagram of CCS scheme under BigSim Emulation

4.2 Suspending Virtual Processors

Another challenge was to figure out how to suspend the execution of a single virtual processor. Notice that while a processor is suspended, we still want to deliver messages to it. For example, requests from the debugger should be honored regardless of the processor's state. At the same time, we do not want other virtual processors emulated inside the same physical processor to be affected. In the non-virtualized environment, the technique we use to suspend a processor is to enter a special scheduler when the processor needs to be suspended. In this mode, regular messages are placed into a queue, and buffered in FIFO order until the processor can handle them. This scheduler is also in charge of driving the network, and receiving incoming messages. In this way, commands from the debugger can still be executed. In the virtualized environment, the scheduler that drives the network and forwards messages to the virtual processes is a separate entity from the scheduler inside each virtual processor. In this case, it is not possible to have each virtual processor driving the network, which will be too chaotic.

We modified our scheme to move the buffering of messages inside each individual virtual processor. When a worker processor needs to suspend due to an explicit debugger "freeze" command or due to a breakpoint, it calls its own scheduler recursively.

Since this scheduler is stateless, such a recursive scheme is feasible. This new scheduler then starts the buffering of messages. When the processor is released by the debugger, and is allowed to continue its normal execution, we terminate the internal scheduler, and return control to the outer one. Buffered messages are guaranteed to be executed in the same order as they were received while we exit from the internal scheduler. Meanwhile, the main CONVERSE scheduler remains the only one that drives the network and receives messages. Moreover, the CONVERSE scheduler is always active, and never enters a buffering mode.

With the techniques described, we can now debug applications in the virtualized environment as if they were running on a real machine. We shall see an example of using CHARMDEBUG on a real application in section 7. In the future work section, we will outline other topics we plan to address.

5 Debugging MPI Applications on BigSim

Debugging a large scale MPI application on a smaller machine requires running multiple MPI "processes" on one processor. This can be done using existing MPI implementations, if allowed by the operating system. However, this is often infeasible for various reasons. First, operating systems often impose hard limits on the total number of processes allowed by a user on one processor, making it challenging to debug a very large scale application. Secondly, processes are heavy-weight in terms of creation and context switching. Finally, there are very few MPI implementations that support out-of-core execution, which is needed for running applications with large memory footprints.

To overcome these challenges, we adopted the same idea of processor virtualization used in CHARM++: each MPI rank is now a virtual processor implemented as a *light-weight* CONVERSE user-level thread. This leads to Adaptive MPI, or AMPI [3], an implementation of the MPI standard on top of CHARM++. As illustrated in Figure 4, each physical processor can host a number of MPI virtual processors (or AMPI threads). These AMPI threads communicate via the underlying CHARM++ and CONVERSE layers. This implementation also takes advantage of CHARM++'s out-of-core execution capability. Since AMPI is a multi-threaded implementation of the MPI standard, global variables in MPI applications may be an issue. AMPI provides a few solutions to automatically handle global variables [12] to ensure that an MPI application compiled against AMPI libraries runs correctly.

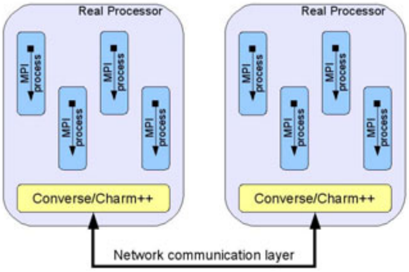
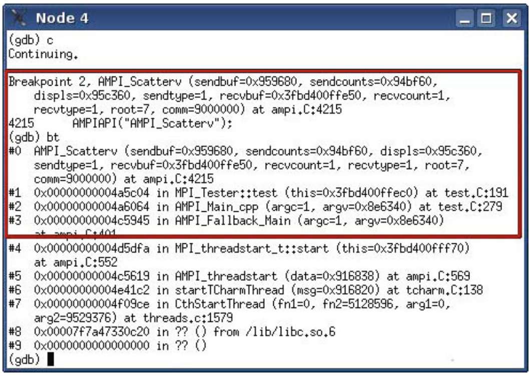


Fig. 4. AMPI virtualization using CHARM++

Debugging MPI applications can now use any arbitrary number of physical processors. For example, when debugging Rocstar [13], a rocket simulation program in MPI developed by the Center for Simulation of Advanced Rockets (CSAR) at the University of Illinois, a developer was faced with an error in mesh motion that only appeared



```
(gdb) c
Continuing.

Breakpoint 2, AMPI_Scatterv (sendbuf=0x959680, sendcounts=0x94bf60,
    displs=0x95c360, sendtype=1, recvbuf=0x3fbd400ffe50, recvcount=1,
    recvtype=1, root=7, comm=9000000) at ampi.C:4215
4215        AMPIAPI("AMPI_Scatterv");
(gdb) bt
#0  AMPI_Scatterv (sendbuf=0x959680, sendcounts=0x94bf60, displs=0x95c360,
    sendtype=1, recvbuf=0x3fbd400ffe50, recvcount=1, recvtype=1, root=7,
    comm=9000000) at ampi.C:4215
#1  0x00000000004a5c04 in MPI_Tester::test (this=0x3fbd400ffec0) at test.C:191
#2  0x00000000004a6064 in AMPI_Main_cpp (argc=1, argv=0x8e6340) at test.C:279
#3  0x00000000004c5945 in AMPI_Fallback_Main (argc=1, argv=0x8e6340)
    at ampi.C:401
#4  0x00000000004d5dfa in MPI_threadstart_t::start (this=0x3fbd400fff70)
    at ampi.C:552
#5  0x00000000004c5619 in AMPI_threadstart (data=0x916838) at ampi.C:569
#6  0x00000000004e41c2 in startTCharmThread (msg=0x916820) at tcharm.C:138
#7  0x00000000004f09ce in CthStartThread (fn1=0, fn2=5128596, arg1=0,
    arg2=9529376) at threads.c:1579
#8  0x00007f7a47330c20 in ?? () from /lib/libc.so.6
#9  0x0000000000000000 in ?? ()
(gdb) █
```

Fig. 5. Screenshot of GDB attached to a specific MPI rank, and displaying its stack trace

when a particular problem was partitioned for 480 processors. Therefore, he needed to run the application on a large cluster at a supercomputer center to find and fix the bug. However, the turn-around time for a 480 processor batch job was fairly long since the batch queue was quite busy at that time, which made the debugging process painfully slow. Using AMPI, the developer was able to debug the program interactively, using 480 virtual processors distributed over 32 physical processors of a *local* cluster, where he could easily make as many runs as he wanted to resolve the bug.

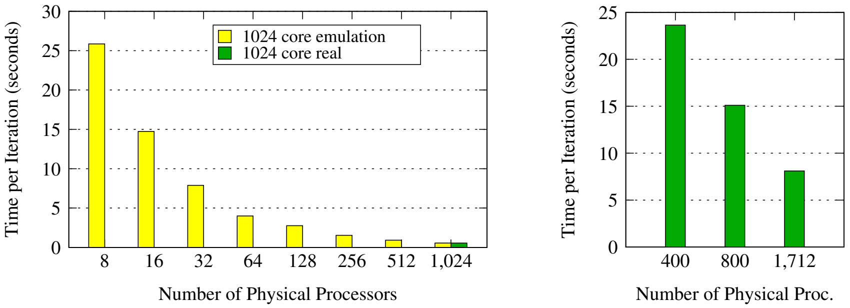
Since AMPI is implemented on top of CHARM++, the basic techniques for debugging as described in Section 4 work on AMPI programs automatically. In addition, if the user desires to perform more in-depth analysis on a specific MPI rank, he can choose to start a GDB sequential debugger attached to the processor hosting that rank, and focus on the desired rank. This GDB process is shown in Figure 5 for a simple test program. In this example, the user has set a breakpoint on MPI_Scatterv function, and when the breakpoint was hit, he printed the stack trace.

6 Debugging Overhead in the Virtualized Environment

In this section, we study the debugging overhead using a synthetic Jacobi benchmark and a real application, NAMD, running on the modified BigSim emulator with CHARMDEBUG support.

Our test environment is Blue Print, a Blue Waters interim system at National Center for Supercomputing Applications (NCSA). It is an IBM Power 5+ system. There are 107 compute nodes actually available for running a job, and each node has 16 cores (i.e. 1712 cores total).

We first tested a Jacobi3D program written in CHARM++ on 1024 virtual processors on a varying number of physical processors with CHARMDEBUG enabled, and measured the execution time per step. Figure 6(a) shows the results of the execution time with varying number of physical processors, from 8 to 1024. The last bar in the figure is the actual execution time of the same code on the 1024 processors with normal CHARM++. We can see that by using exactly same number of processors, Jacobi under



(a) 1024 emulated processors. The last bar is the actual runtime on 1024 processors.

(b) 1M (1,048,576) emulated processors.

Fig. 6. Jacobi3D execution time with varying number of physical processors

BigSim Emulator runs as fast as the actual execution in normal CHARM++, showing almost no overhead of the virtualization in BigSim. When we use fewer physical processors to run the same Jacobi emulation on 1024 virtual processors, the total execution time increases as expected. However, the increase in the execution time is a little less than the time proportional to the loss of processors. For example, when using 1024 physical processors, the execution time is 0.25s, while it takes only 23.96s when using only 8 physical processors. That is about 92 times slower (using 128 times fewer processors). This is largely due to the fact that most communication becomes in-node communication when using fewer processors.

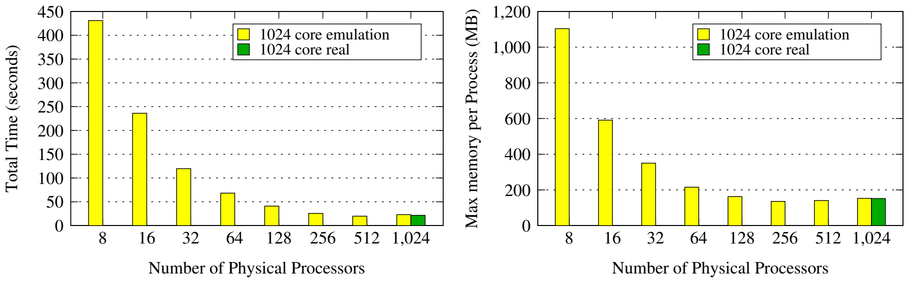
As a stress test, we ran the same Jacobi3D program on one million (1,048,576) emulated processors, while trying to use as fewer number of physical processors as possible. Figure 6(b) shows the execution time when running on 400, 800, and 1712 physical processors. These experiments show that it is feasible to debug an application in a virtualized environment for very large number of target processors using a much smaller machine.

To test how much time typical operations take from the debugger point of view, we used a similar Jacobi3D program, this time written in MPI. Table 1 reports timings for starting the MPI application, loading the list of messages queued on a given processor, and perform a step operation (deliver a single message) on all virtual processors. The latter two operations perform in an almost identical amount of time in all scenarios, including the case when the application is run in the non-virtualized environment.

We also studied the BigSim overhead on a real application. NAMD [14,8] is a scalable parallel application for Molecular Dynamics simulations written using the

Table 1. Time taken by the CHARMDEBUG debugger to perform typical operations, using MPI Jacobi3D application with 1024 emulated processors on varying number of physical processors

	8	16	32	64	128	256	512	1024	original
Startup (seconds)	11.60	11.63	13.34	13.12	15.86	14.41	16.45	17.71	17.85
Load a message queue (ms)	398	399	399	400	400	399	399	379	379
Single step, all pe (ms)	131	99	213	66	41	118	67	118	114



(a) Execution Time

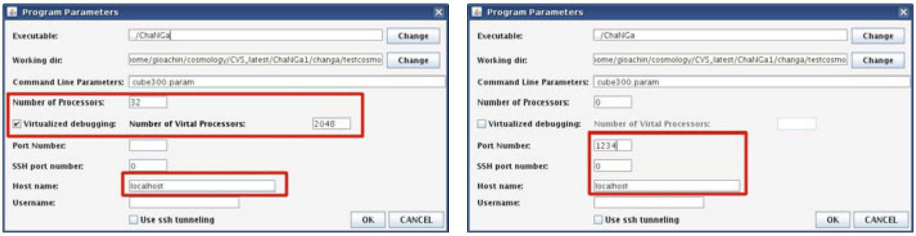
(b) Memory Usage

Fig. 7. NAMD on 1024 emulated processors using varying number of physical processors. The last bar is the actual run on 1024 processors.

CHARM++ programming model. It is used for the simulation of biomolecules to understand their structure. In these experiments, we ran NAMD on 1024 emulated processors with Apolipoprotein-A1 (ApoA1) benchmark for 100 timesteps. We measured the total execution time of each run (including startup and I/O) using a varying number of physical processors, from 8 to 1024. This is illustrated in Figure 7(a). Same as for Jacobi, we ran NAMD also in non-emulated mode using 1024 physical processors. The total execution time is shown in the last bar of the figure. We can see that NAMD running on the BigSim Emulator is only marginally slower (by 6%) compared to the normal execution on 1024 physical processors, showing little overhead of the emulator. On 512 processors, however, NAMD running in the emulation mode is even slightly faster than the actual run on 1024 processors. This is due to savings in the NAMD initial computation phases: faster global synchronization on fewer nodes.

Overall, this demonstrates that in terms of the time cost, debugging in a virtualized environment using much smaller number of processors is possible. Although it takes a longer time (19 times slower from 1024 to 8 processors) to run the application, debugging on a much smaller machine under a realistic scenario is not only easily accessible and convenient, but also simpler for setting up debugging sessions.

We further studied the memory overhead under the virtualized environment. Using the same NAMD benchmark on 1024 virtual processors, we gathered memory usage information for each processor. Figure 7(b) shows the peak memory usages across all physical processors. Again, the last bar is with the non-emulated CHARM++. Note that in emulation mode, the total memory usage is the sum of the application's memory usage across all emulated processors, plus the memory used by the emulator itself. It can be seen that there is no difference in memory usage between the emulation mode and non-emulation mode when using 1024 physical processors. When the number of processors decreases to 512, or even 256, the memory usage remains about the same. This is because NAMD has some constant memory consumption that dominates the memory usage (for example, read-only global data such as molecule database, which is replicated on each node), and the emulator itself tends to use less memory when the number of processors decreases. However, when the number of physical processors keeps reducing, each physical processor hosts a much larger number of emulated



Executable: ../ChaNGa
Working dir: ome/gioachin/cosmology/CVS_latest/ChaNGa1/changa/testcosmo
Command Line Parameters: cube300.param
Number of Processors: 32
☑ Virtualized debugging: Number of Virtal Processors: 2048
Port Number:
SSH port number: 0
Host name: localhost
Username:
□ Use ssh tunneling

(a) Launching scenario

Executable: ../ChaNGa
Working dir: ome/gioachin/cosmology/CVS_latest/ChaNGa1/changa/testcosmo
Command Line Parameters: cube300.param
Number of Processors: 0
□ Virtualized debugging: Number of Virtal Processors:
Port Number: 1234
SSH port number: 0
Host name: localhost
Username:
□ Use ssh tunneling

(b) Attach scenario

Fig. 8. Screenshots of CHARMDEBUG parameter window

virtual processors whose memory usage starts to dominate, therefore the total memory usage increases significantly. Nevertheless, when the number of physical processors is down to 8, the peak memory usage reaches about 1GB, which is still very feasible on machines nowadays. Note that this is an increase of only about 7 fold compared to the 1024 processor case, due to the sharing of the global read-only data at the process level.

In summary, we have demonstrated that debugging under virtualized environment incurs reasonably low overhead, considering the overhead proportional to the loss of processors. This makes it feasible to debug applications running on a large machine using only a portion of it.

7 Case Study

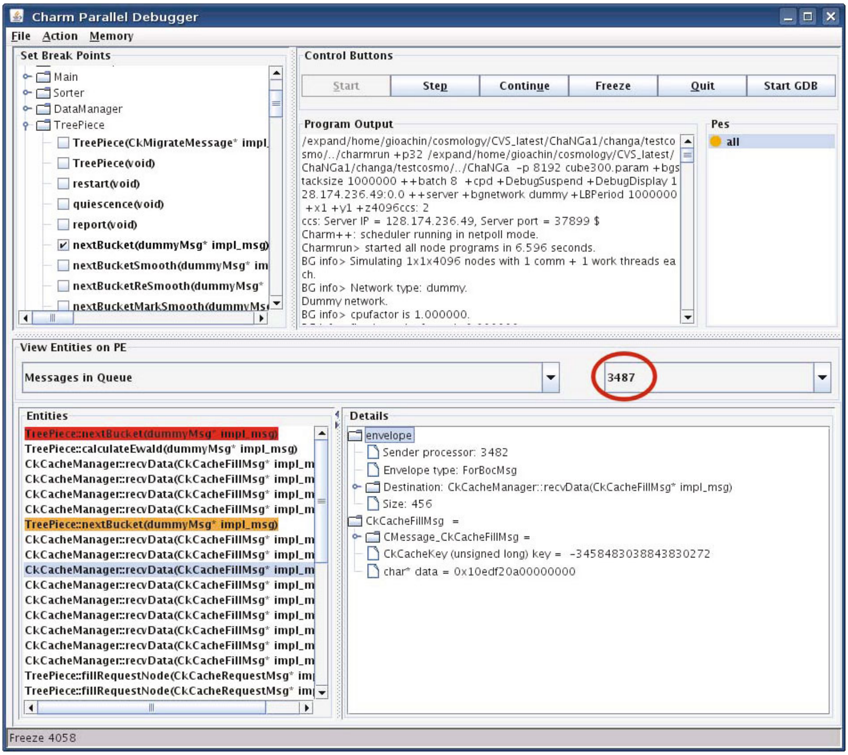
To demonstrate the capabilities of our technique, we used a few examples of complex applications, and debugged them in the virtualized environment. It is not the purpose of this section to describe actual bugs found with this technique, but rather illustrate how the user has access to all the same tools as in a normal scenario. With those tools, the user can search for the bug as he sees fit in the virtualized environment. Some applications have been described in section 6 while considering the overhead our technique imposes to the application under debugging. In this section, we use another real world application as an example.

CHANGA [15] is a production code for the simulation of cosmological evolution, currently in its second release. It is capable of computing standard gravitational and hydrodynamic forces using Barnes-Hut and SPH approaches respectively. This application is natively written in CHARM++, and it uses most of the language abstractions provided by the runtime system. While most of the computation is performed by CHARM++ *array elements*, which are not bound to the number of processors involved in the simulation, the application also uses CHARM++ *groups* and *nodegroups* for performance reasons. The groups have the characteristic of having one entity per processor, thus modifying the application behavior when scaling to larger number of processors. The complexity of this application is one reason why we chose it over other examples.

After the user has built the CHARM++ runtime system with support for BigSim emulation and compiled the CHANGA program over the virtualized CHARM++, he can start CHARMDEBUG's GUI. Figure 8(a) shows the dialogue box for the application parameters. In here, the user will indicate the location of his executable, the arguments, and the number of processors he wants to run on. The only difference from a standard non-virtualized execution is the presence of a checkbox to enable the virtualization. In general, the user will input the number of desired processors in the "Number of Processors" textfield and confirm. In this case, "Number of Processors" refers to the number of physical processors CHARMDEBUG will allocate on the machine. The number of processors the user wants to debug on has to be specified in the field named "Number of Virtual Processors". These fields are highlighted in the Figure. At this point the user can confirm the parameters, and start the execution of the program from CHARMDEBUG's main view.

If the machine to be used for debugging requires jobs to be submitted through a batch scheduler (or if the user desires to start the application himself), only the fields regarding executable location and CCS host/port connection need to be specified. These are highlighted in Figure 8(b). When the attach command is issued from the main view, the CHARMDEBUG plugin will automatically detect the number of processors in the simulation, and if the execution is happening in the virtualized environment.

Once the program has been started, and CHARMDEBUG has connected to it, the user can perform his desired debugging steps, oblivious of the fact that the system in using fewer resources internally. Figure 9 shows the CHANGA application loaded onto four



Main
Sorter
DataManager
TreePiece
□ TreePiece(CkMigrateMessage* impl_
□ TreePiece(void)
□ restart(void)
□ quiescence(void)
□ report(void)
☑ nextBucket(dummyMsg* impl_msg)
□ nextBucketSmooth(dummyMsg* im
□ nextBucketReSmooth(dummyMsg*
□ nextBucketMarkSmooth(dummyMs

```
/expand/home/gioachin/cosmology/CVS_latest/ChaNGa1/changa/testco
smo/../charmrun +p32 /expand/home/gioachin/cosmology/CVS_latest/
ChaNGa1/changa/testcosmo/../ChaNGa  -p 8192 cube300.param +bgs
tacksize 1000000 ++batch 8  +cpd +DebugSuspend +DebugDisplay 1
28.174.236.49:0.0 ++server +bgnetwork dummy +LBPeriod 1000000
 +x1 +y1 +z4096ccs: 2
ccs: Server IP = 128.174.236.49, Server port = 37899 $
Charm++: scheduler running in netpoll mode.
Charmrun> started all node programs in 6.596 seconds.
BG info> Simulating 1x1x4096 nodes with 1 comm + 1 work threads ea
ch.
BG info> Network type: dummy.
Dummy network.
BG info> cpufactor is 1.000000.
```

Messages in Queue

3487

TreePiece::nextBucket(dummyMsg* impl_msg)
TreePiece::calculateEwald(dummyMsg* impl_msg)
CkCacheManager::recvData(CkCacheFillMsg* impl_m
CkCacheManager::recvData(CkCacheFillMsg* impl_m
CkCacheManager::recvData(CkCacheFillMsg* impl_m
CkCacheManager::recvData(CkCacheFillMsg* impl_m
TreePiece::nextBucket(dummyMsg* impl_msg)
CkCacheManager::recvData(CkCacheFillMsg* impl_m
CkCacheManager::recvData(CkCacheFillMsg* impl_m
CkCacheManager::recvData(CkCacheFillMsg* impl_m
CkCacheManager::recvData(CkCacheFillMsg* impl_m
CkCacheManager::recvData(CkCacheFillMsg* impl_m
CkCacheManager::recvData(CkCacheFillMsg* impl_m
CkCacheManager::recvData(CkCacheFillMsg* impl_m
CkCacheManager::recvData(CkCacheFillMsg* impl_m
CkCacheManager::recvData(CkCacheFillMsg* impl_m
TreePiece::fillRequestNode(CkCacheRequestMsg* im
TreePiece::fillRequestNode(CkCacheRequestMsg* im

envelope
Sender processor: 3482
Envelope type: ForBocMsg
Destination: CkCacheManager::recvData(CkCacheFillMsg* impl_msg)
Size: 456
CkCacheFillMsg =
CMessage_CkCacheFillMsg =
CkCacheKey (unsigned long) key = -3458483038843830272
char* data = 0x10edf20a00000000

Freeze 4058

Fig. 9. Screenshot of ChaNGa debugged on 4,096 virtual processors using 32 real processors

thousand virtual processors. Underneath, we allocated only 32 processors from four local dual quad-core machines. In the bottom left part of the view, we can see all the messages that are enqueued in the selected processor (processor 3,487 in the Figure). Some messages have a breakpoint set (7^{th} message, in orange), and one has actually hit the breakpoint (1^{st} message, in red). In the same message list, we can see that some messages have as destination "TreePiece" (a CHARM++ array element), while others have as destination "CkCacheManager", one of the groups mentioned earlier. One such message is further expanded in the bottom right portion of the view (10^{th} message).

When joining multiple processes inside the same address space, the behavior of the system might be altered. First of all, one virtual processor could corrupt the memory belonging to another one. To solve this problem, the techniques described in [16] can be used. Another problem regards the kind of bugs that can be detected, in particular race conditions. By reducing the amount of physical processors available, a race condition might not appear anymore. A solution is to use record-replay techniques to force the execution of a particular message ordering. This is already available in the virtualized environment, as described in [17]. The other possibility is to force the delivery of messages in the virtualized environment in a different order each time.

8 Related Work

In the realm of parallel debugging, there are several tools that a programmer can use to understand why his program is misbehaving and correct the problem. Widely used commercial products are TotalView [18] from TotalView Technologies, and DDT [19] from Allinea. At least one of these tools is generally available in the majority of parallel supercomputers. Within the Open Source community, a tool worth mentioning is Eclipse [20]. Several Eclipse plugins have been developed to address parallel computing, in particular the Parallel Tools Platform (PTP) [21]. All these debuggers target applications written both in C/C++ and Fortran languages, and using MPI and/or OpenMP [22] as programming models. None of them supports the CHARM++ programming model natively. They all could manage CHARM++ programs if CHARM++ were built with MPI as its underlying communication layer. In this case, though, users would be exposed to the CHARM++ implementation, rather than their own program. Most importantly, while all the tools mentioned can scale to large number of processors, they all require the whole set of processors to be allocated. If the users desires to perform his debugging using one hundred thousand processors, then a big machine has to be used and occupied for long periods of time for the debugging to happen.

Virtualization for High Performance Computing has been claimed to be important [23]. Nevertheless, no tool known to the authors does, at present, provide a debugging environment tailored to thousands of processors or more, while utilizing only the few processors that a local cluster can provide. A few techniques have been developed in contexts other than High Performance Computing leveraging the concept of virtualization. These target the debugging of embedded systems [24], distributed systems [25], or entire operating systems using time-travel techniques [26,27]. All of them target virtual machines (such as Xen [28] or IBM Hypervisor [29]) where the entire operating system is virtualized. Using virtual machines may pose problems for a normal user as the installation and configuration of such virtual environments require administration privileges, and most supercomputers do not provide them by default. Our technique, instead, resides entirely in the user space, and does not suffer from this limitation.

9 Conclusions and Future Work

In this paper, we presented an innovative technique to address the issue of debugging applications on very large number of processors without consuming large amount of resources. In order to do this, we extended and integrated CHARMDEBUG, a debugger for CHARM++ applications, with BigSim, an emulator for large machines. By combining these two systems, and solving the resultant challenges in scaling and integration, we were able to provide the user a seamless debugging approach that uses much fewer processors than those requested by the user. This is accomplished by internally allocating multiple *virtual* processors inside each physical processor. We demonstrated the feasibility of this approach by studying the virtualization overhead with real world applications. We showed examples of the debugger used on many processors, displaying information about objects, breakpoint, and the content of each virtual processor. Furthermore, we also extended this technique to applications written in MPI, one of the most popular parallel programming model.

With the co-existence of multiple virtual processors inside the single address space of a physical processor, some memory operations have been disabled. For examples, searching for memory leaks. This and other operations require the debugger to disambiguate which virtual processor allocated the memory. One approach would be to use the same memory tagging mechanism described in [16] and cluster memory allocation by virtual processor.

Another future work regards MPI. As we described in section 5, currently CHARMDEBUG focuses primarily on applications written in CHARM++. While it can debug MPI applications using the AMPI implementation of the MPI standard, we realize that for a programmer debugging his application there may be unnecessary overhead. For the future, we are considering possible extensions to provide a more natural debugging also for MPI programs.

Acknowledgments

This work was supported in part by the NSF Grant OCI-0725070 for Blue Waters, and by the Institute for Advanced Computing Applications and Technologies (IACAT).

References

1. Kale, L.V., Zheng, G.: Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects. In: Parashar, M. (ed.) Advanced Computational Infrastructures for Parallel and Distributed Applications, pp. 265–282. Wiley-Interscience, Hoboken (2009)
2. Kale, L.V., Bohm, E., Mendes, C.L., Wilmarth, T., Zheng, G.: Programming Petascale Applications with Charm++ and AMPI. In: Bader, D. (ed.) Petascale Computing: Algorithms and Applications, pp. 421–441. Chapman & Hall / CRC Press (2008)

3. Huang, C., Zheng, G., Kumar, S., Kalé, L.V.: Performance Evaluation of Adaptive MPI. In: Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006 (March 2006)
4. Gioachin, F., Lee, C.W., Kalé, L.V.: Scalable Interaction with Parallel Applications. In: Proceedings of TeraGrid 2009, Arlington, VA, USA (June 2009)
5. Free Software Foundation, GDB: The GNU Project Debugger, http://www.gnu.org/software/gdb/
6. Zheng, G., Singla, A.K., Unger, J.M., Kalé, L.V.: A parallel-object programming model for petaflops machines and blue gene/cyclops. In: NSF Next Generation Systems Program Workshop, 16th International Parallel and Distributed Processing Symposium (IPDPS), Fort Lauderdale, FL (April 2002)
7. Zheng, G., Wilmarth, T., Jagadishprasad, P., Kalé, L.V.: Simulation-based performance prediction for large parallel machines. International Journal of Parallel Programming 33(2-3), 183–207 (2005)
8. Phillips, J.C., Zheng, G., Kumar, S., Kalé, L.V.: NAMD: Biomolecular simulation on thousands of processors. In: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, Baltimore, MD, September 2002, pp. 1–18 (2002)
9. National Center for Supercomputing Applications, Blue Waters project, http://www.ncsa.illinois.edu/BlueWaters/
10. Mei, C.: A preliminary investigation of emulating applications that use petabytes of memory on petascale machines. Master's thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign (2007), http://charm.cs.uiuc.edu/papers/ChaoMeiMSThesis07.shtml
11. Potnuru, M.: Automatic out-of-core exceution support for charm++. Master's thesis, University of Illinois at Urbana-Champaign (2003)
12. Negara, S., Zheng, G., Pan, K.-C., Negara, N., Johnson, R.E., Kale, L.V., Ricker, P.M.: Automatic MPI to AMPI Program Transformation using Photran. In: 3rd Workshop on Productivity and Performance (PROPER 2010), Ischia/Naples/Italy, vol. (10-14) (August 2010)
13. Jiao, X., Zheng, G., Alexander, P.A., Campbell, M.T., Lawlor, O.S., Norris, J., Haselbacher, A., Heath, M.T.: A system integration framework for coupled multiphysics simulations. Engineering with Computers 22(3), 293–309 (2006)
14. Bhatele, A., Kumar, S., Mei, C., Phillips, J.C., Zheng, G., Kale, L.V.: Overcoming scaling challenges in biomolecular simulations across multiple platforms. In: Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008 (April 2008)
15. Jetley, P., Gioachin, F., Mendes, C., Kale, L.V., Quinn, T.R.: Massively parallel cosmological simulations with ChaNGa. In: Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008 (2008)
16. Gioachin, F., Kalé, L.V.: Memory Tagging in Charm++. In: Proceedings of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD), Seattle, Washington, USA (July 2008)
17. Gioachin, F., Zheng, G., Kalé, L.V.: Robust Record-Replay with Processor Extraction. In: Proceedings of the 8th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD - VIII), Trento, Italy, pp. 9–19 (July 2010)
18. TotalView Technologies, TotalView® debugger, http://www.totalviewtech.com/TotalView
19. Allinea, The distributed debugging tool (DDT), http://www.allinea.com/index.php?page=48
20. T.E. Foundation, Eclipse - an open development platform, http://www.eclipse.org/

21. Watson, G.R., Rasmussen, C.E.: A strategy for addressing the needs of advanced scientific computing using eclipse as a parallel tools platform. Los Alamos National Laboratory, Tech. Rep. LA-UR-05-9114 (December 2005)
22. Dagum, L., Menon, R.: OpenMP: An Industry-Standard API for Shared-Memory Programming. IEEE Computational Science & Engineering 5(1) (January-March 1998)
23. Mergen, M.F., Uhlig, V., Krieger, O., Xenidis, J.: Virtualization for high-performance computing. SIGOPS Oper. Syst. Rev. 40(2), 8–11 (2006)
24. Pan, Y., Abe, N., Tanaka, K., Taki, H.: The virtual debugging system for developing embedded software using virtual machinery. In: Yang, L.T., Guo, M., Gao, G.R., Jha, N.K. (eds.) EUC 2004. LNCS, vol. 3207, pp. 139–147. Springer, Heidelberg (2004)
25. Gupta, D., Vishwanath, K.V., Vahdat, A.: Diecast: Testing distributed systems with an accurate scale model. In: Proceedings of the 5th USENIX Symposium on Networked System Design and Implementation (NSDI 2008). USENIX Association, Berkeley (2008)
26. Ta-Shma, P., Laden, G., Ben-Yehuda, M., Factor, M.: Virtual machine time travel using continuous data protection and checkpointing. SIGOPS Oper. Syst. Rev. 42(1), 127–134 (2008)
27. King, S.T., Dunlap, G.W., Chen, P.M.: Debugging operating systems with time-traveling virtual machines. In: ATEC 2005: Proceedings of the annual conference on USENIX Annual Technical Conference, p. 1. USENIX Association, Berkeley (2005)
28. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: SOSP 2003: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, pp. 164–177. ACM, New York (2003)
29. The research hypervisor: A multi-platform, multi-purpose research hypervisor, IBM Research, http://www.research.ibm.com/hypervisor/

Optimizing the Exploitation of Multicore Processors and GPUs with OpenMP and OpenCL

Roger Ferrer[1], Judit Planas[1], Pieter Bellens[1], Alejandro Duran[1],
Marc Gonzalez[1,2], Xavier Martorell[1,2], Rosa M. Badia[1,3],
Eduard Ayguade[1,2], and Jesus Labarta[1,2]

[1] Barcelona Supercomputing Center, Jordi Girona, 29. Barcelona, Spain
[2] Departament d'Arquitectura de Computadors, Univ. Politècnica de Catalunya,
Jordi Girona, 1–3. Barcelona, Spain
[3] IIIA, Artificial Intelligence Research Institute, CSIC
Spanish National Research Council, Spain
`name.surname@bsc.es`

Abstract. In this paper, we present OMPSs, a programming model based on OpenMP and StarSs, that can also incorporate the use of OpenCL or CUDA kernels. We evaluate the proposal on three different architectures, SMP, Cell/B.E. and GPUs, showing the wide usefulness of the approach. The evaluation is done with four different benchmarks, Matrix Multiply, BlackScholes, Perlin Noise, and Julia Set. We compare the results obtained with the execution of the same benchmarks written in OpenCL, in the same architectures. The results show that OMPSs greatly outperforms the OpenCL environment. It is more flexible to exploit multiple accelerators. And due to the simplicity of the annotations, it increases programmer's productivity.

1 Introduction

In this paper we present OMPSs, the programming model based on OpenMP and StarSs extensions, which can also include OpenCL or CUDA kernels, as a solution for easy programming of heterogeneous architectures. We have developed OMPSs to run on plain SMP machines, the Cell/B.E. processor, and SMP machines with GPUs. We are also making a port of the model for clusters.

From OpenMP we obtain high expressiveness to exploit parallelism using tasks. StarSs extensions allow runtime dependence analisys between tasks, and automatic data transfers. And OpenCL allows the programmer to easily write efficient and portable SIMD kernels to be exploited inside the tasks.

This approach is developed and evaluated for an Intel Xeon server (SMP, with 24 cores), a Cell/B.E.–based blade (with 2 Cell processors), and a machine with two NVIDIA GTX285 GPUs. Results show that OMPSs outperforms the equivalent code written in OpenCL, in the Xeon server, and the Cell/B.E. architectures. The proposal also outperforms the native CUDA implementations in NVIDIA GPUs.

K. Cooper, J. Mellor-Crummey, and V. Sarkar (Eds.): LCPC 2010, LNCS 6548, pp. 215–229, 2011.

Our approach allows the same source code to run on all three architectures (currently, except for the CUDA implementation of the kernels in the NVIDIA architecture), showing that productivity and performance is achievable in these environments.

The rest of the paper is organized as follows: Section 2 shows the proposal presented in this paper and it introduces the benchmarks used in the evaluation of the proposal. Section 3 presents the evaluation of the proposal. Section 4 shows the comparison with the related work, and Section 5 concludes the paper, and presents our future work.

2 Proposal

Our proposal is to have a single programming model covering the different homogeneous and heterogeneous architectures in use today, and opened to future ones. To achieve this goal, we have proposed OpenMP extensions to deal with devices, data movement, and data dependences [3].

Using these extensions, applications are annotated with directives avoiding as much as possible runtime library calls. Avoiding library calls is important to keep the source code of the application clean, and still be able to compile and debug the functionality of the application in a serial manner.

2.1 Brief Description of the Programming Model

In the proposed OMPSs programming model, programmers use tasks to express parallelism, like in OpenMP 3.0. Task pragmas can annotate sections of code to be outlined as functions, or functions written by the programmer. A new pragma, *target*, is used preceeding a task or a user function to specify that their execution can be done in specific *device*s, and whether the data accessed by the task code should be *copy_in*, *copy_out*, or *copy_inout*, from the point of view of the accelerator. The task specification is completed with its needs for data *input*, *output*, or *inout*. The runtime system will then ensure that the task is not activated until the data has been produced (e.g. by another task), and the runtime system will activate dependent tasks when data is available.

As a summary of the syntax used, Listing 1.1 shows the grammar that our Mercurium compiler recognizes. Our support for accelerators currently includes the *smp*, *cell*, and *cuda* devices.

Listing 1.1. Grammar of the proposed OpenMP extensions

```
#pragma omp target device(devnam,...) [implements (function_name)] \
    { [copy_deps] [copy_in(array_spec,...)] [copy_out(...)] [copy_inout(...)] }
#pragma omp task [input(...)] [output(...)] [inout(...)]
    { function or code block }
```

In this paper we have used the benchmarks Matrix Multiply (AMD OpenCL SDK [2]), and BlackScholes, Perlin Noise and Julia Set (IBM OpenCL SDK [8]). In the next subsections, we explain how they have been rewritten using OMPSs.

2.2 Matrix Multiply

The Matrix Multiply benchmark works on 2 input matrices (A, B), and an input/output matrix C, that has been initialized to zero. All accelerators write part of the result on a block of C, after computing the result for this block based on the input blocks of A and B.

Listing 1.2 shows the code implementing this benchmark. Pragmas in lines 2, 8, and 14 annotate the alternative functions computing a block of the matrix, so that each invocation in line 26 gets created as a task. In lines 8 and 14, the *implements* clause indicates that the functions *matmul_block_cl* and *matmul_block_gpu* will be used in devices *cell* and *cuda*. The function at line 3 will be used in the SMP environment. The *copy_deps* clause in lines 8 and 14 indicates to the compiler that the same data areas checked for data dependences (*input/inout* clauses in line 2) should be moved into and out of the accelerators, in case of execution on the Cell SPUs or NVIDIA GPUs.

Listing 1.2. Annotated blocked Matrix Multiply. Each block is NBxNB float values

```
1   const int NB = 512;
2   #pragma omp task inout([NB*NB] C) input([NB*NB] A, [NB*NB] B)
3   void matmul_block(float * A,float * B,float * C)
4   {
5      // plain C kernel code for the SMP environment
6   }
7
8   #pragma omp target device(cell) copy_deps implements(matmul_block)
9   void matmul_block_cl(float * A,float * B,float * C)
10  {
11     // OpenCL kernel code
12  }
13
14  #pragma omp target device(cuda) copy_deps implements(matmul_block)
15  void matmul_block_gpu (float * A,float * B,float * C)
16  {
17     // CUDA kernel code
18  }
19
20  void matmul (int m, int l, int n, int mDIM, int lDIM, int nDIM,
21                float ** A, float ** B, float ** C)
22  {
23    for(i = 0;i < mDIM; i++) {
24     for (j = 0; j < nDIM; j++) {
25      for (k = 0; k < lDIM; k++) {
26       matmul_block (A[i*lDIM+k],B[k*nDIM+j], C[i*nDIM+j]);
27      }
28     }
29    }
30  #pragma omp taskwait
31  }
```

2.3 BlackScholes

The BlackScholes benchmark computes the pricing of European-style options. Its kernel has 6 input arrays, and a single output. Listing 1.3 shows the annotated code. In this example, the pragmas are the same as in Matrix Multiply, but instead of annotating a function, the annotation is done on inline code. Lines 1 to 12 show the parallel loop. It creates a task for each *work group* of iterations, in the same way the OpenCL version does. Each task copies the 6 input parameters in (lines 3-5), executes the kernel (line 9), and at the end it copies the output array(*answer*) out of the accelerator (expressed in line 6).

The original version of this application was written in OpenCL. For the experiments with OMPSs in SMP and the Cell/B.E. processor we simply reuse the same OpenCL kernel code. To reuse the OpenCL code, we compile the code to an object file, and then the OMPSs task simply calls it as if it were a C function, as seen in line 9 of Listing 1.3. Listing 1.4 shows a portion of the OpenCL code. Observe the use of SIMD data types, *int4*, and *float4*, which direct the OpenCL compiler to use the SIMD units available, if any, in the processor cores. Observe also the expressiveness with respect to mathematical operators (addition, multiplication, etc.), and functions (log, sqrt) on SIMD data types.

Listing 1.3. Annotated BlackScholes

```
1   for (i=0; i<array_size; i+=work_group) {
2   #pragma omp target device(smp,cell,cuda) \
3       copy_in ( [work_group] &cpflag[i], [work_group] &S0[i], \
4                 [work_group] &K[i],      [work_group] &r[i],  \
5                 [work_group] &sigma[i],  [work_group] &T[i])  \
6      copy_out ( [work_group] &answer[i])
7   #pragma omp task shared (cpflag,S0,K,r,sigma,T,answer)
8      {
9         bsop_reference_float (&cpflag[i],&S0[i],&K[i],&r[i],
10                                  &sigma[i],&T[i],&answer[i]);
11     }
12  }
13  #pragma omp taskwait
```

Listing 1.4. BlackScholes OpenCL kernel. For reusing the OpenCL code, we compile the code to an object file, and then the OMPSs task simply calls it as if it were a C function

```
__kernel void bsop_reference_float(
       int4 * cpflag, float4 * S0, float4 * K,
     float4 * r,   float4 * sigma, float4 * T, float4 * answer)
{
 float4 d1, expval, Nd1, Nd2, call, put;
 ...
 for (x=0; x<work_group; x++) {
  d1 = log(S0[x]/K[x]) + (r[x] + HALF * sigma[x]*sigma[x])*T[x];
  d1 /= (sigma[x] * sqrt(T[x]));
  ...
  call = S0[x] * Nd1 - K[x] * expval * Nd2
  put =  K[0] * expval * (ONE - Nd2) - S0[x] * (ONE - Nd1);
  answer[x] = bitselect(put, call, as_float4(cpflag));
 }
}
```

For the GPU version, we have not been able to use this same technique yet, so we have translated the kernel into CUDA code. The translation is straightforward. And then we use a similar technique, shown in Listing 1.5 to exploit the CUDA code in the GPU from inside the task. We are currently working to overcome this limitation, and be able to use exactly the same OpenCL code also in the GPUs environment. Our compiler outlines the CUDA code, and then compiles the task with the NVIDIA compiler for the GPU. Observe that in the GPU case, we use the plain, non-SIMD, data types, as those get better performance in the GPU.

Listing 1.5. BlackScholes CUDA code, annotated with the task pragma, and using the CUDA syntax for kernel launching. The kernel has been written using non-SIMD data types for better performance in the GPUs

```
#pragma omp target device(cuda) ...
#pragma omp task ...
   {
   dim3 dimBlock (local_work_group);
   dim3 dimGrid  (work_group);
   cuda_bsop <<<dimGrid, dimBlock>>> (
                         cpflag, S0, K, r, sigma, T, answer);
   }
...
__global__
void  cuda_bsop ( int * cpflag, float * S0, float * K,
     float * r, float * sigma, float * T, float * answer)
{
 int x = blockIdx.x * blockDim.x + threadIdx.x;
 float d1, expval, Nd1, Nd2, call, put;
 ...
 d1 = log(S0[x]/K[x]) + (r[x] + HALF * sigma[x]*sigma[x])*T[x];
 d1 /= (sigma[x] * sqrt(T[x]));
 ...
 call = S0[x] * Nd1 - K[x] * expval * Nd2
 put =  K[0] * expval * (ONE - Nd2) - S0[x] * (ONE - Nd1);
 answer[x] = (cpflag[x])? put, call;
}
```

2.4 Perlin Noise

Perlin Noise has a single output, an image that is filled with noise to improve the realistic view of moving graphics, for example in games. Listing 1.6 shows the annotations used for Perlin Noise. Each task created by the loop starting in line 1, generates an horizontal slice of the image of height BS lines.

Listing 1.6. Annotated Perlin Noise

```
1 for (j = 0; j < img_height; j+=BS) {
2 #pragma omp target device(smp,cell,cuda) \
3              copy_out([BS*rowstride] optr)
4 #pragma omp task shared(optr)
5   {
6       // OpenCL / CUDA kernel
7   }
8 }
9 #pragma omp taskwait
```

2.5 Julia Set

Julia Set computes a series of images of the Julia Set fractal. Listing 1.7 shows the annotations used for this benchmark. Each task has one input, *julia_context*, a structure with the characteristics of the julia image to be generated. Among others, the image number to be generated, light position and intensity, and the spectator position, are passed in julia_context. As output, each task delivers a horizontal slice of the Julia fractal of height BS lines.

Listing 1.7. Annotated Julia Set

```
for (j = 0; j < img_height; j+=BS) {
#pragma omp target device(smp,cell,cuda) \
       copy_in(julia_context) copy_out([BS*rowstride] image)
#pragma omp task shared(out,julia_context) \
  {
      // OpenCL / CUDA kernel
  }
}
#pragma omp taskwait
```

As it can be observed, the benchmarks are easily annotated. Also, the proposed environment does not require to use the low level OpenCL or CUDA runtime calls to allocate and copy memory, compile the kernel code at runtime, or copy the results back to main memory. Our runtime system takes care of implementing memory allocation and data transfers and optimizing them.

3 Evaluation

This section presents the execution environments used for evaluation, and the evaluation of the benchmarks.

3.1 Execution Environments

Three execution environments have been used to evaluate the benchmarks:

- **Intel Xeon server.** This is a machine with 4 Intel Xeon chips, with 6 cores each, for a total of 24 cores. Each chip runs at 2.4 Ghz, and it has a L2 cache of 12Mbytes, shared among the 6 cores, and a peak of 9.6 Gflops. The machine has 48 Gbytes of main memory (RAM). In the Intel server, we run the AMD/ATI OpenCL SDK [2], and we compare its performance to the OMPSs environment. In this SMP environment OMPSs relies on the shared memory, and it avoids all data copies. On the contrary, the native OpenCL runs still do data copying. This is the main source of improvement in this platform.
- **Cell/B.E.** The QS20 Cell/B.E.–based blades contain two Cell/B.E. processors, running at 3.2 Ghz, each with a L2 cache of 512Kbytes. The blade has 1 Gbyte of main memory. Each Cell/B.E. has 8 SPUs, for a total of 16 in the blade. In the Cell/B.E. we run the IBM OpenCL SDK [8], and we compare its performance to the StarSs environment for the Cell processor [17]. In this environment, OMPSs is able to exploit higher coarse granularity in the data transfers. This is because the OpenCL environment has a limit on the size of each work–item, and thus in the associated data used to compute on it.
- **NVIDIA GPUs**
  The NVIDIA GPUs are in a host with dual-chip dual-core Opteron AMD processors. It has 8 Gbytes of main memory. Two NVIDIA GeForce GTX 285 GPUs are connected through the PCI bus. The GPU clock is 1.476 Mhz, it contains 240 CUDA cores, and it has 1 GBytes of global memory. We had also the opportunity to run two of the benchmarks in an Intel host

(dual chip, each with 4 i7 975 cores at 3.33Ghz, and 24 Gbytes of main memory) with a Fermi GTX480 GPU. In the GPUs environment, we run the NVIDIA OpenCL SDK [14], and we compare its performance to the current porting of the OMPSs environment for GPUs. In this environment, a general comparison is more difficult to do. The coding of the kernels in CUDA seems to go against good performance in OMPSs. Nevertheless, the results achieved are still good.

3.2 On an Intel Xeon Server

Figure 1 presents the evaluation of Matrix Multiply on 512x512 float matrices, on the Xeon server. It shows the OpenCL version, and two versions running with OMPSs. 512-nb stands for the non-blocked version, and 512-b for the blocked version. As it can be observed, the OMPSs blocked version outperforms the other two versions. Blocked versions achieve better data locality, and it is the reason for the better performance. As soon as the matrix size is increased from 512x512 elements, the performance obtained by the OpenCL environment heavily decreases, as shown in Figures 2 and 3. Initially, we thought that the reason could be that the data set could exceed the L2 data cache, but this is not the case. We suspect that the ATI OpenCL implementation has some problem when dealing with the transfers of large data sizes.

Figure 4 shows the speedup obtained on the BlackScholes benchmark for the OpenCL and OMPSs versions, running from 1 to 24 cores in the Xeon–based machine. Both the OpenCL and OMPSs versions on a single core achieve a speedup of 2.6 over the serial version due to the vectorization achieved through the SIMD OpenCL kernel. OMPSs is consistently better than OpenCL. We attribute this benefit to the fact that in OMPSs for SMP machines, we do not need to copy any data to work in parallel, while in OpenCL the data copies are done in the same way as for heterogeneous machines, as the data movement is coded in the application itself by the programmer.

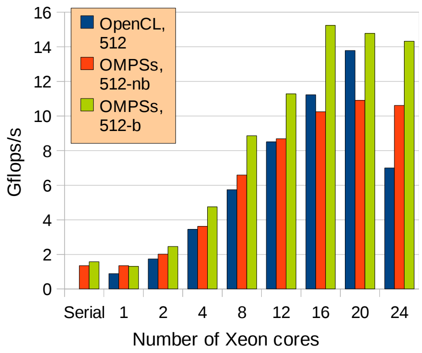


Fig. 1. Evaluation of Matrix Multiply (512x512) in an Intel Xeon server

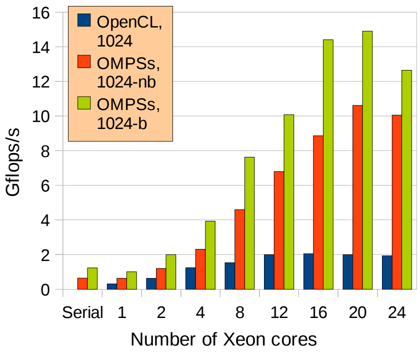


Fig. 2. Evaluation of Matrix Multiply (1024x1024) in an Intel Xeon server

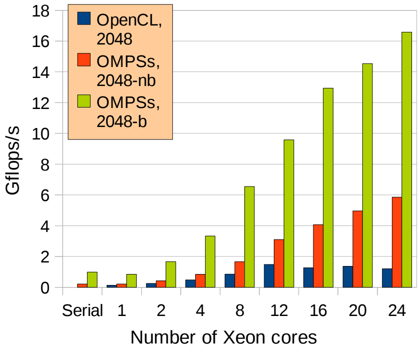


Fig. 3. Evaluation of Matrix Multiply (2048x2048) in an Intel Xeon server

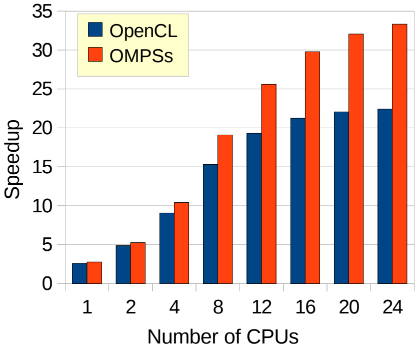


Fig. 4. Evaluation of BlackScholes in an Intel Xeon server

Figure 5 shows the results obtained from the Perlin Noise benchmark. The plot shows millions of pixels per second processed by the benchmark when running from 1 to 24 cores. In this case, we compared the performance of the OpenCL kernel with the C code compiled with gcc. We found out that gcc is achieving better performance on the inner kernel of this application. Our opinion is that for some reason gcc is able to exploit better the SIMD units of the Xeon processor. This situation does not happen in the other benchmarks. This shows the importance of being able to reuse the best code generated for an application, as we can do with OMPSs. This is something that it is not immediate for the OpenCL environment, as in it the kernel is compiled during runtime, and it will not be easy to have gcc generating the code for the kernel at runtime.

Figure 6 shows the results obtained from the Julia Set benchmark. The plot displays millions of pixels per second, obtained when running the benchmark from 1 to 24 cores. From the results obtained we also conclude that our OMPSs approach is performing better with respect the OpenCL version of the benchmark, being much easier to code.

3.3 On the Cell/B.E. Processor

In the Cell processor environment, we have used the CellSs flavor [17] of OMPSs. In this environment, we have found that the OpenCL Matrix Multiply benchmark achieves very poor performance compared to the hand–tuned SDK version, so that it is not interesting to show results on this benchmark. We present the results obtained in BlackScholes, Perlin Noise and Julia Set.

Figure 7 presents the results obtained from the BlackScholes benchmark when run from 1 to 16 SPUs. The OpenCL version can be compiled using different techniques to execute the OpenCL kernel. Those techniques using the OpenCL intrinsic *async_work_group_copy* are the best behaving in the Cell processor. The bar labeled *OpenCL, awgc* shows the results when using OpenCL ranges and such intrinsic. The bar labeled *OpenCL, db* presents the results when using the OpenCL task approach and also such intrinsic, in this case to implement *double*

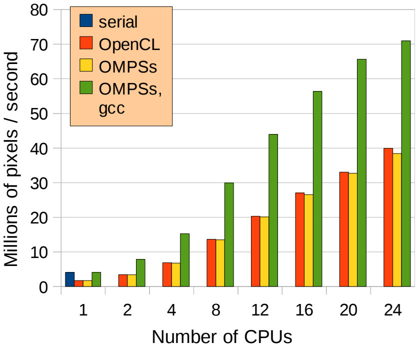


Fig. 5. Evaluation of Perlin-Noise in an Intel Xeon server

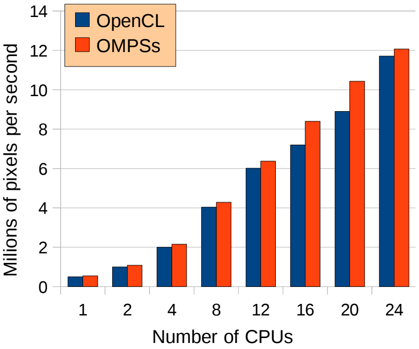


Fig. 6. Evaluation of Julia-Set in an Intel Xeon server

buffering. It is interesting to compare these results, as tasks with double buffering perform a little worse than using ranges and single buffering. This seems to be due to the overhead of creating the individual tasks in OpenCL, compared to the overhead of using OpenCL ranges.

The comparison of the OpenCL versions against OMPSs results highly beneficial for our approach, that reaches a 2x performance increase when running on 12 and 16 SPUs. This is again because our approach is able to reduce the number of data transfers. In OpenCL the application has to create special buffers to put the data on, an then the OpenCL kernel running on the SPUs issues the work group copies that trigger the SPUs DMA data transfers. This two–level copy is avoided in OMPSs, where the SPUs access the application data directly through the DMAs managed by the SPU runtime system.

Figure 8 shows the results obtained from the Perlin Noise benchmark. It shows millions of pixels per second, depending on the number of SPUs used. We also show two different alternatives for OpenCL, ranges with *asynchronous work group copies*, and tasks with *load/store* operations. For this benchmark, the latter performs better, but it is far from the performance of OMPSs. As in BlackScholes, the reduced number of copies done in OMPSs benefit performance.

Figure 9 compares the performance obtained from OpenCL and OMPSs on the Julia Set benchmark. In this case, the kernel of the benchmark is highly computational, so the impact of the data transfers is not so high. Nevertheless, the figure shows that OMPSs is having better scalability than OpenCL, as there is a consistent increase in the difference of performance between both approaches, as long as the number of SPUs is increased.

3.4 On NVIDIA GPUs

In this section we present the first evaluation of the OMPSs proposal on GPUs.

Figure 10 shows the results obtained in Matrix Multiply using the example in the CUDA SDK, the OpenCL example from the ATI SDK, and the OMPSs approach with the CUDA kernel. It presents Gflop/s, on matrix sizes of 512x512,

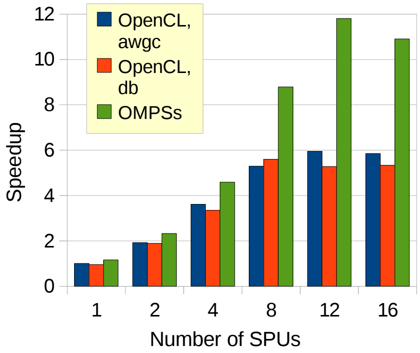


Fig. 7. Evaluation of BlackScholes in a Cell/B.E. blade

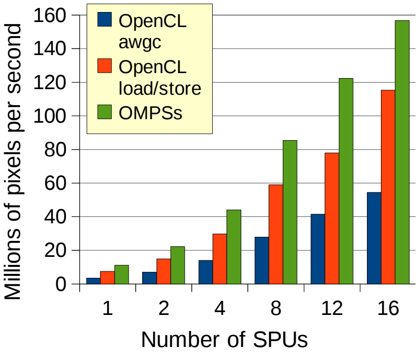


Fig. 8. Evaluation of Perlin-Noise in a Cell/B.E. blade

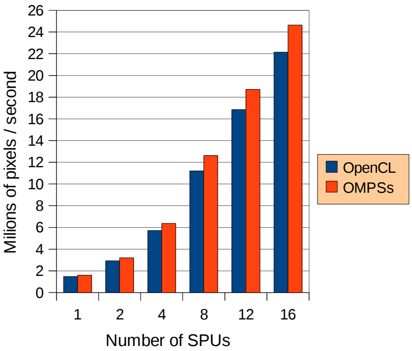


Fig. 9. Evaluation of Julia-Set in a Cell/B.E. blade

1024x1024 and 2048x2048, and in the case of OMPSs also using 1 and 2 GPUs. For the 512x512 matrix size, OMPSs performs very similarly to OpenCL, and both are a little bit better than CUDA. This is despite the fact that our OMPSs approach uses the CUDA kernel. For the 1024x1024 matrix size, OMPSs outperforms the CUDA and OpenCL environments. For the 2048x2048 matrix size, the three environments behave very similarly. In the NVIDIA CUDA, and OpenCL environments, it is not immediate to exploit more than one GPU. The source code of the application needs to be modified to access to the additional devices, and do the memory allocations and transfers.

Instead, with OMPSs it is our runtime system that accesses the number of devices indicated by the user through an environment variable (NX_GPUS, similarly to the traditional OMP_NUM_THREADS), and the application can automatically be exploited in several GPUs.

For matrix sizes larger than 1024x1024, OMPSs scales nicely when going from 1 to 2 GPUs. For the 1024x1024 matrix size and below, there is no gain in using

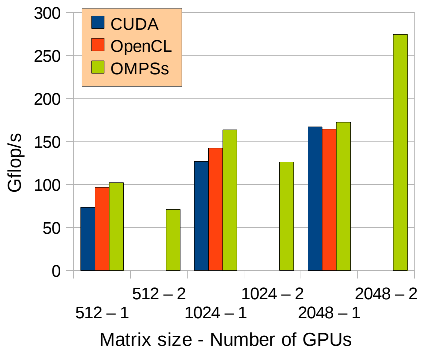


Fig. 10. Evaluation of Matrix Multiply in NVIDIA GTX285 GPUs

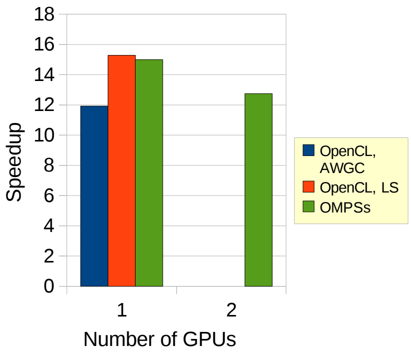


Fig. 11. Evaluation of BlackScholes in NVIDIA GTX285 GPUs

more GPUs because the data set is so small, and the overhead of having several data transfers to different devices is noticeable.

Figure 11 shows the speedup obtained from BlackScholes. It compares two alternative implementations of the OpenCL kernel, with the OMPSs approach. The latter uses the kernel translated to CUDA. As in the case of OMPSs on SMPs, the performance achieved is similar. OMPSs does not obtain any improvement when executing on two GPUs. This is because the data transfers dominate the execution time, as the parallel region has 6 input data arrays. Again, if we compare the results obtained in a single GPU we can state that the performance is the same, and the programming effort will be much less.

Figure 12 shows the results obtained from the Perlin Noise benchmark. Recently, we have had access to a NVIDIA Fermi GTX480 card, and we have executed Perlin Noise and Julia Set on it. The plot compares the performance obtained from the OpenCL and OMPSs versions of the benchmark, on one and two GPUs (two only for the GTX285). Looking at the evaluation on one GPU, we can appreciate that OMPSs outperforms OpenCL for both the GTX285 and the GTX480 GPUs. It is outstanding the increase of performance that the GTX480 hardware represents, and it is also outstanding the 2x speedup that OMPSs obtains in a single GPU over the OpenCL version of the benchmark. Finally, the benchmark scales from 1 to 2 GTX285 GPUs. We have not had access to hardware with two GTX480 GPUs yet.

Figure 13 shows the results obtained from the Julia Set benchmark. The plot shows the OpenCL and OMPSs versions of the benchmark on the GTX285 and GTX480 GPUs. The lower performance that OMPSs gets in the GTX285 GPU is due to the translation of the OpenCL kernel code into CUDA. We had to translate the OpenCL kernel code into CUDA because of the the limitation described in section 2.3, regarding the lack of ability to compile the OpenCL kernel code to be used in the OMPSs applications for the GPUs environment. It is interesting to note that even if the CUDA code obtains lower performance in

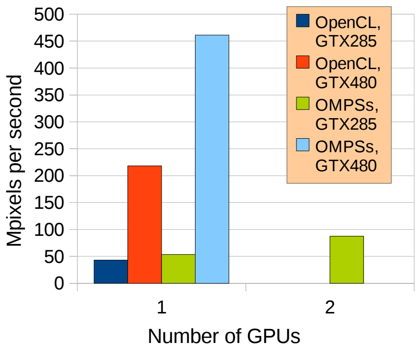


Fig. 12. Evaluation of Perlin Noise in NVIDIA GTX285 and GTX480 GPUs

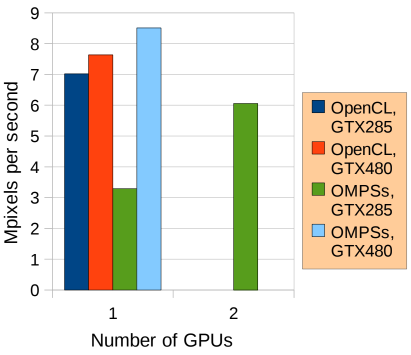


Fig. 13. Evaluation of Julia Set in NVIDIA GTX285 and GTX480 GPUs

GTX285 GPUs, OMPSs outperforms the OpenCL version of the benchmark in the GTX480 GPU. We think that this shows that our approach is well positioned for future GPU hardware. Notice also, that Julia Set scales nicely when executed on two GTX285 GPUs.

4 Related Work

New computer architecture designs based on heterogeneous multicores have raised the question about their programmability. The CAPS HMPP [6] toolkit is a set of compiler directives, tools and software runtime that supports parallel programming in C and Fortran. HMPP works based on *codelets* that define functions that will be run in a hardware accelerator. These codelets can either be hand–written for a specific architecture or be generated by some code generator. Offload [5] is a programming model for offloading portions of C++ applications to run on accelerators. Code to be offloaded is wrapped in an *offload* block, indicating that the code should be compiled for an accelerator, and executed asynchronously as a separate thread. Call graphs rooted at an offload block are automatically identified and compiled for the accelerator. Data movement between host and accelerator memories is also handled automatically.

Recently, general purpose computation on graphic processors has received a lot of attention as it delivers high performance computing at rather low price. Major processor vendors have showed their intent to integrate GPUs as a GPU–core in the CPU chip [9,1]. CUDA [13] is an extension to C++ proposed by NVIDIA. It is based on *kernels* that are run n times in parallel by n threads. Tools to better map the algorithms to the memory hierarchy have been proposed [19]. They advocate for that programmers should provide straight-forward implementations of the application kernels using only global memory, and that tools like CUDA-lite will do the transformations automatically to exploit local memories.

Most of the programming models suitable for heterogeneous multicores allow to express some form of task based parallelism. OpenMP 3.0 [16], the industry standard for parallelism in shared memory machines, introduces a *task* suitable for parallelization of irregular applications [4]. The Sequoia [11] alternative focuses on the mapping of the application kernels onto the appropriate engines to exploit the memory hierarchy. RapidMind [18] is a development and runtime platform that uses dynamic compilation to accelerate code for the accelerators available, being those GPUs or the Cell SPUs. The programmer encapsulates functions amenable for acceleration into program containers. The code in containers is only compiled during the execution of the application, so that it can be optimized dynamically depending on the input data and the target architecture.

Merge [12] encapsulates specialized languages targeting specialized accelerators (GPUs, FPGAs) in C/C++ functions to provide a uniform interface for them. Encapsulation is based on EXOCHI [20], which uses pragmas to offload the domain specific language to be compiled with the compiler of the target device. Merge allows the specification of the same function for different targets, as new intrinsic functions, and it provides the mechanism for dynamic function selection at runtime.

We have found that all solutions make the programmer to split the application in pieces to provide the low level kernels to the acceleration engines. As we propose with OMPSs, we think that the compiler must be the responsible to address this issue. This will increase productivity in multicore processors, specially in the heterogeneous ones. This is also the case of Offload [5] and the IBM compiler [7,15], also known as Octopiler, which takes OpenMP code and places the parallel regions on the Cell SPUs. We propose to augment the C/C++ languages to incorporate vector types like OpenCL does. This will allow to easily obtain performance from OpenMP parallel regions and vectorization at the same time, which is currently difficult with current programming models.

5 Conclusions and Future Work

This paper presents OMPSs, a proposal to improve programming on multicore processors and GPUs. The proposal improves productivity and achieves a performance similar or better than existing environments based on CUDA/OpenCL.

OMPSs is based on program annotations taking the best features from the tasks of OpenMP, StarSs dependence analysis and automatic generation of data transfers, and the expression of SIMD operations in OpenCL kernel codes.

The OMPSs proposal is evaluated with four benchmarks: Matrix Multiply, BlackScholes, Perlin Noise, and Julia Set; and using three distinct architectures: an Intel SMP, an IBM Cell/B.E.–based blade, and NVIDIA GPUs.

Results show that OMPSs outperforms OpenCL/CUDA implementations of the same benchmarks, in the three execution environments. In addition, it achieves a unified way of programming them in different environments. We propose to augment the C/C++ languages to incorporate vector types. This will allow to exploit parallelization with OpenMP and vectorization at the same time.

Our future work is focused on the improvement of OMPSs, including research on task scheduling in heterogenous environments, automatic tuning of data transfer sizes, and task granularity. In addition, we are porting larger applications to the programming model, to show that it is applicable in a general way to a variety of algorithms.

Acknowledgments

This work utilized the AC cluster [10] operated by the Innovative Systems Laboratory (ISL) at the National Center for Supercomputing Applications (NCSA) at the University of Illinois. The cluster was funded by NSF SCI 05-25308 and CNS 05-51665 grants along with generous donations of hardware from NVIDIA, Nallatech, and AMD.

We thankfully acknowledge the support of the European Commission through the ENCORE project (FP7-248647), the TERAFLUX project (FP7-249013), the TEXT project (FP7-261580), the SARC IP project (FP6-27648), and the HiPEAC-2 Network of Excellence (FP7/ICT 217068), the support of the CSIC through the intramural project no. 200850I237, and the support of the Spanish Ministry of Education (TIN2007-60625, and CSD2007-00050), the Generalitat de Catalunya (2009-SGR-980), and the BSC-IBM MareIncognito project.

References

1. AMD Corporation. The AMD Fusion Family of APUs, http://fusion.amd.com
2. AMD/ATI. OpenCL: The Open Standard for Parallel Programming of GPUs and Multi–core CPUs (2010), http://www.amd.com/us/products/technologies/stream-technology/opencl/Pages/opencl.aspx
3. Ayguade, E., Badia, R.M., Cabrera, D., Duran, A., Gonzalez, M., Igual, F., Jimenez, D., Labarta, J., Martorell, X., Mayo, R., Perez, J.M., Quintana-Orti, E.S.: A Proposal to Extend the OpenMP Tasking Model for Heterogeneous Architectures. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 154–167. Springer, Heidelberg (2009)
4. Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Su, E., Unnikrishnan, P., Zhang, G.: A proposal for task parallelism in openMP. In: Chapman, B., Zheng, W., Gao, G.R., Sato, M., Ayguadé, E., Wang, D. (eds.) IWOMP 2007. LNCS, vol. 4935, pp. 1–12. Springer, Heidelberg (2008)
5. Cooper, P., Dolinsky, U., Donaldson, A.F., Richards, A., Riley, C., Russell, G.: Offload – automating code migration to heterogeneous multicore systems. In: Patt, Y.N., Foglia, P., Duesterwald, E., Faraboschi, P., Martorell, X. (eds.) HiPEAC 2010. LNCS, vol. 5952, pp. 337–352. Springer, Heidelberg (2010)
6. Dolbeau, R., Bihan, S., Bodin, F.: HMPP: A Hybrid Multi-core Parallel Programming Environment. In: Workshop on General Processing Using GPUs (2006)
7. Eichenberger, A.E., O'Brien, K., O'Brien, K.M., Wu, P., Chen, T., Oden, P.H., Prener, D.A., Shepherd, J.C., So, B., Sura, Z., Wang, A., Zhang, T., Zhao, P., Gschwind, M., Archambault, R., Gao, Y., Koo, R.: Using advanced compiler technology to exploit the performance of the cell broadband engine$^{(\text{tm})}$ architecture. IBM Systems Journal 45(1), 59–84 (2006)

8. IBM Corporation. OpenCL (2010), http://www.alphaworks.ibm.com/tech/opencl
9. Intel Corporation. Intel Unveils Product Plans for HPC (May 2010), http://www.intel.com/pressroom/archive/releases/2010/20100531comp.htm
10. Kindratenko, V., Enos, J., Shi, G., Showerman, M., Stone, G.A.J., Phillips, J., Hwu, W.: GPU Clusters for High-Performance Computing. In: IEEE Int. Conf. on Cluster Comp. Workshop on Parallel Programming on Accelerator Clusters (2009)
11. Knight, T.J., Park, J.Y., Ren, M., Houston, M., Erez, M., Fatahalian, K., Aiken, A., Dally, W.J., Hanrahan, P.: Compilation for explicitly managed memory hierarchies. In: Proceedings of the 2007 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2007)
12. Linderman, M., Collins, J., Wang, H., Meng, T.: Merge: A Programming Model for Heterogeneous Multi-core Systems. In: Proc. of the 14th Int. Conf. on Arch. Support for Prog. Languages and Operating Systems (ASPLOS) (March 2009)
13. NVIDIA Corporation. NVIDIA CUDA Compute Unified Device Architecture Version 2.0 (2008)
14. NVIDIA Corporation. OpenCL (2010), http://www.nvidia.com/object/cuda_opencl_new.html
15. O'Brien, K., O'Brien, K.M., Sura, Z., Chen, T., Zhang, T.: Supporting openmp on cell. International Journal of Parallel Programming 36(3), 289–311 (2008)
16. OpenMP Architecture Review Board. OpenMP Application Program Interface. Version 3.0 (May 2008)
17. Perez, J.M., Bellens, P., Badia, R.M., Labarta, J.: CellSs: Making it easier to program the Cell Broadband Engine processor. IBM Journal of Research and Development 51(5), 593–604 (2007)
18. RapidMind. RapidMind Multi-core Development Platform, http://www.rapidmind.com/pdfs/RapidmindDatasheet.pdf
19. Ueng, S.-Z., Lathara, M., Baghsorkhi, S.S., Hwu, W.-m.W.: CUDA-Lite: Reducing GPU Programming Complexity. In: Amaral, J.N. (ed.) LCPC 2008. LNCS, vol. 5335, pp. 1–15. Springer, Heidelberg (2008)
20. Wang, P., Collins, J., Chinya, G., Jiang, H., Tian, X., Girkar, M., Yang, N., Lueh, G.-Y., Wang, H.: EXOCHI: Architecture and programming environment for a heterogeneous multi-core multithreaded system. In: Proc. of PLDI, pp. 156–166 (2007)

CnC-CUDA: Declarative Programming for GPUs

Max Grossman, Alina Simion Sbîrlea, Zoran Budimlić, and Vivek Sarkar

Department of Computer Science, Rice University
{jmg3,alina,zoran,vsarkar}@rice.edu

Abstract. The computer industry is at a major inflection point in its hardware roadmap due to the end of a decades-long trend of exponentially increasing clock frequencies. Instead, future computer systems are expected to be built using homogeneous and heterogeneous many-core processors with 10's to 100's of cores per chip, and complex hardware designs to address the challenges of concurrency, energy efficiency and resiliency. Unlike previous generations of hardware evolution, this shift towards many-core computing will have a profound impact on software. These software challenges are further compounded by the need to enable parallelism in workloads and application domains that traditionally did not have to worry about multiprocessor parallelism in the past. A recent trend in mainstream desktop systems is the use of graphics processor units (GPUs) to obtain order-of-magnitude performance improvements relative to general-purpose CPUs. Unfortunately, hybrid programming models that support multithreaded execution on CPUs in parallel with CUDA execution on GPUs prove to be too complex for use by mainstream programmers and domain experts, especially when targeting platforms with multiple CPU cores and multiple GPU devices.

In this paper, we extend past work on Intel's Concurrent Collections (CnC) programming model to address the hybrid programming challenge using a model called CnC-CUDA. CnC is a declarative and implicitly parallel coordination language that supports flexible combinations of task and data parallelism while retaining determinism. CnC computations are built using steps that are related by data and control dependence edges, which are represented by a CnC graph. The CnC-CUDA extensions in this paper include the definition of multithreaded steps for execution on GPUs, and automatic generation of data and control flow between CPU steps and GPU steps. Experimental results show that this approach can yield significant performance benefits with both GPU execution and hybrid CPU/GPU execution.

1 Introduction

The computer industry is at a major inflection point in its hardware roadmap due to the end of a decades-long trend of exponentially increasing clock frequencies. Instead, future computer systems are expected to be built using homogeneous and heterogeneous many-core processors with 10's to 100's of cores per chip,

K. Cooper, J. Mellor-Crummey, and V. Sarkar (Eds.): LCPC 2010, LNCS 6548, pp. 230–245, 2011.

and complex hardware designs to address the challenges of concurrency, energy efficiency and resiliency. Unlike previous generations of hardware evolution, this shift towards many-core computing will have a profound impact on software.

These software challenges are further compounded by the need to enable parallelism in mainstream workloads and application domains that have traditionally not had to worry about multiprocessor parallelism in the past. Despite over four decades of research, there are few choices of high-level parallel programming models available to domain experts who are not computer science experts. Fortunately, this situation is starting to change. Systems like MapReduce [7] are succeeding based on implicit parallelism, albeit with a restricted applicability. Other systems like CUDA [20] and OpenCL [17] are partially there for GPU accelerators, providing a restricted programming model to the user but also exposing a fair amount of hardware details.

Intel's Concurrent Collections[1] (CnC) is a declarative and implicitly parallel coordination language that supports flexible combinations of task and data parallelism while retaining determinism. CnC computations are built using *steps* that are related by data and control dependence edges, which in turn are represented by a CnC *graph*. CnC is provably deterministic [2]. While this restricts CnC's scope, it is more general than other deterministic programming models including dataflow and stream-processing, and can incorporate static and dynamic forms of task, data, loop, pipeline, and tree parallelism. However, all known implementations of CnC to date have been on homogeneous multicore SMP's.

A recent trend in mainstream desktop systems is the use of general-purpose graphics processor units (GPGPUs) to obtain order-of-magnitude performance improvements. As an example, NVIDIA's Compute Unified Device Architecture (CUDA) has emerged as a popular hybrid programming model for CPUs and GPGPUs [20]. While it can be fairly straightforward for mainstream programmers to write the device code for specific kernels in CUDA, the CPU-GPU interactions necessary for deploying a complete CUDA application can be complicated to implement because of the necessary control flow for launching new kernels, data flow for communicating inputs and outputs, and synchronization to ensure proper coordination between the CPU and GPU. Further, debugging the execution of a CUDA program across a multicore SMP and a GPU is especially onerous because of the loose coupling via the device interface and the lack of integrated debugging tools. For these reasons, we believe that writing and deploying full CUDA applications is beyond the scope of mainstream domain experts, from the viewpoints of both programmability and productivity.

In this paper, we extend past work on Intel's Concurrent Collections (CnC) programming model to address the hybrid programming challenge using a model called CnC-CUDA. The CnC-CUDA extensions in this paper include the definition of *multithreaded steps* for execution on GPUs, and *automatic generation of data and control flow* between CPU steps and GPU steps. Further, given the widespread use of managed-runtime execution environments, such as the Java Virtual Machine (JVM) and .Net platforms, we have developed a Java-based

[1] An earlier version of CnC was called TStreams [18].

implementation of CnC which provides the foundation for the CnC-CUDA implementation. In this way, the programmer has the choice of writing CPU Steps in Java or C (since C code can be invoked from Java) and GPU steps in CUDA, and can leave all the remaining details of creating and managing parallel tasks and data transfers to the CnC-CUDA framework. Experimental results show that this approach can yield significant performance benefits with both GPU execution and hybrid CPU/GPU execution. To the best of our knowledge, this is the first experience with mapping the CnC model on to hybrid systems with accelerators (GPUs).

The rest of the paper is organized as follows. Section 2 briefly summarizes the CnC and CUDA programming models. Section 3 introduces the CnC-CUDA Programming Interface and Implementation. Section 4 presents preliminary experimental results for CnC-CUDA. Related work is discussed in Section 5, and our conclusions are contained in Section 6.

2 Background

2.1 Concurrent Collections Programming (CnC) Model

In this section, we give a brief summary of the CnC model, as described in [3]. As in dataflow and stream-processing languages, a CnC program is a graph of serial kernels, communicating with one another. The three main constructs in CnC are *step collections*, *data item collections*, and *control tag collections*. Statically, each of these constructs is a *collection* representing a set of dynamic *instances*. Step instances are the unit of distribution and scheduling. Item instances are the unit of synchronization and communication. Control tag instances are the unit of control.

The program is represented as a graph. In textual form, the graph is denoted using `()` to suggest circles for computation steps, `[]` to suggest boxes for data items and `<>` to suggest triangles for control tags. The edges in the graph specify the partial ordering constraints required by the semantics. One type of ordering constraint arises from a *data dependence*. This relationship occurs when an instance of a step, say `(F1)`, produces an instance of an item, say `[X]`, which is later consumed by an instance of another step, say `(F2)`. Clearly the producing step instance must occur before the consuming step instance. Another type of ordering constraint arises from a *control dependence*, where one computation step determines if another computation step will execute. In that case, the controller step puts a control tag in a tag collection, which in turn *prescribes* the controllee step. The execution order of step instances is constrained only by their dynamic data and control dependences.

Control, data, and step instances are all identified by a unique *tag* within each collection. In CnC, tags are arbitrary values that support an equality test and hash function. Each type of collection uses tags as follows:

- Putting a tag into a control collection will cause the corresponding steps (in the prescribed step collections) to eventually execute. A control collection C with tag i is denoted $\langle C : i \rangle$.

– Each step instance is a computation that takes a single tag (originating from the prescribing control collection) as an argument. The step instance of collection (foo) at tag i is denoted $(foo : i)$.
– A data collection is an associative container indexed by tags. The entry for a tag i, once written, cannot be overwritten (dynamic single assignment). The immutability of entries within a data collection is necessary for determinism. An instance in data collection x with tag "i, j" is denoted $[x : i, j]$.

A CnC specification can optionally include *tag functions* [13] and use them to specify the mapping between a step instance and the data instances that it *consumes* or *produces*. A tag function can be the identity function, or can define nearest neighbor computations, a parent/child in a tree, neighbors in a graph, or any other relationship useful in the application.

2.2 Habanero-Java Implementation of CnC

Habanero-Java (HJ) is a programming language being developed in the Habanero Multicore Software Research project at Rice University [1]. We chose it for the baseline implementation of the CnC runtime system because it includes constructs that serve as a convenient target for implementing CnC primitives. We were pleasantly surprised to see how straightforward it has been to map CnC primitives to HJ, as summarized in Table 1.

Table 1. Summary of mapping from CnC primitives to HJ primitives

CnC construct	Translation to HJ
Tag	Java *String* object or *point* object
Prescription	*async* or *delayed async*
Item Collection	`java.util.concurrent.ConcurrentHashMap`
put() on Item Collection	Nonblocking *put()* on `ConcurrentHashMap`
get() on Item Collection	Blocking or nonblocking *get()* on `ConcurrentHashMap`

Additional details of the mapping from CnC to HJ are summarized below.

Tags. We allow tags to be instances of *String* or *point* value types. A *point* in HJ is an integer tuple that can be declared with an unspecified rank. A multidimensional tag is implemented by a multidimensional point.

Prescriptions. We have optimized away all prescription tags in the HJ implementation. When a step needs to put a prescription tag in the tag collection, we perform a normal *async* or a *delayed async* for each step prescribed by that tag. The normal async statement, *async* $\langle stmt \rangle$ causes the parent activity to create a new child activity to execute $\langle stmt \rangle$. Execution of the async statement returns immediately i.e., the parent activity can proceed immediately to its next statement. The delayed async statement, *async* $(\langle cond \rangle)$ $\langle stmt \rangle$, is similar to a normal async except that execution of $\langle stmt \rangle$ is guaranteed to be delayed until after the boolean condition, $\langle cond \rangle$, evaluates to true.

Item Collections. We use the `java.util.concurrent.ConcurrentHashMap` class to implement item collections. Our HJ implementation of item collections supports the following operations:

- `new ItemCollection(String` *name*`)`: create and return a new item collection. The string parameter, *name*, is used only for diagnostic purposes.
- C.`put(point` p`, Object` O`)`: insert item O with tag p into collection C. Throw an exception if C already contains an item with tag p.
- C.`awaitAndGet(point` p`)`: return item in collection C with tag p. If necessary, the caller blocks until item becomes available.
- C.`containsTag(point` p`)`: return true if collection C contains an item with tag p, false otherwise.
- C.`get(point` p`)`: return item in collection C with tag p if present; return `null` otherwise. The HJ implementation of CnC ensures that this operation is only performed when tag p is present *i.e.,* when C.`containsTag(point` p`)` = `true`. Unlike `awaitAndGet()`, a `get()` operation is guaranteed to always be nonblocking.

Put and Get Operations. A CnC `put` operation is directly translated to a `put` operation on an HJ item collection, but implementing `get` operations can be more complicated. A naive approach is to translate a CnC `get` operation to an `awaitAndGet` operation on an HJ item collection. However, this approach does not scale well when there are a large number of steps blocked on `get` operations, since each blocked activity in the current HJ work-sharing scheduler gets bound to a separate Java thread. A Java thread has a larger memory footprint than a newly created async operation. Typically, a single heavyweight Java thread executes multiple lightweight async's; however, when an async blocks on an `awaitAndGet` operation it also blocks the Java thread, thereby causing additional Java threads to be allocated in the thread pool [9]. In some scenarios, this can result in thousands of Java threads getting created and then immediately blocking on `awaitAndGet` operations.

This observation lead to some interesting compiler optimization opportunities of `get` operations using delayed asyncs. Consider a CnC step S that performs two `get` operations followed by a `put` operation as follows (where T_x, T_y, T_z are distinct tags):

$$S\text{: } \{\ x := C.\texttt{get}(T_x);\ y := C.\texttt{get}(T_y);\ z := F(x, y);\ C.\texttt{put}(T_z, z);\ \}$$

Instead of implementing a prescription of step S with tag T_S as a normal async like "`async` $S(T_S)$", a compiler can implement it using a delayed async of the form "`async when(`C.`containsTag(`T_x`) &&` C.`containsTag(`T_y`))` $S(T_S)$". With this boolean condition, we are guaranteed that execution of the step will not begin until items with tags T_x and T_y are available in collection C.

2.3 GPU Architecture and the CUDA Programming Model

The NVIDIA CUDA programming model is an interface designed to allow programmers access to the extremely parallel hardware of programmable Graphics

Processing Units (GPUs). With an architecture originally intended for graphics rendering, the GPU's strengths lie with highly parallel applications, streaming data, and low inter-thread communication and synchronization. For instance, the GPU used in our performance evaluation, an NVIDIA GTX 480, has 480 processing cores. From this we can see that GPUs' architecture make them easily applicable to scientific and mathematical computing problems.

CUDA is an extension on the C/C++ programming language, with the CUDA runtime library providing a collection of device memory management, host-device stream synchronization, and execution control functions (among others). The general flow of a CUDA program consists of the following steps [23], where all allocation and copying of device memory is controlled explicitly or implicitly by the host using the CUDA runtime library:

1. Copy data from main memory to GPU memory
2. CPU instructs GPU to start a kernel
3. GPU executes kernel in parallel and accesses GPU memory
4. Copy the results from GPU memory to main memory

CUDA is a data parallel SIMT architecture, in which the same programmer-defined kernel executions on all launched threads. These threads are launched in batches of blocks and grids, where blocks are collections of threads and grids are collections of blocks.

3 Programming Interface and Implementation

We have implemented several extensions to the CnC programming model in order to support CUDA steps, which we outline in this section.

3.1 Graph File

Some of the features added require using new syntax in the graph file. First, we introduce a new syntax for CUDA steps. CUDA steps are declared with braces, {}, instead of parentheses, (), for CPU steps. The graph file can now define both CPU and GPU steps.

Second, we have added support for the programmer to specify constants in the graph file using the following notation:

|*const_name const_value*|;

where *const_value* is of an integer type. This definition generates constant values to be used both in HJ and CUDA. These constants are used for specifying the exact type of item and tag collections that are passed to CUDA, ensuring the copying of the correct amount of data from Java arrays onto the CUDA device. For example, if each CUDA thread takes 1000 integers, this item collection would be declared as:

[*int items*[1000]];, `or` |*size* 1000|; [*int items*[*size*]];

Using the second method allows the CnC programmer access to the constant value inside the computation step and the program's entry point, ensuring no stale values for those constants caused by multiple definitions.

3.2 Item Collections

Access functions for all item collections are automatically generated from the CnC parser in order to enforce strict typing rules on items and tags. Item collections maintain the standard interface for adding or retrieving items, the Put and Get methods for individual items. With this approach, each of the items is put into a ConcurrentHashMap, similarly to what is done in CnC-HJ. Once a threshold number of tags have been put, the items corresponding to those tags are collected from the ConcurrentHashMap, converted to a C friendly format (i.e., java.lang.Integer→int) and passed to CUDA. While this approach correctly handles individual Put and Get operations, it also results in a significant performance overhead.

In CUDA-CnC we introduce a much more efficient alternative. The PutRegion/GetRegion primitives allow the programmer to put a (potentially multidimensional) region of integers associated with a similarly dimensioned array of items. This approach eliminates putting and then extracting individual items since the array is directly passed to the kernel. Currently we only support arrays of primitive types (int[], float[], e.t.c.), which we believe to be a reasonable limitation as the CUDA kernels are usually coded using primitive types anyway. Optimizing the individual item puts and gets, adding support for getting individual items from region puts, and implementing PutRegion and GetRegion for arrays of more complex structures (i.e., user defined classes) are beyond the scope of this paper and a subject of future research.

Currently, if an error occurs in copying an item collection (i.e., insufficient device memory) no error is reported to the user. This is simple to implement as the library written for CnC-CUDA host-device memory transfer returns error values, and will certainly be in future work in CnC-CUDA.

3.3 Tag Collection - PutRegion and GetRegion

Tag collections are automatically generated using type definitions in the graph file. Our preliminary implementation only supports integer tags, which can be easily extended to any type that can be hashed, as in traditional CnC.

As described earlier, tag collections control the execution and synchronization of computation steps. Synchronization between computation steps using tag collections is a different matter in CUDA. First, a pthread mutex is used to indicate that the device is currently in use by a computation step and inaccessible by any new computation steps for non-Fermi architecture GPUs which do not support concurrent kernel execution. Second, we limit the number of CUDA computation steps that can be prescribed by another CUDA computation step to 1, which considerably simplifies the complexity of synchronizing multiple CUDA computation steps. If a CUDA step prescribes another CUDA step (as determined by analyzing the CnC graph file), the second step is invoked immediately following the first without returning to HJ. No limitation is placed on a CUDA step prescribing multiple HJ steps. Last, synchronization between host and device computation steps is used by a call to a CUDA tag collection's Wait() method.

This call blocks until all launched CUDA kernels have returned and their output has been placed in host memory.

Tag Collections also implement the PutRegion operation, which places a region of integer tags into the tag collection. PutRegion immediately launches a CUDA kernel for all tags in the range once the required items are available. On individual tag Puts, the tag collection waits for a threshold number of tags to be put, and then launches a CUDA kernel with those tags and their associated items. We have currently empirically set this threshold to 8192 tags. Once all tags have been put into a GPU tag collection, the programmer has to issue a call to that tag collection's *Wait()* function to be certain that all CUDA threads have completed as the transferring of data to and from device memory and launching of CUDA kernels is handled by a separate CPU thread.

For more advanced CUDA programmers we introduce the option of defining a two dimensional tag:

```
<int tag:two_region>
```

This offers the opportunity of placing a tag with 2 regions on the graph. Those two regions will be interpreted as number of blocks per grid and threads per block to be used in a kernel launch. In addition, one can specify the number of desired threads in a block by compiling the graph file with the flag: *-t <number of threads>*.

We also support the item collection property One-For-All (OFA), which passes the same data to each thread on a device. This property follows the format:

```
[int item:ofa]
```

where int can be any supported data type. This can result in both considerable saving in device and host memory (less memory allocated to copy from and to) as well as better performance with less time spent copying data to the device.

3.4 CUDA Kernel

The advantage of using CnC CUDA is that the user need not worry about the allocating and copying of data, but just about writing the actual CUDA kernel. The translator is the one responsible for generating stub codes that will allocate memory and copy the data structures to the device before a step is executed as well as free the device memory after these finish. During the execution of a kernel, no puts or get are done on GPUs. The actual step code is written as a CUDA __device__ function by the CnC programmer in a file named "XXXXXKernel.cu" where 'XXXXX' is the CUDA step name as declared in the graph file. The step function needs to be defined as a __device__ function because the __global__ entry point to the device is auto-generated by the translator to protect against unwanted threads entering the programmer-defined kernel (i.e. 513 tags are put, but 2x512 threads are launched. The auto-generated __global__ function will ensure the upper 511 threads of the second block do not enter the programmer-defined kernel). Also, CUDA kernels are limited to putting a single item on each output item collection. This limitation is a result of CUDA requiring preallocation of all device memory before kernel launch. Future work will allow the programmer to specify the number of items output from a kernel.

3.5 Implementation Details

The CnC-CUDA execution model requires a Java-to-native code interface. The approach outlined below builds on our past experience with the JCUDA system [23].

Java provides the *native* keyword which is used to define functions that are implemented in native code. Our implementation creates a libJVMToC.so library which contains CUDA and C code needed to communicate the data to and from the device and execute the kernel. This library encapsulates the C and CUDA code auto-generated by the translator and is generated by the compiler. In CnC-CUDA, all of the complexity of inter-language function-calls and device memory management is hidden from the CnC programmer and auto-generated at compile time, allowing them to focus on developing and implementing the algorithms in their application.

Habanero Java also offers the *extern* keyword — similar to *native* in Java — which greatly simplifies programming with native code. Compiling an HJ class with extern functions generates C stubs which will be included in the file with the native function implementation, named *ClassName_FunctionName*.

The C and CUDA code that are generated by our CnC translator are responsible for creating a data collection for every data structure declared in the graph file, copying the said data structure to the GPU before launching the kernel and from the device back to the CPU afterwards, and actual launching of the CUDA computation step(s). The CnC graph is analyzed to determine which item and tag collections are only accessed from the device (i.e., Put from one CUDA step and Get from another), and this analysis is used to remove extraneous device memory copies from the generated code.

4 Preliminary Experimental Results

4.1 Experimental Setup

In order to compare the performance of CnC-CUDA to CnC-HJ (available at [6]) and other programming models and languages we used three benchmarks from the Java Grande Forum (JGF) benchmark suite [15], as well as the Heart Wall Tracking program from the Rodinia benchmark suite [5]. The three benchmarks from the JGF suite are Fourier coefficient analysis (Series), successive over-relaxation (SOR), and IDEA encryption (Crypt). Each was run on varying data sizes using CnC-CUDA, CnC-HJ, Serial C, hand-coded CUDA, and the original single-threaded JGF Java benchmark. Additionally, the Crypt benchmark was run using CnC-CUDA and CnC-HJ computation steps running in parallel. The timing of each benchmark was started just before the first set of tag puts were performed to launch the CnC graph or the function call to launch the actual computation for non-CnC benchmarks. Timing was stopped when all

CnC steps completed (as detected by the HJ finish construct) or the core function completed. For GPU execution, this included the overhead of copying data to and from device memory.

For these evaluations, we used an NVIDIA GTX 480 GPU. The GTX 480 has 480 processing cores and 1.6 GB of memory. The CPU host of this GPU is a AMD Phenom 9850 Quad-Core Processor with a 1.25 GHz clock, 512 KB cache, and 8 GB of memory. The installed software includes a Java HotSpot 64-bit virtual machine from version 1.6.0_20 of the Java Development Kit (JDK), a GNU C compiler v. 4.1.2, and version 3.1 of the NVIDIA CUDA Toolkit.

There are a few limitations in the current CnC-CUDA implementation which will need to be addressed in future work. First, the current implementation limits tags to only be integers. Second, both parent and CUDA steps are assumed to always have the same block/grid structure; giving the CnC programmer the ability to change the number of threads used across parent-child CUDA computation steps could further increase the flexibility of our CnC-CUDA implementation.

4.2 Evaluation and Analysis

Tables 2, 3, and 4 display the average execution times across ten runs of the respective benchmarks in different programming models or languages. In the CnC versions (HJ or CUDA), the CnC Parser was used to auto-generate all the glue code, leaving the programmer to only provide the CnC step code in CUDA or HJ and the code for launching the CnC graph in the main HJ program. The CnC-CUDA measurements on the GPU were compared to CnC-HJ runs of the same benchmarks on the CPU and the results listed in the Speedup column of each table. Each CUDA kernel launch was performed with a constant 256

Table 2. Execution times in seconds of JGF Crypt benchmark implemented in several programming models, and Speedup of CnC-CUDA relative to CnC-HJ

Crypt	GPU Performance		CPU Performance			
Data Size (bytes)	CnC-CUDA	CUDA (16 cores)	CnC-HJ	Serial C	Serial Java	Speedup
50,000,000 (JGF Size C)	0.886	0.161	2.067	7.367	2.92	2.33
75,000,000	1.208	0.253	3.239	11.033	4.387	2.68
100,000,000	1.488	0.341	4.460	14.678	5.818	3.00
150,000,000	2.311	0.550	6.903	22.039	8.716	2.99

Table 3. Execution times in seconds of JGF Series benchmark implemented in several programming models, and Speedup of CnC-CUDA relative to CnC-HJ

Series	GPU Performance		CPU Performance			
Data Size	CnC-CUDA	CUDA	CnC-HJ (16 cores)	Serial C	Serial Java	Speedup
10,000 (JGF Size A)	0.332	0.0095	3.587	3.157	6.777	10.80
100,000 (JGF Size B)	0.441	0.116	36.588	31.832	69.074	60.78
1,000,000 (JGF Size C)	1.411	1.279	572.86	321.553	N/A	406.00

Table 4. Execution times in seconds of JGF SOR benchmark implemented in several programming models, and Speedup of CnC-CUDA relative to CnC-HJ

SOR	GPU Performance		CPU Performance			
Data Size (Dim)	CnC-CUDA	CUDA	CnC-HJ (16 cores)	Serial C	Serial Java	Speedup
1,000 (JGF Size A)	0.403	0.021	0.714	1.691	1.247	1.77
1,500 (JGF Size B)	0.448	0.045	3.015	3.811	3.872	6.73
2,000 (JGF Size C)	0.498	0.078	5.400	6.769	6.891	10.84
3.000	0.602	0.186	12.079	15.309	15.677	20.06
4,000	0.795	0.475	21.512	27.262	27.658	27.06
5,000	0.952	0.813	33.547	42.600	43.129	35.24

Table 5. Execution times in seconds, and Speedup of a hybrid CnC-CUDA/HJ version of Crypt against only CnC-CUDA

Crypt (150,000,000 bytes)	Hybrid Performance			
Percent of Data on GPU	Average	Slowest	Fastest	Speedup (Relative to CnC-CUDA)
10	3.042	3.806	2.493	0.76
20	3.066	3.765	2.727	0.75
30	2.720	3.048	2.223	0.85
40	2.289	2.750	1.878	1.01
50	2.139	2.397	1.973	1.08
60	2.035	2.242	1.538	1.14
70	2.076	2.799	1.755	1.11
80	2.189	2.511	1.883	1.06
90	2.143	2.344	1.968	1.08

Table 6. Execution times in seconds on GPU - CnC-CUDA and hand-coded (Rodinia) CUDA versions - and CPU - Serial, OpenMP on 16 cores and CnC-HJ on 16 cores - of the Heart Wall Tracking benchmark

Heart Wall Tracking	GPU		CPU		
Data Size (# of frames)	CnC-CUDA	CUDA	Serial	Open MP 16 cores	CnC-HJ 16 cores
1	0.427	0.985	0.005	0.005	0.246
104	4.4842	3.6133	156.977	13.863	11.058

threads per block, with the grid size determined by the iteration size[2]. Each CnC-HJ execution was performed using 16 worker threads. (Over-provisioning the number of worker threads per CPU degraded performance for the benchmarks and hardware studied in this paper.)

No special CUDA memory (e.g., texture, shared, constant) was used in the execution of these benchmarks.

[2] An evaluation of alternate grid/block sizes is a subject for future work.

These results demonstrate that performance potential of GPUs can be made accessible to non-expert programmers through CnC-CUDA. Without any knowledge of CUDA's memory or threading model, a CnC-CUDA programmer can go from working with CnC-HJ to exploiting the computational power of a GPU using CnC-CUDA quickly and easily, achieving a magnitude of performance better than a quad-core CPU. For example, building the SOR benchmark from a CnC-HJ version required 3 hours of time to port the step code to CUDA, and resulted in a 35× speedup. Auto-generation of CUDA code using techniques such as those reported in [19] could result in a further productivity boost for non-expert programmers.

We observe that the speedup of CUDA over HJ increases as the size of the data set increases, with the maximum average speedup (406.00×) observed for the embarrassingly parallel Series benchmark at its largest data size. The minimum average speedup of 1.77× was observed for SOR executed with its smallest data size. While not observed in these results, it is of course possible for a GPU version of an application to run slower than a CPU version, when the relative overheads of host-device data transfers, CUDA initialization, or of control flow divergence lead to performance degradation on the GPU.

The results in Table 5 shows the potential for performance improvement using hybrid CPU-GPU execution in the CnC model. For consistency, the hybrid CUDA/HJ tests were performed using 256 threads per block for GPU execution. These results were obtained by evaluating different load distributions between the CPU and GPU for the Crypt benchmark with its largest size, for which the average speedup of the GPU over the CPU in Table 2 was 1.82×. In Table 5, we see that an additional 1.14× speedup can be obtained over the pure CnC-CUDA version by a hybrid execution in which 60% of the load is placed on the GPU and 40% of the load on the CPU. An interesting topic for future research is to extend the CnC runtime to perform this load distribution adaptively and automatically, allowing for a single CnC-CUDA graph to dynamically and efficiently handle a wide range of data set sizes.

Finally, Table 6 shows the execution times in seconds for the CnC-CUDA and hand-coded CUDA versions of the Heart Wall Tracking benchmark (the hand-coded version was obtained from the Rodinia benchmark set [5]). When comparing the fastest times, we see that both versions have comparable performance when processing a single frame, but the CnC-CUDA version is 1.25× slower than the hand-coded version for 104 frames thereby reflecting the extra coordination overhead in CnC involved in sequencing the computation across frames.

5 Related Work

We discuss related work according to their attributes in three dimensions: *Declarative*, *Deterministic* and *Efficient*. A number of lower-level programming models in use today — *e.g.,* Intel TBB [22], .Net Task Parallel Library, OpenMP [4], Nvidia CUDA, Java Concurrency [21] — are non-declarative, non-deterministic, and efficient[3]. Deterministic Parallel Java [10] is an interesting variant of Java; though imperative (non-declarative), it includes a subset that is provably deterministic, as well as constructs that explicitly indicate when determinism cannot be guaranteed for certain code regions.

Higher-level languages such as High Performance Fortran (HPF) [16], X10 [8], and Linda [14] contain hybrid combinations of imperative and declarative programming in different ways. HPF combines a declarative language for data distribution and data parallelism with imperative (procedural) statements, X10 contains a functional subset that supports declarative parallelism, and Linda is a coordination language in which a thread's interactions with the tuple space is declarative. Linda was a major influence on the CnC design, but CnC also differs from Linda in many ways. For example, an in() operation in Linda atomically removes the tuple from the tuple space, but a CnC get() operation does not remove the item from the collection. This is a key reason why Linda programs can be non-deterministic in general, and why CnC programs are provably deterministic. Further, there is no separation between tags and values in a Linda tuple; instead, the choice of tag is implicit in the use of wildcards. In CnC, there is a separation between tags and values, and control tags are first class constructs like data items.

Both streaming and dataflow languages have also had major influence on the CnC design. The CnC semantic model is based on dataflow in that steps are functional and execution can proceed whenever data is ready, without unnecessary serialization. However, CnC differs from dataflow in some key ways. The use of control tags elevates control to a first-class construct in CnC. In addition, item collections allow more general indexing (as in a tuple space) compared to dataflow arrays (I-structures). CnC is like streaming in that the internals of a step are not visible from the graph that describes their connectivity, thereby establishing an isolation among steps. A producer step in a streaming model need not know its consumers; it just needs to know which buffers (collections) to perform read and write operations on. However, CnC differs from streaming in that *put* and *get* operations need not be performed in FIFO order, and (as mentioned above) control is a first-class construct in CnC. We observe that CnC's dynamic put/get operations on data and control collections is a general model that can be used to express many kinds of applications (such as Cholesky factorization) that would not be considered to be dataflow or streaming applications.

With respect to our experimental results, we are not claiming that the GPU by itself offers a certain speedup [12], rather that the speedup we get is from taking advantage of the high amount of data parallelism in our test applications, having 256-512 GPU threads run in parallel instead of 16-32 on a CPU, and an easy to use programming model that hides the use of resources from the user while offering lower execution times. The overhead of copying data from and to the device is hidden by a much larger number of tasks run in parallel. The innovation we offer is an easy way for a programmer to specify the algorithm

[3] We call a programming model efficient if there are known implementations that deliver competitive performance for a reasonably broad set of programs.

while taking advantage of the available resources. Like Oregami [11], CnC is based on a graph description of the algorithm, however in Oregami the programmer needs to design the program as a "set of parallel processes that communicate through explicit message passing. The identity of all of the processes are known at compile time [...]". Such restrictions are not applicable to CnC-CUDA.

In summary, CnC has benefited from influences in past work, but we are not aware of any other parallel programming model that shares CnC's fundamental properties as a coordination language, a declarative language, a deterministic language, and a language amenable to efficient implementation. To the best of our knowledge, this is the first experience with mapping the CnC model on to hybrid systems with accelerators (GPUs).

6 Conclusions and Future Work

In this paper, we extended past work on Intel's Concurrent Collections (CnC) programming model to address the hybrid programming challenge using a model called CnC-CUDA. The CnC-CUDA extensions in this paper include the definition of multithreaded steps for execution on GPUs, and automatic generation of data and control flow between CPU steps and GPU steps. Further, given the widespread use of managed-runtime execution environments, such as the Java Virtual Machine (JVM) and .Net platforms, we have developed a Java-based implementation of CnC which provides the foundation for the CnC-CUDA implementation. In this way, the programmer has the choice of writing CPU Steps in Java or C (since C code can be invoked form Java) and GPU steps in CUDA, and can leave all the remaining details of creating and managing parallel tasks and data transfers to the CnC-CUDA framework. The CnC-CUDA extensions in this paper include the definition of *multithreaded steps* for execution on GPUs, and *automatic generation of data and control flow* between CPU steps and GPU steps. Experimental results show that this approach can yield significant performance benefits with both GPU execution and hybrid CPU/GPU execution.

There are multiple opportunities for future research. We would like to support richer (non-primitive) element data types in the PutRegion and GetRegion primitives. In addition, we would like to extend the types accepted for tags to more than integers. There is a large amount of overhead incurred by transfers to and from device memory, and further experimentation with transfer patterns may yield better performance. Finally, our longer-term plan is to extend the CnC-CUDA implementation to serve as a unified runtime for heterogeneous combinations of CPUs, GPUs, and FPGAs in the CDSC project.

Acknowledgments

We would like to thank all members of the Habanero team at Rice University and the CnC team at Intel for valuable discussions and feedback related to the Concurrent Collections programming model. We gratefully acknowledge support from an Intel award during 2009-2010. This research is partially supported by the

Center for Domain-Specific Computing (CDSC) funded by the NSF Expedition in Computing Award CCF-0926127. Finally, we would like to thank Tim Warburton for providing access to the GPU systems used to obtain the performance results reported in this paper.

References

1. Habanero Multicore Software Project, `http://habanero.rice.edu`
2. Budimlić, Z., Burke, M., Cavé, V., Knobe, K., Lowney, G., Newton, R., Palsberg, J., Peixotto, D., Sarkar, V., Schlimbach, F., Taşrlar, S.: The CnC Programming Model. In: SIAM PP10, Special Issue on Scientific Programming (2010)
3. Burke, M.G., Knobe, K., Newton, R., Sarkar, V.: The Concurrent Collections Programming Model. In: Padua, D. (ed.) Encyclopedia of Parallel Computing. Springer, New York (to be published 2011)
4. Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., Menon, R.: Programming in OpenMP. Academic Press, London (2001)
5. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.-H., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: IEEE International Symposium on Workload Characterization (October 2009)
6. Concurrent Collections in Habanero-Java, HJ (2010), `http://habanero.rice.edu/cnc-download`
7. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM 51(1), 107–113 (2008)
8. Charles, P., et al.: X10: An object-oriented approach to non-uniform cluster computing. In: Proceedings of OOPSLA 2005, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, pp. 519–538 (2005)
9. Barik, R., et al.: Experiences with an smp implementation for x10 based on the java concurrency utilities. In: Workshop on Programming Models for Ubiquitous Parallelism (PMUP), held in conjunction with PACT 2006 (September 2006)
10. Bocchino, R.L., et al.: A type and effect system for Deterministic Parallel Java. In: Proceedings of OOPSLA 2009, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, pp. 97–116 (2009)
11. Lo, V.M., et al.: Oregami: Tools for mapping parallel computations to parallel architectures. IJPP: International Journal of Parallel Programming 20(3), 237–270 (1991)
12. Lee, V.W., et al.: Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu. In: ISCA 2010: ACM IEEE International Symposium on Computer Architecture (June 2010)
13. Budimlić, Z., et al.: Declarative aspects of memory management in the concurrent collections parallel programming model. In: DAMP 2009: the Workshop on Declarative Aspects of Multicore Programming, pp. 47–58. ACM, New York (2008)
14. Gelernter, D.: Generative communication in linda. ACM Trans. Program. Lang. Syst. 7(1), 80–112 (1985)
15. The Java Grande Forum benchmark suite, `http://www.epcc.ed.ac.uk/javagrande`
16. Kennedy, K., Koelbel, C., Zima, H.P.: The rise and fall of High Performance Fortran. In: Proceedings of HOPL 2007, Third ACM SIGPLAN History of Programming Languages Conference, pp. 1–22 (2007)

17. Khronos OpenCL Working Group. The OpenCL Specification - Version 1.0. Technical report, The Khronos Group (2009)
18. Knobe, K., Offner, C.D.: Tstreams: A model of parallel computation (preliminary report). Technical Report HPL-2004-78, HP Labs (2004)
19. Lee, S., Min, S.-J., Eigenmann, R.: Openmp to gpgpu: a compiler framework for automatic translation and optimization. In: PPoPP 2009, pp. 101–110. ACM, New York (2009)
20. Nickolls, J., Buck, I., Garland, M., Nvidia, Skadron, K.: Scalable Parallel Programming with CUDA. ACM Queue 6(2), 40–53 (2008)
21. Peierls, T., Goetz, B., Bloch, J., Bowbeer, J., Lea, D., Holmes, D.: Java Concurrency in Practice. Addison-Wesley Professional, Reading (2005)
22. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism. O'Reilly Media, Sebastopol (2007)
23. Yan, Y., Grossman, M., Sarkar, V.: Jcuda: A programmer-friendly interface for accelerating java programs with cuda. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 887–899. Springer, Heidelberg (2009)

Parallel Graph Partitioning on Multicore Architectures

Xin Sui[1], Donald Nguyen[1], Martin Burtscher[2], and Keshav Pingali[1,3]

[1] Department of Computer Science, University of Texas, Austin
[2] Department of Computer Science, Texas State University, San Marcos
[3] Institute for Computational Engineering and Sciences,
University of Texas, Austin

Abstract. Graph partitioning is a common and frequent preprocessing step in many high-performance parallel applications on distributed- and shared-memory architectures. It is used to distribute graphs across memory and to improve spatial locality. There are several parallel implementations of graph partitioning for distributed-memory architectures.

In this paper, we present a parallel graph partitioner that implements a variation of the Metis partitioner for shared-memory, multicore architectures. We show that (1) the parallelism in this algorithm is an instance of the general amorphous data-parallelism pattern, and (2) a parallel implementation can be derived systematically from a sequential specification of the algorithm. The resulting program can be executed in parallel using the Galois system for optimistic parallelization. The scalability of this parallel implementation compares favorably with that of a publicly available, hand-parallelized C implementation of the algorithm, ParMetis, but absolute performance is lower because of missing sequential optimizations in our system. On a set of 15 large, publicly available graphs, we achieve an average scalability of 2.98X on 8 cores with our implementation, compared with 1.77X for ParMetis, and we achieve an average speedup of 2.80X over Metis, compared with 3.60X for ParMetis. These results show that our systematic approach for parallelizing irregular algorithms on multicore architectures is promising.

1 Introduction

Graph partitioning is a common preprocessing step in many high-performance parallel algorithms. It is used to find partitions of graph nodes such that each partition has roughly the same number of nodes and the sum of the weights of cross-partition edges is minimized. If the number of nodes in a partition is proportional to the amount of work involved in processing that partition and the edge weights are a measure of communication costs, such a partition attempts to achieve load balance while minimizing the inter-processor communication cost.

Graph partitioning is useful in distributed architectures where partitioning can reduce the amount of explicit communication between distributed processing elements. In shared memory settings, partitioning is useful for reducing memory contention and increasing spatial locality.

K. Cooper, J. Mellor-Crummey, and V. Sarkar (Eds.): LCPC 2010, LNCS 6548, pp. 246–260, 2011.

More formally, we define a weighted, undirected graph in the usual way: $G = (V, E)$, and w is a function assigning weights to each edge $(u, v) \in E$. For any subset of vertices $V_i \subseteq V$, the *cut set* induced by V_i is $C_i = \{(u, v) \in E \mid u \in V_i, v \in V - V_i\}$. The *value* (or *edge cut*) of the cut set C_i is $w_i = \sum_{e \in C_i} w(e)$. The subsets $P = V_1, V_2, \ldots, V_k$ are a *k-way partitioning* iff (1) $\cup_i V_i = V$ and (2) $\forall i, j : i \neq j \rightarrow V_i \cap V_j = \emptyset$. The balance of P is

$$B(V_1, V_2, \ldots, V_k) = \frac{k * \max_{i=1}^{k} w_i}{\sum_{i=1}^{k} w_i}$$

The *graph partitioning problem* is, given k and G, to find a k-way partitioning P such that the balance of P and the edge cut of P (i.e., $\sum w_i$) are minimized.

The most widely used graph partitioner is the sequential partitioner Metis from Karypis *et al.* [8]. There are several parallel graph partitioners: ParMetis [6] from Karypis *et al.* as well as PT-Scotch [1] and JOSTLE [13]. All of these implementations are explicitly parallel programs for distributed-memory architectures. With the rise of multicore, we are interested in parallel implementations that can take advantage of the lightweight synchronization and communication available on multicore machines. Additionally, we prefer implementations that hew as close as possible to their original sequential implementations as it is usually easier to write and determine the correctness of sequential programs.

The approach that we adopt in this paper is as follows. First, we recognize that the Metis algorithm exhibits a generalized form of data-parallelism that we call amorphous data-parallelism [12]. Second, we implement a version of Metis that exploits amorphous data-parallelism, and we show that it is possible to achieve better scalability than existing parallelizations without resorting to explicit parallel programming.

This paper is organized as follows. Section 2 describes the existing implementations of Metis, sequential and parallel, in detail. Section 3 introduces amorphous data-parallelism, and Section 4 describes the Galois system, a programming model and runtime system designed to exploit amorphous data-parallelism. In Section 5, we show how the Metis algorithm exhibits amorphous data-parallelism and how to implement it using the Galois system. Section 6 discusses results, and Section 7 presents conclusions from this study.

2 Graph Partitioning Algorithms

Metis is one of the most widely used graph partitioners. Recent work has shown that ParMetis is generally the best parallel graph partitioner for distributed-memory systems [5]. In this section, we describe both of these partitioners in more detail.

2.1 Metis

Metis is composed of a set of algorithms for graph and mesh partitioning. The key algorithm is KMetis. It uses a multilevel scheme that coarsens input graphs until they are small enough to employ more direct techniques and then interpolates the results of the smaller graph back onto the input graph. It consists of three phases: coarsening, initial partitioning, and refinement (see Figure 1).

Coarsening. The goal of coarsening is to construct a smaller graph from the input graph that preserves most of the connectivity information. This is achieved by constructing a sequence of graphs, each of which is obtained by contracting edges in the previous graph (see Figure 3). When contracting an edge (a, b), a new node is created in the coarse graph, and its weight is set to the sum of the weights of nodes a and b. If nodes a and b are both connected to a node c in the fine graph, a new edge with a weight equal to the sum of the weights of edges (a, c) and (b, c) is created in the coarse graph.

To find edges to collapse, KMetis computes a maximal matching. A matching is a set of edges in the graph that do not share any nodes. A matching is maximal if it is not possible to add any edges into it. There are several heuristics for finding the maximal matching. KMetis employs a heuristic called heavy edge matching. Heavy edge matching finds maximal matchings whose edges have large weight (Figure 2(b)). Collapsing such edges will greatly decrease the edge weights in the coarse graph and improve the quality of the resulting partitioning. The nodes in the graph are visited randomly, and each node is matched to the unmatched neighbor with the maximal edge weight. If there is no such neighbor, the node is matched to itself.

The coarsening phase ends when the number of nodes in the coarse graph is less than some threshold or the reduction in the size of successive graphs is less than some factor.

Initial partitioning. In this phase, a recursive bisection algorithm called PMetis is used to partition the coarsest graph. Each bisection has the same phases as KMetis except that PMetis (1) uses a breadth-first traversal to perform an initial bisection and (2) employs the Kernighan-Lin heuristic [9] to improve the quality of the bisection. Compared to KMetis, PMetis is slower when the desired number of partitions is large because it coarsens the input graph multiple times. Usually, this phase represents only a small fraction of the overall partitioning time.

Refinement. In this phase, the initial partitioning is projected back on the original graph via the sequence of graphs created in the coarsening phase (Figure 2(c)). KMetis uses a simplified version of the Kernighan-Lin heuristic called random k-way refinement. The nodes of the graph that are on the boundary between partitions are visited randomly. A boundary node is moved to its neighboring partition if doing so reduces the edge cut without leading to a significant imbalance in the partitioning.

2.2 ParMetis

ParMetis is a parallelization of KMetis using MPI. In ParMetis, each process owns a random portion of the input graph. Each of the phases of KMetis is parallelized as follows.

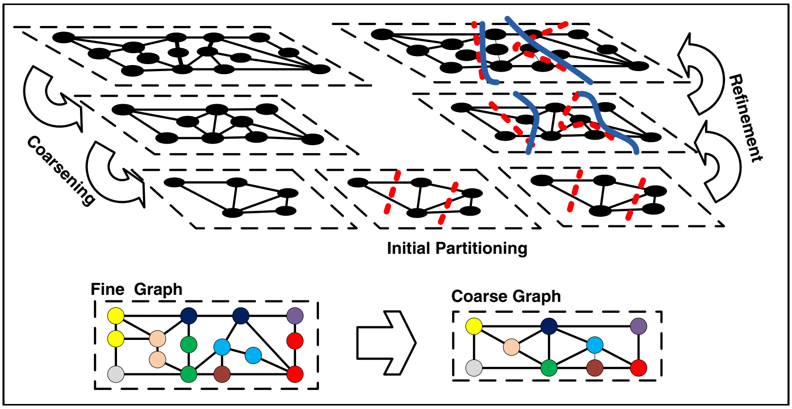


Fig. 1. The phases of a multilevel partitioning algorithm. Dashed lines illustrate the partitioning projected from the coarse graph, and solid shaded lines illustrate the refined partitioning.

```
Graph g = // Read in graph
int k = // Number of partitions
Graph original = g;
do {
  heavyEdgeMatching(g);
  Graph cg = coarsen(g);
  cg.setFinerGraph(g);
  g = cg;
} while (!g.coarseEnough());
PMetis.partition(g, k);
while (g != original) {
  Graph fg = g.finerGraph();
  g.projectPartitioning(fg);
  fg.makeInfoForRefinement();
  refine(fg);
  g = fg;
}
```

(a) Pseudocode for main Metis algorithm.

```
void heavyEdgeMatching(Graph g) {
  // Randomly access
  foreach (Node n : g) {
    if (n.isMatched()) continue;
    Node match = n.findMatch();
    g.setMatch(n, match);
  }
}
```

(b) Pseudocode for heavy edge matching.

```
void refine(Graph g) {
  Worklist wl = new Worklist();
  foreach (Node n : g.boundaryNodes()) {
    if (...) // Moving n to neighbor partition reduces edge cut
      wl.add(n);
  }
  foreach (Node n : wl) {
    part = // Neighbor partition with max edge cut gain;
    if (...) // Balancing condition is not violated
      moveNode(n, part);
  }
}
```

(c) Pseudocode for random k-way refinement.

Fig. 2. Pseudocode for Metis algorithm. Foreach loops indicate the presence of amorphous data-parallelism.

```
Graph coarsen(Graph graph) {
  Graph cg = new Graph(); // Coarse graph
  for (Node n : graph) {
    if (n.visited) continue;
    Node cn = cg.createNode(n.weight+n.getMatch().weight);
    n.setRepresentative(cn);
    n.getMatch().setRepresentative(cn);
    n.getMatch().setVisited(true);
  }
  ... // Reset visited field for each node in the graph
  foreach (Node n : graph) {
    if (n.visited) continue;
    // Add edges in cg according to n's neighbors
    for (Node nn : n.getNeighbors()) {
      Edge e = graph.getNeighbor(n, nn);
      Node cn = n.getRepresentative();
      Node cnn = nn.getRepresentative();
      Edge ce = cg.getEdge(cn, cnn);
      if (ce == null) { cg.addEdge(cn, cnn, e.getWeight()); }
      else { ce.increaseWeight(e.getWeight()); }
      ... // Add edges in cg according to n.getMatch()'s neighbors,
      ... // and similarly for n
      n.getMatch().setVisited(true);
    }
  }
  return cg;
}
```

Fig. 3. Pseudocode for creating a coarse graph. Foreach loops indicate the presence of amorphous data-parallelism.

Coarsening. The parallelization of heavy edge matching proceeds in alternating even and odd rounds. There are two steps in each round. In the first step, each process scans its local unmatched nodes, and for each node v, each process tries to find a neighbor node u to match using the heavy edge matching heuristic. There are three cases for u: (1) if u is stored locally, then the matching is completed immediately; (2) if the round is odd and $v < u$ or the round is even and $v > u$, the process sends a matching request to the process owning u; or (3) otherwise, the match is deferred to the next round. In the second step, each process responds to its matching requests. Processes break conflicts arbitrarily and notify senders on whether their matching requests were successful. Heavy edge matching terminates when some large fraction of the nodes has been matched.

Initial partitioning. ParMetis does not use PMetis for its initial partitioning phase. Instead, it uses the following parallel algorithm: all the pieces of the graph are scattered to all threads using an all-to-all broadcast operation. Then, each process explores a single path of the recursive bisection tree. The recursive bisection is based on nested dissection [4].

Refinement. The random k-way refinement algorithm is parallelized similarly to heavy edge matching. The algorithm runs in rounds. Each round is divided into two steps. In the first step, nodes can only be moved to higher partitions. During the second step, nodes can only be moved to lower partitions. This alternation pattern helps avoid situations where moves of nodes to new partitions, when considered in isolation, would decrease the edge cut but, when considered *en masse*, actually increase the overall edge cut.

ParMetis also implements optimizations specific to the message-passing programming model. It coalesces small messages into larger messages, and it detects situations where the graph can be redistributed to a smaller set of processes.

3 Amorphous Data-Parallelism

Amorphous data-parallelism is a form of parallelism that arises in irregular algorithms that operate on complex data structures like graphs [12]. At each point during the execution of such an algorithm, there are certain nodes or edges in the graph where computation might be performed. Performing a computation may require reading or writing other nodes and edges in the graph. The node or edge on which a computation is centered is called an *active element*, and the computation itself is called an *activity.* It is convenient to think of an activity as resulting from the application of an *operator* to the active node. We refer to the set of nodes and edges that are read or written in performing the activity as the *neighborhood* of that activity. Note that in general, the neighborhood of an active node is distinct from the set of its neighbors in the graph. Activities may modify the graph structure of the neighborhood by adding or removing graph elements.

In general, there are many active nodes in a graph, so a sequential implementation must pick one of them and perform the appropriate computation. In some algorithms such as Metis, the implementation is allowed to pick *any* active node for execution. We call these algorithms *unordered algorithms.* In contrast, other algorithms dictate an order in which active nodes must be processed. We call these *ordered algorithms.*

A natural way to program these algorithms is to use the Galois programming model [10], which is a *sequential*, object-oriented programming model (such as Java) augmented with two *Galois set iterators*:

- **Unordered-set iterator: foreach (e : Set S) { B(e) }**
  The loop body B(e) is executed for each element e of set S. The order in which iterations execute is indeterminate and can be chosen by the implementation. There may be dependences between the iterations. When an iteration executes, it may add elements to S.
- **Ordered-set iterator: foreach (e : OrderedSet S) { B(e) }**
  This construct iterates over an ordered set S. It is similar to the unordered set iterator above, except that a sequential implementation must choose a minimal element from set S at every iteration. When an iteration executes, it may add new elements to S.

Opportunities for exploiting parallelism arise if there are many active elements at some point in the computation, each one is a site where a processor can perform computation. When active nodes are unordered, multiple active nodes may be processed concurrently as long as their neighborhoods do not overlap.

For ordered active elements, there is an additional constraint that activities must appear to commit in the same order as the ordering on the set elements.

Definition 1. *Given a set of active nodes and an ordering on active nodes,* amorphous data-parallelism *is the parallelism that arises from simultaneously processing active nodes, subject to neighborhood and ordering constraints.*

Amorphous data-parallelism is a generalization of conventional data-parallelism in which (1) concurrent operations may conflict with each other, (2) activities can be created dynamically, and (3) activities may modify the underlying data structure.

4 The Galois System

The *Galois system* is a set of data structures and a runtime system for Java that allows programs to exploit amorphous data-parallelism. The runtime system uses an optimistic parallelization scheme, and the data structures implement the necessary features for optimistic execution: conflict detection and rollback.

Data structure library. The system provides a library of concurrent implementations of data structures, such as graphs, maps, and sets, which are commonly used in irregular algorithms. Programmers can either use one of the existing implementations or provide new ones. For a data structure implementation to be suitable for the Galois system it must satisfy three properties: (1) operations on the data structure must appear to execute atomically, (2) it should enforce the appropriate neighborhood constraints, and (3) it should enable rollback in case of conflicts.

Execution Model. The data structures are stored in shared-memory, and active nodes are processed by some number of threads. A free thread picks an arbitrary active node and speculatively applies the operator to that node. Each data structure ensures that its neighborhood constraints are respected. Note that this is performed by the library code not the application code. If a neighborhood constraint is violated, a conflict is reported to the runtime system, which rolls back one of the conflicting activities. To enable rollback, each library method that modifies a data structure makes a copy of the data before modification. Like lock manipulation, rollbacks are a service implemented by the library and runtime system.

Runtime System. The Galois runtime system coordinates the parallel execution of the application. A `foreach` construct is executed by some number of threads. Each thread works on an active node from the Galois iterator and executes speculatively, rolling back the activity if needed. Library code registers with the runtime to ensure that neighborhood conflicts and rollbacks are implemented correctly.

5 GMetis

In this section, we show how the Galois system can be used to parallelize Metis. Parallelization proceeds in three steps: (1) we identify instances of amorphous data-parallelism, (2) we modify those instances to use a Galois set iterator, and (3) we modify the algorithm to use the graph data structure from the Galois library. Once we identify the amorphous data-parallelism loops, the subsequent steps (2–3) are straightforward.

Coarsening. The heavy edge matching algorithm is amorphous data-parallel. All the graph nodes are active nodes, and the neighborhood of an active node consists of itself and its direct neighbors in the graph. Nodes can be processed in any order, so this is an unordered algorithm.

Creating a coarser graph is also amorphous data-parallel. All the graph nodes in the finer graph are the active elements, and nodes in the coarser graph can be created in any order. When processing an active node n, an activity will access the neighbors of n and the neighbors of the node with which n matches. Edges are added between this set of nodes and the corresponding set in the coarser graph. This entire set of elements is the neighborhood of an activity.

Initial Partitioning. This phase generally accounts for only a small fraction of the overall runtime of Metis, so we did not investigate parallelizing it.

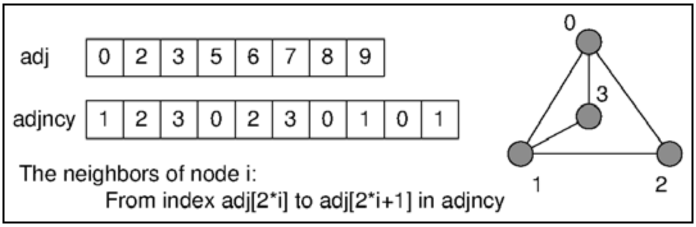
Refinement. Random k-way refinement is amorphous data-parallel. The boundary nodes can be processed in any order. They are the active nodes. When moving a node n to a neighbor partition, the partitioning information of the direct neighbors of n has to be updated. Thus, the neighborhood of each active node consists of these direct neighbors.

5.1 Optimizations

Graph Representation. The graphs in the Galois library are object-based adjacency lists. The graph keeps a list of node objects, and each node object keeps a list of its neighbors. However, this implementation is very costly compared to the compressed sparse row (CSR) format based on arrays used in Metis (see Figure 4). For instance, a serial version of Metis using Galois and using object-based adjacency lists is about an order of magnitude slower than the standard implementation of Metis written in C (see Section 6). Note that this includes the overhead of Java over C. This slowdown has nothing to do with parallelism *per se* but rather is a sequential optimization that is difficult to perform starting from the Galois library of graph implementations. The API of the CSR graph is incompatible with the more general graph API used by the library.

We have developed a variant of the Galois parallelization of Metis, which differs only in that it has been modified by hand to use a CSR graph. It is this variant that will be our main point of comparison in Section 6.

One-shot. Each one of the instances of amorphous data-parallelism identified above benefits from the one-shot optimization [11]. Briefly, if the neighborhood of an activity can be determined before executing it, then the neighborhood constraints of the activity can evaluated eagerly. This provides three benefits: (1) no rollback information needs to be stored during the execution of the activity because the activity is guaranteed to complete after the initial check, (2) the cost of an aborted activity is less because conflicts are found earlier, and (3) there are no redundant checks of neighborhood constraints because all the constraints are checked at once.



adj	0	2	3	5	6	7	8	9		
adjncy	1	2	3	0	2	3	0	1	0	1

The neighbors of node i:
From index adj[2*i] to adj[2*i+1] in adjncy

Fig. 4. The graph data structure of GHMetis is a variation of the compressed sparse row (CSR) format that allows creating the graph in parallel because the maximum degree of a node is specified beforehand.

6 Evaluation

6.1 Methodology

To evaluate the impact of exploiting amorphous data-parallelism in graph partitioning, we implemented a version of Metis written in Java and using the Galois system to exploit amorphous data-parallelism (GMetis) and a version of GMetis that additionally implements the data structure optimization mentioned in Section 5.1 by hand (GHMetis). We compared the performance of our implementations with two publicly available graph partitioners: the sequential Metis program and ParMetis, a MPI parallelization of Metis by the same authors.

Table 1 summarizes the graph partitioners that we evaluated. As we described before, Metis and ParMetis have the same algorithm framework, but they differ in (1) heuristics, for example, ParMetis gives priority to internal nodes owned by a process; (2) parameter values, such as the coarsening threshold; and (3) initial partitioning algorithm. We configured Metis to use the heavy edge matching (HEM) and random k-way refinement (KWAYRANDOM) options. This makes the implementation similar to the algorithm described in [7]. GMetis is adapted directly from Metis (with the same heuristics and parameter values) but (1) with a general-purpose graph implementation, (2) with a different algorithm to randomize visiting nodes, and (3) written in Java. GHMetis is a modification of GMetis that replaces the general-purpose graph implementation with the CSR representation described in Section 5.1.

We conducted two sets of experiments. A small-scale experiment with all four partitioners, and a large-scale experiment with only Metis, ParMetis, and GHMetis. For all experiments, we partitioned the input graph into 64 partitions. We transformed the input graphs or sparse matrices into suitable inputs to graph partitioning by making them symmetric with unit edge weights and removing all self edges.

For the small-scale experiment, we selected three graphs of road networks from the University of Florida Sparse Matrix Collection [2] and from the DIMACS shortest path competition [3]. For the large-scale experiment, we chose the 15 inputs from the University of Florida collection with the largest number of edges (number of non-zeros) whose fraction of edges to nodes was less than 20 (to select sparse matrices). The choice of cutoff was arbitrary, but, generally, more dense matrices would not benefit from exploiting amorphous data-parallelism because the number of conflicts would be high. In cases where there were multiple matrices from the same problem family, we selected only the largest input of the family except for the wikipedia family of inputs where we selected the smallest input because we had trouble running the largest input with Metis. Table 2 shows the matrices selected.

We ran all the implementations on the same test machine: a Sun Fire X2270 running Ubuntu Linux 8.04.4 LTS 64-bit. The machine contains two quad-core 2.93 GHz Intel Xeon X5570 processors. The two CPUs share 24 GB of main memory. Each core has a 32 KB L1 cache and a unified 256 KB L2 cache. Each processor has an 8 MB L3 cache that is shared among the cores.

We used Metis 5.0pre2 and ParMetis 3.1.1 compiled with gcc 4.2.4 and with the default 64-bit build options. For both programs, we configured the graph index data type to be a 64-bit integer as well. With ParMetis, we used OpenMPI 1.4.2, and multiple MPI processes were launched on the same machine. For GMetis and GHMetis, we used the Sun JDK 1.6.0 to compile and ran the programs with a heap size of 20 GB. To control for JIT compilation, we ran each input 5 times within the same JVM instance and report the median run time.

Table 1. Summary of graph partitioning algorithms evaluated

	Language	Parallelization	Graph Data Structure	Adapted From
Metis	C		CSR	-
ParMetis	C	MPI	Distributed CSR	-
GMetis	Java	Galois	Object-based adjacency list	Metis
GHMetis	Java	Galois	CSR	GMetis

6.2 Results

Tables 5 and 3 show the results from the small-scale experiment. The results are typical of many of the trends we see in the large-scale experiment as well. We define scalability as the runtime relative to the single-threaded runtime of the same program. Speedup is the runtime relative to the runtime of Metis.

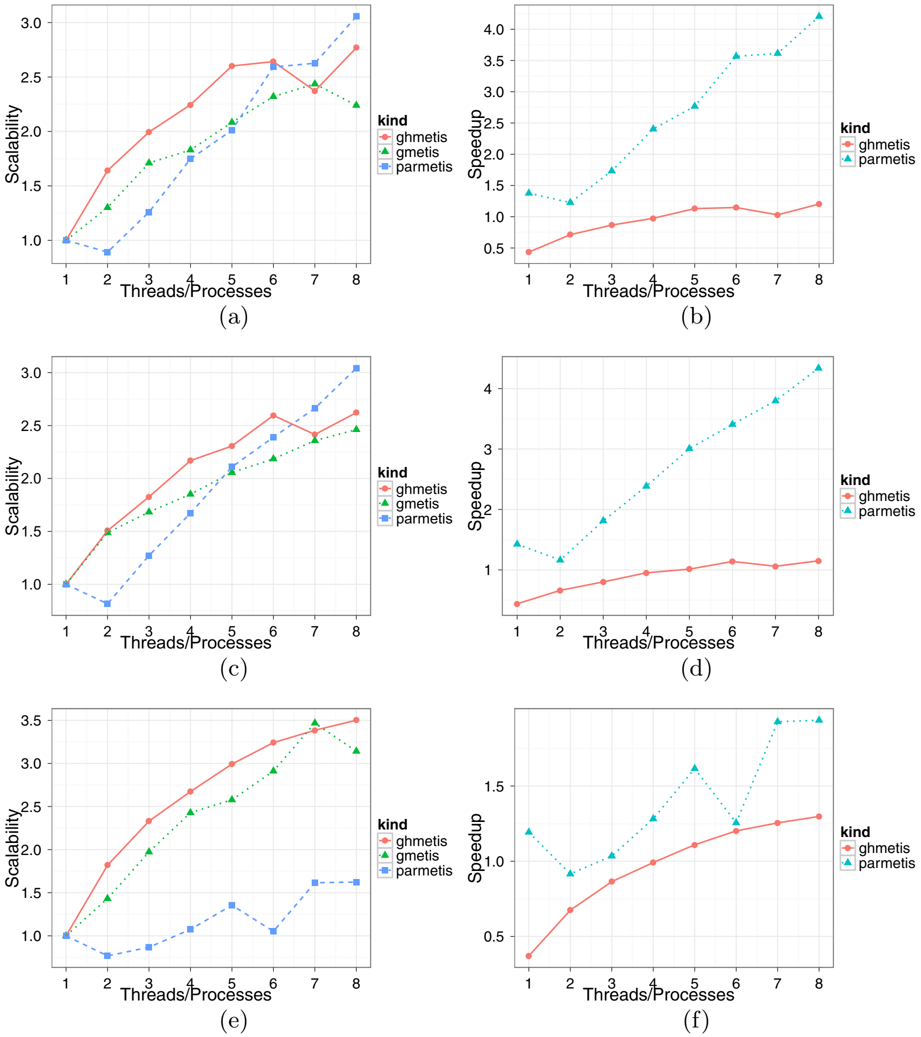
Table 2. Summary of inputs used in evaluation. The top portion lists the small-scale inputs; the bottom portion lists the large-scale inputs. All inputs are from [2] except USA-road-d.W, which is from [3].

	$\|V\|$	$\|E\|$	$\|E\|/\|V\|$	Description
roadNet-CA	1,965,206	2,766,607	1.41	Road network of California
roadNet-TX	1,379,917	1,921,660	1.39	Road network of Texas
USA-road-d.W	6,262,104	7,559,642	1.21	Road network of western USA
as-Skitter	1,696,415	11,095,298	6.54	Internet topology graph
cage15	5,154,859	47,022,346	9.12	DNA electrophoresis
circuit5M_dc	3,523,315	8,562,474	2.43	Large circuit, DC analysis
cit-Patents	3,774,768	16,518,947	4.38	Citation network among US patents
Freescale1	3,428,754	8,472,832	2.47	Circuit problem
GL7d19	1,955,309	37,322,139	19.09	Differentials of Voronoi complex
kkt_power	2,063,494	6,482,320	3.14	Nonlinear optimization
memchip	2,707,524	6,621,370	2.45	Memory chip
patents	3,750,822	14,970,766	3.99	NBER US patent citations
rajat31	4,690,002	7,813,751	1.67	Circuit simulation matrix
rel9	5,921,786	23,667,162	4.00	Relations
relat9	9,746,232	38,955,401	4.00	Relations
Rucci1	1,977,885	7,791,154	3.94	Ill-conditioned least-squares problem
soc-LiveJournal1	4,846,609	42,851,237	8.84	LiveJournal online social network
wikipedia-20051105	1,598,534	18,540,603	11.60	Link graph of Wikipedia pages

Table 3. Time, edge cut and balance of ParMetis, GMetis and GHMetis as a function of input and number of threads. All times are in milliseconds. Metis results (Time, Cut, Balance) for roadNet-CA, roadNet-TX and USA-road.d.W are (1644, 6010, 1.02), (1128, 4493, 1.02), and (5704, 3113, 1.01) respectively.

		ParMetis			GMetis			GHMetis		
	T	Time	Cut	Bal.	Time	Cut	Bal.	Time	Cut	Bal.
roadNet-CA	1	1,195	5,752	1.03	16885	9886	1.03	3785	5525	1.02
	2	1,340	6,455	1.04	12982	9495	1.03	2307	5882	1.02
	4	684	6,341	1.04	9226	9401	1.02	1686	5605	1.02
	8	391	6,472	1.03	7540	9707	1.03	1367	5865	1.03
roadNet-TX	1	791	4,426	1.03	12067	4760	1.03	2570	4592	1.02
	2	970	4,715	1.05	8114	4443	1.02	1706	4237	1.03
	4	473	4,705	1.05	6517	4433	1.03	1185	4165	1.02
	8	260	4,611	1.04	4901	4329	1.02	980	4232	1.02
USA-road-d.W	1	4,781	11,012	1.18	68151	3057	1.01	15384	2930	1.02
	2	6,230	6,382	1.23	47598	3007	1.01	8457	2951	1.00
	4	4,449	5,868	1.22	28064	2951	1.00	5754	2971	1.01
	8	2,944	5,455	1.22	21691	3050	1.00	4394	3175	1.01

For single-threaded runs, GMetis is about five times slower than GHMetis, and GHMetis is about twice as slow as Metis and ParMetis. The scalability of ParMetis, GMetis and GHMetis is similar for the smaller inputs, roadNet-CA and roadNet-TX, but for USA-road-d.W, GMetis and GHMetis have better scalability than ParMetis. In the large-scale experiments, the scalability gap



(a)

(b)

(c)

(d)

(e)

(f)

Fig. 5. Scalability and speedup of ParMetis, GMetis and GHMetis. Scalability is runtime relative to runtime with one thread/process. Speedup is runtime relative to runtime of sequential Metis. (a) and (b) are scalability and speedup for roadNet-CA. (c) and (d) are scalability and speedup for roadNet-TX. (e) and (f) are scalability and speedup for USA-road-d.W.

between GHMetis and ParMetis becomes more pronounced. For USA-road-d.W, ParMetis produces worse partitions in terms of balance and edge cut than the other three partitioners. We believe this is due to the different initial partitioning phase. Also, ParMetis uses a different partitioning strategy when run with one process than with more than one process. This may explain the much larger edge cut in the one process case. ParMetis has better speedup than GHMetis largely

Table 4. Performance of Metis, ParMetis and GHMetis. m/t is speedup, runtime relative to sequential Metis (m). t/t1 is scalability, runtime relative to single-threaded runtime. All times are in milliseconds. Blank entries correspond to runs that terminated abnormally or exceeded the timeout of 30 minutes.

	Metis	Best ParMetis		Best GHMetis	
	Time	m/t	t1/t	m/t	t1/t
as-Skitter	273,581			14.86	2.50
cage15	23,677	2.86	2.31	1.04	3.16
circuit5M_dc	3,164	1.63	1.38	1.26	3.20
cit-Patents	58,740	4.34	1.62	2.22	2.93
Freescale1	2,944	1.90	1.60	1.22	3.61
GL7d19	168,199	4.51	6.64	2.43	2.50
kkt_power	10,445	3.80	1.34	1.22	2.91
memchip	2,368	2.14	1.84	1.35	3.27
patents	51,695	4.10	1.65	2.08	2.91
rajat31	4,044	4.27	3.49	1.35	3.37
rel9			1.13		
relat9	1,106,377	2.63	1.06		
Ruccil	1,065,551	20.26	1.03	48.45	3.15
soc-LiveJournal1	304.295				
wikipedia-20051105	358,030			8.98	2.54
Geomean		**3.60**	**1.77**	**2.80**	**2.98**

Table 5. Balance and EdgeCut of Metis, ParMetis and GHMetis. Blank entries correspond to runs that terminated abnormally or exceeded the timeout of 30 minutes.

	Metis		Best ParMetis		Best GHMetis	
	Cut	Bal.	Cut	Bal.	Cut	Bal.
as-Skitter	3,054,856	1.03			1,991,020	1.03
cage15	4,536,885	1.03	4,697,417	1.05	4,629,593	1.03
circuit5M_dc	13,187	1.03	14,764	1.05	14,133	1.02
cit-Patents	3,036,598	1.02	3,258,823	1.05	2,900,222	1.02
Freescale1	13,429	1.03	15,093	1.05	13,828	1.02
GL7d19	31,168,010	1.03	34,248,358	1.26	31,295,495	1.03
kkt_power	453,357	1.02	578,264	1.04	392,115	1.01
memchip	16,235	1.02	18,524	1.05	16,534	1.02
patents	2,672,325	1.02	2,841,655	1.05	2,550,783	1.01
rajat31	27,391	1.01	27,907	1.04	26,851	1.00
rel9			12,774,163	1.05		
relat9	22,417,154	1.03	21,532,960	1.05		
Ruccil	1,890,352	1.03	1,928,212	1.01	1,074,278	1.00
soc-LiveJournal1	13,838,247	1.03				
wikipedia-20051105	10,081,144	1.03			9,389,056	1.03
Geomean		**1.02**		**1.06**		**1.02**

due to starting with a better single-threaded runtime. Recall that GHMetis is implemented in Java whereas ParMetis and Metis are implemented in C.

Tables 4 and 5 show the results from the large-scale experiment. Instead of showing results for each number of threads/processes, we only show the best performing result for ParMetis and GHMetis and its corresponding edge cut and balance. The trends from the small-scale experiment show up here as well. GHMetis achieves better scalability and produces better partitions than ParMetis, but ParMetis is faster than GHMetis. In fact, it is often faster than Metis as well. Observe that the speedup of ParMetis is greater than its scalability. Over a large set of inputs, we see that GHMetis scales better than ParMetis, suggesting that amorphous data-parallelism is a fruitful form of parallelism to exploit.

The missing runs for GHMetis are generally due to lack of memory. They occur on larger inputs when GHMetis spends most of its time doing garbage collection. For inputs as-Skitter, rel9 and Ruccil, Metis performs particularly poorly compared to ParMetis or GHMetis. We believe that this is due to the randomization strategy used in Metis, which causes the coarsening phase to stop early and consequentially produces a very large input to the initial partitioning phase. When we used the same randomization strategy in GHMetis, we observed similarly poor performance.

7 Conclusion

Graph partitioning is an important problem in parallel computing, and we have shown how one common graph partitioning application, Metis, naturally exhibits a form of parallelism that we call amorphous data-parallelism. Using the Galois system, we can exploit this parallelism to achieve reasonable parallel scalability from a sequential specification, and this scalability is comparable to that of an explicitly parallel implementation over a suite of large test matrices. An advantage of the Galois version is that it is derived directly from the sequential application.

Our naïve implementation still does not obtain consistent speedup over sequential Metis, but we have shown how changing the graph data structure bridges the gap considerably. In addition, we have not parallelized the initial partitioning phase of the algorithm. We also believe that a significant overhead exists because our implementation is in Java, whereas Metis is hand-tuned C.

The previous approaches to parallelizing graph partitioning [1,6,13] are complementary to our approach. On a hybrid architecture consisting of multiple multicore machines, amorphous data-parallelism can exploit intra-machine parallelism while message-passing can exploit inter-machine parallelism.

References

1. Chevalier, C., Pellegrini, F.: Pt-scotch: A tool for efficient parallel graph ordering. Parallel Computing 34(6-8), 318–331 (2008)
2. Davis, T.A., Hu, Y.F.: The university of florida sparse matrix collection (in submission). ACM Transactions on Mathematical Software (2010)

3. DIMACS. 9th dimacs implementation challenge—shortest paths (2005), http://www.dis.uniroma1.it/~challenge9
4. George, A.: Nested dissection of a regular finite element mesh. SIAM Journal on Numerical Analysis 10(2), 345–363 (1973)
5. Gupta, A.: An evaluation of parallel graph partitioning and ordering software on a massively parallel computer. Technical Report RC25008 (W1006-029), IBM Research Division, Thomas J. Watson Research Center (2010)
6. Karypis, G., Kumar, V.: A coarse-grain parallel formulation of multilevel k-way graph-partitioning algorithm. In: Proc. 8th SIAM Conference on Parallel Processing for Scientific Computing (1997)
7. Karypis, G., Kumar, V.: Multilevel k-way partitioning scheme for irregular graphs. Journal of Parallel and Distributed Computing 48(1), 96–129 (1998)
8. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM Journal on Scientific Computing 20(1), 359–392 (1999)
9. Kernighan, B.W., Lin, S.: An effective heuristic procedure for partitioning graphs. The Bell System Technical Journal, 291–308 (February 1970)
10. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Chew, L.P.: Optimistic parallelism requires abstractions. SIGPLAN Not. (Proceedings of PLDI 2007) 42(6), 211–222 (2007)
11. Mendez-Lojo, M., Nguyen, D., Prountzos, D., Sui, X., Hassaan, M.A., Kulkarni, M., Burtscher, M., Pingali, K.: Structure-driven optimizations for amorphous data-parallel programs. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 3–14 (2010)
12. Pingali, K., Kulkarni, M., Nguyen, D., Burtscher, M., Mendez-Lojo, M., Prountzos, D., Sui, X., Zhong, Z.: Amorphous data-parallelism in irregular algorithms. regular tech report TR-09-05, The University of Texas at Austin (2009)
13. Walshaw, C., Cross, M.: Jostle: Parallel multilevel graph-partitioning software—an overview. In: Magoules, F. (ed.) Mesh Partitioning Techniques and Domain Decomposition Techniques, pp. 27–58. Civil-Comp. Ltd. (2007)

The STAPL pView*

Antal Buss, Adam Fidel, Harshvardhan, Timmie Smith,
Gabriel Tanase, Nathan Thomas, Xiabing Xu, Mauro Bianco,
Nancy M. Amato, and Lawrence Rauchwerger

Parasol Lab, Dept. of Computer Science and Engineering, Texas A&M University
stapl@cse.tamu.edu

Abstract. The Standard Template Adaptive Parallel Library (STAPL) is a C++ parallel programming library that provides a collection of distributed data structures (pContainers) and parallel algorithms (pAlgorithms) and a generic methodology for extending them to provide customized functionality. STAPL algorithms are written in terms of `pViews`, which provide a generic access interface to pContainer data by abstracting common data structure concepts. Briefly, `pViews` allow the same pContainer to present multiple interfaces, e.g., enabling the same pMatrix to be 'viewed' (or used) as a row-major or column-major matrix, or even as a vector. In this paper, we describe the STAPL `pView` concept and its properties. `pViews` generalize the iterator concept and enable parallelism by providing random access to, and an ADT for, collections of elements. We illustrate how `pViews` provide support for managing the tradeoff between expressivity and performance and examine the performance overhead incurred when using `pViews`.

1 Introduction

Decoupling of data structures and algorithms is a common practice in generic programming. STL, the C++ Standard Template Library, obtains this abstraction by using *iterators*, which provide a generic interface for algorithms to access data that is stored in containers. This mechanism enables the same algorithm to operate on multiple containers. In STL, different containers support various types of iterators that provide appropriate functionality for the data structure, and algorithms can specify which types of iterators they can use. For example,

* This research supported in part by NSF awards CRI-0551685, CCF-0833199, CCF-0830753, IIS-096053, IIS-0917266, NSF/DNDO award 2008-DN-077-ARI018-02, by the DOE NNSA under the Predictive Science Academic Alliances Program by grant DE-FC52-08NA28616, by THECB NHARP award 000512-0097-2009, by Chevron, IBM, Intel, Oracle/Sun and by Award KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST). This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Tanase is now a Research Staff Member at IBM T.J. Watson Research Center. Bianco is now a scientist at the Swiss National Supercomputing Centre.

K. Cooper, J. Mellor-Crummey, and V. Sarkar (Eds.): LCPC 2010, LNCS 6548, pp. 261–275, 2011.

algorithms requiring write operations cannot work on input iterators and lists do not support random access iterators. The major capability provided by the iterator is a mechanism to traverse the data of a container.

The Standard Template Adaptive Parallel Library (STAPL) [3] provides building blocks for writing parallel programs – parallel algorithms (pAlgorithms), parallel and distributed containers (pContainers), and pViews to abstract data accesses to pContainers. pAlgorithms are represented in STAPL as task graphs called pRanges. The STAPL runtime system includes a communication library (ARMI) and an executor that executes pRanges. The STAPL pView generalizes the iterator concept by providing an abstract data type (ADT) for the data it represents. While an iterator corresponds to a single element, a pView corresponds to a collection of elements. Also, while an iterator primarily provides a traversal mechanism, pViews provide a variety of operations as defined by the ADT. For example, all STAPL pViews support size() operations that provide the number of elements represented by the pView. A STAPL pView can provide operations that return new pViews. For example, a pMatrix supports access to rows, columns, and blocks of its elements through row, column and blocked pViews, respectively.

pViews are designed to enable parallelism. In particular, pViews provide random access to partitioned collections of elements of each ADT supported by STAPL. This characteristic is essential for the scalability of STAPL programs. The size of these collections can be dynamically controlled and typically depends on the desired degree of parallelism. For example, the pList pView provides concurrent access to segments of the list, where the number of segments could be set to match the number of parallel processes. The pView provides random access to a partitioned data space. To mitigate the potential loss of locality incurred by the flexibility of the random access capability, pViews provide, to the degree possible, a remapping mechanism of a user specified pView to the container's physical distribution (known as the native pView).

In this paper, we describe the STAPL pView concept and its properties. As outlined above, critical aspects of the pView are:

– STAPL pViews generalize the iterator concept — a pView corresponds to a collection of elements and provides an ADT for the data it represents.
– STAPL pViews enable parallelism — this is done by providing random access to the elements, and support for managing the tradeoff between the expressivity of the views and the performance of the parallel execution.

2 STAPL Overview

STAPL [3,20,15,16] is a framework for parallel C++ code development (Figure 1). Its core is a library of parallel algorithms (pAlgorithms) and distributed data structures (pContainers)[18] that have interfaces similar to the (sequential) C++ standard library (STL) [12]. Analogous to STL algorithms that use *iterators*, STAPL pAlgorithms are written in terms of pViews so that the same algorithm can operate on multiple pContainers.

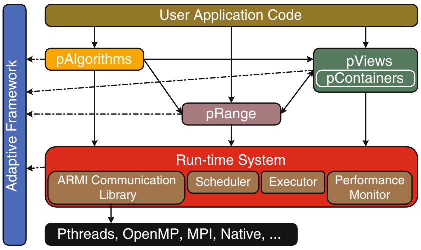


Fig. 1. STAPL Overview

`pAlgorithms` are represented by `pRanges`. Briefly, a `pRange` is a graph whose vertices are tasks and edges are dependencies, if any, between tasks. A task includes both *work* (*workfunctions*) and *data* (from `pContainers`, generically accessed through `pViews`). The `executor`, itself a distributed shared object, is responsible for the parallel execution of computations represented by `pRanges`. Nested parallelism can be created by invoking a `pAlgorithm` from within a task.

STAPL `pContainers` are distributed, thread-safe, concurrent objects, i.e., shared objects that provide parallel methods that can be invoked concurrently. They are composable and extendible via inheritance. Currently, STAPL provides counterparts of all STL containers (e.g., `pArray`, `pVector`, `pList`, `pMap`, etc.), and two `pContainers` that do not have STL equivalents: parallel matrix (`pMatrix`) and parallel graph (`pGraph`). `pContainers` include a set of `bContainers`, that are the basic storage components for the elements, and distribution information that manages the distribution of the elements across the parallel machine.

`pContainers` provide methods corresponding to the STL container methods, and some additional methods specifically designed for parallel use. For example, STAPL provides an `insert_async` method that can return control to the caller before its execution completes, or an `insert_anywhere` that does not specify where an element is going to be inserted and is executed asynchronously. While a `pContainer`'s data may be distributed, `pContainers` offer the programmer a *shared object view*, i.e., they are shared data structures with a global address space. This is supported by assigning each `pContainer` element a unique global identifier (GID) and by providing each `pContainer` an internal translation mechanism that can locate, transparently, both local and remote elements. The physical distribution of `pContainer` data can be determined automatically by STAPL or it can be user-specified.

The runtime system (RTS) and its communication library ARMI (Adaptive Remote Method Invocation) provide the interface to the underlying operating system, native communication library and hardware architecture. ARMI uses the remote method invocation (RMI) communication abstraction to hide the lower level implementation (e.g., MPI, OpenMP, etc.). The RTS provides *locations* as an abstraction of processing elements in a system. A *location* is a component of a parallel machine that has a contiguous memory address space and has associated execution capabilities (e.g., threads).

3 Related Work

The view concept has been used in different areas.

One of first uses of the view concept was in database systems. In particular, views can be defined by a database query and used to represent a virtual table in a relational database or an entity in an object oriented database. Generally, database views are read-only. However, views can be updatable (writable) if the database supports reverse mappings from a view to the database. Some systems implement updatable views using an "instead of" trigger that is executed when an insert, delete or update over the view is executed. A similar approach is presented in [1] where lenses implement bidirectional transformations.

GIL (Generic Image Library) [2,8] is a C++ image library from Adobe for image manipulation. It provides the concept of an image view, which generalizes STL's range concept [13] to multiple dimensions. GIL's image views are specialized for operating on two-dimensional images which may have different storage distributions in memory, but are always in the same address space.

The VTL (View Template Library) [21] project worked with views as an adaptor layer on top of STL. This project, which has been inactive since 2000, was heavily inspired by the Views library of Jon Seymour [17]. A VTL view is a container adaptor, that provides a container interface to access a portion of the data, to rearrange the data, to transform data, or to combine data. The STAPL `pView` provides similar capabilities for `pContainers`.

The view concept has been used in some PGAS (Partitioned Global Address Space) languages. X10 [6] provides the notion of a region to specify a section of data. Chapel [5] provides the user a global view over a container, specifies subarrays with domains, and uses iterators [10] as abstractions for algorithms. STAPL `pViews` provide a container interface in addition to support for iterators.

The Hierarchically Tiled Array (HTA) data type [7] provides a rich interface to specify array views. It also implements advanced support for handling the boundary communication of common patterns arising in scientific computing. STAPL overlap `pViews` are similar to HTA overlapped tiling, though STAPL supports arbitrary, static and dynamic data types.

4 STAPL `pView` Concept

In this section, we introduce the `pView` concept and explain how it can be exploited in the parallel and distributed environment of STAPL.

A `pView` is a class that defines an abstract data type (ADT) for the *collection of elements* it represents. As an ADT, a `pView` provides *operations* to be performed on the collection, such as read, write, insert, and delete.

`pViews` have *reference semantics*, meaning that a `pView` does not own the actual elements of the collection but simply *references* to them. The collection is typically stored in a `pContainer` to which the `pView` refers; this allows a `pView` to be a relatively light weight object as compared to a container. However, the collection could also be another `pView`, or an arbitrary object that provides a container interface. With this flexibility, the user can define `pViews` over `pViews`, and also `pViews` that generate values dynamically, read them from a file, etc.

All the operations of a `pView` must be routed to the underlying collection. To support this, a mapping is needed from elements of the `pView` to elements of the underlying collection. This is done by assigning a unique identifier to each `pView` element (assigned by the `pView` itself); the elements of the collection must also have unique identifiers. Then, the `pView` specifies a *mapping function* from the `pView`'s *domain* (the union of the identifiers of the `pView`'s elements) to the collection's domain (the union of the identifiers of the collection's elements).

More formally, a `pView` $\mathcal{V}$ is a tuple

$$\mathcal{V} \overset{def}{=} (\mathcal{C}, \mathcal{D}, \mathcal{F}, \mathcal{O}) \tag{1}$$

where $\mathcal{C}$ represents the underlying typed collection, $\mathcal{D}$ defines the domain of $\mathcal{V}$, $\mathcal{F}$ represents the mapping function from $\mathcal{V}$'s domain to $\mathcal{C}$'s domain, and $\mathcal{O}$ is the set of operations provided by $\mathcal{V}$, which must also be supported by $\mathcal{C}$.

To support parallel use, the $\mathcal{C}$ and $\mathcal{D}$ components of the `pView` can be partitioned so that they can be used in parallel. Also, most generally, the mapping function $\mathcal{F}$ and the operations $\mathcal{O}$ may differ for each component of the partition. That is, $\mathcal{C} = \{c_0, c_1, \ldots, c_{n-1}\}$, $\mathcal{D} = \{d_0, d_1, \ldots, d_{n-1}\}$, $\mathcal{F} = \{f_0, f_1, \ldots, f_{n-1}\}$, and $\mathcal{O} = \{o_0, o_1, \ldots, o_{n-1}\}$. This is a very general definition and not all components are necessarily unique. For example, the mapping functions f_i and the operations o_i may often be the same for all $0 \leq i < n$. The tuples (c_i, d_i, f_i, o_i) are called the *base views* (`bViews`) of the `pView` $\mathcal{V}$. The `pView` supports parallelism by enabling random access to its `bViews`, which can then be used in parallel by `pAlgorithms`.

Note that we can generate a variety of `pViews` by selecting appropriate components of the tuple. For instance, it becomes straightforward to define a `pView` over a subset of elements of a collection, e.g., a `pView` of a block of a `pMatrix` or a `pView` containing only the even elements of an array. As another example, `pViews` can be implemented that transform one operation into another. This is analogous to backinserter iterators in STL in which a write operation is transformed into a pushback in a container.

Example. A common concept in generic programming is a one-dimensional array of size n supporting random access. The `pView` corresponding to this has an integer domain $\mathcal{D} = [0, n)$ and operations $\mathcal{O}$ including the random access read and write operators. This `pView` can be applied to any container by providing a mapping function $\mathcal{F}$ from the domain $\mathcal{D} = [0, n)$ to the desired identifiers of the container. If the container provides the operations, then they can be inherited using the mechanisms provided in the base `pView`. If new behavior is needed, then the developer can implement it explicitly.

Composition of views. Since a `pView` and the collection it represents can be used interchangeably, the `pView` definition (Equation 1) naturally enables *composition*, i.e., `pViews` defined over other `pViews`. Figure 2(a) shows the construction of `pViews` over other `pViews`, and the possibility of having multiple `pViews`

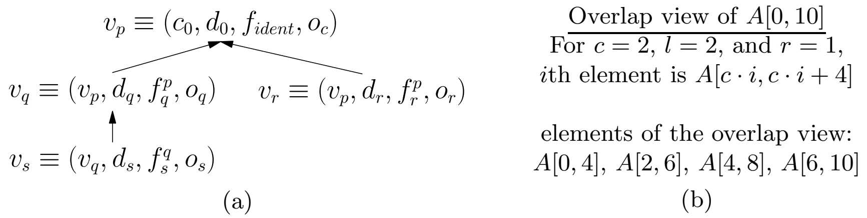

$v_p \equiv (c_0, d_0, f_{ident}, o_c)$
$v_q \equiv (v_p, d_q, f^p_q, o_q)$
$v_r \equiv (v_p, d_r, f^p_r, o_r)$
$v_s \equiv (v_q, d_s, f^q_s, o_s)$
(a)
Overlap view of $A[0, 10]$
For $c = 2$, $l = 2$, and $r = 1$,
ith element is $A[c \cdot i, c \cdot i + 4]$
elements of the overlap view:
$A[0, 4]$, $A[2, 6]$, $A[4, 8]$, $A[6, 10]$
(b)

Fig. 2. (a) Example of construction of `pViews` over the same collection. v_p is a `pView` over the collection c_0. v_q and v_r are two, possibly different, `pViews` over v_p, and v_s is a `pView` over v_q. (b) Example overlap `pView` for $A[0, 10]$.

concurrently referencing the same container. Thus, composition makes possible the representation of complex data organizations and naturally supports the recursive partitioning of domains.

4.1 Useful Views

There are several types of `pViews` worthy of note because they enable optimizations or are useful in expressing computations.

By providing certain operations and not others, `pViews` can be classified as *read-only* or *write-only*. This is analogous to the STL input and output iterators.

Some special cases of `pViews` are particularly useful in the context of parallel programming. For instance the *single-element partition*, where the domain of the collection is split into single elements and all mapping functions are identity functions. This partition enables maximum parallelism and is the default partition adopted by STAPL when calling a `pAlgorithm`.

Other important `pViews` include the *balanced* `pView` where the data is split into a specified number of approximately equal-sized chunks, and the *native* `pView`, which provides `bViews` that are *aligned* with the `pContainer` distribution. These turn out to be very useful in the context of STAPL.

Transform `pViews` apply a user specified function to the elements returned from the collection. This feature can be used to change the value, type, or both, of the returned element. Important aspects of the transform `pView` are that the elements in the underlying collection are not modified and the result is computed and made available only when an element accessed through the `pView` is actually referenced in the program. In contrast, for example, a `for_each` algorithm applied to a `pContainer`, would traverse and modify all elements of the container within the relevant range.

There are also a number of useful views that have more complex elements. One example is a *zip* `pView`, which takes two (or more) collections and provides a `pView` where each element is a pair (or tuple) including an element from each collection. Zip views are useful for expressing algorithms that operate on multiple collections. Another `pView` heavily used in STAPL is the *overlap* `pView`, in which one element of the `pView` overlaps another element of the view. This `pView` is naturally suited for specifying many algorithms, such as adjacent differences,

Table 1. Major views implemented in STAPL and corresponding operations. `tranform_view` implements an overridden read operation that returns the value produced by a user specified function, the other operations depends on the `pView` the transform `pView` is applied to. `insert_any` refers to the special operations provided by STAPL `pContainers` that insert elements in unspecified positions.

	read	write	[ ]	begin	insert	insert
				end	erase	any
`array_1d_pview`	✔	✔	✔	✔		
`array_1d_ro_pview`	✔		✔	✔		
`static_list_pview`	✔			✔		
`list_view`	✔	✔		✔	✔	✔
`matrix_pview`	✔	✔	✔			
`graph_pview`	✔	✔			✔	✔
`strided_1D_pview`	✔	✔	✔	✔		
`transform_pview`	✔		-	-		
`balanced_pview`	✔		✔	✔		
`overlap_pview`	✔		✔	✔		
`native_pview`	✔		✔	✔		
`repeated_pview`	✔		✔	✔		

string matching, etc. The *repeated* `pView` is a special case of an overlapped `pView` in which each element includes the entire collection. As an example, we can define an overlap `pView` for a one-dimensional array $A[0, n-1]$ using three parameters, c (core size), l (left overlap), and r (right overlap), so that the ith element of the overlap `pView` $v^o[i]$ is $A[c \cdot i, c \cdot i + l + c + r - 1]$. See example in Figure 2(b).

5 The `pView` Class

The `pView` is an object that builds on the STAPL `pContainer` framework. To create a `pView`, the user specifies the partitioned collection (often a `pContainer`), the partitioned domain $\mathcal{D}$, and the mapping functions $\mathcal{F}$, as (template) arguments of the `pView` class, while the operations $\mathcal{O}$ must be implemented by the class itself. All STAPL `pViews` are derived from the `core_view` templated base class. This class provides constructors, and stores references to $\mathcal{C}$, $\mathcal{D}$, and $\mathcal{F}$.

To ease the implementation of the basic operations, and thus the implementation of the generic `pView` concepts needed by STAPL algorithms, the user can derive the `pView` class from classes implementing those operations, e.g., a `pContainer`. Usually, the `pView` can directly invoke the `pContainer` methods. An exception is the transform `pView`, where the read operation is implemented as `return F(container.operation(f(i), ...))` and `F` is the transformation function, and `f` is the mapping function.

`pViews` *in* STAPL. Table 1 shows a list of some `pViews` available in STAPL. These `pViews` are implemented using the schema discussed above, and new `pViews` can be implemented and created in the same way. The *native* `pView` is a `pView` whose partitioned domain $\mathcal{D}$ matches the data partition of the underlying `pContainer`, allowing data references to it to be local. The *balanced* `pView` partitions the data set into a user specified number of pieces. The sizes of the pieces differs by at most by one. This `pView` can be used to balance the amount of work in a parallel computation. If STAPL algorithms can use balanced or native `pViews`, then performance is greatly enhanced.

Optimizations. There are trade-offs between the expressivity offered by the `pViews` and performance. For this reason, the `pViews` are designed to allow the implementation of different optimizations to improve the performance of data access. Below, we present a few such examples.

The repeated composition of `pViews`, an important technique to develop new `pViews`, can result in an increasing chain of indirect data references due to the repeated composition of the mapping functions ($\mathcal{F}$s). In certain cases, such as when where $\mathcal{F}$ is statically known, and has a relatively simple closed form expression, STAPL can reduce the chain of indirections to one. For instance, composing identity functions results in another identity function, while composing an arbitrary function $\mathcal{F}$ with an identity function is the same $\mathcal{F}$.

Another important optimization is localization of memory references. STAPL `pViews` can determine which sections of consecutive references are local (within the same address space). This allows the `pView` to use a much simpler, and thus much faster mechanism to reference local data.

6 Results: Expressivity, Genericity, and Performance

In this section, we present experimental results to study the trade-offs between the enhanced expressivity enabled by `pViews` and their performance. For this purpose, we compare the performance of functionally equivalent STAPL programs written using `pViews` and C++ MPI programs.

We conducted our experimental studies on two architectures: an 832 processor IBM cluster with p575 SMP nodes (16 cores per node) available at Texas A&M University (called P5-CLUSTER) and a 38,288 processors Cray XT4 with quad core Opteron processors (4 cores per node) available at NERSC (called CRAY4-CLUSTER). The compiler used for the experiments was gcc (version 4.3.1 on P5-CLUSTER and version 4.4.1 on CRAY4-CLUSTER) with the `-O3` optimization flag. In all experiments, a location contains a single processor, and the terms can be used interchangeably.

Genericity. We can solve many problems using the `stapl::count_if(view, pred)` algorithm which takes an `array_1D_view` and counts how many times the referenced elements satisfy a user provided predicate `pred`.

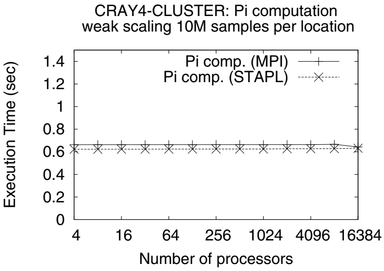
For instance, we can compute π using the well known Monte Carlo method [14]. Random points are generated inside the unit square and we count how many of these fall inside the unit circle. The ratio between these and the total number of points generated is $\pi/4$. The `pView` used to represent the input does not need a reference to storage because the points can be generated on demand. Hence, the container provided to the `pView` is a simple class that exports the

```
struct in_circle {
 template<typename Reference>
 bool operator()(Reference ref) const {
   std::pair<double, double> val = ref;
   return sqrt(pow(val.first,2.0) +
        pow(val.second,2.0)) < 1.0;
 }
}

void stapl_main(int argc, char** argv){
  typedef array_1D_ro_view
     <rand_gen_cont> rand_gen_view;

  rand_gen_cont rgc(N);
  rand_gen_view rgenv(rgc);

  int result =
      stapl::count_if(rgenv,in_circle());
  double pi = 4*(result/(double)N);
}
```

(a)

```
MPI_Init(&argc, &argv);
int P;

MPI_Comm_size(MPI_COMM_WORLD, &P);

N=N/P;

int pid;
MPI_Comm_rank(MPI_COMM_WORLD,&pid);

int cnt(0);
for (int i=0; i<N; ++i) {
  double xcoord = drand48();
  double ycoord = drand48();

  double dist =
          sqrt(pow(xcoord, 2.0)
          + pow(ycoord, 2.0));

  if (dist < 1.0)
     ++cnt;
}

int res;
MPI_Reduce(&cnt,&res,1,MPI_INT,MPI_SUM,0,
           MPI_COMM_WORLD);
MPI_Finalize();

double pi = 4*(res/(double)(N*P));
```

(b)



(c)

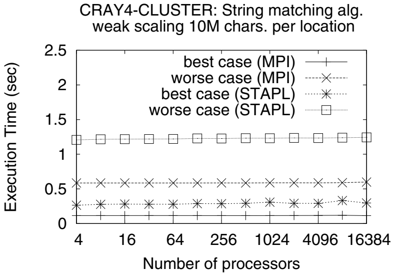
Fig. 3. Computing π. (a) STAPL code using a view over a generator container. (b) MPI version. (c) Execution times on CRAY4-CLUSTER.

container interface and whose read method returns a randomly generated point in the unit square. Passing this `pView` to `stapl::count_if`, with a predicate to check if the point lies within the unit circle, will execute the π computation. We also evaluated an equivalent C++ MPI program for computing π. The code snippets are shown in Figures 3(a) and 3(b), respectively. Note that the two programs are comparable in terms of complexity for this embarrassingly parallel algorithm. Figure 3(c) shows that the performance for the two implementations is comparable, with the STAPL program slightly outperforming the MPI version.

String matching can also be implemented by calling `stapl::count_if(view, pred)` with an appropriate `pView` and predicate. In this case, given a pattern of length M, we create an overlap `pView` over the text, with a core of length 1, left overlap of size 0 and right overlap of size $M - 1$. This will give a `pView` over all the sub-strings of size M of the input text. The code sample is shown in Figure 4(a). In Figure 4(b), an MPI version of the program is shown. In this case it becomes possible to appreciate the additional complexity of the MPI code with respect to the STAPL version, since in MPI the programmer must take explicit care of the boundary regions (this is a special case of the use of ghost nodes, a well known technique in parallel processing [11,7]). Figure 4(c)

```
struct strmatch {
  const string& S;
  strmatch(const string& s): S(s) {}

  template<typename View>
  bool operator()(View v) const {
    return equal(S.begin(),S.end(),
                 v.begin());
  }
};

void stapl_main(int argc, char** argv) {
  typedef stapl::p_array<char>
      p_string_type;
  typedef stapl::array_1D_view
      <p_string_type> pstringView;
  ...
  result=stapl::count_if(
           stapl::overlap_view(text,
           1,0,pattern.size()-1),
           strmatch(pattern));
  ...
}
```

(a)



(c)

```
int main(int argc, char** argv) {
  ...
  MPI_Comm_size(MPI_COMM_WORLD, &P);

  N=N/P;
  std::vector<char> V(N);
  int M=S.length();

  for (int i=0; i <= N-M+1; ++i)
    if (equal(S.begin(), S.end(),
             V.begin()+i)) ++cnt;
  if (pid>0)
    MPI_Send((&V[0]), M-1, MPI_CHAR,
          pid-1, 1, MPI_COMM_WORLD);

  if (pid<P-1) {
    vector<char> BUFF(2*(M-1));
    copy(V.begin()+N-M+1, V.end(),
              BUFF.begin());

    MPI_Recv( &BUFF[M-1], M-1, MPI_CHAR,
              pid+1, 1, MPI_COMM_WORLD,
              &status );

    for (int i=0; i <= M-1; ++i)
      if (equal(S.begin(), S.end(),
            BUFF.begin()+i ))
         ++cnt;
  }

  int res;
  MPI_Reduce ( &cnt, &res, 1, MPI_INT,
               MPI_SUM, 0, MPI_COMM_WORLD );
  ...
}
```

(b)

Fig. 4. String matching. (a) STAPL code using an overlap partitioned view. (b) MPI version. (c) Execution times on CRAY4-CLUSTER.

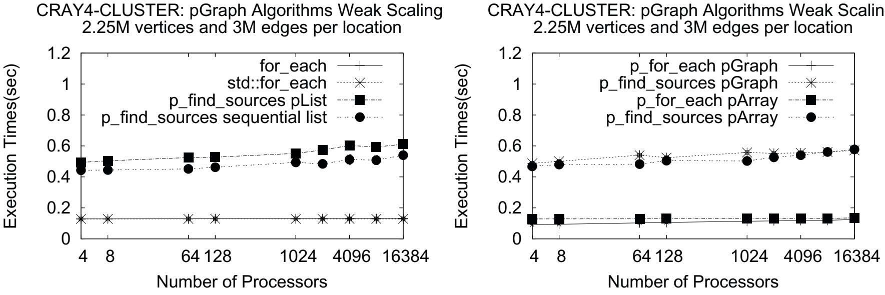
shows that performance of the two versions is comparable. In the best case, the first character of the substring is not in the text and the number of occurrences is zero. In the worse case, both text and substring are composed of the same character, maximizing the number of occurrences.

Graph views. STAPL provides the `pGraph`, a parallel and distributed graph data structure. Algorithms operating on `pGraphs` are written generically in terms of graph `pView` concepts. In this section, we describe `pGraph` specific `pViews` and discuss the performance of generic algorithms using them.

For simple operations such as initializing the data of each vertex or edge, we can use a view over the set of vetrices and edges. These views implement the `static_list` concept and support efficient parallel map and map_reduce operations. In Figure 5, we show a STAPL program that performs an initialization of the `pGraph` vertex properties (Figure 5, Line 5), and then computes and stores the set of source vertices in a parallel list (Figure 5, Line 6). The `list_view` (Figure 5, Line 4) defined over a parallel list [19] supports an interface to efficiently

```
stapl::p_graph<vertex_property>                          graph;
stapl::graph_view<stapl::p_graph<vertex_property> >  graph_view(graph);
stapl::p_list<vertex>                                    list;
stapl::list_view<stapl::p_list<vertex> >                 result_view(list);
stapl::for_each(graph_view.vertices(), init_property());
stapl::p_find_sources(graph_view.edges(), result_view);
```

Fig. 5. Find sources and sinks in a graph



(a) (b)

Fig. 6. Weak scaling on CRAY4-CLUSTER of `pGraph` methods with 2.25M vertices and ∼3M edges per location. (a) `pView` overhead: comparison of calling `stapl::for_each` versus STL `for_each`, and of storing sources in a `pList` versus a sequential STL list; (b) Comparison of graph algorithms on graph views defined over `pGraph` and `pArray`.

insert and erase elements concurrently. The `stapl::find_sources` algorithm uses the `insert_anywhere` method of the `list_view` to populate the parallel list with source vertices. To evaluate the algorithms we perform a weak scaling experiment using a 2D sparse mesh as input, where each processor holds a stencil of 1500×1500 vertices. The number of edges per location is on average two thirds the maximum number of edges in a 2D mesh while the number of remote edges is ∼1500 (0.3%) per location.

Figure 6 shows the performance of the two algorithms on the CRAY4-CLUSTER. `stapl::for_each` is a simple do-all operation that applies a functor to every element of a view. It scales well when the number of processors is varied from 4 to 16384. `stapl::find_sources` performs a `stapl::for_each` on a `pView` over the edges of the graph, marking their targets as non source vertices. To evaluate the overhead of using views and STAPL containers, we performed the following experiments: first we compared the performance of the `stapl::for_each` using a `vertex_set_view` versus a simple STL `for_each` applied to individual elements stored inside the `pGraph`'s `bContainers`. We observe in Figure 6(a) that `stapl::for_each` has no overhead relative to the STL `for_each`. A second experiment performed was to evaluate the overhead of storing the source vertices in a `pList` through a `list_view` versus storing the vertices directly in sequential STL lists, one for each location considered. As we can see from Figure 6(a), the `pList` incurs an overhead of only 4%.

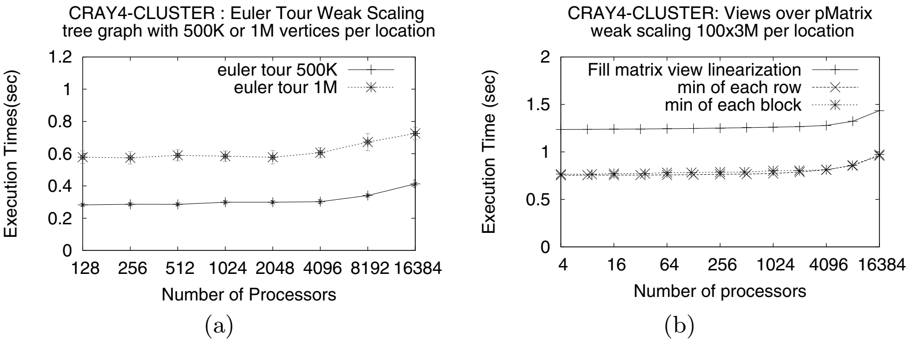


Fig. 7. Weak scaling experiments on CRAY4-CLUSTER. (a) Euler Tour computation on binary tree with 500K or 1M subtrees per processor. (b) `pMatrix` fill with random values using a linearization view and computing the minimum of each row or block.

Another important feature of using views is that new interfaces can be defined on top of existing data structures. For example, a graph view can be defined for a `pArray` of lists of edges. Generic parallel graph algorithms in STAPL will operate properly on the data stored in a `pArray`, provided a suitable graph `pView` is implemented. In Figure 6(b) we show the performance of `stapl::for_each` and `stapl::find_sources` when accessing data using a graph view defined on a `pGraph` and a graph view defined on a `pArray`. We observe that both views provide good scaling. When data is stored in the `pArray`, `stapl::for_each` is slightly faster. On the `stapl::find_sources` algorithm there is additional overhead because `stapl::find_sources` uses additional graph methods (e.g., `find_vertex`) that are more efficiently implemented in the native `pGraph`.

The Euler Tour (ET) is an important `pView` of a graph for parallel processing. In particular, the ET, which traverses every edge of the graph exactly once, corresponds to an edge view of the graph. Since the ET represents a depth-first-search traversal, when it is applied to a tree it can be used to compute a number of tree functions such as rooting a tree, postorder numbering, computing the vertex level, and computing the number of descendants [9]. The parallel Euler Tour algorithm [9] implemented in STAPL uses a STAPL `pGraph` to represent the tree and a `pList` to store the final Euler Tour. The algorithm executes parallel traversals on the `pGraph` view, generating Euler Tour segments that are stored in a temporary `pList`. Then, the segments are linked together to form the final `pList` containing the Euler Tour. The performance is evaluated by performing a weak scaling experiment on CRAY4-CLUSTER using as input a tree distributed across all locations. The tree is generated by first building a binary tree in each location and then linking the roots of these trees in a binary tree fashion. The number of remote edges is at most six for each location (one to the root and two to the children of the root in each location, with directed edges for both directions). Figure 7(a) shows the execution time on CRAY4-CLUSTER for different sizes of the tree. The running time increases with the number of vertices

```
block_partition_t     blkpart(m,n);
p_matrix_t            pmat(N1,N2,blkpart);
matrix_view_t         vmat(pmat);

// Row major linearization of the p_matrix
typedef array_1D_view<p_matrix_t,
    dom1D<size_t>,
    f1d_row_major_2d<size_t,p_matrix_index_type> >      linear_row_t;
linear_row_t lrow = vmat.linear_row();
// Fill the matrix using the linearization view

// One dimensional view over the p_matrix's rows
typedef partitioned_view<matrix_view_t,
    rows_partition<matrix_domain_t,row_domain_t>,
    map_fun_gen1<fcol_2d<size_t,matrix_dom_t::index_type> >,
    matrix_view_t::row_type>                            rows_view_t;
rows_view_t       rowsv( vmat, rows_partition_t(vmat.domain()) );

// Computing the minimum of each row
stapl::transform(rowsv, resv, stapl::min_value<int>());

// One dimensional view of blocks over the p_matrix
typedef partitioned_view<matrix_view_t,
    block_partition_t,
    map_fun_gen<f_ident<mat_view_t::index_type> > >     blocks_view_t;
blocks_view_t     blocksv(vmat,blkpart);

// Computing the minimum of each block
stapl::transform(blocksv, resv, stapl::min_value<int>());
```

Fig. 8. Snippets of code used to create different types of views over `pMatrix`: row major linearization of the matrix, partition the matrix view in rows and partition the matrix view in blocks

per location because the number of edges in the ET to be computed increases correspondingly.

Matrix views. The `pMatrix` is a `pContainer` that implements a dense, two-dimensional array [4]. We can create different types of views over a `pMatrix` to adapt the container to the algorithm requirements. For example, we can initialize the values of a container using `stapl::generate` or `stapl::copy`. Both algorithms require the data layout in a one-dimensional container. Using a mapping function to translate indices from one to two dimensions, we can define a linearization view over the `pMatrix` (e.g., `f1d_row_major_2d` in Figure 8). Similarly, we can create row and blocked `pViews` of the `pMatrix`. Figure 8 shows two of these views: a `pView` over the rows (`rows_view_t`) and a `pView` over blocks (`blocks_view_t`). They differ in the partitioner and the mapping function generator used.

Figure 7(b) shows the execution time on CRAY4-CLUSTER of three algorithms using `pMatrix`: filling the `pMatrix` using `stapl::copy` from a generator container through the row major linearization `pView`, computing the minimum element of each row and computing the minimum element of each block using `stapl::transform(input_view,output_view,functor)` algorithm, where the `functor` finds the minimum of a sequence of elements.

7 Conclusion

In this paper we have introduced the `pView` a higher level concept that allows programmers to be more expressive. Furthermore, it is a concept that hides some of the details of parallel programming. It has been assumed that programming at higher levels of abstraction inevitably reduces performance, an unwelcome side-effect in general, and in parallel programming in particular. In this paper we have shown that, at least as far the `pView` is concerned, performance does not always have to suffer. In fact, in some cases we have shown that the `pView` offers more structural and semantic information than, for example, the STL iterator, and thus enables better performance. We believe that a programming environment such as STAPL will prove to be both expressive and productive as well as high performance.

References

1. Barbosa, D.M.J., Cretin, J., Foster, N., Greenberg, M., Pierce, B.C.: Matching Lenses: Alignment and View Update. In: Proc. ACM SIGPLAN Int. Conf. on Functional Programming, Baltimore, Maryland (September 2010)
2. Bourdev, L.: Generic Image Library. Software Developer's Journal, 42–52 (2007)
3. Buss, A., Harshvardhan, Papadopoulos, I., Pearce, O., Smith, T., Tanase, G., Thomas, N., Xu, X., Bianco, M., Amato, N.M., Rauchwerger, L.: STAPL: Standard template adaptive parallel library. In: Proc. Annual Haifa Experimental Systems Conference (SYSTOR), pp. 1–10. ACM, New York (2010)
4. Buss, A., Smith, T., Tanase, G., Thomas, N., Bianco, M., Amato, N.M., Rauchwerger, L.: Design for interoperability in STAPL: `pMatrices` and linear algebra algorithms. In: Amaral, J.N. (ed.) LCPC 2008. LNCS, vol. 5335, pp. 304–315. Springer, Heidelberg (2008)
5. Callahan, D., Chamberlain, B.L., Zima, H.P.: The Cascade High Productivity Language. In: The 9th Int. Workshop on High-Level Parallel Programming Models and Supportive Environments, Los Alamitos, vol. 26, pp. 52–60 (2004)
6. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an Object-Oriented Approach to Non-Uniform Cluster Computing. In: ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications, pp. 519–538. ACM Press, New York (2005)
7. Guo, J., Bikshandi, G., Fraguela, B.B., Padua, D.: Writing Productive Stencil Codes with Overlapped Tiling. Concurr. Comput.: Pract. Exper. 21(1), 25–39 (2009)
8. Adobe Inc.: Generic Image Library, `http://opensource.adobe.com/wiki/display/gil/Generic+Image+Library`

9. JàJà, J.: An Introduction Parallel Algorithms. Addison-Wesley, Reading (1992)
10. Joyner, M., Chamberlain, B.L., Deitz, S.J.: Iterators in Chapel (April 2006)
11. Lumsdaine, A., Gregor, D., Hendrickson, B., Berry, J.W.: Challenges in Parallel Graph Processing. Parallel Processing Letters 17(1), 5–20 (2007)
12. Musser, D., Derge, G., Saini, A.: STL Tutorial and Reference Guide, 2nd edn. Addison-Wesley, Reading (2001)
13. Ottosen, T.: Range Library Proposal. Technical report, JTC1/SC22/WG21 - The C++ Standards Committee (2005), http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1871.html
14. Quinn, M.: Parallel Programming in C with MPI and OpenMP. McGraw-Hill, New York (2003)
15. Rauchwerger, L., Arzu, F., Ouchi, K.: Standard Templates Adaptive Parallel Library (STAPL). In: O'Hallaron, D.R. (ed.) LCR 1998. LNCS, vol. 1511, pp. 402–409. Springer, Heidelberg (1998)
16. Saunders, S., Rauchwerger, L.: ARMI: An Adaptive, Platform Independent Communication Library. In: Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog (PPoPP), San Diego, California, USA, pp. 230–241 (2003)
17. Seymour, J.: Views - a C++ Standard Template Library Extension (January 1996), http://www.zeta.org.au/~jon/STL/views/doc/views.html
18. Tanase, G., Buss, A., Fidel, A., Harshvardhan, P.I., Pearce, O., Smith, T., Thomas, N., Xu, X., Mourad, N., Vu, J., Bianco, M., Amato, N.M., Rauchwerger, L.: The STAPL Parallel Container Framework. In: Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog (PPoPP), San Antonio, Texas, USA (2011)
19. Tanase, G., Xu, X., Buss, A., Harshvardhan, Papadopoulos, I., Pearce, O., Smith, T., Thomas, N., Bianco, M., Amato, N.M., Rauchwerger, L.: The STAPL pList. In: Gao, G.R., Pollock, L.L., Cavazos, J., Li, X. (eds.) LCPC 2009. LNCS, vol. 5898, pp. 16–30. Springer, Heidelberg (2010)
20. Thomas, N., Tanase, G., Tkachyshyn, O., Perdue, J., Amato, N.M., Rauchwerger, L.: A framework for adaptive algorithm selection in STAPL. In: Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog (PPoPP), Chicago, IL, USA, pp. 277–288 (2005)
21. Weiser, M., Powell, G.: The View Template Library. In: 1st Workshop on C++ Template Programming, Erfurt, Germany (October 2000)

Author Index