Michael Klemm
Bronis R. de Supinski
Jannis Klinkenberg
Brandon Neth (Eds.)

LNCS 13527

OpenMP in a Modern World: From Multi-device Support to Meta Programming

18th International Workshop on OpenMP, IWOMP 2022
Chattanooga, TN, USA, September 27–30, 2022
Proceedings

IWOMP
INTERNATIONAL WORKSHOP

Springer

Lecture Notes in Computer Science 13527

More information about this series at https://link.springer.com/bookseries/558

Michael Klemm · Bronis R. de Supinski ·
Jannis Klinkenberg · Brandon Neth (Eds.)

OpenMP in a Modern World: From Multi-device Support to Meta Programming

18th International Workshop on OpenMP, IWOMP 2022
Chattanooga, TN, USA, September 27–30, 2022
Proceedings

Springer

Editors
Michael Klemm
OpenMP ARB
Beaverton, OR, USA

Jannis Klinkenberg
RWTH Aachen University
Aachen, Germany

Bronis R. de Supinski
Lawrence Livermore National Laboratory
Livermore, CA, USA

Brandon Neth
University of Arizona
Tucson, AZ, USA

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-031-15921-3 ISBN 978-3-031-15922-0 (eBook)
https://doi.org/10.1007/978-3-031-15922-0

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

The OpenMP API is a widely used application programming interface (API) for high-level parallel programming in Fortran, C, and C++. The OpenMP API has been supported in most high-performance compilers and by hardware vendors since it was introduced in 1997. Under the guidance of the OpenMP Architecture Review Board (ARB) and the diligent work of the OpenMP Language Committee, the OpenMP specification has evolved to version 5.2, which was released in November 2021. It supports parallelism at several levels: offloading in heterogeneous systems, task-based processing across processors, and vectorization in SIMD units. It also goes beyond parallel computing by supporting processor affinity and through policies and mechanisms for using memory and for matching directives and functions to computing environments.

Many of these advances were realized through major new features in version 5.0: context selectors and the declare variant construct and metadirectives that use them; the requires directive; memory allocators and support for deep copy of pointer-based data structures; acquire and release semantics; task (memory) affinity; the descriptive loop construct; reverse offloading; affinity display; and first- and third-party tools interfaces. OpenMP version 5.0 also significantly enhanced many existing features, such as implicit declare target semantics, support for task reductions, discontiguous array shaping in target updates, and imperfectly nested loop collapsing. Versions 5.1 and 5.2 refined these capabilities and augmented them for increased expressiveness and improved ease of use.

With version 5.2 of the OpenMP API specification, the OpenMP ARB undertook a great effort to regularize OpenMP directive syntax. While this effort involved deprecation of existing syntax, it makes the OpenMP API easier to understand and to apply. The new features that OpenMP API version 5.2 introduced include the ompx/omx sentinel and API prefix for OpenMP extensions; extensions to metadirectives for Fortran programs; improvements to memory allocators; and additions to the OpenMP tools interface.

While these changes are small advancements, work is already well advanced for the definition of the OpenMP API specification version 6.0. Larger changes that we anticipate for that version include the ability for threads to create tasks to be executed by threads in a different parallel team and to enable free-agent threads to execute tasks in addition to the threads explicitly created for that team. For heterogeneous programming, the OpenMP Language Committee is exploring worksharing across target devices. This volume includes a paper on this topic. Other features under consideration include support for scoping memory consistency and cross-device atomic operations; descriptive array language offload support in Fortran; and extensions of deep-copy features.

The OpenMP API remains important both as a stand-alone parallel programming model and as part of a hybrid programming model for massively parallel, distributed memory systems that consist of homogeneous manycore nodes and heterogeneous node architectures, as found in leading supercomputers. As much of the increased parallelism in the next exascale systems is expected to be within a node, OpenMP will become even more widely used in top-end systems. Importantly, the features in OpenMP versions

5.0 through 5.2 support applications on such systems in addition to facilitating portable exploitation of specific system attributes.

The first IWOMP meeting was held in 2005, in Eugene, Oregon, USA. Since then, meetings have been held each year, in Reims, France; Beijing, China; West Lafayette, USA; Dresden, Germany; Tsukuba, Japan; Chicago, USA; Rome, Italy; Canberra, Australia; Salvador, Brazil; Aachen, Germany; Nara, Japan; Stony Brook, USA; Barcelona, Spain, and Auckland, New Zealand. In 2020 and 2021, IWOMP continued the series with technical papers and tutorials presented in a virtual conference setting, due to the SARS-CoV-2 pandemic. Each workshop draws participants from research and development groups and industry throughout the world. We are delighted to continue the IWOMP series with a hybrid event hosted by University of Tennessee at Chattanooga, TN, USA. We are grateful for the generous support of sponsors that help make these meeting successful, they are cited on the conference pages (present and archived) at the IWOMP website.

The evolution of the specification would be impossible without active research in OpenMP compilers, runtime systems, tools, and environments. The many additions in the OpenMP versions 5.0 through 5.2 reflect the contribution by a vibrant and dedicated user, research, and implementation community that is committed to supporting the OpenMP API. As we move beyond the present needs, and adapt and evolve OpenMP to the expanding parallelism in new architectures, the OpenMP research community will continue to play a vital role. The papers in this volume demonstrate the use and evaluation of new features found in the OpenMP API. These papers also demonstrate the forward thinking of the research community, and highlight potential OpenMP directions and further improvements for systems on the horizon.

The IWOMP website (www.iwomp.org) has the latest workshop information, as well as links to archived events. This publication contains the proceedings of the 18th International Workshop on OpenMP, IWOMP 2022. The workshop program included eleven technical papers, six vendor updates, two keynote talks, and two tutorials related to the OpenMP API. All technical papers were peer reviewed by at least four different members of the Program Committee. The work evidenced by these authors and the committee demonstrates that the OpenMP API will remain a key technology well into the future.

September 2022

Michael Klemm
Bronis R. de Supinski
Jannis Klinkenberg
Brandon Neth

Organization

General Chair

Michael Klemm	AMD, OpenMP ARB, Germany

Program Committee Chairs

Bronis R. de Supinski	Lawrence Livermore National Laboratory, USA
Jannis Klinkenberg	RWTH Aachen University, Germany

Local Arrangements Chairs

Anthony Skjellum	University of Tennessee at Chattanooga, USA
Purushotham Bangalore	University of Alabama, USA

Tutorial Chair

Henry Jin	NASA Ames Research Center, USA

Publications and Registrations Chair

Brandon Neth	University of Arizona, USA

Steering Committee

Matthias S. Müller (Chair)	RWTH Aachen University, Germany
Eduard Ayguadé	BSC, Universitat Politècnica de Catalunya, Spain
Mark Bull	EPCC, University of Edinburgh, UK
Barbara Chapman	Hewlett Packard Enterprise, USA
Rudolf Eigenmann	University of Delaware, USA
William Gropp	University of Illinois, USA
Michael Klemm	AMD, OpenMP ARB, Germany
Kalyan Kumaran	Argonne National Laboratory, USA
Simon McIntosh-Smith	University of Bristol, UK
Dieter an Mey	RWTH Aachen University, Germany
Kent Milfeld	Texas Advanced Computing Center, USA
Brandon Neth	University of Arizona, USA
Stephen L. Olivier	Sandia National Laboratories, USA
Ruud van der Pas	Oracle, USA

Alistair Rendell	Flinders University, Australia
Mitsuhisa Sato	RIKEN R-CCS, Japan
Sanjiv Shah	Intel, USA
Oliver Sinnen	University of Auckland, New Zealand
Josemar Rodrigues de Souza	SENAI CIMATEC, Brazil
Bronis R. de Supinski	Lawrence Livermore National Laboratory, USA
Christian Terboven	RWTH Aachen University, Germany
Matthijs van Waveren	OpenMP ARB, France

Program Committee

Eduard Ayguade	Technical University of Catalunya, Spain
Mark Bull	University of Edinburgh, UK
Florina Ciorba	University of Basel, Switzerland
Sunita Chandrasekaran	University of Delaware, USA
Tom Deakin	University of Bristol, UK
Johannes Doerfert	Argonne National Laboratory, USA
Alex Duran	Intel, Spain
Deepak Eachempati	Hewlett Packard Enterprise, USA
Jini Susan George	AMD, USA
Sheikh Ghafoor	Tennessee Tech University, USA
Oscar Hernandez	NVIDIA, USA
Paul Kelly	Imperial College London, UK
Michael Kruse	Argonne National Laboratory, USA
Kelvin Li	IBM, USA
Chunhua Liao	Lawrence Livermore National Laboratory, USA
Georgios Markomanolis	CSC, Finland
Stephen Olivier	Sandia National Laboratories, USA
Swaroop Pophale	Oak Ridge National Laboratory, USA
Joachim Protze	RWTH Aachen University, Germany
Mitsuhisa Sato	RIKEN R-CCS, Japan
Tom Scogland	Lawrence Livermore National Laboratory, USA
Xavier Teruel	Barcelona Supercomputing Center, Spain
Terry Wilmarth	Intel, USA

Contents

OpenMP and Multiple Nodes

Enhancing MPI+OpenMP Task Based Applications for Heterogeneous Architectures with GPU Support 3
Manuel Ferat, Romain Pereira, Adrien Roussel, Patrick Carribault, Luiz-Angelo Steffenel, and Thierry Gautier

Towards Efficient Remote OpenMP Offloading 17
Wenbin Lu, Baodi Shan, Eric Raut, Jie Meng, Mauricio Araya-Polo, Johannes Doerfert, Abid M. Malik, and Barbara Chapman

Exploring New and Recent OpenMP Extensions

Characterizing the Performance of Task Reductions in OpenMP 5.X Implementations 35
Jan Ciesko and Stephen L. Olivier

Using Off-the-Shelf Hardware Transactional Memory to Implement Speculative While in OpenMP 50
Juan Salamanca and Alexandro Baldassin

Effective Use of Advanced Heterogeneous Node Architectures

Towards Automatic OpenMP-Aware Utilization of Fast GPU Memory 67
Delaram Talaashrafi, Marc Moreno Maza, and Johannes Doerfert

Feasibility Studies in Multi-GPU Target Offloading 81
Anton Rydahl, Mathias Gammelmark, and Sven Karlsson

Reducing OpenMP to FPGA Round-Trip Times with Predictive Modelling 94
Julian Brandner, Florian Mayer, and Michael Philippsen

OpenMP Tool Support

Improving Tool Support for Nested Parallel Regions with Introspection Consistency 111
Vladimir Indic and John Mellor-Crummey

On the Migration of OpenACC-Based Applications into OpenMP 5+ 127
Harald Servat, Giacomo Rossi, Alejandro Duran, and Ravi Narayanaswamy

OpenMP and Multiple Translation Units

Just-in-Time Compilation and Link-Time Optimization for OpenMP Target Offloading 145
Shilei Tian, Joseph Huber, John Tramm, Barbara Chapman, and Johannes Doerfert

Extending OpenMP to Support Automated Function Specialization Across Translation Units 159
Giorgis Georgakoudis, Thomas R. W. Scogland, Chunhua Liao, and Bronis R. de Supinski

Author Index 175

OpenMP and Multiple Nodes

Enhancing MPI+OpenMP Task Based Applications for Heterogeneous Architectures with GPU Support

Manuel Ferat[1], Romain Pereira[2,4,5], Adrien Roussel[3,4](✉), Patrick Carribault[3,4], Luiz-Angelo Steffenel[1], and Thierry Gautier[5]

[1] Université de Reims Champagne Ardenne, LICIIS, LRC DIGIT, 51097 Reims, France
{manuel.ferat,luiz-angelo.steffenel}@univ-reims.fr

[2] CEA, DAM, DIF, 91297 Arpajon, France
romain.pereira@cea.fr

[3] CEA, DAM, DIF, LRC DIGIT, 91297 Arpajon, France
{adrien.roussel,patrick.carribault}@cea.fr

[4] Université Paris-Saclay, CEA, Laboratoire en Informatique Haute Performance pour le Calcul et la simulation, 91680 Bruyères-le-Châtel, France

[5] Project Team AVALON INRIA, LIP, ENS-Lyon, Lyon, France
thierry.gautier@inrialpes.fr

Abstract. Heterogeneous supercomputers are widespread over HPC systems and programming efficient applications on these architectures is a challenge. Task-based programming models are a promising way to tackle this challenge. Since OpenMP 4.0 and 4.5, the `target` directives enable to offload pieces of code to GPUs and to express it as tasks with dependencies. Therefore, heterogeneous machines can be programmed using MPI+OpenMP(task+target) to exhibit a very high level of concurrent asynchronous operations for which data transfers, kernel executions, communications and CPU computations can be overlapped. Hence, it is possible to suspend tasks performing these asynchronous operations on the CPUs and to overlap their completion with another task execution. Suspended tasks can resume once the associated asynchronous event is completed in an opportunistic way at every scheduling point. We have integrated this feature into the MPC framework and validated it on a *AXPY* microbenchmark and evaluated on a MPI+OpenMP(tasks) implementation of the LULESH proxy applications. The results show that we are able to improve asynchronism and the overall HPC performance, allowing applications to benefit from asynchronous execution on heterogeneous machines.

Keywords: OpenMP · GPU Computing · Distributed Application · Task programming

M. Klemm et al. (Eds.): IWOMP 2022, LNCS 13527, pp. 3–16, 2022.
https://doi.org/10.1007/978-3-031-15922-0_1

1 Introduction

Supercomputers race to the Exascale by significantly increasing the number of computing units per node and diversifying the types of computing resources with accelerators (e.g., GPU). Numerical simulation applications need to handle computational complexity to remain efficient, but it is burdensome. Thus, parallel programming models require some evolutions to follow this trend. They attempt to leverage these hardware features in an almost transparent way while minimizing the stress put on application developers. Fine-grain parallelism with synchronization reductions is a solution to take advantage of many-core compute nodes.

The task programming model is thus an excellent alternative to the fork-join model. A task represents a piece of code and dependencies can be applied between tasks to represent partial order execution. Hence, a Task Dependency Graph (TDG) depicts the parallel section. Once a task is created, it can be executed instantly or deferred: programming makes asynchronous execution easier. Asynchronism is essential for the performance of applications designed for supercomputers in order to maintain a high level of parallelism and not to lose efficiency. A task-based programming model with dependencies has been added in OpenMP 4.0 [7] and has continued to grow in importance since then. In addition, supercomputers are composed of many compute nodes linked through a high-speed network. When coupling with MPI, the *de facto* standard used for distributed environment, the task programming model favors asynchronism to overlap communication with computation in distributed software [20–22].

OpenMP 4.0 introduces the `target` directives [18] to offload pieces of code to computing accelerators like GPUs. Since the advent of OpenMP 4.5 [19], `target` directives are explicit tasks with dependencies. This addition allows developers to write their entire application with a task-based approach homogeneously for heterogeneous architectures. Task representation may help overlapping data transfers with computation and then favor asynchronous execution between the host and the device.

Programming efficient codes exploiting all the parallelism provided by heterogeneous supercomputers remains challenging. While MPI+OpenMP programming model with tasks appears to be a good candidate for composing such applications, it still requires some effort at the runtime level to be efficient. Runtime stacking remains challenging, and the close collaboration between all the components is not granted. MPC [5] is a parallel framework that proposes an OpenMP implementation relying on a user-level thread scheduler. Recent work demonstrates that MPC can take advantage of interoperability between MPI and OpenMP(tasks) runtimes to enhance overall application performances [20]. In this paper, we propose the addition of GPU task support with the `target` directives in MPI+OpenMP(tasks) applications to enhance asynchronous executions thanks to user-space mechanisms. We have implemented this approach in the MPC framework and evaluated it on the LULESH mini application [12].

The contributions of this paper are the followings:

- integration of `target` directives in the MPC-OpenMP task based programming model,

- addition of user-space scheduling mechanisms to enhance asynchronous GPU executions,
- porting and evaluating MPI+OpenMP(tasks) LULESH mini application for heterogeneous architectures.

The paper is organized as follows. Section 2 presents related work in task-based programming models for distributed heterogeneous applications. Section 3 details the port of OpenMP `target` directives and its integration into the MPC framework to enhance asynchronism thanks to the GPU task suspension support. Section 4 sketches the porting of MPI+OpenMP(tasks) with GPU support for some applications. Section 4.2 presents the performance evaluation of our work. Finally, Sect. 5 concludes this work and depicts future work.

2 Related Work

Heterogeneous programming gained importance as GPU computing accelerators are widespread in HPC systems. Tasked-based runtime systems have shown that task programming is well designed for the composition of heterogeneous applications. StarPU [1], X-Kaapi [8] and OmpSs [2] enable developers to compose an application with various types of tasks that can be performed both on the GPU and the CPU. These runtime systems can overlap data transfers with computation through data prefetching techniques thanks to CUDA streams. Legion [3, 9] is also a task-based runtime designed for distributed machines with heterogeneous nodes. More recently, CudaGraph [14] enables developers to write or capture GPU operations only and organizes them into graphs to reduce kernel launch overheads of CUDA. OpenMP [4, 18] target constructs have been introduced in the specification version 4.0. OpenMP 4.5 [19] defines target constructions as tasks, and task programming seems to be growing for the future OpenMP 6.0. OpenMP standard guarantees the perpetuation of simulation codes by integrating heterogeneous task programming [24].

Offloading data and kernels on GPU are blocking operations by default but can be transformed into asynchronous ones to gain parallel efficiency. In the same way that MPI communications can block the execution of the tasks [23], synchronous GPU operations may degrade the performances. In 2015, MPI+ULT (User Level Thread, Argobots) [16] proposed to run MPI code within a user-level thread and make it yield whenever an MPI communication blocks. MPI Detach [21] advocates programming through continuations. It enables the addition of asynchronous callbacks on communication completion but implies heavy code restructuration. TAMPI [22] and MPC [20] propose to transform synchronous to asynchronous MPI calls finely nested into OpenMP tasks. These works tackle the interoperability issue between MPI and OpenMP tasks but do not consider heterogeneous applications. As OpenMP target constructs are now explicit tasks, these approaches may be applied to GPU task programming.

In [25], several approaches are presented to overlap GPU operations with computations thanks to OpenMP target constructions. They proposed to run asynchronous target tasks within dedicated threads, which are preempted by blocking operations. These threads are called Hidden Helper Threads (HTT) and are implemented as *kernel* threads, hence delegating scheduling decisions to the operating system. Standardization of this approach was suggested in [15]. This paper presents a different design for asynchronous GPU calls overlapping through user-space cooperative scheduling. This approach targets efficient MPI+OpenMP(tasks) applications for heterogeneous supercomputers.

3 Targets with Asynchronous Tasks

The main goal of asynchronous offloading is to overlap accelerator operations with useful work on the CPUs. With the task programming model, this is achieved through task switching during asynchronous operation progression.

3.1 Cooperative Target Task Design

The general idea of the Cooperative Target Task Design is to have the target tasks make asynchronous calls and explicitly preempt right before synchronization points without the operating system scheduler intervening. Polling functions ensure asynchronous events progression. On completion, the preempted task is unblocked and may resume. This design strictly distinguishes *threads* and *tasks* as depicted in the Fig. 1.

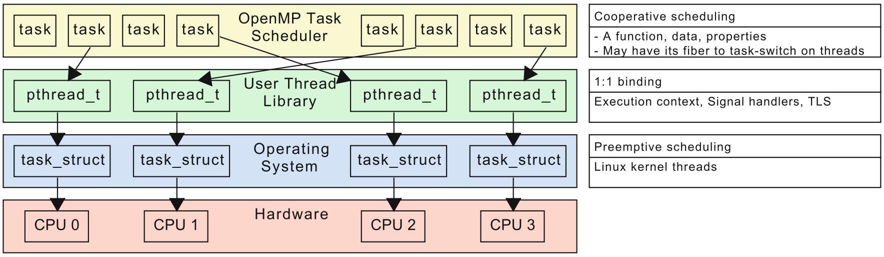


Fig. 1. Fiber task design : from physical core to user-space tasks

Threads are standard OpenMP threads represented as `pthreads` and mapped 1:1 with kernel threads and physical cores. They have their own user and kernel space execution *context* made of a stack and registers copy. They also have their own signal handlers, file descriptor table, and Thread Local Storage (TLS).

Tasks are standard OpenMP tasks made of a function, some data, and properties. In addition, they may also have their own user-space execution context known as a *fiber* [13]. A task can run on top of a thread on its own fiber if it has one or directly on the thread fiber otherwise. With the fiber capability, a task can pause itself explicitly at any time of its execution and may resume on any threads (if untied) once it unblocks. Yet, asynchronous events still have to be polled at some point. In our approach, this is done opportunistically on every scheduling point defined in the OpenMP Specification.

3.2 OpenMP Target in MPC

Multi Processor Computing (MPC) [5] implements its own OpenMP runtime through LLVM and GCC Application Binary Interface (ABI). It also improves standard OpenMP task capabilities with fibers through Linux `<ucontext.h>` interface for cooperative scheduling. Previous work has been done to finely nest MPI communications into OpenMP(task) [20]. This suspend/resume task mechanism allows generating asynchronism that can be usefully exploited by an application to overlap MPI communications with computations. Implementing OpenMP targets with the introduced design now only becomes a question of Application Binary Interface (ABI) support and tasks cooperativity.

Implementation. To add target directive support to MPC, we chose to use the LLVM *libomptarget* library because it is built as a sub-module of the LLVM OpenMP project. The OpenMP Target part is slightly coupled with the rest of the project, which will ease its integration with another OpenMP implementation such as MPC-OpenMP.

MPC implements the LLVM ABI which makes possible to compile with *clang* and to link with both the MPC-OpenMP runtime and the LLVM `libomptarget` library. To make this combination functional, we had to modify several entry points in the MPC-OpenMP runtime. First, when starting a program using LLVM's `libomptarget`, even before the main execution, the `libomptarget` library is initialized through `__kmpc_get_target_offload` ABI which was implemented in the MPC-OpenMP runtime. It reads the `OMP_TARGET_OFFLOAD` environment variable from the OpenMP runtime and passes it to `libomptarget`. Then, the device's specific libraries are loaded. We also had to implement the support of the `omp_get_default_device` and `omp_is_initial_device` OpenMP API in MPC-OpenMP.

Finally, to enable the expression of target directives as tasks with dependencies, we added the support of the `__kmpc_omp_target_task_alloc` ABI. This creates a task that encapsulates the incoming target region, generating a standard and opaque task for the OpenMP runtime. The internal task function points to an entry point in the `libomptarget` responsible of starting and completing the asynchronous GPU operation. The completion of GPU operations implies synchronizations that end up blocking threads. Hence, the LLVM OpenMP runtime executes asynchronous target tasks on dedicated Hidden Helper Threads (HHT) [25] implemented as *kernel* threads. Thus, the operating system can preempt threads blocking on GPU operations, and Standard OpenMP threads can be rescheduled onto physical cores to progress other tasks in parallel.

In MPC-OpenMP, we decided to implement the Cooperative Target Task design instead of the HHT design for its user-space scheduling flexibility. Still, the target tasks end up synchronizing at some points blocking threads and cores.

Enabling Asynchronism Through Cooperativity. In practice, synchronizations happens at the end of target execution with a CUDA stream synchronization within the `libomptarget`. Without cooperativity, the target tasks ends up retaining the physical core. To tackle this issue, we patched the LLVM `libomptarget` CUDA Runtime Library (RTL)[1] to modify the implementation of `DeviceRTLTy.synchronize`. It now inserts a CUDA Stream progression polling function into MPC relying on `cudaStreamQuery` and explicitly yields until the progression is completed. Moreover, programmers can also decide whether target tasks should have their own fibers and mark them as `untied`, so that they may resume on any thread at any scheduling point. Hence, MPC threads will not block cores and the thread will overlap the stream progression with useful computation through pure user-space task switches. This approach enables fully asynchronous support of OpenMP targets in MPC-OpenMP.

3.3 State-of-the-Art Comparison

Microbenchmark. In order to compare our approach with existing solutions, we extended the microbenchmarks from [25]. The Listing 1.2 depicts our microbenchmark B5. The x and y vectors are of size $n.T$ with $(n, T) \in \mathbb{N}^2$. A single thread produces $T \in \mathbb{N}$ host-to-device data transfers (line 5), computation kernels (line 7) and device-to-host data transfers (line 10). These operations are not grouped in one construction to ensure the most asynchronism. The target tasks triplets are ordered with dependencies, and up to T triplets can be consumed concurrently by any threads. Each data transfers complexity is $O(n)$ bytes while computation kernels time complexity is $O(n^2)$ as shown in the Listing 1.1 and 1.2.

Listing 1.1. daxpy-like function

```
# define daxpy(A, X, Y, I0, IF)             \
    for (uint64_t I = I0 ; I < IF ; ++I)    \
        for (uint64_t J = 0 ; J <= I ; ++J) \
            Y[I] = Y[I] + A * X[J];
```

Listing 1.2. Target microbenchmark B5

```
 1  void B5(double a, double * x, double * y, int T, int n) {
 2      # pragma omp parallel
 3      # pragma omp single
 4      for (int t = 0 ; t < T ; ++t) {
 5          # pragma omp target update to(y[t*n:n]) nowait depend(inout:
                y[t*n])
 6
 7          # pragma omp target teams distribute parallel for nowait
                depend(inout: y[t*n])
 8          daxpy(a, x, y, t*n, n);
 9
10          # pragma omp target update from(y[t*n:n]) nowait depend(inout:
                y[t*n])
11      }
12  }
```

[1] https://gitlab.inria.fr/ropereir/iwomp2022.

Environment. Our experiments run onto nodes made of two AMD EPYC 7H12 64-core processors. Each processor has 4 NUMA domains with 32 GB memory and 16 cores. Moreover, each processor has 2 Nvidia A100 GPUs connected to their NUMA domain 0 and 2. We use a compact threads pinning on NUMA domains 0 or 2 and we only use one GPU at a time. We set $T = 64$ and vary n in $\{2^4, 2^5, ..., 2^{16}\}$. Each point is the median of 5 runs, and the error bars represent extremums. We used LLVM release 14.x suite, Nvidia/PGI 22.2 suite and MPC-OpenMP runtime with LLVM release 14.x compiler and the patched libomptarget. Every software was compiled with $O3$ optimizations enabled.

Results. Each run was wrapped into NVIDIA Nsight Systems tracing and each runtime showed a similar execution time on the compute kernel and the two data transfers, which means that the observed performance differences mostly come from task scheduling. Figure 2 depicts the performances varying the number of threads. For each runtime, the number of threads corresponds to the parallel region, but for LLVM, it also corresponds to the number of Hidden Helper Threads (HHT) using `LIBOMP_NUM_HIDDEN_HELPER_THREADS` environment variable. The performances are represented as the speedup relative to LLVM with 8 OpenMP threads and 8 HHT. The best performance reaches about 625 GFlop/s for $n = 2^{16}$ with both LLVM using 16 threads and MPC using 4 threads.

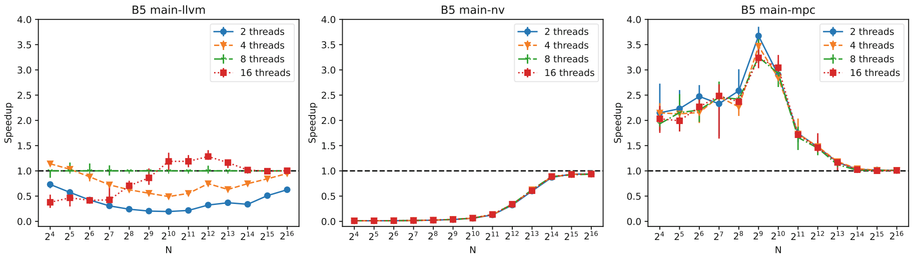


Fig. 2. OpenMP runtimes speedup relative to LLVM with 8 hidden helper threads

LLVM. We observe that the number of HHT can significantly impact performances regarding the state of art performances. While the 8-threads default configuration seems a reasonable compromise on average. We still observed up to 1.14 speedup on fine-grain using 4 threads and 1.3 speedup using 16 threads on medium grain.

Nvidia/PGI. Our installation is low-performing on fine-grain offloading (1% to 5% of LLVM performances). For coarser grains, performances improve but are still about 8% slower. We also observe that the number of threads does not impact performances. CUDA driver calls are only performed on a single thread, making the GPU underloaded on the fine-grain configurations.

MPC. Cooperative task scheduling on target tasks significantly improve performances on fine-grain offloading. The performances are converging to LLVM-OpenMP for coarser grains, likely because the GPU cores are becoming overloaded. Another benefit of the user-space scheduling approach is the small performance variation with the number of threads. As LLVM results show, performances vary by a factor up to 6 depending on the number of threads, on $n = 2^{10}$ between 2 and 16 threads. With MPC, this variation is maximum between 2 and 16 threads on $n = 2^9$ with a factor of 1.2.

Conclusion. This microbenchmark shows that our cooperative target tasks approach efficiently schedules asynchronous GPU operations. Moreover, it relieves the burden of hidden helper thread configuration from users - which can significantly impact performances on LLVM - by delegating the scheduling decisions from the operating system to the OpenMP task scheduler. Now comes the question of performance on real-world applications.

4 Heterogeneous and Hybrid Task-Based LULESH

Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) is a proxy application representing the core of hydrodynamics in real-world applications. It has been widely studied under different programming models but not under fully asynchronous, hybrid, and heterogeneous task programming. Hence, we decided to port this application to MPI+OpenMP(tasks+target) standards and evaluate our scheduling approach.

4.1 Porting

Tasking. Original application `parallel for` loops were transformed to tasks generating loops with dependencies in a single-producer/multi-consumer scheme. Thus, we defined the parameter `-te` as the number of tasks decomposing a loop.

Heterogeneity. We separate computation offloading through the target construct from data transfers between host to device with the target enter/exit data directives. Order of execution is guaranteed through data dependencies which allows independant operations to operate concurrently. In practice, we offloaded the first loop of `IntegrateStressForElems` and the `CalcKinematicsForElems` loops because they depend on host-to-device data transfers that can be overlapped with other CPU computation such as `CalcAccelerationForNodes` loops for instance. Moreover, we also switch the standard `malloc` memory allocator to `cudaMallocHost` to allocate pinned memory. Otherwise, the device to host memory transfers cannot be overlapped with computation on the CPUs.[2] To adjust GPU tasks granularity, GPU tasks were made "super-tasks" merging several original CPU tasks. The `-st` parameter controls the number of original CPU tasks composing a single GPU super-task.

[2] https://docs.nvidia.com/cuda/cuda-driver-api/api-sync-behavior.html.

Hybridation. MPI communications were finely nested within OpenMP tasks and ordered with dependencies to support distributed computing. We are using the MPC-OpenMP interoperability layer presented in [20] to automatically overlap blocking calls through user-space task switches. Hence, tasks with MPI calls can be inserted into the Task Dependency Graph (TDG), just like regular tasks.

Optimizations. We backported the *global allocation of temporary work arrays* optimization proposed in [10] in the baseline version and our task-based version. This optimization consists of preallocating reusable memory buffers instead of reallocating them on every iteration in the original code.

Representativity, Correctness, Standard. Original authors provide guidelines on porting to keep the code representative of ALE3D. Moreover, authors also provide minimal cases results to ensure the correctness of various studies [11,12] When porting the application, we fully respected those guidelines and ensure the correctness of our approach. Moreover, the source code of the application and our MPC-OpenMP runtime are available online.[3] Note that our version is not fully standard compliant. In particular, we are using some specific MPC-OpenMP runtime call to bypass restrictions on `depobj` and `iterators`, and the lack of support for the `inoutset` dependency type by compilers.

4.2 Evaluation

The upcoming experiments aims to demonstrate that the hybrid and heterogeneous task-based version of LULESH manage to overlap GPU/MPI asynchronous operations with computation within a unified OpenMP task scheduler.

Experimental Setup. In this section, we always run 5 times the application and presents the median and extremum values. We are using the same AMD+A100 nodes positioning MPI ranks per 16-core NUMA domain with 1 GPU. We run our experiments with the same software stack presented in the Sect. 3.3. The tasks versions always run with MPC-OpenMP and the patched `libomptarget` presented in the Sect. 3.2. The problem dimension was set to `-s=264` which fills 80% of the NUMA domain memory capacity and 20% of the GPU memory capacity. The parameters `-te` and `-st` represents the loop cutting into tasks and must be evaluated for fine tuning.

Loop Cutting into Tasks. Our first experiment consisted in determining the best value for `-te` on the given problem dimension. We fixed `-st=1` and disable GPU offloading, so that every tasks run onto CPUs. We varied the parameter `-te` in $[8, 2048]$ on single rank runs and `-i=32` iterations. This parameter changes both the parallelism and the tasks grain: the more tasks, the more parallelism,

[3] https://gitlab.inria.fr/ropereir/iwomp2022.

but the finer the grain. The best performances were found for `-te=1280` so this is the value we used in next experiments.

In a second phase, we enabled GPU offloading and varied `-st` in $[1, 1280]$. We compared the performance results obtained for each task cutting with the OpenMP `parallel-for` baseline version and our task-based version disabling GPU offloading. The results are depicted in the Fig. 3 and show that the lowest execution time for `-st=4` and `-st=8`. The right side is a zoom of the left side to see the lowest execution time better. First, we observe that the task version

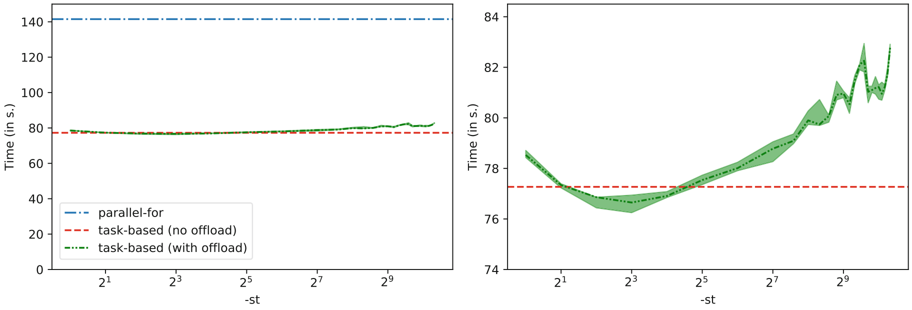


Fig. 3. Task merging into super-tasks study

significantly improves performances over the reference parallel-for version. After investigating with `perf` [6], we found out that this important performance gain comes from the data cache reuse. We are using a LIFO scheduling strategy that favors the execution of the tasks direct successors. This strategy ends up following the data movements, which significantly reduce cache misses and the process pipeline stalls, improving the number of instructions per cycle from 0.52 to 0.90. This phenomenon is called *work time inflation* and was already shown in [17]. Regarding the GPU offloading, we measured the best performances with `-st=4`. We do not observe a significant performance gain with the GPU offloading and tasks fibers enabled even with this finely tuned parameter. We have looked at some possible explanations.

Overlapping and Scheduling. The Fig. 4 depicts the Gantt chart on an instance of execution for the same problem size. The highlighted task is a GPU kernel from the `CalcKinematicsForElem` loop. As you can see, the target task starts

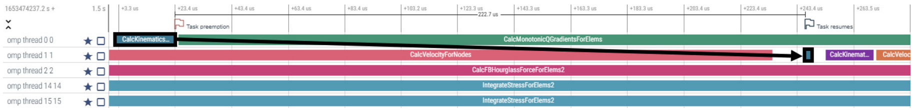


Fig. 4. LULESH CalcKinematicsForElem offloading overlap

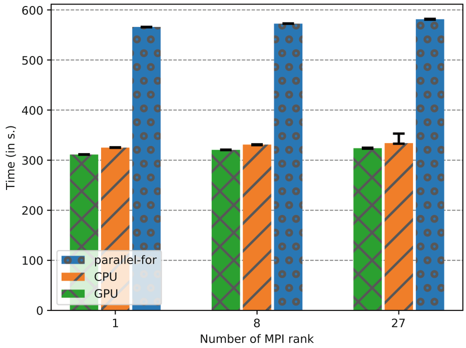
and blocks on thread 0 to resume later on thread 1. Thread 0 overlaps the kernel execution by switching to a `CalcMonotonicQGradientsForElems` task. While the overlapping mechanism is working as expected, we found a few cases where the threads have no more ready tasks on the CPU-side to overlap asynchronous operations leading to short idle periods. Yet, we measured less than 1% of idle time overall, which removes this hypothesis to explain the small performance gain.

Workload, Arithmetic Intensity, Data Transfers. Using MPC tracing, we measured that the offloaded loops represent 15% of the overall work when running on CPUs. This limits the amount of work available to the GPU, which ends up being underused. Using NVIDIA Nsight System tracing, we measured 0.7 s. Spent by the GPU in the computation kernels and 18.9 s. Spent on the data transfers. This also shows the short *arithmetic intensity* of the offloaded kernel. This may perturbate the task execution on CPUs leading to *work time inflation.* Still, the host and device memory transfers are mostly overlapped, as depicted in the Fig. 4 so we would have expected higher performances.

Work Time Inflation. NVIDIA tools reported 512,000 data transfers of 0.38 MB on average that our OpenMP runtime blindly schedules as normal tasks. We decided to measure the impact of those many data transfers on CPU performances, by running the task with GPU offloading under `perf`. We measured that the offloading version has 14% fewer instructions than the pure-CPU version - which corresponds to the 15% CPU work offloaded to GPU we measured previously with MPC. But more importantly, `perf` shows that the instruction/-cycle drops from 0.90 to 0.62 and MPC shows a +12% work time inflation on the CPU tasks when enabling the offloading. In other words, the same tasks running on CPUs take 12% more time when we offload some work to the GPU. We do not yet have an exact explanation for this.

Conclusion. To conclude this preliminary experiments, setting `-te=1280` and `-st=4` enables efficient loop cutting into tasks for the given problem size. Asynchronous tasks execution between host and device increases the performance of the LULESH application. However, this gain is compensated by the work time inflation on CPU tasks and would need more investigation.

Weak-Scaling. This last experiment is a weak-scaling from 1 to 27 processes distributed over 7 nodes. We are running the same problem but increasing the number of iterations from 32 to 128. We used Open MPI 4.0.5 with MPC-OpenMP interoperability library. Every version of the application scales very well to 27 MPI ranks. Compared to the baseline version, both task versions show a significant speedup, which comes from better cache reuse, as shown in the previous experiment. We also observe a slight performance gain when offloading work to the GPU. This result shows that (almost) standard OpenMP asynchronous hybrid and heterogeneous task-based programming model enables scalable applications (Fig. 5).



	Median time (in s.)		
ranks	parallel-for	no offloading	offloading
1	565.84	325.02	309.14
8	572.45	331.09	317.91
27	581.26	333.98	318.83

Fig. 5. LULESH weak-scaling from 1 to 27 GPUs

5 Conclusion and Perspectives

Heterogeneous supercomputers achieve high performance but add a new level of asynchronicity. It makes the programming of efficient HPC applications challenging. OpenMP task-based programming is promising to compose every parallelism level. However, the overlapping of data transfers and GPU offloading with computations on CPUs are only possible if task suspension is efficiently supported. In this paper, we propose a user-space cooperative target task design to enable the execution of asynchronous and heterogeneous applications on distributed machines.

We have integrated this mechanism into the MPC framework and the LLVM `libomptarget` for the GPU offloading support. We show that the Cooperative Target Task design offers more flexibility to the task scheduler than the Hidden Helper Threads (HHT) state-of-art design which delegates scheduling decisions to the operating system. We also have ported the LULESH proxy applications to the MPI+OpenMP(tasks) programming standards for heterogeneous and distributed architectures. The results show that performances can benefit from asynchronous executions. However, the gains observed are hidden by work time inflation of CPU tasks, and, in future works, it requires more investigation to understand this behavior.

Currently, we only support the asynchronous target task execution on NVIDIA GPUs. As a perspective, we plan to add support for other GPU vendors (AMD, Intel). It will likely be through patching other `libomptarget` RTL plugin implementations to support asynchronous offloading in a portable way. In this paper, the asynchronous target task design relies on a few non-standard runtime capabilities. In particular, MPC-OpenMP tasks can run onto their own *fiber*, block/unblock explicitly and resume on any threads. The current standard `taskyield` directive and `untied` clause does not give such guarantees to the programmer.

Acknowledgements. Present results have been obtained within the frame of DIGIT, Contractual Research Laboratory between CEA, CEA/DAM-Île de France center and the University of Reims Champagne Ardenne.

References

1. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurr. Comput. Pract. Exp. **23**(2), 187–198 (2011). https://doi.org/10.1002/cpe.1631
2. Ayguadé, E., Badia, R.M., Igual, F.D., Labarta, J., Mayo, R., Quintana-Ortí, E.S.: An extension of the StarSs programming model for platforms with multiple GPUs. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 851–862. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03869-3_79
3. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: expressing locality and independence with logical regions. In: SC 2012: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp. 1–11 (2012). https://doi.org/10.1109/SC.2012.71
4. Beyer, J.C., Stotzer, E.J., Hart, A., de Supinski, B.R.: OpenMP for accelerators. In: Chapman, B.M., Gropp, W.D., Kumaran, K., Müller, M.S. (eds.) IWOMP 2011. LNCS, vol. 6665, pp. 108–121. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21487-5_9
5. Carribault, P., Pérache, M., Jourdren, H.: Enabling low-overhead hybrid MPI/OpenMP parallelism with MPC. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., de Supinski, B.R. (eds.) IWOMP 2010. LNCS, vol. 6132, pp. 1–14. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13217-9_1
6. De Melo, A.C.: The new Linux 'perf' tools. In: Slides from Linux Kongress, vol. 18, pp. 1–42 (2010)
7. Duran, A., Ferrer, R., Ayguadé, E., Badia, R.M., Labarta, J.: A proposal to extend the OpenMP tasking model with dependent tasks. Int. J. Parallel Program. **37**(3), 292–305 (2009). https://doi.org/10.1007/s10766-009-0101-1
8. Gautier, T., Lementec, F., Faucher, V., Raffin, B.: X-kaapi: a multi paradigm runtime for multicore architectures. In: Workshop P2S2 in Conjunction of ICPP, Lyon, France, p. 16, October 2013. https://hal.inria.fr/hal-00727827
9. Jia, Z., Kwon, Y., Shipman, G., McCormick, P., Erez, M., Aiken, A.: A distributed multi-GPU system for fast graph processing. Proc. VLDB Endow. **11**(3), 297–310 (2017). https://doi.org/10.14778/3157794.3157799
10. Karlin, I., McGraw, J., Keasler, J., Still, B.: Tuning the LULESH Mini-App for Current and Future Hardware (2013). https://www.osti.gov/biblio/1070167
11. Karlin, I.: LULESH Programming Model and Performance Ports Overview (2012)
12. Karlin, I., Keasler, J., Neely, R.: LULESH 2.0 updates and changes. Technical report LLNL-TR-641973, August 2013
13. Kowalke, O.: Distinguishing coroutines and fibers (2014)
14. Lin, D.-L., Huang, T.-W.: Efficient GPU computation using task graph parallelism. In: Sousa, L., Roma, N., Tomás, P. (eds.) Euro-Par 2021. LNCS, vol. 12820, pp. 435–450. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85665-6_27
15. Lopez, V., Criado, J., Peñacoba, R., Ferrer, R., Teruel, X., Garcia-Gasulla, M.: An OpenMP free agent threads implementation. In: McIntosh-Smith, S., de Supinski, B.R., Klinkenberg, J. (eds.) IWOMP 2021. LNCS, vol. 12870, pp. 211–225. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85262-7_15

16. Lu, H., Seo, S., Balaji, P.: MPI+ULT: overlapping communication and computation with user-level threads, pp. 444–454, August 2015. https://doi.org/10.1109/HPCC-CSS-ICESS.2015.82
17. Olivier, S.L., de Supinski, B.R., Schulz, M., Prins, J.F.: Characterizing and mitigating work time inflation in task parallel programs. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, Washington, DC, USA. IEEE Computer Society Press (2012)
18. OpenMP Architecture Review Board: OpenMP Application Program Interface Version 4.0 (2013). http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf
19. OpenMP Architecture Review Board: OpenMP Application Program Interface Version 4.5 (2015). http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
20. Pereira, R., Roussel, A., Carribault, P., Gautier, T.: Communication-aware task scheduling strategy in hybrid MPI+OpenMP applications. In: McIntosh-Smith, S., de Supinski, B.R., Klinkenberg, J. (eds.) IWOMP 2021. LNCS, vol. 12870, pp. 197–210. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85262-7_14
21. Protze, J., Hermanns, M.A., Demiralp, A., Müller, M.S., Kuhlen, T.: MPI detach - asynchronous local completion. In: 27th European MPI Users' Group Meeting, EuroMPI/USA 2020, pp. 71–80. Association for Computing Machinery, New York (2020). https://doi.org/10.1145/3416315.3416323
22. Sala, K., Teruel, X., Pérez, J., Peña, A., Beltran, V., Labarta, J.: Integrating blocking and non-blocking MPI primitives with task-based programming models. Parallel Comput. **85**, 153–166 (2019). https://doi.org/10.1016/j.parco.2018.12.008
23. Schuchart, J., Tsugane, K., Gracia, J., Sato, M.: The impact of Taskyield on the design of tasks communicating through MPI. In: de Supinski, B.R., Valero-Lara, P., Martorell, X., Mateo Bellido, S., Labarta, J. (eds.) IWOMP 2018. LNCS, vol. 11128, pp. 3–17. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98521-3_1
24. de Supinski, B.R., et al.: The ongoing evolution of OpenMP. Proc. IEEE **106**(11), 2004–2019 (2018). https://doi.org/10.1109/JPROC.2018.2853600
25. Tian, S., Doerfert, J., Chapman, B.: Concurrent execution of deferred OpenMP target tasks with hidden helper threads. In: Chapman, B., Moreira, J. (eds.) LCPC 2020. LNTCS, vol. 13149, pp. 41–56. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-95953-1_4

Towards Efficient Remote OpenMP Offloading

Wenbin Lu[1](✉), Baodi Shan[1], Eric Raut[1], Jie Meng[2],
Mauricio Araya-Polo[2], Johannes Doerfert[3], Abid M. Malik[4],
and Barbara Chapman[1,4]

[1] Stony Brook University, Stony Brook, NY 11794, USA
{wenbin.lu,baodi.shan,eric.raut,barbara.chapman}@stonybrook.edu
[2] TotalEnergies EP R&T, Houston, TX 77002, USA
[3] Argonne National Laboratory, Lemont, IL 60439, USA
jdoerfert@anl.gov
[4] Brookhaven National Laboratory, Upton, NY 11793, USA
amalik@bnl.gov

Abstract. On modern heterogeneous HPC systems, the most popular way to realize distributed computation is the hybrid programming model of MPI+X (X being OpenMP/CUDA/etc.), as it has been proven to perform well with various scientific applications. However, application developers prefer to use a single coherent programming model over a hybrid model, as maintainability and portability decrease per additional model. Recent work [14] has shown that the OpenMP device offloading model could be used to program distributed accelerator-based HPC systems with minimal changes to the application.

In this paper, we improve the performance of OpenMP remote offloading through various runtime optimizations, guided by a detailed overhead analysis. Evaluation of our work is conducted using an industrial-level seismic modeling code, *Minimod*, as well as two proxy-apps, *XSBench* and *RSBench*. Results show that, compared to the baseline version, our optimizations can reduce offloading latencies by up to 92%, and raise application parallel efficiency by at least 25.2% when running with 16 GPUs. We then point out why strong scaling is still difficult with OpenMP remote offloading, and propose further improvements to the runtime to increase scalability.

Keywords: OpenMP · GPGPU · distributed computing

1 Introduction

As we move towards extreme heterogeneity, it is increasingly important to utilize HPC accelerators like GPUs efficiently in the distributed setting. Also, the great variety in accelerator software/hardware makes the portability and maintainability of applications as important as reducing the time to solution.

M. Klemm et al. (Eds.): IWOMP 2022, LNCS 13527, pp. 17–31, 2022.
https://doi.org/10.1007/978-3-031-15922-0_2

As many of the distributed programming models were designed with only the CPUs as their primary processing elements, users often have to add additional layers of domain decomposition and use vendor-specific APIs to take advantage of the accelerators. For code initially written in intra-node programming models, they often require even more effort to port to a hybrid programming model that can run across multiple nodes. This increases the development burden and poses great portability challenges.

OpenMP [13] is the *de facto* HPC programming model for shared-memory parallelism. Version 4.0 of OpenMP introduced device offloading to execute code on accelerators, without the user having to write device kernels in vendor-specific APIs. Recent work by Patel and Doerfert [14] has shown that through extensions in the LLVM/OpenMP runtime system, specification-conforming OpenMP offloading applications can seamlessly utilize accelerators attached to remote compute nodes. This shows the potential for transforming OpenMP to an all-encompassing programming model for writing performance portable and maintainable scientific application in the era of heterogeneous supercomputing, as an alternative to hybrid programming models like MPI+X.

In this paper, we optimize the previous work and push OpenMP remote offloading performance further. We then point out limitations that are deeply rooted in the design of the LLVM/OpenMP runtime, in the hope to stir up discussions about remote offloading in the OpenMP community, and inspire future OpenMP runtime designs. Our main contributions are the following:

- A detailed overhead analysis of the OpenMP remote offloading plugin.
- A series of runtime optimizations that significantly reduce the overhead of the remote offloading process.
- An evaluation of the optimized plugin using proxy-apps of real-world HPC applications.
- A discussion of the limitations of the plugin, as well as proposals for further improvements.

The paper is organized as follows: Sect. 2 describes LLVM/OpenMP remote offloading and the related work, Sect. 3 identifies performance issues and present the corresponding optimizations, evaluation of our work is described in Sect. 4, and finally Sect. 5 concludes the work and talk about future directions.

2 Background

In this section, we describe the details of how OpenMP remote offloading works in LLVM, as well as related work about the technologies and cases used later in the experimental section.

2.1 OpenMP Offloading in LLVM/OpenMP

An OpenMP application's offloading directives are lowered to functions calls into the Host Runtime (`libomptarget.so`), which is the entry point of all offloading

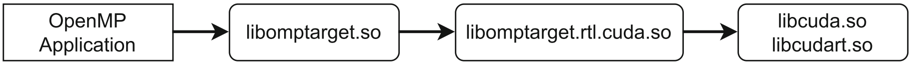


Fig. 1. LLVM/OpenMP device offloading workflow for CUDA devices

operations in the LLVM/OpenMP offloading workflow shown in Fig. 1. The Host Runtime is target-agnostic, and it loads the Device Runtime plugin according to the type of device code embedded in the fat application binary. For example, it can load the CUDA plugin `libomptarget.rtl.cuda.so`, which makes CUDA calls to transfer data and launch kernels.

The Host Runtime talks to the Device Runtime through the Device Plugin Interface, which is a small set of target-agnostic C functions, to perform offloading operations (e.g. `__tgt_rtl_data_alloc()`). All Device Runtime plugins must implement this interface, hiding low-level details from the Host Runtime.

2.2 The Remote Offloading Plugin

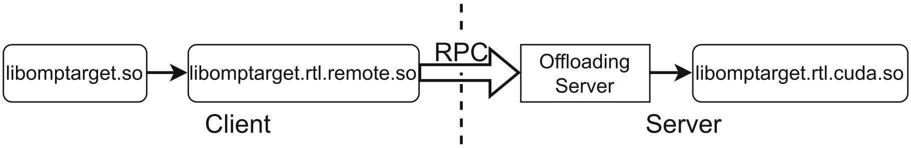


Fig. 2. LLVM/OpenMP remote offloading workflow for CUDA devices

The idea of the remote offloading plugin is to use remote procedure calls (RPC) to relay the plugin interface calls to remote processes, so the application can access devices on other machines transparently. This plugin implements the core set of the Device Plugin Interface to talk to the Host Runtime. OpenMP application are compiled as usual and no code modification is required except for doing multi-device offloading using the `device(n)` clause. The plugin is also compatible with OpenMP asynchronous offloading (e.g. the `nowait` clause), thanks to the orthogonal design of the concurrent offloading mechanism in LLVM/OpenMP [24].

In Fig. 2, the Client stands for the OpenMP application, while the Server is a binary provided by the remote offloading plugin. The user runs one instance of the Client, and one instance of the Server per GPU node. Once connected, the Client can offload to all the GPUs managed by all the Servers. Both the Host Runtime and the Device Runtime plugin are working as they normally would, unaware of the Client-Server pair in the middle.

For each plugin interface call, the Client serializes the function arguments and sends one blocking RPC request to the Server. The server will deserialize incoming requests and execute the corresponding device operation, then reply to the Client, so it can proceed. For a `map(to/from)` operation, the buffer must also be serialized. The Server uses a task queue and a pool of threads to handle the RPC requests, so that offloading activities on one GPU does not block activities on other GPUs managed by the same Server.

Originally, the remote offloading plugin uses gRPC [5] as its RPC backend, since it is the natural choice. Then an UCX [22] backend was added since it can utilize high-performance interconnects like Infiniband, thus bringing the potential to improve the offloading application's performance to a level comparable to that of its MPI+X equivalent.

2.3 Related Work

OpenMP single-node multi-GPU offloading has been previously explored [7,28], but standard OpenMP does not go beyond the node boundary and the most popular approach to program distributed-memory GPUs is still the hybrid model of MPI+X, X being a local GPU programming model such as CUDA or OpenMP. An alternative that has received some attention is to replace MPI with a task-based programming model that also interoperates with CUDA; among them are Charm++ [1], UPC++ [2], Legion [3], and Chapel [4]. NVSHMEM [6] enables remote communication directly from the CUDA kernels. Intel Cluster OpenMP uses a Distributed Shared Memory runtime system to run OpenMP CPU parallel regions across nodes [23]. Kokkos Remote Spaces [8] is an extension to the Kokkos programming model [27] to support distributed shared memory for programming GPUs and other devices. rCUDA [19] is a framework for remote GPU virtualization, in which a set of GPUs can be shared and remotely accessed by several clients simultaneously.

Minimod [10] is one of the applications used in this study. Implementations of Minimod have been evaluated with OpenMP tasks [18], and in distributed setups using the Legion programming model targeting CPUs [17] and GPUs [16,21]. The present paper evaluates a version of Minimod using OpenMP target regions wrapped in tasks to make use of multiple GPUs simultaneously.

3 Performance Analysis and Optimizations

This work focuses on the UCX backend of the remote offloading plugin for its performance potentials. Additionally, the devices are running the exact same kernel, so scalability issues mostly arise from runtime and communication overheads.

3.1 Runtime Optimizations

To identify the performance issues of the remote offloading process, we profiled a microbenchmark that only does host-device transfers of fixed-size buffers, using the `map` clause. The selected results of running on a single remote GPU are shown in Fig. 3. Clearly, the remote offloading plugin's internal overhead dominates the communication latency for all message sizes. This is a direct result of the RPC

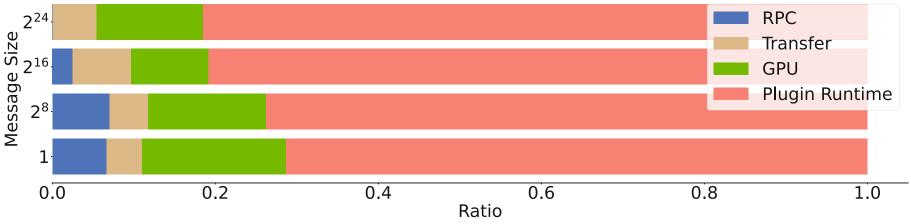


Fig. 3. Breakdown of remote `map(from)` latency

mechanism: the layers of abstractions to handle different offloading operations and communication backends, communication progression/completion tracking, data (de)serialization, and other bookkeeping operations. Since the plugin uses blocking RPCs, all these overheads are translated to `map` latencies, which lead to longer idle periods on the device.

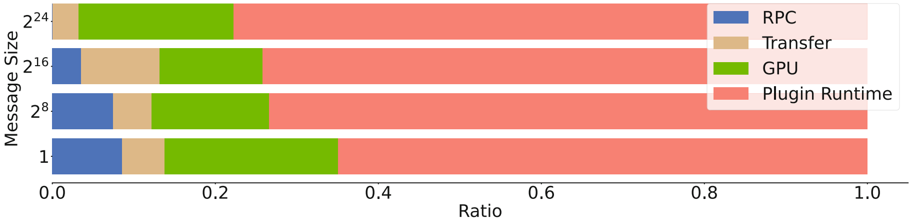


Fig. 4. Breakdown of remote `map(from)` latency, with runtime optimizations

UCX provides active messages for inter-node RPC, but its API is too primitive to relay the plugin interface functions. The original remote offloading plugin implemented its own RPC mechanism based on UCX's message passing API. We first improve the plugin from a software engineering point of view: use suitable C++ features to reduce the overhead of the serializer and other internal abstractions, speedup UCX communication progression to reduce send-receive latency, etc. The latency breakdown after applying our runtime optimizations is shown in Fig. 4. Although we have achieved around 6% overhead reduction for small buffers, the optimizations' effectiveness drops for larger ones.

3.2 CUDA-Aware Communication

The plugin's RPC serializer is a major source of overhead. Similar to MPI, UCX is mostly designed to send contiguous chunks of data between buffers on different processes. To perform RPC, which is a high-level operation, the Client must serialize function arguments and all the buffers involved before passing them to UCX, and the Server must do the reverse. Additionally, for `map(to/from)` the

Server needs a staging buffer to store a temporary copy of the mapped buffer, to pass the mapped data between the serializer and the device API.

To map a host buffer to the device using the `__tgt_rtl_datasubmit` plugin interface, a regular device plugins does a device allocation, an `HtoD memcpy` and a device synchronization call. But for the remote offloading plugin, the RPC serializer does multiple extra memory allocations and `memcpy`'s. These overheads are repeated on both the Client and the Server and grows linearly with the size of the buffer, which is why the runtime overhead accounts for such a large percentage of the total latency.

To further reduce the overhead, we utilize the UCX's CUDA support to send and receive data directly from the GPU, eliminating the need for a staging buffer on the Server and other serializer overhead. When this mechanism is active, a `map(to/from)` operation will be broken into two steps: an RPC request that only sends the metadata (device buffer address, buffer size, etc.), and a second UCX send/receive request that transfers data directly between the Client's host memory and the Server's device memory. The two-step approach has its own associated overhead: if the mapped buffer is too small, then it may be faster to serialize everything and use a single RPC request. Therefore, we introduce an environment variable to specify the smallest buffer size to activate the CUDA-aware UCX mechanism (`SPLIT_THRESH`). This threshold is affected by many factors and should be experimentally determined for different software and hardware combinations.

3.3 Thread Contention and NUMA-GPU Affinity

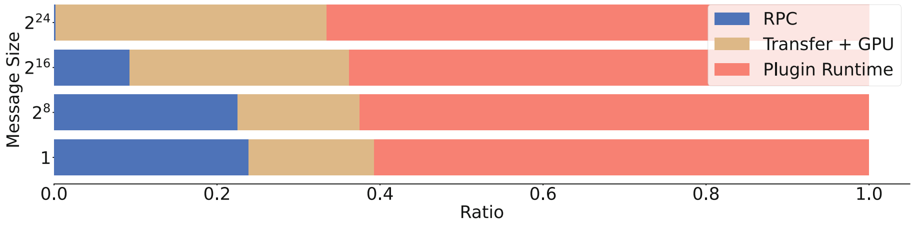


Fig. 5. Breakdown of remote `map(from)` latency, with all optimizations applied

There are several other factors that affect `map` latency. In UCX, the Worker object provides independent progression and completion of communication operations. This means that even if all traffic goes through the same network card, parallel injection from multiple Workers can still improve message throughput through overlapping send-receive operations in the higher levels of the runtime system, especially when the combined message flow is not large enough to saturate the hardware bandwidth. Originally, the Server uses a single Worker to serve all the GPUs it has access to. This means the Worker must use locks to prevent data races caused by concurrent access from different threads, essentially serializing many overlappable operations and reducing the injection rate [9]. In this work, we use one Server (Worker) per GPU, to reduce thread contention.

Also, it is important to make sure that the Server is running in the NUMA node that is the closest to the GPU it is offloading to. Modern heterogeneous systems tend to have more than one GPU per compute node, and different GPUs are local to different CPU sockets, if there is more than one socket. Additionally, HPC systems like ORNL Summit and LLNL Sierra split the PCIe lanes of the Infiniband network card evenly between the two sockets [29]. This means the application must drive communication from both NUMA nodes in a balanced fashion, to maximize communication performance. In our experiments, host-device transfers that cross the NUMA boundary can have up to 23% higher latencies than that of the transfers within the same NUMA node. In all our experiments, we pin the Server process to the NUMA node that the GPU is connected to, and UCX will pick up the closest network port.

Figure 5 shows the latency breakdown after applying the all optimizations mentioned in this section, with `SPLIT_THRESH` set to 2^{22} bytes. The results show that we have achieved at least 11% reduction in overheads when compared to the results of the original plugin in Fig. 3, for all buffer sizes. Now, the plugin internal overhead is always below 70% of the remote data mapping latency. Note that the RPC overhead is a constant (5.328 µs), but its percentage in the total latency becomes over 20% for the smaller buffers, as a result of significantly reduced absolute latency.

4 Evaluation

We evaluate our optimizations using microbenchmarks and proxy-apps on the Cypress computing system at TotalEnergies R&T in Houston. Each Cypress node contains one AMD EPYC 7F52 16-core CPU, one Mellanox ConnectX-6 200 Gb/s Infiniband network card, and four NVIDIA A100 GPUs. The system runs CentOS 8 with Linux kernel 4.18.0, CUDA 11.5.119 and MOFED-5.1-2.5.8.0. We use UCX commit `5879c44` of the `v1.13x` branch, with the GPUDirect RDMA [12] and GDRCopy [11] transports enabled.

As the baseline, we use commit `6120be4` of the original remote offloading plugin, which is based on commit `67ab4c0` of the LLVM trunk. The remote offloading plugin with only runtime optimizations is referred to as Opt1; while plugin version Opt2 has all optimizations applied. Since we can eliminate unnecessary overhead by allowing the Client to offload to its local GPUs directly (instead of go through the remote plugin), we have included results that enabled Client-side offloading, which we refer to as Opt2L. All three proxy-apps are tested in small and large problem sizes.

4.1 Microbenchmarks

Figure 6 shows the remote GPU to/from mapping latencies of different buffer sizes. Compared to the baseline, the Opt2 version of the plugin reduces the

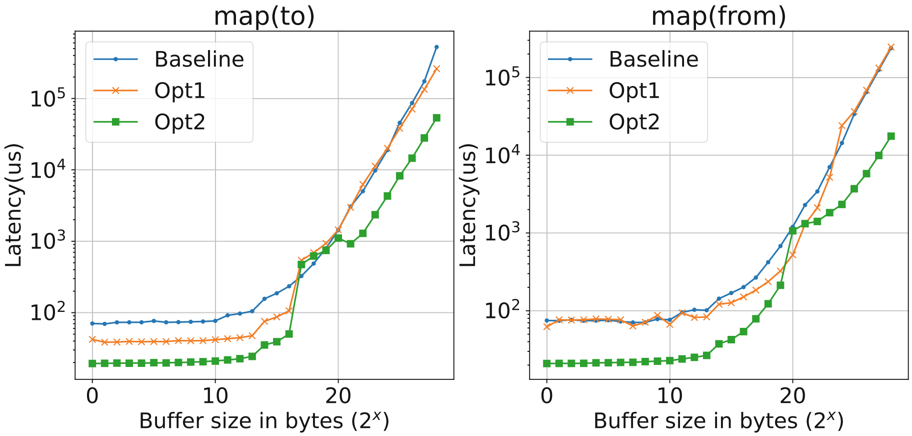


Fig. 6. Remote GPU `map(to/from)` latency

message latency by ∼72% for small buffers, and ∼90% for large ones. With Opt2's lower device buffer mapping latencies, the OpenMP application can launch `target` regions faster, thus obtaining better scalability.

Opt1 and Opt2 show little to no speedup for buffer size around 2^{19} bytes. This is caused by the compound effects of UCX switching its internal communication protocol and our `SPLIT_THRESH` setting for enabling CUDA-aware communication, since we used the same setting for all buffer sizes instead of the best settings for each size. For real applications, the user should adjust the `SPLIT_THRESH`, as well as UCX's protocol switching thresholds (zero-copy, rendezvous, etc.), so that the latencies of the most frequently mapped buffer sizes are optimal.

4.2 Weak Scaling - RSBench and XSBench

XSBench [26] and RSBench [25] are proxy-apps that capture the core computation of the Monte Carlo neutron transport code OpenMC [20], while XSBench is memory-bound and RSBench is compute-bound. We use the OpenMP offloading version of both proxy-apps, with the same modifications used in [14] to enable multi-device offloading. We run the proxy-apps on 4 to 16 GPUs, and normalized the run times with respect to the run time of using 4 local GPUs without the remote offloading plugin. We keep the amount of work and data transferred per GPU constant to evaluate the weak scalability of the plugin.

Kernels execution time, total host-device transfer size per kernel, and the total number of host-device transfers per kernel are listed in Table 1. The number of transfers per kernel launch is important since it is proportional to the number of RPC requests for device buffer allocation/free, and the actual data transfer. So XSBench has $19 \times 3 + 1 = 58$ RPC requests per kernel launch, while the number is $27 \times 3 + 1 = 82$ for RSBench.

Table 1. XSBench/RSBench kernel durations and per-kernel launch data transfers

	Kernel-Small	Kernel-Large	Transfer-Small	Transfer-Large	No. Transfers
XSBench	56.38 ms	271.6 ms	240.4 MB	5648 MB	19
RSBench	231.4 ms	1371 ms	5.325 MB	28.92 MB	27

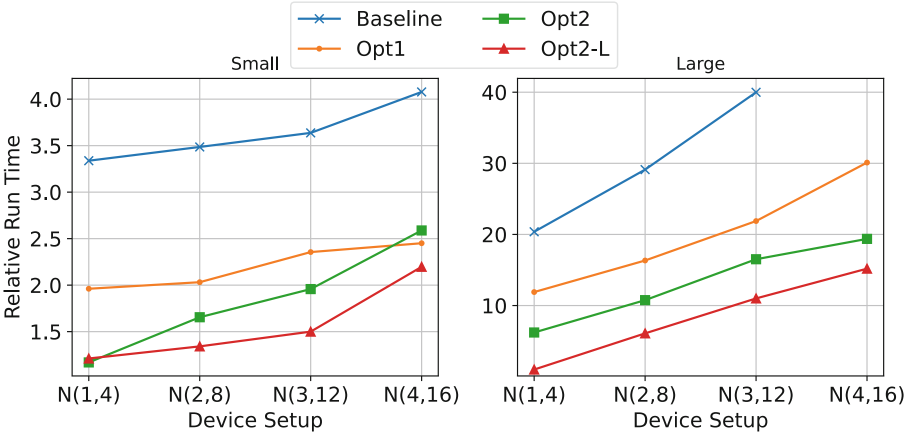


Fig. 7. Weak scaling results of XSBench

XSBench results are presented in Fig. 7. For the horizontal axis, $N(n, 4n)$ stands for running the benchmark on n nodes and using all $4n$ GPUs. The baseline version running the large setup crashes since it exhausts all available host memory. Again, our optimized implementations are significantly faster than the baseline. For 16 GPUs, Opt2L increases the parallel efficiency by 25.2%. While we can achieve a 56% parallel efficiency on 16 GPUs for the small setup, only 6% was obtained on 16 GPUs for the large setup. This is because the XSBench-large transfers 5.5 GB of data per kernel launch, and all of them must go through the Client's network card without effective overlap. XSBench-large is therefore severely communication-bound and does not scale well.

RSBench results in Fig. 8 shows the effectiveness of our optimizations for compute-bound applications with longer kernel execution times and smaller data transfers. On 16 GPUs, all three optimized versions obtained at least 66% parallel efficiency, while the baseline version goes as low as 34%.

4.3 Strong Scaling - Minimod

Minimod [10] is a proxy application that simulates the propagation of waves through subsurface models, by solving a finite difference discretized form of the wave equation. In this work, we use one of the kernels contained in Minimod: the acoustic isotropic propagator in a constant-density domain [15].

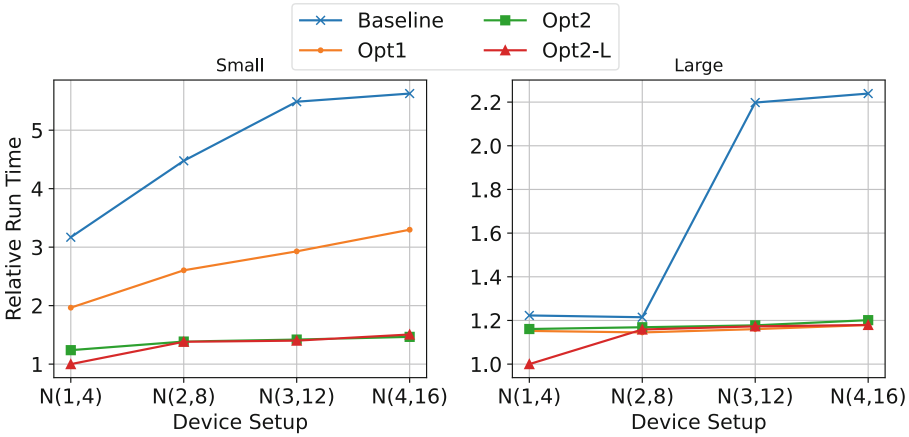


Fig. 8. Weak scaling results of RSBench

Table 2. Minimod total kernel durations and data transfers (per iteration)

Kernel-Small	Kernel-Large	Transfer-Small	Transfer-Large	No. Transfers
171.9 μs	6733 μs	182.2 KB	4032 KB	2

```
for (int g = 0; g < nGPUs; g++) {
  #pragma omp task depend(...)
  #pragma omp target teams distribute parallel for device(g)
  for (...)
    // Stencil computation
}
// Halo exchange
for (int g = 0; g < nGPUs; g++) {
    // Left halo region: DtoH
    #pragma omp task depend(...)
    #pragma omp target update from(...) device(g)
    // Left halo region: HtoD
    #pragma omp task depend(...)
    #pragma omp target update to(...) device(g-1)
    // Repeat for the right halo regions ...
}
```

Fig. 9. Simplified Minimod multi-GPU offloading and halo exchange workflow

Minimod natively supports multi-device OpenMP offloading using `target` regions wrapped in OpenMP tasks (see Fig. 9), and is strong-scaling in nature [18]. The 3D grid used in Minimod is partitioned along the X-axis (i.e. sliced parallel to the YZ-plane), regardless of the number of devices it is running on. Therefore, the amount of halo data exchanged between the devices is only related to the size of the grid, since the area of the cross-section of the grid is always $dimY \times dimZ$. However, since the offloading Servers are only connected to the Client, not to each other, the halo data must all be relayed by the Client, creating a central communication bottleneck. Minimod's kernel durations and data transfer sizes for running on two devices are listed in Table 2.

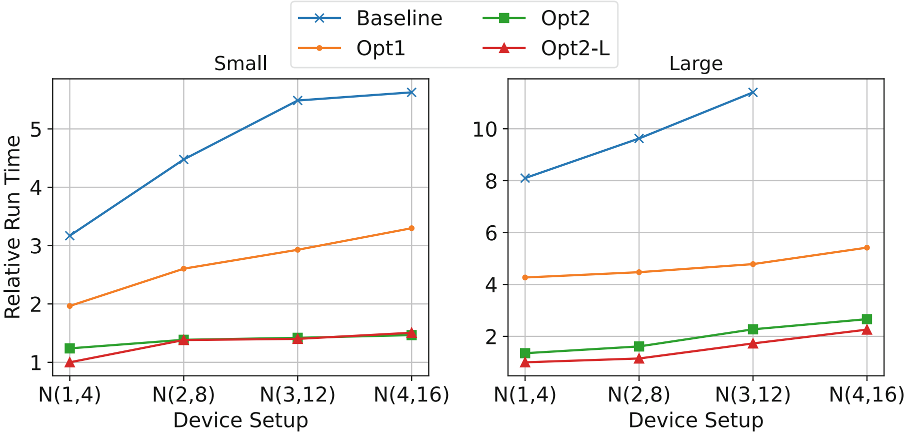


Fig. 10. Strong scaling results of Minimod

Figure 10 shows Minimod strong scaling results. Again, Opt2 and Opt2L outperforms other versions of the plugin significantly, showing the effectiveness of our optimizations. However, for all configurations, the run time increases as we use more GPUs. The reason is: as the number of device increases, the (already short) execution time of the kernel decreases linearly, but the halo exchange overhead grows linearly. Additionally, all halo exchange traffic must go through the Client, which leads to high communication contention. Since the communication overhead dominates the execution time, we see no performance improvement for all configurations.

4.4 Discussions

Our work has reduced the overhead of the LLVM/OpenMP remote offloading plugin by a large margin, and is especially effective for weak scaling of compute-bound applications. But as shown in Fig. 5, plugin overhead still accounts for more than 60% of the communication latency, which is a road blocker for higher strong scalability. One solution is to implement the asynchronous Device Plugin Interface functions in the remote plugin, using UCX asynchronous communication APIs. We will need to replicate CUDA Stream functionalities to keep track of asynchronous events and handle dependencies, but this will hide and/or reduce the aforementioned runtime overheads. We could also implement message aggregation to reduce the total number of RPC requests, as many transfers listed in Table 1 are only sending a single scalar.

OpenMP's flat device model can be extended to expose the device topology to the users, so they can do hierarchical computation decomposition. Similar to an MPI shared-memory communicator, an OpenMP `node` construct can enumerate the `legion` of devices attached to the same machine. We could also do implicit hierarchical offloading by presenting all GPUs attached to the same NUMA node as a single device, and `map` buffers to CUDA Managed Memory.

The current design of the LLVM/OpenMP runtime can also be extended to push the scalability further. Currently, the Device Plugins know very little about the big picture and rely on the Host Runtime's prescriptive offloading instructions, creating the central bottleneck. We could increase the autonomy of the Device Plugins and give descriptive orders whenever possible. A partitioned global address space model can also be introduced to support direct inter-node device-to-device transfers.

Lastly, the single-Client multi-Server architecture must be replaced with a more SPMD-like one, for the remote offloading plugin to work for a wider spectrum of applications. Then current centralized approach not only creates a communication bottleneck, but also limits the amount of host memory available to the application to be the amount of memory installed on the Client's node. One specification-breaking solution is to encourage the users to allocate all host buffers that will be interacting with the device i inside a `target data device(i)` construct. Then in run-time we "offload" the entire `target data` region to device i's node, so that the host code inside that region also runs on the remote node and can utilize its main memory. Alternatively, we could use a page migration-based mechanism to transparently extend the amount of host memory, similar to Intel Cluster OpenMP [23].

5 Conclusions and Future Work

Remote OpenMP device offloading is a promising alternative to MPI+X, as it improves the portability and maintainability of the application by covering both inter-node and intra-node computation in a single programming model. In this work, we analyzed the performance bottlenecks of the LLVM/OpenMP remote offloading plugin, and have identified the RPC serializer and the buffered communication mechanism as two major sources of overhead. We then applied optimizations that reduce the plugin's internal overhead, and enabled CUDA-aware UCX communications to accelerate the transportation of large buffers. Evaluation using microbenchmarks shows that our optimizations have reduced `map` latencies by up to 92%. With lower data transfer latencies, we have achieved a minimum of 25.2% increase of parallel efficiency in proxy-apps running on 16 GPUs. Our optimizations are especially effective for weak scaling proxy-apps that have relatively long-running kernels and small data transfers.

However, weak scaling with large data transfers and strong scaling still prove to be challenging despite our optimizations. We propose a few runtime modifications to further reduce the plugin's latency and improve its scalability, but ultimately we believe extending the OpenMP runtime design is required to reach performance comparable to MPI+X.

For future work, we can implement the asynchronous plugin interface to hide plugin internal overhead, add an MPI backend to further improve the plugin's portability, and extend the offloading Server to support direct Server-to-Server transfers.

Acknowledgements. We would like to thank TotalEnergies EP Research and Technologies for their support of this work. This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, in particular its subproject on Scaling OpenMP with LLVM for Exascale performance and portability (SOLLVE).

This research was also funded in part by the United States Department of Defense, and was supported by resources at Los Alamos National Laboratory, operated by Triad National Security, LLC under Contract No. 89233218CNA000001.

References

1. Acun, B., et al.: Parallel programming with migratable objects: Charm++ in practice. In: SC 2014: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 647–658 (2014). https://doi.org/10.1109/SC.2014.58
2. Bachan, J., et al.: UPC++: a high-performance communication framework for asynchronous computation. In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 963–973 (2019). https://doi.org/10.1109/IPDPS.2019.00104
3. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: expressing locality and independence with logical regions. In: SC 2012: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp. 1–11, November 2012. https://doi.org/10.1109/SC.2012.71
4. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the Chapel language. Int. J. High Perform. Comput. Appl. **21**(3), 291–312 (2007). https://doi.org/10.1177/1094342007078442
5. gRPC community: grpc. https://grpc.io/about/
6. Hsu, C.H., Imam, N., Langer, A., Potluri, S., Newburn, C.J.: An initial assessment of NVSHMEM for high performance computing. In: 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 1–10 (2020). https://doi.org/10.1109/IPDPSW50202.2020.00104
7. Kale, V., Lu, W., Curtis, A., Malik, A.M., Chapman, B., Hernandez, O.: Toward supporting multi-GPU Targets via taskloop and user-defined schedules. In: Milfeld, K., de Supinski, B.R., Koesterke, L., Klinkenberg, J. (eds.) IWOMP 2020. LNCS, vol. 12295, pp. 295–309. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58144-2_19
8. Kokkos: kokkos remote spaces. https://github.com/kokkos/kokkos-remote-spaces
9. Lu, W., Curtis, T., Chapman, B.: Enabling low-overhead communication in multithreaded OpenSHMEM applications using contexts. In: 2019 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM), pp. 47–57 (2019). https://doi.org/10.1109/PAW-ATM49560.2019.00010

10. Meng, J., Atle, A., Calandra, H., Araya-Polo, M.: Minimod: a finite difference solver for seismic modeling (2020). https://arxiv.org/abs/2007.06048
11. NVIDIA: Gdrcopy. https://github.com/NVIDIA/gdrcopy
12. NVIDIA: Nvidia cuda gpudirect rdma. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html
13. OpenMP architecture review board: OpenMP application programming interface, November 2018. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf, version 5.0
14. Patel, A., Doerfert, J.: Remote openmp offloading. In: Varbanescu, A.L., Bhatele, A., Luszczek, P., Marc, B. (eds.) High Performance Computing. pp. 315–333. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-031-07312-0_16
15. Qawasmeh, A., Hugues, M.R., Calandra, H., Chapman, B.M.: Performance portability in reverse time migration and seismic modelling via OpenACC. Int. J. High Perform. Comput. Appl. **31**(5), 422–440 (2017). https://doi.org/10.1177/1094342016675678
16. Raut, E., Anderson, J., Araya-Polo, M., Meng, J.: Evaluation of distributed tasks in stencil-based application on GPUs. In: 2021 IEEE/ACM 6th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2), pp. 45–52 (2021). https://doi.org/10.1109/ESPM254806.2021.00011
17. Raut, E., Anderson, J., Araya-Polo, M., Meng, J.: Porting and evaluation of a distributed task-driven stencil-based application. In: Proceedings of the 12th International Workshop on Programming Models and Applications for Multicores and Manycores, pp. 21–30. PMAM 2021. Association for Computing Machinery, New York (2021). https://doi.org/10.1145/3448290.3448559
18. Raut, E., Meng, J., Araya-Polo, M., Chapman, B.: Evaluating performance of OpenMP tasks in a seismic stencil application. In: Milfeld, K., de Supinski, B.R., Koesterke, L., Klinkenberg, J. (eds.) IWOMP 2020. LNCS, vol. 12295, pp. 67–81. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58144-2_5
19. Reaño, C., Silla, F., Shainer, G., Schultz, S.: Local and remote GPUs perform similar with EDR 100g InfiniBand. In: Proceedings of the Industrial Track of the 16th International Middleware Conference. Middleware Industry 2015. Association for Computing Machinery, New York (2015). https://doi.org/10.1145/2830013.2830015
20. Romano, P.K., Forget, B.: The OpenMC monte Carlo particle transport code. Ann. Nucl. Energy **51**, 274–281 (2013). https://doi.org/10.1016/j.anucene.2012.06.040
21. Sai, R., Mellor-Crummey, J., Meng, X., Araya-Polo, M., Meng, J.: Accelerating high-order stencils on GPUs. In: 2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), pp. 86–108 (2020). https://doi.org/10.1109/PMBS51919.2020.00014
22. Shamis, P., et al.: UCX: an open source framework for HPC network APIs and beyond. In: 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects, pp. 40–43 (2015). https://doi.org/10.1109/HOTI.2015.13
23. Terboven, C., Mey, D., Schmidl, D., Wagner, M.: First experiences with intel cluster OpenMP. In: Eigenmann, R., de Supinski, B.R. (eds.) IWOMP 2008. LNCS, vol. 5004, pp. 48–59. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-79561-2_5
24. Tian, S., Doerfert, J., Chapman, B.: Concurrent execution of deferred OpenMP target tasks with hidden helper threads. In: Chapman, B., Moreira, J. (eds.) Languages and Compilers for Parallel Computing, pp. 41–56. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-030-95953-1_4

25. Tramm, J.R., Siegel, A.R., Forget, B., Josey, C.: Performance analysis of a reduced data movement algorithm for neutron cross section data in Monte Carlo simulations. In: Markidis, S., Laure, E. (eds.) EASC 2014. LNCS, vol. 8759, pp. 39–56. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-15976-8_3
26. Tramm, J.R., Siegel, A.R., Islam, T., Schulz, M.: XSBench - the development and verification of a performance abstraction for Monte Carlo reactor analysis. In: PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future, Kyoto (2014). https://www.mcs.anl.gov/papers/P5064-0114.pdf
27. Trott, C.R., et al.: Kokkos 3: programming model extensions for the exascale Era. IEEE Trans. Parallel Distrib. Syst. **33**(4), 805–817 (2022). https://doi.org/10.1109/TPDS.2021.3097283
28. Yan, Y., Lin, P.H., Liao, C., de Supinski, B.R., Quinlan, D.J.: Supporting multiple accelerators in high-level programming models. In: Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores, pp. 170–180. PMAM 2015. Association for Computing Machinery, New York (2015). https://doi.org/10.1145/2712386.2712405
29. Zimmer, C., et al.: An evaluation of the coral interconnects. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC 2019. Association for Computing Machinery, New York (2019). https://doi.org/10.1145/3295500.3356166

Exploring New and Recent OpenMP Extensions

Characterizing the Performance of Task Reductions in OpenMP 5.X Implementations

Jan Ciesko and Stephen L. Olivier(✉)

Center for Computing Research, Sandia National Laboratories,
Albuquerque, NM 87123, USA
{jciesko,slolivi}@sandia.gov

Abstract. OpenMP 5.0 added support for reductions over explicit tasks. This expands the previous reduction support that was limited primarily to worksharing and parallel constructs. While the scope of a reduction operation in a worksharing construct is the scope of the construct itself, the scope of a task reduction can vary. This difference requires syntactical means to define the scope of reductions, e.g., the task_reduction clause, and to associate participating tasks, e.g., the in_reduction clause. Furthermore, the disassociation of the number of threads and the number of tasks creates space for different implementations in the OpenMP runtime. In this work, we provide insights into the behavior and performance of task reduction implementations in GCC/g++ and LLVM/Clang. Our results indicate that task reductions are well supported by both compilers, but their performance differs in some cases and is often determined by the efficiency of the underlying task management.

Keywords: OpenMP · reduction · worksharing · tasking

1 Introduction

Since the first OpenMP specifications for Fortran and C/C++, the API has always included support for a defined set of reduction operations in worksharing and

This article has been authored by an employee of National Technology & Engineering Solutions of Sandia, LLC under Contract No. DE-NA0003525 with the U.S. Department of Energy (DOE). The employee owns all right, title and interest in and to the article and is solely responsible for its contents. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this article or allow others to do so, for United States Government purposes. The DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (https://www.energy.gov/downloads/doe-public-access-plan). Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

M. Klemm et al. (Eds.): IWOMP 2022, LNCS 13527, pp. 35–49, 2022.
https://doi.org/10.1007/978-3-031-15922-0_3

parallel constructs. This support covers C/C++ `for` and Fortran `do` loops with a defined iteration space, suitable for many iterative algorithms. OpenMP 3.0 introduced support for explicit task parallelism in OpenMP, enabling more irregular computations such as recursive algorithms and pointer chasing. That class of capabilities was further developed in subsequent versions of the specification to include the support of task dependencies in OpenMP 4.0 and the taskloop construct in OpenMP 4.5.

OpenMP 4.0 also added support for user-defined reductions over non-trivial data types and arbitrary operations. OpenMP 5.0 [8] finally brought support for explicit tasks to contribute to reductions, adding the means to define both the scope of such reductions and the tasks participating in them. As with many OpenMP features, implementors required time to incorporate support of task reductions, but both LLVM/Clang and GCC/g++ now include this feature.

This paper examines the current state of task reduction support in GCC/g++ and LLVM/Clang, including support of the language constructs as well as the performance of their implementations. For this purpose, we have selected three synthetic benchmarks that stress-test implementations and expose the cost of reduction support as well as tasking overheads for each compiler.

The rest of the paper is structured as follows. Section 2 provides background on reductions and currently supported syntax in OpenMP. Section 3 describes the benchmarks used in the evaluation, and Sect. 4 provides details of the experimental setup. Section 5 discusses performance results from the evaluation. Section 6 provides insight into reduction support in GCC/g++ and LLVM/-Clang and matches observed benchmark behavior with implementation choices. Section 7 surveys related work. Lastly, Sect. 8 summarizes this work and gives an outlook on further research directions.

2 Background

In this section we provide some context for our work. First, we consider general implementation strategies for reduction operations. Second, we give an overview of OpenMP task reductions.

2.1 Reductions and Their Common Implementation Strategies

In mathematical terms, a reduction algorithm is a numerical fold over a sequence of numbers. As the name suggests, it implies an iterative update (accumulation) of a result variable. For parallel formulations on parallel hardware in which the sequence of numbers is traversed concurrently and where the result variable is thus updated concurrently, data races can occur.

Two common strategies exist to avoid data races. One strategy makes memory updates atomic. The other creates thread-private data copies for the duration of the traversal of the sequence by each thread. In this strategy, a second step is required, during which the privatized copies are combined.

Atomic updates depend on software or hardware support of atomic memory updates and have implications for cache coherency traffic as a result of contention among threads to access the location of the reduction variable. Privatization avoids concurrent accesses to a memory location at the cost of private memory allocation, initialization, and the need for the final combination step. Which particular implementation is preferable on a given architecture depends on the size of the reduction variable and the access frequency.

OpenMP supports atomic memory accesses through the `atomic` construct which can be used directly by the developer to implement reductions. Programing model runtimes commonly rely on privatization for their internal implementation, which together with the possibility of using atomics, offers flexibility of choice to the programmer depending on the use case. Alternatively, the developer can implement privatization using the `threadprivate` directive. In this case, the developer is responsible for combining per-thread results as well.

From an OpenMP implementation perspective, privatization can be achieved by privatizing per thread or by privatizing per task. Privatization per task can incur significant overheads if the number of tasks is disproportionally larger than the number of threads. Since large numbers of tasks are common, implementations rely on per thread privatization where each task acquires the thread-private copy of the reduction variable at execution time.

Finally, the developer can avoid privatization or the need for atomic accesses in recursive task parallel programs by passing the reduction variable to each task as a function argument by value and returning the intermediate results as a return value. Unfortunately, the use of stack for the purpose of privatization is equivalent to per-task privatization and incurs the highest memory use and overheads due to repetitive initialization of stack variables. We call this approach *stack* in the evaluation section.

2.2 OpenMP Task Reduction Syntax and Semantics

The OpenMP specification admits the formulation of task reductions through *reduction scoping clauses* and a *reduction participating clauses*. The former "defines the region in which a reduction is computed", while the latter "specifies a task (or SIMD lane) as a participant in a reduction defined by a reduction scoping clause" [8].

The clauses are as follows, taking an operation *op* and reduction variable *var*.

- `reduction(task`, *op: var*): Scopes a task reduction for a parallel or worksharing region
- `reduction(`*op: var*): Scopes a task reduction for a taskloop region and makes the tasks created to execute the loop participants in the task reduction
- `task_reduction(`*op: var*): Scopes a task reduction for a taskgroup region
- `in_reduction(`*op: var*): Denotes participation of a task, target task, or taskloop in a task reduction

In Listing 1.1, a task reduction is scoped using the `task_reduction` clause on the `taskgroup` construct. Explicit tasks participating in the reduction use the `in_reduction` clause on the `task` construct. In Listing 1.2, the task reduction is scoped using the `reduction` clause of the `parallel` construct with the task modifier. As before, tasks bearing the in_reduction clause participate in the reduction. Finally, Listing 1.3 shows the use of the reduction clause on the taskloop construct, which acts as both a reduction scoping and reduction participating clause. References to the variable in the explicit tasks created by the OpenMP implementation to execute the iteration of the loop will all contribute to the reduction. Though not demonstrated here, the taskloop construct can also take an in_reduction clause to participate in a reduction already scoped in an enclosing region. The in_reduction clause can also be applied to a target construct, allowing a potentially offloaded target task to contribute to a task reduction.

```
#pragma omp parallel
#pragma omp single
#pragma omp taskgroup task_reduction(+: sum)
{
  #pragma omp task in_reduction(+: sum)
    sum += 1;
  #pragma omp task in_reduction(+: sum)
    sum += 2;
}
```

Listing 1.1. Simple Example of OpenMP Task Reduction

3 Benchmark Programs

To benchmark the implementations of OpenMP reductions, we consider a set of programs with distinct properties. They demonstrate the use of all OpenMP task reductions clauses and critically depend on an efficient implementation due to their high access frequency to the reduction variable.

In real-word applications, this frequency relative to other computation is significantly lower. This is the case where tasks are larger and spend more time in unrelated code. Instead, these benchmark applications stress-test implementations and show the hypothetical limitations, similar to roofline analysis.

For completeness, we contrast the OpenMP reduction support against other approaches to implement reductions such as atomics, user-managed thread-private copies, and returning partial values through the call tree. Prior to the availability of task reductions, these approaches would have been the only options for users attempting to combine results from OpenMP tasks.

3.1 Fibonacci

Fibonacci is a recursive program to calculate the *nth* number in the Fibonacci series. It represents a typical use case for task parallel programs where recursive formulations result in compact code or where an unknown iteration space at a given nesting level disallows the use of work sharing constructs.

```
int n, sum;

void fib (int n, int &sum)
{
  if (n < 2)
```

```
    sum += n;
  else
  {
    #pragma omp task in_reduction(+: sum)
      fib(n-1, sum);
    #pragma omp task in_reduction(+: sum)
      fib(n-2, sum);
  }
}

int main (int argc, char *argv[])
{
  n = atoi(argv[1]);

  #pragma omp parallel reduction (task, +: sum)
  #pragma omp single
  #pragma omp task in_reduction (+: sum)
    fib(n, sum);

  std::cout << "fib(" << n << ") = " << sum << std::endl;
  return 0;
}
```

Listing 1.2. Fibonacci calculation using OpenMP task reduction

Listing 1.2 shows an implementation using OpenMP task reductions. Here the program scopes the reduction using the `reduction` clause on the `parallel` construct with the `task` modifier, and tasks participating in that reduction scope use the `in_reduction` clause. Note that the reduction variable is passed by reference to the recursive function, because the reduction variable in the recursive function is not in the lexical scope of the parallel region. Further, using `taskwait` for synchronization is not required. The barrier at the end of the parallel region ensures that all tasks complete.

The program can be further augmented to provide “cut-off” values below which tasks would not be generated, thus coarsening parallelism. The effect is to reduce the number of tasks which in return lowers overheads of task creation and management. For example, if a cut-off of 10 were specified, then the calculation of $fib(9)$ would be handled by a direct sequential function call rather than an OpenMP task. Cut-offs could be either be implemented manually (if-then-else block) or using the `final` and `mergeable` clauses. Though not shown in the simplified code listing, for our evaluation we implemented manual cut-offs.

An alternative to task reductions would be the use of the `atomic` construct to update the reduction variable. In practice, this method is expected to introduce contention and limit effective parallelism as all threads compete to update the reduction variable.

Another alternative is to create thread-private copies of the reduction variable, accumulate partial sums in thread-local copies and combine the partial sums into the final sum at the end of the program. The `threadprivate` directive can be used to manage the copies. However, the addition of the final combining step makes this option somewhat cumbersome.

A third alternative with minimal requirements on compiler support is to transmit per-task partial sums as return values through the call stack. A drawback of this option is that a `taskwait` construct is needed at each nesting level in order to wait for the contributions of the child tasks.

3.2 Dot Product

Dot product implements the vector dot product of two arrays of numbers. Unlike Fibonacci, this benchmark is iterative rather than recursive. Listing 1.3 uses the `taskloop` construct to decompose the iteration space of the loop into tasks. In addition to task reduction, atomic, and thread-privatization versions, we also compare to a version using a worksharing construct with no explicit tasks.

For Fibonacci, the number of recursive function calls determines the number of tasks created and thus requires the use of cut-offs to limit the number of tasks. For Dot, the number of tasks is orthogonal to the algorithm itself and can be specified through the `num_tasks` clause on the `taskloop` construct.

```
#pragma omp parallel shared(x, y) num_threads(nthreads)
#pragma omp single
#pragma omp taskloop num_tasks(ntasks) reduction(+ : sum)
   for (unsigned long i = 0; i < n; ++i) {
      double tmp = x[i] * y[i];
      sum += tmp;
   }
```

Listing 1.3. Task reduction for vector dot product using the `taskloop` construct

3.3 Powerset

The Powerset benchmark computes the number of permutations of n elements by expanding a binary tree with a height of *log(n)*. While similar in algorithmic structure to Fibonacci, the Powerset produces a balanced tree which makes it less sensitive to task-stealing features in task schedulers. In addition to a variation of this algorithm using reduction variables of integer type, the Powerset benchmark also exercises implementations with user-defined reductions over a configurable type that is variable in size. We refer to it as *Powerset-UDR*. This configuration enables us to quantify and further differentiate the overheads originating from task management versus privatization. As with Fibonacci, we have implemented manual cut-offs (not shown in the simplified code listing).

```
int thr_priv_sum, cut_off;
#pragma omp threadprivate(thr_priv_sum)

void powerset(int n, int index) {
  for (int i = index; i < n; ++i){
#pragma omp task
    {
      powerset(n, i + 1);
      thr_priv_sum++;
    }
  }
}

int main(int argc, char *argv[]) {
  int n = atoi(argv[1]);
  int nthreads = atoi(argv[2]);
  cut_off = atoi(argv[3]);
  int sum = 0;

#pragma omp parallel num_threads(nthreads)
  {
    thr_priv_sum = 0;
#pragma omp single
#pragma omp task
    powerset(n, 0);
  }

//Reduce thread-private copies
#pragma omp parallel num_threads(nthreads)
  {
#pragma omp single
    nthreads = omp_get_num_threads();
#pragma omp for reduction(+ : sum)
    for (int i = 0; i < nthreads; i++)
      sum += thr_priv_sum;
  }

  std::cout << "powerset(" << n << ") = " << sum << std::endl;
  return 0;
}
```

Listing 1.4. Powerset using thread-private reduction variables obtained with manual thread privatization

The Powerset benchmark includes the variations described in the previous section for Fibonacci. Listing 1.4 shows the implementation of Powerset using the `threadprivate` directive and the subsequent manual reduction of private copies into the final reduction variable.

4 Experimental Setup

The test machine comprises Intel® Xeon® "Skylake" Platinum 8160 Processors in a dual socket configuration with 24 cores per socket (48 cores total) and 2 hardware threads per core running at 2.1 GHz. The memory is 192 GB DDR4. The operating system is Red Hat® Enterprise Linux® 7.9. The compiler and runtime versions are LLVM/Clang 14.0 (release) and GCC 13 (not yet released, code version dated 20220518).

For both compilers we have used the following sequence of options *-fopenmp, -Wall, -Wextra, -pedantic, -Werror* and *-O3*. Further, we have set the environment variables *OMP_PROC_BIND* and *OMP_PLACES* to *close* and *cores* respectively during execution. On the test machines, this results in a thread mapping of one thread per core.

5 Evaluation

We have evaluated the set of presented benchmarks for the various implementations, using a variable number of threads ranging from one to 128 and for a variable number of tasks. Further, we have compiled all implementations of all benchmarks with both the LLVM/Clang and the GCC/g++ compilers using the same compiler options. Lastly, the evaluation of a version of Powerset using user-defined reductions includes results for variable reduction type sizes. This section summarizes key finding and provides representative figures for configurations with 48 threads only. Executions with smaller thread counts exhibit similar behavior, while executions with more than 48 threads result in non-representative data due to over-subscription of the system and the resulting effects. All benchmark results show the average total execution time of 5 repetitions for a respective constant problem size and given range of tasks.

Figures 1 and 2 show performance results for the Fibonacci and Powerset computations. Key insights for these two benchmarks are as follows.

- Performance of implementations using the OpenMP language features for reductions is the same order of magnitude as prior available implementations using stack-local variables (*stack*) or manual privatization (*threadprivate*).
- The use of `parallel` with the `task` modifier (*parallel-task-red*) yields similar results to the use of the `task_reduction` clause on the `taskgroup` construct (*taskgroup-red*).
- Atomic accesses (*atomic*) incur high overhead regardless of the number of tasks due to threads contending for the same memory location.
- Implementations relying on stack-local variables and the *taskwait* construct depend on the efficiency of the underlying tasking implementation. GCC/g++ performs well only for low task counts, while LLVM/Clang outperforms for large task counts.
- Implementations using the *threadprivate* clause to manually privatize variables (*threadpriv*) underperform compared to other techniques for small task counts. Recall that they incur the cost of the additional step of manually combining the thread-private copies.
- When using GCC/g++, the performance of `taskloop` reductions (*taskloop-red*) degrades with large numbers of tasks.
- No significant differences were observed when using the `untied` task modifier (*parallel-task-red-untied*) compared to the tied default (*parallel-task-red*).

Figure 3 shows results for the dot-product with two to 131k tasks for LLVM/-Clang and an input problem size of 2^{24} values per array. This input size corresponds to array allocations of 128 MB each. The results indicate a similar behavior for the atomic implementation as described for Fibonacci and Powerset. The implementation using the `parallel for` construct uses no tasks and is provided for reference. As it is invariant to the number of tasks, its performance represents a horizontal line (*parallel-for-red*). Lastly, all other techniques perform similarly: Performance degrades when the number of tasks is too low to provide enough parallelism and when the number of tasks is too large with the resulting granularity being too fine. However, the amount of work per task is significantly higher compared to Powerset or Fibonacci, potentially underexposing some technique-specific performance variations. We have observed comparable performance for GCC/g++ on this benchmark.

The graph in Fig. 4 shows results for the Powerset benchmark using user-defined reductions with a constant number of 262k tasks for LLVM/Clang and GCC/g++. Results indicate that all techniques except *stack* are invariant to the size of the reduction variable, and thus the cost of memory allocation is equal. For LLVM/Clang, the implementation using *stack* degrades in performance with increasing type sizes due to increasingly distant memory accesses for stack operations. Results for GNU/g++ resemble performance results shown in Fig. 2b corresponding to 262k tasks: In both those results and the results for UDRs of all sizes, tasking overheads dominate.

To summarize, the performance of task reductions is determined by the task granularity and task count, by properties of a reduction technique and by its implementation in the runtime system.

Techniques available prior to the support of task reductions in OpenMP vary significantly in performance. For higher degrees of concurrency and frequent accesses to the reduction variable, atomics achieve the lowest performance. The use of stack-local variables requires task synchronization and relies on efficient task management in the runtime. Finally, manual privatization using `threadprivate` variables requires a final reduction of all private copies once the reduction completes. If the final reduction incurs additional overhead, performance degrades.

Task reduction support in OpenMP using the `parallel` and `taskgroup` constructs exhibits performance asymptotic to the *threadprivate* version, suggesting that both LLVM/Clang and GCC/g++ internally use per-thread privatization with tasks acquiring and reusing such thread-private allocations. In this case, internal optimizations can raise the performance beyond that of manual privatization coded at user level. In particular, the implementation does not need to expose OpenMP semantics for its internal mechanism used to combine results and may employ a more sophisticated reduction of thread-private copies such as an in-line parallel tree based reduction.

The next section examines implementations of task reduction in both LLVM/Clang and GCC/g++, as well as relevant differences in their general task management approaches.

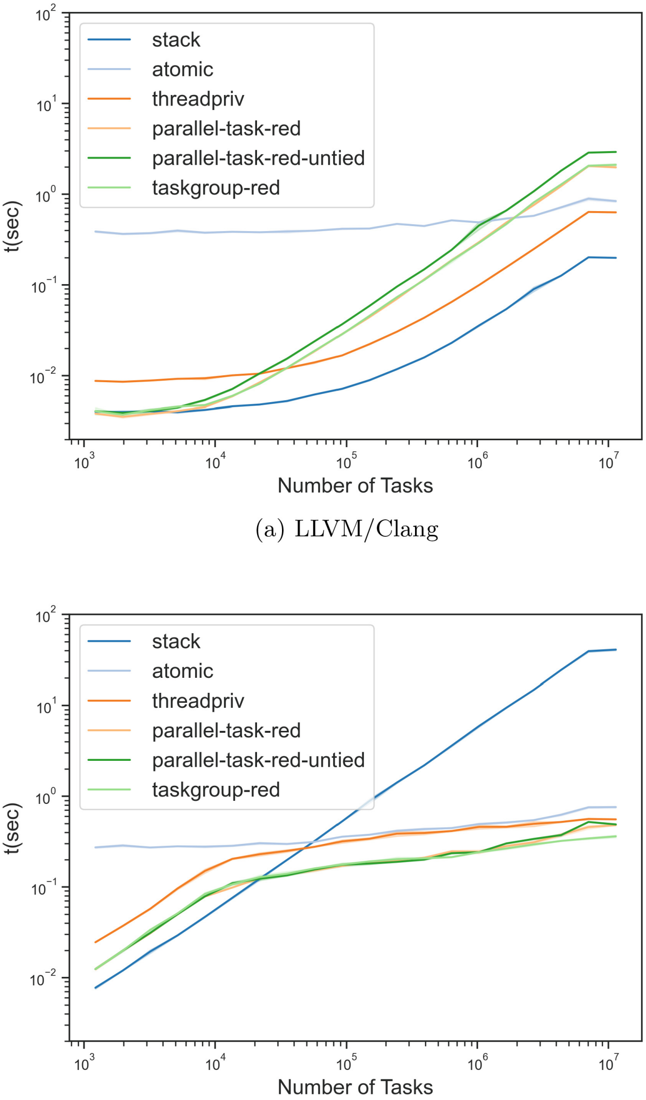
6 Implementations in GCC and LLVM/Clang

The understanding of performance characteristics described in the previous section requires inspection of tasking and task reduction support in the front-end compiler as well as the runtime. Of particular interest is the implementation of memory privatization and whether it occurs on thread or task level.

Listing 1.5 shows example code for a task participating in a task reduction. Listing 1.6 shows the corresponding intermediate code representation produced by the GCC/g++ (compiler *-fdump-tree-optimized*). Accesses to the original memory location are redirected to a new memory location obtained by calling *__builtin_GOMP_tas_reduction_remap*. This function obtains the associated thread-private memory location corresponding to the reduction variable registered by the reduction clause.

```
void func(int &sum) {
#pragma omp task in_reduction(+ : sum)
    sum++;
}
```

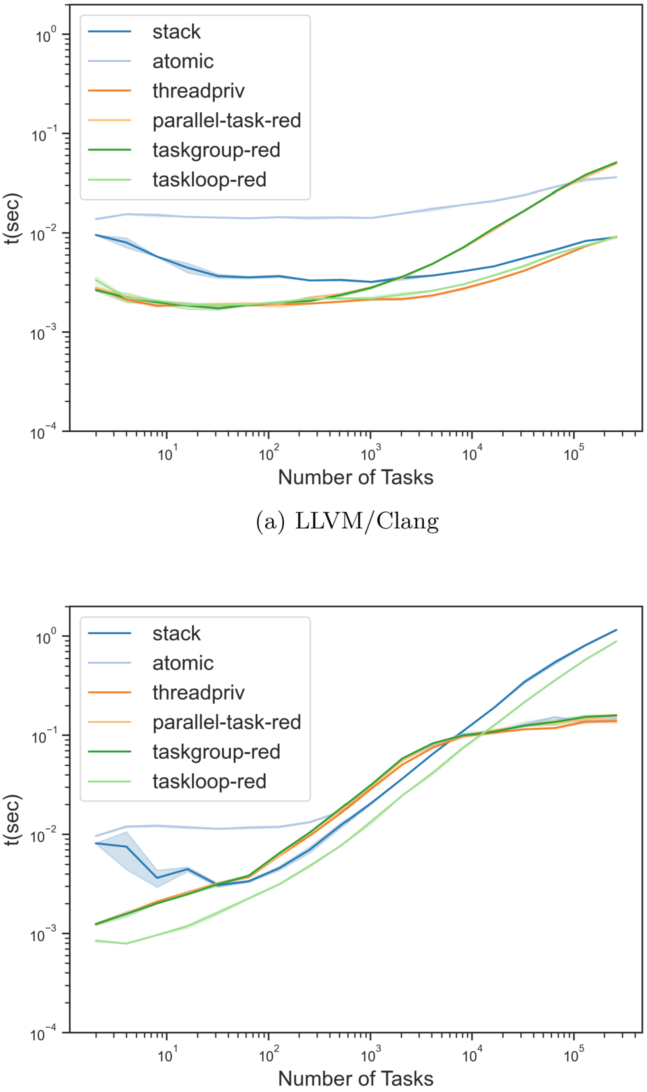
Listing 1.5. Sample code for a task participating in a reduction



(a) LLVM/Clang

(b) GCC/g++

Fig. 1. Fibonacci computation with a constant problem size of $N = 33$, 48 threads and a variable task cutoff resulting in a range of 1.2k–11405k tasks, showing differences between compilers and techniques



(a) LLVM/Clang

(b) GCC/g++

Fig. 2. Powerset computation with a constant problem size of $N = 18$, 48 threads and a variable cutoff with a range of 2–262k tasks, compiled with both compilers

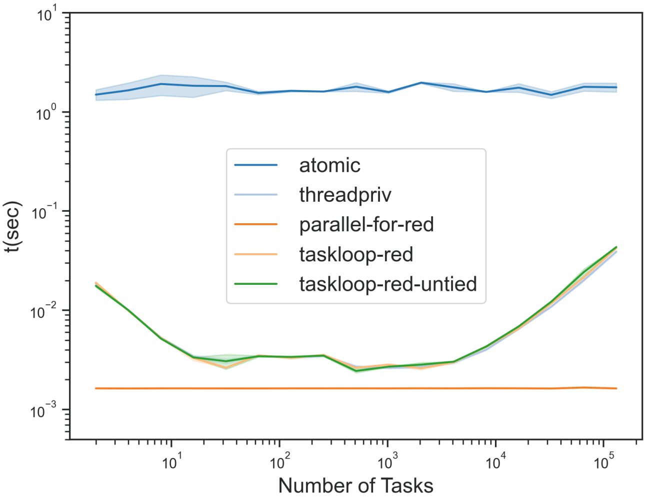


Fig. 3. Dot-product compiled with LLVM/Clang with a constant problem size of $N = 2^{24}$, 48 threads and a variable cutoff resulting in a range of 2–131k tasks

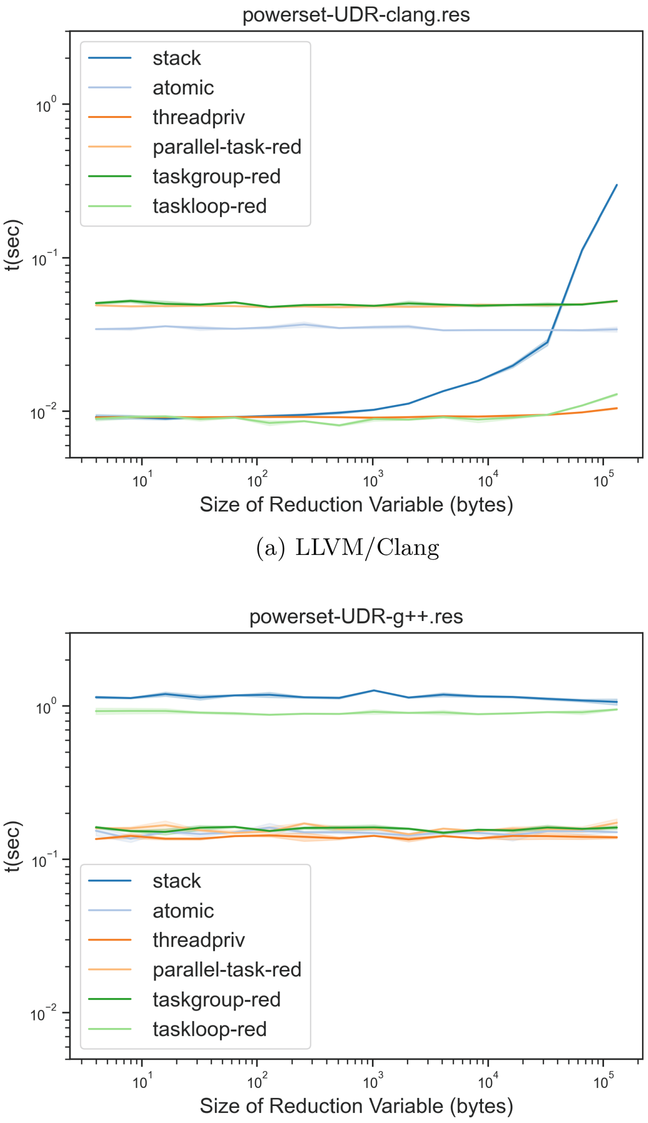
```
void func (int & sum) {
  struct .omp_data_s.0 .omp_data_o.1;
  ...
  .omp_data_o.1.sum = sum_2(D);
  __builtin_GOMP_task (_Z4funcRi._omp_fn.0, &.omp_data_o.1, 0B, 8, 8, 1,
     0, 0B, 0, 0B);
  return;
}

void _Z4funcRi._omp_fn.0 (struct .omp_data_s.0 & restrict .omp_data_i) {
  ...
  void * D.2516[1];
  _3 = .omp_data_i_2(D)->sum;
  D.2516[0] = _3;
  __builtin_GOMP_task_reduction_remap (1, 0, &D.2516);
  sum_6 = D.2516[0];
  _10 = *sum_6;
  _11 = _10 + 1;
  *sum_6 = _11;
  return;
}
```

Listing 1.6. Intermediate code fragments generated by the GCC/g++ front-end compiler for the example code in Listing 1.5

LLVM/Clang supports task reductions through per-thread privatization as well. Similar to the approach in GCC, the intermediate code calls the function *__kmpc_task_reduction_get_th_data* to access the thread-private copy.

The overhead costs of task management are critical to performance both with and without task reductions. The use of per-thread task queues in the LLVM runtime contributes to lower task management costs compared to the



(a) LLVM/Clang

(b) GCC/g++

Fig. 4. Powerset with a constant problem size of $N = 18$, 48 threads, 262k tasks and variable reduction type with type size range of 4B–131KB

GCC runtime with its centralized queue that is shared among all threads in the team. High overhead costs particularly impact our *stack* benchmark versions that pass partial results through the call stack, because they require taskwait synchronizations that induce additional accesses to task queues in the runtime.

7 Related Work

Prior to the addition of task reductions in OpenMP 5.0, evaluation studies [2,4] of the proposed feature had been demonstrated using the Nanos runtime system[1] and Mercurium compiler[2] [1]. In addition to OpenMP tasking, they implement the OmpSs programming model[3] [5], a tasking-centric programming model with close ties to OpenMP and support for task reductions. Previous work also explored array reductions over OmpSs tasks [3]. User-defined reductions for OpenMP were proposed by Duran et al. [6]. Reductions in other task parallel languages and language extensions include X10/Habanero-Java phaser accumulators [11] and finish accumulators [10], as well as Cilk++ hyperobjects [7]. Blaze-Tasks is a C++17-based framework for task scheduling and reductions [9].

8 Conclusions and Future Work

Our study provides evidence that the task reduction features in OpenMP are well supported by GCC and LLVM/Clang today. Performance insights indicate that the use of the language features is meaningful and provides performance commensurate to efficient manually implemented reductions for reasonable task sizes. For reproducibility, we intend to make available the benchmarks, the scripts to build and run them, and the complete set of graphs, upon approval.

Key topics that warrant further investigation are performance differences among compilers and the efficiency of their support for `taskwait` synchronizations (stressed by our manually-coded stack-based reductions) and reductions on `taskloop` constructs. Further inspection of their implementations along with a deeper experimental evaluation is a subject for future work. Based on the results in this paper, our recommendation to users is that they can confidently employ the convenience and performance of task reductions for their OpenMP applications on multicore CPUs.

References

1. Balart, J., Duran, A., Gonzàlez, M., Martorell, X., Ayguadé, E., Labarta, J.: Nanos Mercurium: a research compiler for OpenMP. In: European Workshop on OpenMP (EWOMP 2004), pp. 103–109 (2004)
2. Ciesko, J., et al.: Task-parallel reductions in OpenMP and OmpSs. In: DeRose, L., de Supinski, B.R., Olivier, S.L., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2014. LNCS, vol. 8766, pp. 1–15. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11454-5_1
3. Ciesko, J., Mateo, S., Teruel, X., Martorell, X., Ayguadé, E., Labarta, J.: Supporting adaptive privatization techniques for irregular array reductions in task-parallel programming models. In: Maruyama, N., de Supinski, B.R., Wahib, M. (eds.) IWOMP 2016. LNCS, vol. 9903, pp. 336–349. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45550-1_24

[1] https://pm.bsc.es/nanox.

[2] https://pm.bsc.es/mcxx.

[3] https://pm.bsc.es/ompss.

4. Ciesko, J., et al.: Towards task-parallel reductions in OpenMP. In: Terboven, C., de Supinski, B.R., Reble, P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2015. LNCS, vol. 9342, pp. 189–201. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24595-9_14
5. Duran, A., et al.: OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Process. Lett. **21**(02), 173–193 (2011)
6. Duran, A., Ferrer, R., Klemm, M., de Supinski, B.R., Ayguadé, E.: A proposal for user-defined reductions in OpenMP. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., de Supinski, B.R. (eds.) IWOMP 2010. LNCS, vol. 6132, pp. 43–55. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13217-9_4
7. Frigo, M., Halpern, P., Leiserson, C.E., Lewin-Berlin, S.: Reducers and other Cilk++ hyperobjects. In: Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures (SPAA 2009), pp. 79–90. ACM, New York (2009)
8. OpenMP Architecture Review Board: OpenMP Application Programming Interface Version 5.0, November 2018. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf
9. Pirkelbauer, P., Wilson, A., Peterson, C., Dechev, D.: Blaze-Tasks: A framework for computing parallel reductions over tasks. ACM Trans. Archit. Code Optim. **15**(4), 1–25 (2019)
10. Shirako, J., Cavé, V., Zhao, J., Sarkar, V.: Finish accumulators: an efficient reduction construct for dynamic task parallelism. In: Kasahara, H., Kimura, K. (eds.) LCPC 2012. LNCS, vol. 7760, pp. 264–265. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37658-0_18
11. Shirako, J., Peixotto, D.M., Sarkar, V., Scherer, W.N.: Phaser accumulators: a new reduction construct for dynamic parallelism. In: IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2009), Rome, Italy, pp. 1–12. IEEE (2009)

Using Off-the-Shelf Hardware Transactional Memory to Implement Speculative While in OpenMP

Juan Salamanca(✉) and Alexandro Baldassin

Sao Paulo State University (Unesp), Rio Claro, SP, Brazil
{juan,alex}@rc.unesp.br

Abstract. This paper presents Speculative While (SWh), a technique that enables Speculative Task Execution (STE) in OpenMP to accelerate `while` loops marked by the proposed `while` construct and the `swh` clause. With SWh, the speculative tasks are generated by the OpenMP `task` construct in `while` loops (from linear algebra or goal finding algorithms) where control dependencies between iterations can be speculated. This paper also presents a detailed analysis of the application of Hardware Transactional Memory (HTM) support to implement Speculative While and describes a preliminary evaluation of SWh implementation using HTM. As a result, it provides evidence to support the performance benefits of using STE over HTM to parallelize some well-known benchmarks. Experimental results reveal that by implementing SWh over HTM, speed-ups of up to 1.8× can be obtained for the `Gauss-Seidel` benchmark.

Keywords: Speculative While · OpenMP · Transactional Memory

1 Introduction

Speculative Task Execution (STE) [17,19] is a technique that enables the speculation of dependencies between (OpenMP) tasks coming from a hot-code region and thus allowing them to be executed in parallel, achieving an acceleration of that region of code. This form of speculative parallelization can be generated in several ways, such as when the speculative tasks are generated from the partition of the iterations of a `for` loop (hot-code region) by the OpenMP `taskloop` construct; in that case, the speculated dependencies are of data type and between iterations (loop-carried). This form of STE is known as Speculative Taskloop (STL) [20] and can be implemented using Hardware Transactional Memory (HTM) and code transformations similar to how TLS has been implemented before on HTM (HTM-TLS) [16]. In this way, STL needs four primary features: conflict detection, speculative storage, transaction rollback, and ordered

This work is supported by the Sao Paulo Research Foundation (grants 18/07446-8, 20/01665-0, and 18/15519-5).

M. Klemm et al. (Eds.): IWOMP 2022, LNCS 13527, pp. 50–64, 2022.
https://doi.org/10.1007/978-3-031-15922-0_4

transactions. Commodity hardware through HTM already implements the first three of them; ordered transactions can be implemented in software.

There are certain problems (hot-code regions) where it is not possible to know in advance the number of iterations that the loop will have. In those cases, `while` statements are used to compute the problem solution and, generally, the execution of the next iteration $(i + 1)$ in this kind of loop depends on the values computed in the current iteration (i). This fact tremendously limits the parallel execution of the loop iterations on multicore architectures. Many of these loops are widely used for convergence in linear algebra or goal finding problems in graph algorithms [13]. These loops run until a goal or threshold is reached, therefore the iteration space is unknown until the end and the execution must be sequential. In practice these loops execute for many iterations before their termination; however, the compiler needs to be conservative and it cannot parallelize these loops.

In order to overcome this limitation, this paper presents another form of STE called Speculative While (SWh), which unlike STL does not speculate data dependencies but control dependencies and has a `while` loop as the hot-code region. Thus, the tasks that are inside the loop body are speculated for several iterations. These tasks may have loop-carried data dependencies but these must be resolved through the OpenMP `depend` clause. The type of `while` loop for which SWh can be applied does not have a defined final boundary and the stop condition depends on the result of the tasks of the current iteration i. Therefore there is a control dependency from iteration i to iteration $i + 1$, from $i + 1$ to $i + 2$, and so on; then, it is possible to speculate the generation of all the tasks of different iterations and it is up to the runtime to resolve the loop-carried dependencies of these tasks of different iterations to be executed. However, if on iteration j the `while` loop's stop condition is set to true, all tasks speculated on iterations greater than j must be aborted and roll-backed. In the event that the tasks do not have data dependencies between iterations, tasks from different iterations can be executed simultaneously, accelerating the execution.

For instance, Fig. 1a shows a `while` statement where three tasks are generated for each iteration; however, the generation of tasks is paused by a `taskwait` at the end of each iteration (Line 11) because the condition `cond` depends on the termination of function `h`. Therefore, the execution flow of tasks is similar to the one shown in Fig. 1c, where the work for each iteration is serialized. This paper proposes the use of a new construct called `while`, which allows the use of STE to generate and speculate tasks of iterations where the condition has not yet been evaluated. As shown in Fig. 1b, the `while` construct is used instead of `taskwait` (Line 3), thus the next iterations create speculative tasks. Speculative tasks of different iterations are executed in parallel as shown in Fig. 1d, and they are committed when the variable `cond` is true for the previous iteration. When a task that executes function `h` for some iteration evaluates `cond` to `false`, it aborts all tasks that are waiting to commit.

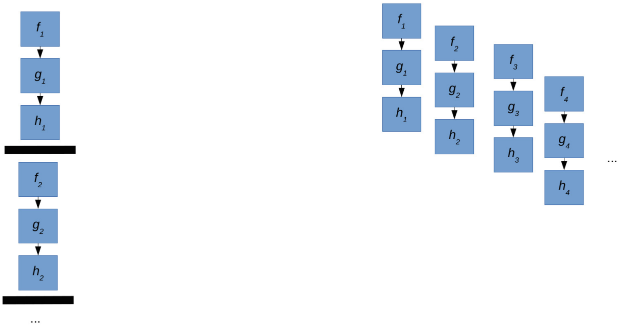
The key use of SWh is to speculate tasks with control dependencies on the stop condition whose value is probably invariant for many iterations. SWh, in a

```
#pragma omp parallel{
#pragma omp single{
      while(cond) {
        a=pop_queue();
        #pragma omp task depend(in:a)
              depend(out:b)
              firstprivate(a) shared(b)
        f(a,b);
        #pragma omp task depend(in:b)
              depend(out:c) shared(b,c)
        g(b,c);
        #pragma omp task depend(in:c)
              depend(out:cond,d)
              shared(c,d,cond)
        h(c,d, cond);
        #pragma omp taskwait
      }
      i(d);
}
}
```

(a) Using `taskwait` clause

```
#pragma omp parallel{
#pragma omp single{
      #pragma omp while swh(cond,iter)
      while(cond) {
        a=pop_queue();
        #pragma omp task depend(in:a)
              depend (out:b)
        f(a,b);
        #pragma omp task depend(in:b)
              depend (out:c)
        g(b,c);
        #pragma omp task depend(in:c)
              depend (out:cond,d)
        h(c,d, cond);
        iter++;
      }
      i(d);
}
}
```

(b) Using the proposed `while` construct.



(c) Using `taskwait`

(d) Using `while`

Fig. 1. Example of loop parallelization with tasks and possible flow executions

similar way to STL, needs mechanisms that support conflict detection, speculative storage, rollback of transactions, and ordered transactions. Current commodity off-the-shelf microprocessors provide support for speculation by means of hardware transactions [6,7]. In the same spirit of STL, this work proposes a novel utility for HTM enabling the implementation of three key features required by SWh: (a) conflict detection; (b) speculative storage; and (c) transaction rollback. The optimistic approach to parallelism in SWh is similar to the execution of transactions in HTM. However, in SWh, when tasks are generated within the Speculative-While construct, the data dependencies of tasks between iterations or in the same iteration are resolved by the runtime using the annotations of the `depend` clause. Furthermore, the control dependencies of each iteration with respect to the stop condition must also be respected implementing ordered transactions (in-order-transaction commit). In contrast, HTM transactions can execute in any order and in-order-commit feature is not provided.

In this paper we make the following contributions:

- We propose Speculative While (SWh) in OpenMP—the novel `while` construct and `ste` clause—to parallelize iterations speculating control dependencies of tasks through HTM's speculative support in `while` loops, which are widely used, for instance, in algorithms to determine convergence in linear algebra;
- We describe an algorithm to implement Speculative While using Hardware Transactional Memory and code transformations, enabling the speculative generation and execution of tasks from multiple `while`-loop iterations (Sect. 4);
- We evaluate SWh performance using the `Gauss-Seidel` and `Jacobi` benchmarks. We further compare against the parallelization of these benchmarks using only the `task` construct and the `depend` clause from standard OpenMP. The experimental results are promising with an average increase in the speedup of 1.36× in `Gauss-Seidel` when compared to the non-speculative version using OpenMP tasks (Sect. 6).

This paper is organized as follows. Section 2 describes the background material. Section 3 discusses related works. Section 4 details the design and implementation of Speculative While on HTM. Benchmarks, methodology and settings are described in Sect. 5. Section 6 presents the experimental evaluation for the implementation of SWh over HTM along with an analysis of the preliminary results. Finally, Sect. 7 concludes the work.

2 Background

This section explains fundamental concepts to understand the paper: Transactional Memory, Task-based Parallelism, and Thread-Level Speculation on Hardware Transactional Memory.

2.1 Transactional Memory

Transactional Memory (TM) uses the concept of transactions, borrowed from the Database community, to provide atomic and isolated updates to volatile memory (DRAM). Implementing transactions require devising a version management and a conflict detection scheme. Version management decides where new (speculative) and old data are stored. In *eager versioning*, the speculative data is stored in place, while the old one is kept in an internal undo-log. On the other hand, *lazy versioning* does not update the original data, storing the speculative one in a kind of internal write buffer. Conflict detection determines whether two operations executed in separate transactions cause a conflict, *i.e.*, if they access a common memory location and at least one of the operations is a write. Conflict detection can be eager (detection is done immediately when the conflict occurs) or lazy (detection is done when transactions attempt to commit) [8]. A conflict causes at least one of the transactions involved in the conflict to abort and it may re-execute, but other actions could also be carried out to support a conflict-resolution policy. Resolution can happen eagerly when the conflict occurs or lazily when the transaction attempts to commit.

TM can be implemented in hardware (HTM) [5] or software (STM) [21]. HTMs are easier to use because programmers only need to specify the start and the end of a transaction. STM systems can have a large overhead because conflict detection is performed in software. On the other hand, STMs have the advantage that they can be executed on any available hardware, and in principle have no limit on the amount of speculative state that a transaction may use. *Hybrid Transactional Memory* (HyTM) is an approach to implementing TM in software so that it can use best-effort HTM to boost performance but it does not depend on HTM. This approach exploits HTM if it is available to achieve hardware performance for transactions that do not exceed the HTM's limitations [3].

Intel TSX-NI. The Intel *Transactional Synchronization Extensions New Instructions* (Intel TSX-NI) was introduced in the 6th generation of the Intel Core processor [6]. Intel TSX-NI provides developers an instruction-set interface to specify transactional execution with two software interfaces: *Hardware Lock Elision* (HLE) and *Restricted Transactional Memory* (RTM). The RTM is an instruction-set extension that includes the instructions `xbegin`, `xend`, and `xabort`. When a transaction aborts, the state of the program immediately before the `xbegin` instruction is recovered, all speculatively written data are dismissed, and the values stored in registers are rolled back to their values prior to the transaction. The execution restarts at a program point specified by the address given as argument to the `xbegin` instruction. Data written transactionally are not visible to other transactions until the transaction commits by executing the `xend` instruction.

Intel TSX-NI does not guarantee that a conflict-free transaction will commit. Aborts may be caused by excess transactional reads or writes, conflicts due to false sharing, and instructions that cause aborts (*e.g.*, system calls) [26]. All data conflicts are detected at the granularity of the 64-byte cache line because the implementation of TSX uses the L1 data cache to track transactional states using physical addresses and the cache coherence protocol.

2.2 Task-Based Parallelism

Using tasks, the execution can be modeled as a directed acyclic graph, where nodes are tasks and edges define data dependencies between tasks. A runtime system schedules tasks whose dependencies are resolved over available worker threads, thus enabling load balancing and work stealing [4].

StarSs. StarSs [12] is a programming model which supports out-of-order execution of tasks by enabling the programmer to identify data dependencies between tasks through annotations of kernel functions as `input`, `output`, or `inout`. At runtime, the task creation code packs the kernel code pointer and the task operands, and then adds them to the task pipeline; in this way, the generating thread can continue creating additional tasks. The pipeline decodes task dependencies, generates the dependence graph using this information, and schedules tasks when they are ready [4].

Tasks in OpenMP. Tasks in OpenMP are blocks of code that the compiler packs and arranges to be executed in parallel. Tasks were added to OpenMP in version 3.0 [1]. In OpenMP 4.0 [9], the `depend` clause and the `taskgroup` construct were incorporated and, in OpenMP 4.5, the `taskloop` construct was proposed and added to the specification [10]. Like worksharing constructs, tasks are generally created inside of a `parallel` region. To spawn each task once, the `single` or `master` constructs are used. The ordering of tasks is not defined, but there are ways to express it: (a) with directives such as `taskgroup` or `taskwait`; or (b) with task dependencies (`depend` clause).

Variables that are used in tasks can be specified with data-sharing attribute clauses (`private`, `firstprivate`, `shared`, etc.) or, by default, data accessed by a task is `shared`. The `depend` clause takes a type (`in`, `out`, or `inout`) followed by a variable or a list of variables. These types establish an order between sibling tasks. The `taskwait` clause waits for the child tasks of the current task. `taskgroup` is similar to `taskwait` but it waits for all descendant tasks created in the block. Moreover, task `reduction` was introduced in OpenMP 5.0 [11].

2.3 Thread-Level Speculation on Hardware Transactional Memory (HTM-TLS)

When a compiler cannot prove that a loop can be executed in parallel (DOALL) but it can estimate with a high probability that the loop iterations will be independent at runtime, it can schedule the parallel execution of the loop speculatively. A mechanism is then necessary to detect when a dependency does occur at runtime (Speculative DOACROSS) and to re-execute the loop iterations that were miss-speculated. This technique is known as Thread-Level Speculation. TLS has been widely studied in the past [22–24]. For performance, TLS requires hardware mechanisms that support four primary features: conflict detection, speculative storage, in-order commit of transactions, and transaction roll-back. However, to this day, there is no off-the-shelf processor that provides direct support for TLS. Speculative execution is supported, however, in the form of Hardware Transactional Memory (HTM) available in processors such as the Intel Core and the IBM POWER [16]. HTM implements three out of the four key features required by TLS: conflict detection, speculative storage, and transaction rollback. And thus these architectures have the potential to be used to implement TLS (HTM-TLS) [16]. Speculative While is based on this approach.

3 Related Works

Rauchweger *et al.* proposed a general framework for the automatic transformation of any `while` loop for parallel execution [14]. Their strategy is to evaluate in parallel the recurrences (statically identified) and speculatively execute the remainder concurrently. If necessary, the approach undoes the effects of any iterations that overshot the termination condition. The methods can even be applied to loops whose data dependency relations cannot be analyzed at compile time. Experimental results on loops from Perfect benchmarks show that this technique can yield significant speed-ups.

Gayatri *et al.* proposed the speculative generation and possible execution of loop iterations ahead of time to overcome the problem of parallelizing `while` loops because of the unknown number of iterations that heavily restricts parallelism [4]. They optimistically predict the execution of the future iterations of the loop based on the fact that such loops execute for multiple iterations before finishing. They propose a technique to speculatively create parallel tasks from the next iterations before the current one completes using STM. Their results indicated an average speed-up of around 1.2× with 16 threads when compared to non-speculative parallel execution of the applications. However, they require significant changes to the StarSs runtime to support speculative tasks using STM. On the other hand, our approach with SWh does not make major changes to the OpenMP runtime because speculation is done through code transformations via HTM.

Azuelos *et al.* argued that speculation leads to increased parallelism in the coarse-grain task dataflow paradigm [2]. They show how simple language additions can allow programmers to make use of speculation. They specify a set of additions to the OmpSs language and the changes required in its runtime environment. Their evaluation using a simple benchmark leads to a promising 10% speed-up.

Salamanca *et al.* proposed adding Hardware-Transactional-Memory-based TLS (HTM-TLS) to `taskloop` through the clause `tls` in a previous work [19]. This clause can be used to speculate about data dependencies between tasks generated by a `taskloop` construct in non-DOALL loops, thus STL manipulates multiple tasks of loop iterations in order to exploit task parallelism (load balancing, work stealing, efficient creation of `parallel`, etc.) and to accelerate the execution of *may*-DOACROSS loops [19,20].

4 Speculative While (SWh)

Usually, when OpenMP tasks are used to parallelize `while` loops where the execution of the next iteration $(i + 1)$ depends on the values computed in the current iteration (i), a task synchronization construct (`taskwait` or `taskgroup`) is placed at the end of the iteration, serializing the execution of the `while` loop. For instance, Fig. 2a shows the parallelization of the hottest `Gauss-Seidel`'s loop using the OpenMP `task` construct, the `depend` clause for `task`, and the `taskgroup` construct. `taskgroup` is used to synchronize the result of `diff` so that it can be compared with `TOL` (Line 20) to decide about the next iteration. Notice that the `task_reduction` clause (proposed in OpenMP 5.0 [11]) is also used to calculate `diff`.

```
done=iter=0;
#pragma omp parallel num_threads(num_ths)
#pragma omp single
while(!done)){ //Gauss-Seidel's loop
 diff = 0;
 #pragma omp taskgroup task_reduction(max:diff)
 {
  for (i = 1; i<n-BS; i+=BS)
   for (j = 1; j<m-BS; j+=BS)
    #pragma omp task in_reduction(max:diff)
          depend(in:(*mat)[i+BS:1][j:BS],
          (*mat)[i-1:1][j:BS], ...)
          depend(inout: (*mat)[i:BS][j:BS])
          shared(mat) firstprivate(iter,...)
    {
     for (ii=i; ii<i+BS; ii++)
      for (jj=j; jj<j+BS; jj++){
       temp = (*mat)[ii][jj];
               (*mat)[ii][jj]= 0.25*(
                     (*mat)[ii][jj-1]+
                     (*mat)[ii-1][jj]+
                     (*mat)[ii][jj+1]+
                     (*mat)[ii+1][jj]);
       diff = fmaxf(diff,
             fabs((*mat)[ii][jj]-temp));
      }
    }
 }//barrier
 if (diff< TOL)
  done = 1;

 iter++;
}
```

(a) Parallelization using `task`/`taskgroup`

```
done=iter=0;
#pragma omp parallel
      num_threads(num_ths)
#pragma omp single
#pragma omp while swh(done,iter)
      spec_reduction(max:diff)
while(!done)){ //Gauss-Seidel's loop
 diff = 0;
 for (i=1; i<n-BS; i+=BS)
  for (j=1; j<m-BS; j+=BS)
   #pragma omp task
         in_reduction(max:diff)
         depend(out:diff)
         depend(in:(*mat)[i+BS:1][j:BS],
         (*mat)[i-1:1][j:BS], ...)
         depend(inout:
         (*mat)[i:BS][j:BS]) shared(mat)
         firstprivate(iter,...)
   {
    ... //same code
   }

 #pragma omp task spec_private(done)
       depend(in:diff) shared(diff)
 { //to set the stop condition
  if (diff< TOL){
   done = 1;
   #pragma omp tls if_write(done)
  }
 }

 iter++;
}
```

(b) Parallelization using SWh

Fig. 2. `Gauss-Seidel`'s loop

We propose another way to parallelize this type of `while` loops using Speculative Task Execution (STE) called Speculative While (SWh). SWh differs from Speculative Taskloop (STL) in that the loop iterations are speculated on control rather than data dependencies. SWh generates tasks of the next iterations speculatively and, if these iterations are only control-dependent, tasks of different iterations can be executed simultaneously. The mechanism relies on the fact that the tasks that were created in iterations greater than i are rolled back when the stop condition in a task of iteration i is `true`. Moreover, if the iterations have loop-carried dependencies, these must be respected by the OpenMP runtime through the `depend` clause, but the execution of iteration i can still be overlapped with the generation of iterations $i + 1$, $i + 2$, etc.

4.1 The `while` Construct and the `swh` Clause

Speculative While is implemented over OpenMP using HTM through the `while` construct and the `swh` clause. The design of the construct is as follows:

```
#pragma omp while swh(stop_ cond,ind_ var)  [clause]
    while-loop
```

where:

- *stop_ cond* is the variable to stop the `while` loop;
- *ind_ var* is the induction variable of the `while` loop;
- *clause* can be `spec_reduction`.

Figure 2b shows the parallelization of `Gauss-Seidel`'s loop using the proposed construct and clauses. Notice that we create a new task (Lines 14–20 of Fig. 2b) to set the stop condition (in the example, variable `done`) because the barrier that we use in Fig. 2a is removed to speculate iterations. Also, we set a dependency on `diff` between tasks, marking `depend(out:diff)` in the first tasks and `depend(in:diff)` in the last task. The creation of this last task is mandatory when using SWh and it will have a comparison (e.g., `diff < TOL`) to set the value of the stop-condition variable. To improve performance and remove false dependencies, we reuse code transformations for STL [17,18], such as `spec_private` and the `tls` construct. Finally, we mark `diff` as `spec_reduction` since `task_reduction` cannot be used because `taskgroup` is removed. The design of `spec_reduction` is explained next.

The `spec_reduction` Clause for the `while` Construct. This clause is necessary to improve the SWh-parallelization performance when the `while` loop has variables with a reduction pattern in speculative tasks inside the `while` construct (for the example, `diff` in Fig. 2). In this case, the standard OpenMP `task_reduction` cannot be used because this clause needs a barrier to wait for tasks within the loop (e.g., `taskgroup`) but the SWh mechanism does not allow a barrier because it is speculative. Another alternative would be to use the `reduction` clause of the `parallel` construct; however, in the `while` construct, we need the value of the reduction for each iteration of the `while` loop and, using `parallel`, we can only get the value of the reduction at the end of this construct. The use of the `spec_reduction` clause is possible in `while` constructs when the clause `swh` is present. It is syntactically similar to the standard `task_reduction` clause. The syntax of the `while` construct with `spec_reduction` is as follows:

```
#pragma omp while swh(stop_ cond,ind_ var) spec_reduction(reduction-
    identifier: list)
  while-loop
```

where:

- *reduction-identifier* is one of the following operators: `+`, `-`, `*`, `&`, `|`, `^`, `&&`, `||`, `max`, and `min`;
- *list* consists of a collection of one or more scalar variables separated by commas.

4.2 Implementing Speculative While (SWh) on HTM

We implement Speculative While using HTM and similar code transformations as proposed in HTM-TLS [15,16], FOR-TLS [16,18], and STL [17,19]. However, an important difference between SWh and the techniques mentioned is that the

Algorithm 1: Mechanism for Speculative While

```
Data: while construct (directive $D$ and while-loop $L$), $stop\_condition$, and $induction\_var$
Result: Transformed code to be parallelized with SWh on HTM
Create BEGIN and END functions;
Outside of the construct, create a new variable $next$ whose identifier is next of the same type
  of the induction variable;
Initialize $next$ with the initial value of $induction\_var$;
foreach $task \in L.task\_list$ do
    Set $induction\_var$ as firstprivate in $task$;
    Set $next$ as shared in $task$;
    Create a new variable $spec$ whose identifier is spec of type char;
    Create a statement $st\_begin$ to attribute to $spec$ the value returned by the call to the
      BEGIN function;
    Insert $st\_begin$ before $task.body$;
    Insert a call to the END function after $task.body$;
    if $task.spec\_private\_list \neq$ NULL then
        foreach scalar $var \in task.spec\_private\_list$ do
            Run Spec_Private_Scalar_Algorithm;
    if $task \neq L.last\_task$ then
        Set $stop\_condition$ as shared in $task$;
    else
        Create a statement $st\_next$ to increment the value of $next$ by $L.step.value$;
        Create an if statement $if\_st$ with condition (!<$stop\_condition.id$>), and set
          $st\_next$ as the then-part;
        At the end of $task.body$, insert $st\_next$;
    At the end of $task.body$, insert a label $exit$ whose identifier is Exit;
    Create an goto statement $st\_goto$ to label $exit.id$;
    Create an if statement $st\_exit$ with condition (<$spec.id$>==-1), and set $st\_goto$ as the
      then-part;
    Insert $st\_exit$ after $st\_begin$;
if $D.spec\_reduction\_list \neq$ NULL then
    foreach scalar $var \in D.spec\_reduction\_list$ do
        Run Spec_Task_Reduction_Algorithm
```

transactions are started by each task in SWh and no longer by each iteration (or iteration strip) as in HTM-TLS. For instance, Fig. 3a shows a sketch of the Fig. 2b's code (SWh) converted to standard OpenMP with HTM intrinsics using Algorithm 1. Each speculative task in Fig. 3a has a BEGIN function inserted at the beginning and an END function inserted at the end.

Another difference with respect to STL is the creation of the **last task** inside the `while` construct in SWh. This is essential because this task sets the stop condition (Line 12 of Fig. 3a) and updates the variable `next` (Line 43 of Fig. 3a), which is used to implement **ordered transactions** at the END function. As shown in Fig. 3c, the variable `next` will be used by all speculative tasks to validate wether they have to commit or abort. Moreover, another important difference with respect to STL is that SWh, before starting the transaction, verifies that the stop-condition variable (`done` in the example) is not `true` at the BEGIN function (Line 5 of Fig. 3b). If it is `true`, the execution jumps to label `Exit`, without starting any transaction and finishing the task. `Spec_Private_Scalar_Algorithm` is explained in previous work [17].

`Spec_Task_Reduction_Algorithm` takes as input the `while` construct, the scalar *var*, the list of tasks marked with `in_red`, the induction variable *ind_ var*,

```
done=0;
iter=0;
int next=iter;
float diff_arr[MAX_ITER];

#pragma omp parallel num_threads(N_CORES)
#pragma omp single
while(!done)){
  diff_arr[iter] = 0;
  for (i=1; i<n-BS; i+=BS)
    for (j=1; j<m-BS; j+=BS)
      #pragma omp task depend(in:(*mat)[i+BS:1][j:BS], (*mat)[i-1:1][j:BS], ...)
            depend(inout: (*mat)[i:BS][j:BS]) shared(mat,next,done,diff_arr)
            firstprivate(iter,...) depend(out:diff_arr[iter])
      {
        float diffL=0;
        char spec= BEGIN(&next,iter,&done);
        if (spec==-1) goto Exit;
        for (ii=i; ii<i+BS; ii++)
          for (jj=j; jj<j+BS; jj++){
            temp = (*mat)[ii][jj];
            (*mat)[ii][jj]= 0.25*( (*mat)[ii][jj-1] + (*mat)[ii-1][jj] + (*mat)[ii][jj+1] +
                  (*mat)[ii+1][jj]);
            diffL= fmaxf(diffL, fabs((*mat)[ii][jj]-temp));
          }
        END(spec,&next,iter);
        diff_arr[iter]= fmaxf(diff_arr[iter],diffL);
        Exit:;
      }

  #pragma omp task depend(in:diff_arr[iter]) shared(next,done,diff_arr) firstprivate(iter)
  {
    char doneL;
    char flag_w_done=0;

    char spec=BEGIN(&next,iter,&done);
    if (spec==-1) goto Exit;

    if (diff_arr[iter]< TOL){
      doneL = 1;
      flag_w_done=1;
    }

    END(spec,&next,current);
    if (flag_w_done) done=doneL;
    if (!done) next++;
    Exit:;
  }

  iter++;
}
```

(a) Fig. 2b's code converted to standard OpenMP

```
char BEGIN(int *ptr_next, int current,
      char *ptr_done){
  char spec;
  unsigned int status;
  Retry:
  if ((*ptr_done)) return -1;
  if ((*ptr_next) != current){
    spec=1;
    status=_xbegin();
    if (status!=_XBEGIN_STARTED)
      goto Retry;
  }
  else spec=0;

  return spec;
}
```

(b) Function BEGIN in SWh

```
void END(char spec, int *ptr_next, int
      current) {
  if (spec) {
    if ((*ptr_next)!=current) xabort();
      xend();
  }
}
```

(c) Function END in SWh

Fig. 3. SWh Parallelization of `Gauss-Seidel`

the reduction operator *op_ red*, and the reduction statement *statement_ red*. Unlike the STL-spec_reduction clause [17], the speculative task reduction is applied to each `task` marked with the `in_reduction` clause (reused from standard `task_reduction`). Moreover, the algorithm creates an array *var_ arr* to replace the scalar variable that participates in the reductions (`diff_arr` is created to replace `diff` in Line 4 of Fig. 3a)[1]. This transformation is carried out because loop-carried dependencies could be generated in the speculative parallelization, although previously these did not exist because the generation of tasks was sequential due to the `taskgroup` construct. For each task marked with `in_reduction` for *var*, the algorithm sets *var_ arr* (`diff_arr` in the example) to `shared`. Then, it creates a local copy of *var* (`diffL` in the example) and initialize its value to the identity value of operator *op_ red* (Line 14 of Fig. 3a). This local copy replaces *var_ arr* at the position *ind_ var* (`diff_arr[iter]`) in the reduction pattern (*statement_ red*) as shown in Line 21 of Fig. 3a. Finally, after committing, it accumulates the partial results in the `shared` variable using *op_ red* as shown in the Line 24 of the example.

5 Benchmarks, Methodology and Experimental Setup

The performance assessment in this work reports speed-ups and abort/commit ratios (transaction outcome) for the SWh (Speculative While) parallelization and speed-ups for the `task-depend` (non-speculative) parallelization of the hottest `while` loops from the `Gauss-Seidel` and `Jacobi` problems running on Intel Core. For all experiments, a random entry of a matrix is used. The baseline for speed-up comparisons is the serial execution of the same benchmark program compiled at the same optimization level. Whole-program times are used to calculate speed-ups. Each software thread is bounded to a unique core. Each benchmark was run twenty times and the average time is used. Runtime variations were negligible and are not presented. We made manual code transformations to the evaluated loops following the algorithms described in Sect. 4, thus obtaining the SWh parallelization of the benchmarks.

Both benchmarks are linear algebra applications:

- **Gauss-Seidel** is a technique for solving n equations of a linear system of equations, $Ax = b$. The equation coefficients are improved in every iteration using the previous solution, $x^{(k+1)} = L^{-1}(b - Ux^{(k)})$, where L is the lower triangular component of A and U is the strictly upper triangular component. The algorithm runs until the absolute approximate error is less than a tolerance. Thus, at the end of every iteration, the convergence is checked.
- **Jacobi** is also an iterative technique for determining the solutions of a strictly diagonally dominant system of linear equations. The solution is then obtained iteratively via $x^{(k+1)} = D^{-1}(b - Rx^{(k)})$, where D is the diagonal component of A and R is the remainder. Unlike the Gauss-Seidel method, we cannot overwrite $x^{(k)}$ with $x^{(k+1)}$, as those values will be needed by the rest of the computation.

[1] This technique is called Scalar Expansion.

The benchmarks were compiled with Clang 12.0 [25] and the set of flags specified in each benchmark program. Code compiled with `clang -fopenmp` was linked against the `libomp12` runtime (with `monotonic` scheduling [20]). The problem size used was 64 for both benchmarks. To guarantee that each software thread is bound to a unique core, the environment variable `KMP_AFFINITY` was set to `granularity = fine,balanced`. This experimental evaluation was carried out on an Intel Core i7-6700HQ processor with 4 cores with 2-way SMT, running at 2.6 GHz, with 16 GB of memory on Ubuntu 18.04.5 LTS (GNU/Linux 4.15.0-139-generic x86_64). The cache-line prefetcher is enabled by default. Each core has a 32 KB L1 data cache and a 256 KB L2 unified cache. The four cores share a 6144 KB L3 cache.

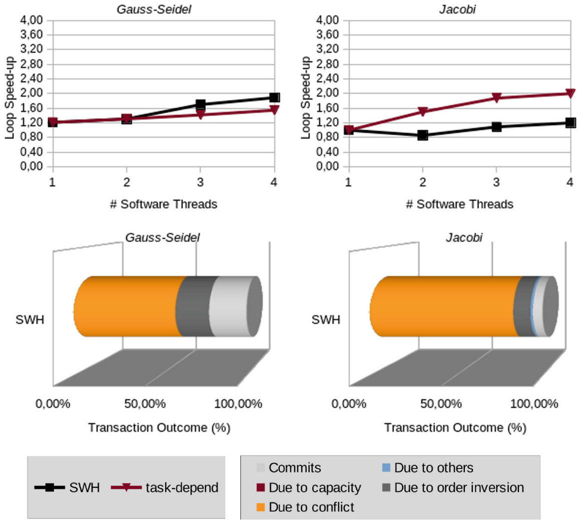


Fig. 4. Speed-ups and abort ratios (4 threads) for SWh and `task-depend` parallelizations on Intel Core

6 Experimental Results

This section presents results and analysis. We assess Speculative While on Gauss-Seidel and Jacobi methods using an Intel Core machine with HTM support. Figure 4 shows the speed-ups with respect to serial for SWh and `task-depend` parallelizations. Jacobi performance is worse using SWh because there is a copy of the array at the end of each iteration of the `while` loop which makes the loop more sequential in nature as the next iteration depends entirely on the array computed in the current iteration. This causes a greater number of violations of loop-carried dependencies in the speculative parallelization and consequently a greater number of aborts due to conflict. The gain that can be obtained from SWh is mainly due to being able to overlap the generation of tasks from some iterations with the execution of tasks from another iteration.

In the case of Gauss-Seidel, there are fewer loop-carried dependencies in the `while` loop, so it is also possible to overlap the execution of tasks of different iterations, thus achieving a higher commit rate and a lower number of aborts due to conflict. The SWh speed-up over `task-depend` is 1.22×. This speed-up is promising because this loop has loop-carried dependencies and there is a significant performance improvement in an already accelerated version (using the OpenMP standard).

7 Conclusions

This paper presents Speculative While (SWh), a technique that speculates about control dependencies in a `while` statement to execute in parallel many iterations and thus accelerating the loop execution. SWh is implemented in OpenMP through the `while` construct and clauses `swh` and `spec_reduction`. A first evaluation using the Gauss-Seidel and Jacobi methods shows promising results. In particular, as observed with Gauss-Seidel, SWh achieved an average increase in the speed-up of 1.22× compared to the non-speculative version using standard OpenMP tasks.

Acknowledgements. The authors would like to thank the anonymous reviewers for the insightful comments.

References

1. Ayguade, E., et al.: The design of OpenMP tasks. IEEE Trans. Parallel Distrib. Syst. (TPDS) **20**(3), 404–418 (2009)
2. Azuelos, N., Etsion, Y., Keidar, I., Zaks, A., Ayguadé, E.: Introducing speculative optimizations in task dataflow with language extensions and runtime support. In: 2012 Data-Flow Execution Models for Extreme Scale Computing, pp. 44–47, September 2012
3. Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., Nussbaum, D.: Hybrid transactional memory. In: International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), San Jose, California, USA, pp. 336–346. ACM (2006)
4. Gayatri, R., Badia, R.M., Ayguade, E.: Loop level speculation in a task based programming model. In: 20th Annual International Conference on High Performance Computing, pp. 39–48 (2013)
5. Herlihy, M., Moss, J.E.: Transactional memory: architectural support for lock-free data structures. In: International Conference on Computer Architecture (ISCA), San Diego, CA, USA, pp. 289–300, May 1993
6. Intel Corporation: Intel architecture instruction set extensions programming reference. Chapter 8: Intel transactional synchronization extensions (2015)
7. Le, H., et al.: Transactional memory support in the IBM POWER8 processor. IBM J. Res. Dev. **59**(1), 8:1–8:14 (2015)
8. Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D., Wood, D.A.: LogTM: Log-based transactional memory. In: High Performance Computer Architecture (HPCA), pp. 254–265 (2006)

9. OpenMP-ARB: OpenMP application program interface version 4.0 (2013)
10. OpenMP-ARB: OpenMP application program interface version 4.5 (2015)
11. OpenMP-ARB: OpenMP application program interface version 5.0 (2018)
12. Perez, J.M., Badia, R.M., Labarta, J.: A dependency-aware task-based programming environment for multi-core architectures. In: 2008 IEEE International Conference on Cluster Computing, Tsukuba, Japan, pp. 142–151 (2008)
13. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes 3rd Edition: The Art of Scientific Computing. Cambridge University Press, Cambridge (2007)
14. Rauchwerger, L., Padua, D.: Parallelizing while loops for multiprocessor systems. In: International Parallel Processing Symposium, pp. 347–356 (1995)
15. Salamanca, J., Amaral, J.N., Araujo, G.: Evaluating and improving thread-level speculation in hardware transactional memories. In: IEEE International Parallel and Distributed Processing Symposium (IPDPS), Chicago, USA, pp. 586–595 (2016)
16. Salamanca, J., Amaral, J.N., Araujo, G.: Using hardware-transactional-memory support to implement thread-level speculation. IEEE Trans. Parallel Distrib. Syst. **29**(2), 466–480 (2018)
17. Salamanca, J., Baldassin, A.: Evaluating the performance of speculative DOACROSS loop parallelization with taskloop. In: International Conference on High Performance Computing and Simulation (HPCS), Barcelona, Spain (2020)
18. Salamanca, J., Baldassin, A.: Using hardware transactional memory to implement speculative privatization in OpenMP. In: Chapman, B., Moreira, J. (eds.) LCPC 2020. LNTCS, vol. 13149, pp. 57–73. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-95953-1_5
19. Salamanca, J., Baldassin, A.: A proposal for supporting speculation in the OpenMP `taskloop` construct. In: Fan, X., de Supinski, B.R., Sinnen, O., Giacaman, N. (eds.) IWOMP 2019. LNCS, vol. 11718, pp. 246–261. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-28596-8_17
20. Salamanca, J., Baldassin, A.: Improving speculative `taskloop` in hardware transactional memory. In: McIntosh-Smith, S., de Supinski, B.R., Klinkenberg, J. (eds.) IWOMP 2021. LNCS, vol. 12870, pp. 3–17. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85262-7_1
21. Shavit, N., Touitou, D.: Software transactional memory. Distrib. Comput. **10**(2), 99–116 (1997). https://doi.org/10.1007/s004460050028
22. Sohi, G.S., Breach, S.E., Vijaykumar, T.N.: Multiscalar processors. In: International Symposium on Computer Architecture (ISCA), S. Margherita Ligure, Italy, pp. 414–425 (1995)
23. Steffan, J., Mowry, T.: The potential for using thread-level data speculation to facilitate automatic parallelization. In: High-Performance Computer Architecture (HPCA), Washington, USA, pp. 2–13 (1998)
24. Steffan, J.G., Colohan, C.B., Zhai, A., Mowry, T.C.: A scalable approach to thread-level speculation. In: International Conference on Computer Architecture (ISCA), Vancouver, British Columbia, Canada, pp. 1–12 (2000)
25. The LLVM Project: LLVM 12.0.0 (2021). https://github.com/llvm/llvm-project
26. Yoo, R.M., Hughes, C.J., Lai, K., Rajwar, R.: Performance evaluation of Intel transactional synchronization extensions for high-performance computing. In: International Conference on High Performance Computing, Networking, Storage and Analysis (SC), Denver, Colorado, pp. 19:1–19:11 (2013)

Effective Use of Advanced Heterogeneous Node Architectures

Towards Automatic OpenMP-Aware Utilization of Fast GPU Memory

Delaram Talaashrafi[1](✉), Marc Moreno Maza[1], and Johannes Doerfert[2]

[1] Western University, London, ON, Canada
dtalaash@uwo.ca, moreno@csd.uwo.ca
[2] Argonne National Laboratory, Lemont, IL, USA
jdoerfert@anl.gov

Abstract. OpenMP has supported target offloading since version 4.0, and LLVM/Clang supports its compilation and optimization. There have been several optimizing transformations in LLVM aiming to improve the performance of the offloaded region, especially for targeting GPUs. Although using the memory efficiently is essential for high performance on a GPU, there has not been much work done to automatically optimize memory transactions inside the target region at compile time.

In this work, we develop an inter-procedural LLVM transformation to improve the performance of OpenMP target regions by optimizing memory transactions. This transformation pass effectively prefetches some of the read-only input data to the *fast* shared memory via compile time code injection. Especially if there is reuse, accesses to shared memory far outpace global memory accesses. Consequently, our method can significantly improve performance if the right data is placed in shared memory.

Keywords: OpenMP · target offloading · GPU · shared memory · compiler optimization · LLVM/Clang

1 Introduction

On modern GPUs, the global memory is off-chip with high access latency. Therefore, using the global memory efficiently and reducing the number of transactions to/from it is essential to maximize a GPU's computation capability utilization. An alternative to global memory is *shared memory* which is limited on-chip and fast memory space. The shared memory is allocated for each block (or team in OpenMP terminology), and it can be used for optimizing a program in different ways, including *prefetching*. For prefetching, a programmer first utilizes the threads in a team to copy (most often read-only) global memory content into a shared memory buffer. Then, all accesses to these locations in the global memory are replaced with accesses to the prefetched data in the faster shared memory. This method is especially beneficial if the data is reused multiple times as each access is sped up while the initial copy costs are fixed.

M. Klemm et al. (Eds.): IWOMP 2022, LNCS 13527, pp. 67–80, 2022.
https://doi.org/10.1007/978-3-031-15922-0_5

In this work, we develop a compiler optimization technique to improve the performance of OpenMP programs containing device offloading regions by *automatically prefetching parts of the required data to the shared memory through code injected at compile time*. In the current version of OpenMP, runtime functions and directives exist to explicitly allocate and use memory in the shared space [4]. Also, the `OpenMPOpt` pass [14], developed as a part of the LLVM framework [15], implements different OpenMP-aware optimization techniques that utilize shared memory. These have proven to effectively improve the performance of a program's target regions. We leverage the `OpenMPOpt` pass infrastructure and the LLVM/OpenMP GPU runtime functions for allocating (dynamic) shared memory for our own optimization. By identifying suitable candidate memory regions and prefetching them into the shared memory buffer automatically, we can improve the program's performance as each original load from the global memory is now significantly faster served from shared memory instead.

For this paper, we only consider target regions consisting of a loop nest that exploits both the team and thread level parallelism of a GPU. In other words, the outer-most loop has to have an attached `distribute` and a `parallel for` construct. The method is developed as a part of the LLVM framework and operates on the program's intermediate representation (LLVM-IR), thus can be reused for Fortran programs as the LLVM/Flang frontend matures. In addition to the general scheme and results of our evaluation, we also discuss a method to use the limited space of the shared memory more efficiently and a method to reduce bank conflicts, since those can severely degrade performance. Our evaluation uses a simple matrix multiplication algorithm for simplicity. By comparing the running times of the kernel with and without our experimental optimization, we can observe a significant performance improvement potential that is likely to translate to more realistic programs too.

The rest of this paper is organized as follows: We first briefly introduce background information related to this work in Sect. 2. Then, we detail our method and implementation in Sect. 3. Further optimization techniques are described in Sect. 4. Our experimental results are reported and discussed in Sect. 5 before we conclude the paper in Sect. 6.

2 Background

In this section, we briefly explain some of the topics related to OpenMP target offloading and GPU programming models relevant to our work. In the context of this paper, and without loss of generality, we assume our target device is an NVIDIA GPU.

2.1 OpenMP Target Offloading Support in LLVM/Clang

OpenMP has supported device offloading since version 4.0 [3]. Compiling and optimizing OpenMP programs with target offload regions has been supported by LLVM/Clang since version 11 [2]. The primary approach for compiling programs with OpenMP constructs is outlining [11]; however, the compiler takes some extra stages to handle device offloading. To this end, the compiler first generates two separate modules, the *host module* for regions of the program running on the host and the *device module* for the offloaded region to run on the device. Therefore, the compilation flow of OpenMP programs with target offloading regions is different from programs with other constructs. For more details on the complete compilation flow of an OpenMP program with target offloading and GPU runtime, refer to [10,12,13,16]. In this work, we only need to manipulate the device module. Moreover, we focus on the *single program multiple data* (SPMD) execution mode and do not consider the generic mode [14].

To compile an OpenMP program with the `omp target` pragma, the Clang compiler outlines the target region to a *kernel* function and uses OpenMP runtime functions to call the kernel from the host module. Parallel loops in an OpenMP program are handled similarly. Clang first outlines the body of the parallel loop, the *parallel region*, to a function. We call this outlined function the *parallel region function*. Then, the compiler replaces the `parallel for` pragma with the call to an OpenMP runtime function, e.g., `kmpc_parallel`. This function takes a pointer to the parallel region function and all variables needed for the execution as inputs. The work-sharing loop logic is explicitly generated through more runtime calls placed inside the parallel region function. Distribution of iterations is based on the thread id initialized by the parallel region.

As explained in Sect. 1, we only consider kernels that expose two levels of parallelism on the outer-most loop level. To handle these kernels, Clang first distributes the iterations of the "distribute component" between teams based on the number of threads in each team. Then it parallelizes the execution of all iterations in each chunk assigned to a team by utilizing the team's threads.

Figure 1 shows the high-level structure of the kernel created for a target region with a single `distribute parallel for`. The compiler first inserts a call to the runtime function `kmpc_distribute_init` in the kernel to determine the lower and upper bounds of the chunks assigned to the team. After that, the compiler inserts a for loop in the kernel to iterate over the chunks. We call this loop the *distribute loop*. The compiler then inserts a call to the `kmpc_parallel` function

```
void target_region(/* inputs */)
{
  kmpc_distribute_init(/* loop bounds */);
  // begining of the distribute region
  foreach chunk { //<- distribute loop
    // parallel region (par_region) with enclosed
    // workshare (=for) loop executed by all threads
    kmpc_parallel(par_region, ...);
  }
  // end of the distribute region
}
```

Fig. 1. Compiler view of the kernels we consider in this paper.

in the distribute loop to distribute the iterations between threads of the teams. The inputs to this runtime call are the pointer to the parallel region function and its inputs, and the chunk of the work-sharing loop specified by each iteration of the distribute loop. For easier reference, we call the region of the kernel after the call to the `kmpc_distribute_static_init` function the *distribute region*.

The `OpenMPOpt` is an inter-procedural optimization (IPO) pass in LLVM, which is implemented for optimizing OpenMP GPU execution. This pass is enabled by default since LLVM 11 when compiling with `O2` and `O3` options. It first runs on the module and later it runs on the call graph of the program. It uses domain knowledge about OpenMP runtime calls to better optimize the LLVM-IR of the program.

2.2 CUDA Memory Hierarchy

While executing, GPU threads can have access to different memory spaces. All threads across all teams have access to the *global memory*. Each team of threads has access to the *shared memory*. Finally, all threads have private *local memory*.

In this work, our focus is on the shared memory. Shared memory is an on-chip memory; therefore, it is faster than global memory. There are two kinds of shared memory: static and dynamic. Static shared memory is used when the required size of the shared memory is known at compile time, and dynamic shared memory is used when this size is unknown at compile time.

A challenge while using the GPU's shared memory is to avoid *bank conflict*. The shared memory is managed in modules of equal size or memory banks. Different memory banks can be accessed simultaneously. However, multiple threads cannot access different locations in the same bank in parallel. Therefore, having multiple threads accessing the same memory bank causes the bank conflict problem, and the accesses will be serialized. More detailed information can be found in [1,8].

In OpenMP, `#pragma omp allocate(X)allocator(omp_pteam_mem_alloc)` is for allocating static shared memory, where `X` should be replaced by the variable's name in the shared memory.

With LLVM, the function `llvm_omp_target_dynamic_shared_alloc` returns a pointer to the beginning of the dynamic shared memory "allocated" with the `LIBOMPTARGET_SHARED_MEMORY_SIZE` environment variable. See: https://openmp.llvm.org/design/Runtimes.html#libomptarget-dynamic-shared.

3 Implementation

This section explains the method we use and our implementation details. We first explain the problem we are solving in more detail in Sect. 3.1. After that, Sects. 3.2, 3.3, and 3.4 explain our method and implementation phases for prefetching and retrieving the memory locations correctly and efficiently. Finally, in Sect. 3.5, we discuss an application of our method to load input arrays of small sizes to the static shared memory.

3.1 Problem Statement

Considering the arrays accessed in a parallel region, we can categorize their access relations in two cases: (1) they are a function of the work-sharing loop's induction variable, and (2) they do not depend on this variable. The former case is more challenging for prefetching because different teams access different locations of the input array. We propose a solution for prefetching an input array in the first case under some conditions; however, the method and the implementation can be extended to relax these limitations and also to handle the second case.

This work considers one of the input arrays to the kernel that is read in the parallel region in a loop for prefetching. That means the read access instruction is surrounded by a loop-nest of the depth of at least two, where the outer-most loop is the work-sharing loop. Also, the array's access relation is a function of the work-sharing loop's induction variable and the induction variable of one of the inner loops. We call that inner loop the *access loop*. Figure 2 shows the work-sharing loop, the access loop, and an eligible read access for prefetching (`v1`). We also assume there are no conditional branches in the target region, and the total number of available threads (number of teams multiplied by the number of threads per team) is equal to the number of iterations of the work-sharing loop. Furthermore, we assume the amount of shared memory usage per team does not exceed the shared space allocated for the program.

We perform shared memory prefetching in the distribute region before the distribute loop. That is before entering the parallel region. This way, we can prefetch those locations needed in each team to the shared memory before the threads begin computation. After that, in the parallel region, we access those locations from the shared memory instead of the global memory. We implement this procedure as a part of `OpenMPOpt` pass of LLVM, and if activated, it executes as a part of the `O3` compiler optimization passes.

3.2 Finding Memory Locations to Prefetch

The first step is to find the memory locations that we want to prefetch for each team. In fact, our goal in this phase is to find all the global memory locations of the considered array that are read by the threads of each team in the input

```
#pragma omp target teams map(to:v1[0:N*M])
{
   #pragma omp distribute parallel for
      for (int i=0; i<N; i++)  //-->work-sharing loop
         for(int j=0; j<N; j++)
            for(int k=0; k<M; k++)  //-->access loop
               sum += v1[i*M+k] * 3;
               //        /\-->eligible access for prefetching
}
```

Fig. 2. Example of the supported read access.

program. To find these memory locations, we need to have (1) the chunks of the work-sharing loop assigned to each team and (2) the memory locations accessed in each iteration of the chunks. Then, we get all the accessed locations by iterating over all locations in each iteration assigned to the team.

As we explained in Sect. 1, the compiler inserts a call to the runtime function `kmpc_distribute_init` in the kernel to get the lower (`team_LB`) and upper (`team_UB`) bounds of the chunks of the work-sharing loop assigned to each team. Therefore, we can access their values after this function call in the kernel (after line 3 in Fig. 1).

For each team, we want to find the locations accessed in all iterations `i` of the work-sharing loop, where `i` is between `team_LB` and `team_UB`. For the kernels we consider, we can find these memory locations by having the integer values of the first location of the array that is accessed in iteration `i` ($\mathtt{Base_i}$), the distance between two consecutive accesses in iteration `i` ($\mathtt{Step_i}$), and the number of accessed locations by iteration `i` ($\mathtt{Number_i}$). Having these numbers, we can find all the accessed memory locations in iteration `i` by computing:

$$(\mathtt{Base_i} + \mathtt{k} \times \mathtt{Step_i}), \mathtt{0} \leq \mathtt{k} < \mathtt{Number_i}. \tag{1}$$

The thread that executes iteration `i` of the work-sharing loop reads $\mathtt{Number_i}$ locations of the array, starting from $\mathtt{Base_i}$, and the difference between two indexes it accesses consecutively is $\mathtt{Step_i}$.

We apply the LLVM scalar evolution (`scev`) [5] analysis pass to the parallel region function to get these values. Using the result of `scev` analysis, we can get the required information as objects of the LLVM `Value` class [9]. To be more precise, we begin by getting the result of scalar evolution of the array's access relation (using the `getSCEV` function) in the parallel region function and casting it to `SCEVAddRecExpr` [6]. After that, we call the `getStart` and `getStepRecurrence` functions of the result of the previous step. The outputs of these functions are objects of the `SCEV` class, and we call them `BaseSCEV` and `StepSCEV`, respectively. In the next step, we use an object of the `SCEVExpander` class [7] to expand code for `BaseSCEV` and `StepSCEV`.

The code expanded for the `BaseSCEV` is an instruction that computes the $\mathtt{Base_i}$ values corresponding to each iteration `i` of the work-sharing loop. Because of the kernel's structure, the value of $\mathtt{Base_i}$ is a function of `i` and the input parameters of the kernel. If the expanded instruction has any operands resulting from other instructions, we get those instructions and store them in a stack. We repeat this operation on the newly added instructions to the stack until we reach an instruction that all its operands are either integer numbers, the variable `i`, or the input parameters. Then, the instructions in the stack make the instruction sequence that, given the iteration number `i`, computes the corresponding value of $\mathtt{Base_i}$.

The code expanded for the `StepSCEV` is an object of the LLVM `Value` class. It is the value of $\mathtt{Step_i}$ and can be inserted into the program as an integer number.

Moreover, we can get the access loop from the `SCEVAddRecExpr` object we got in the first step. The value of $\mathtt{Number_i}$ is the number of iterations of the access loop, and we can get it as an object of the LLVM `Value` class using the bounds of the loop. Then, it can be inserted into the program as an integer value or as a kernel input parameter. Note that the values of $\texttt{Step}_\texttt{i}$ and $\texttt{Number}_\texttt{i}$ are equal for all iterations and we can drop the `i` subscript.

For example, in the kernel of Fig. 2, the code expanded for $\texttt{Base}_\texttt{i}$ is `(M*i)`, the `Step` is the integer `1`, and the `Number` is the kernel input parameter `M`.

Up to this point, we have generated code for computing $\texttt{Base}_\texttt{i}$, `Step`, and `Number`. Based on Eq. 1, to have access to all the locations we want to prefetch, we need to generate code for iterating over the values of $\texttt{Base}_\texttt{i}$ corresponding to the iterations assigned to each team. Therefore, we create a loop iterating from `team_LB` to `team_UB`, and insert the instruction sequence for computing the $\texttt{Base}_\texttt{i}$ value in the body of this loop. We call this loop the *base computing loop*. We insert the base computing loop in the distribute region before the distribute loop. In the following steps, we will explain how to use the generated $\texttt{Base}_\texttt{i}$ values to prefetch their corresponding memory locations.

Notice that the function parameters in the instruction sequence for computing the $\texttt{Base}_\texttt{i}$ value and in the instruction for `Number` are the ones in the parallel region function. Therefore, to keep the program's semantic correctness, the next step is to replace the parameters with their correspondences in the distribute region. Also, the work-sharing loop's iteration number variable (for example, variable `i` in the instruction for computing `Base` for Fig. 2) should be replaced with the induction variable of the base computing loop to compute $\texttt{Base}_\texttt{i}$ for the team's iteration chunk.

3.3 Loading Data to the Shared Memory

After generating code for iterating over values of $\texttt{Base}_\texttt{i}$ for `i` iterating over the team's chunk, and also for `Step` and `Number`, the next step is to prefetch their corresponding locations from the global memory to consecutive locations in the shared memory. For this purpose, we develop a high-level function called `copy_to_shared_mem`. This function prefetches memory locations accessed in *one* iteration of the work-sharing loop. As a result, to prefetch all the locations of the considered array read by each team, we should call this function in the base computing loop. This function gets the lower bound of the team's chunk (`team_LB`), the iteration number of the work-sharing loop (the induction variable of the base computing loop `i`), a pointer to the array we want to prefetch (`V`), the value of $\texttt{Base}_\texttt{i}$, and the values of `Step` and `Number` as its inputs. Therefore, after inserting the instruction sequence for computing $\texttt{Base}_\texttt{i}$ in the base computing loop, we have all the input values and we can insert a call to the `copy_to_shared_mem` function. Figure 3 shows the call to this function in the transformed kernel.

The first global memory location accessed by each team is $\texttt{V}[\texttt{Base}_\texttt{i}]$, for $\texttt{i} = \texttt{team\_LB}$, and it should be stored in location `0` of the shared buffer. The global memory locations accessed in iteration `i` are $\texttt{V}[(\texttt{Base}_\texttt{i} + \texttt{k} \times \texttt{Step})]$, for `k` between `0` and `Number`. The `copy_to_shared_mem` function stores these locations in consecutive indexes of the shared buffer beginning from $\texttt{S}_\texttt{i}$ to $\texttt{S}_\texttt{i} + \texttt{Number}$, where $\texttt{S}_\texttt{i}$ is the starting storing location computed for iteration `i`. To avoid over-writing already prefetched data, $\texttt{S}_\texttt{i}$ is equal to the total number of the

```
void kernel(/* inputs */)
{
  kmpc_distribute(/* loop bounds */);
  for(int i=team_LB; i<team_UB; i++) //-->base computing loop
    //instruction sequence for computing Base_i
    copy_to_shared_mem(team_LB, i, V, Base_i, Step, Number);
  // distribute region
}
```

Fig. 3. Call to the `copy_to_shared_mem` function in the transformed kernel.

locations prefetched in the previous iterations of the base computing loop, and it is equal to (`i-team_LB`)×`Number`. With these relations, each team requires (`team_UB-team_LB`)×`Number` × `B` bits of the shared space, where `B` is the size of the prefetched array's type.

Moreover, we want to prefetch data in the shared memory in parallel to get better performance. The function is called in the target region outside the parallel region. Therefore, we distribute the work between threads based on their ids. Figure 4 shows the `copy_to_shared_mem` function for prefetching an integer array into the shared memory.

```
1  void copy_to_shared_mem(int team_Lb, int i, int *V,
2                          int Base_i, int Step, int Number)
3  {
4      int Tid = get_thread_id();
5      int *DataBuf = (int*) get_dynamic_shared();
6      int bufOff = (i-teamLb)*Number+Tid;
7      int stride = omp_get_max_threads();
8      for(int k=0; k<num; k+=stride)
9          DataBuf[bufOff+k] = V[Base_i+Tid*Step+k*Step];
10 }
```

Fig. 4. The implementation of `copy_to_shared_mem` function.

3.4 Retrieving Data from the Shared Memory

The final step is to generate code for replacing accesses to the global memory with their corresponding accesses in the shared memory in the parallel region function.

Based on the explanation in Sect. 3.3 and line 9 of Fig. 4, the index ($\texttt{Base}_\texttt{i} + \texttt{k} \times \texttt{Step}$) of the considered array in the global memory is stored in index ($\texttt{i}-\texttt{team\_LB}$)×`Number` + `k` of the shared buffer. In these relations `k` iterates from `0` to `Number` and it is the induction variable of the access loop.

To access the shared memory locations, we first generate code for getting the pointer to the dynamic shared memory in the parallel region function. Then, we use the values of `team_LB` and induction variable of the access loop in the parallel region function and generate code for the access relation to the shared memory. Finally, we replace the access to the global memory with the (newly generated) access to the shared memory.

3.5 Static Shared Memory Prefetching

We can take advantage of the method explained in the previous parts to prefetch input arrays of small sizes to the static shared memory. In this case, we prefetch the whole considered arrays into the shared memory for all teams. We first select input arrays that fit in the shared memory. Then, we use the approach of inserting a high-level copy function in the distribute region and replacing read instructions from the global memory with read instructions from the shared memory in the parallel region function to prefetch them.

4 Optimization

In this section we explain two optimization methods that can be applied at compile time for some special cases to improve the performance and applicability of the method explained in Sect. 3 in these cases.

4.1 Space Optimization

The method explained in Sect. 3 stores every read location in the team to the shared memory. Although this method works correctly for all the cases, its time and space usage is not efficient if some iterations in the chunks assigned to each team read the same locations. In other words, if $\texttt{Base}_\texttt{i}$ is equal for some values of `i` ranging from `team_LB` to `team_UB`, the corresponding iterations of the base computing loop prefetch redundant data.

We extend the method of Sect. 3 to more efficiently handle the special case that *all*[1] iterations of the team's chunk read the exact same locations. For this purpose, we add an option that can be set at compile time to `different`, or to `same`. The `different` option is the default one and works as explained in Sect. 3. The `same` option can be used when it is known in advanced that all iterations assigned to each team read the exact same locations. In this case, the base computing loop is unnecessary, and we can load all the required data for a team by finding $\texttt{Base}_\texttt{i}$ for `i=team_LB` and call the `copy_to_shared_mem` function only once. In this case, the global memory locations are prefetched in the shared buffer from index `0` to `Number-1` and we can retrieve them in the parallel region by the induction variable of the access loop.

[1] It is better to use the default option for cases where *some* (but not all) of the team's chunk iterations read the same locations. The reason is that avoiding prefetching redundant data in these cases complicates the `copy_to_shared_mem` function in different ways (e.g., adds conditional branches to it) that degrades the performance.

4.2 Reducing Bank Conflict

We explained in Sect. 2.2 that bank conflict might happen when a kernel uses GPU's shared memory. In the kernels we consider, the number of accesses with bank conflict depends on the value of `Number`. In the worst case, where `Number` is a multiple of 32, almost all of the accesses have bank conflict and we get slowdown by prefetching. The reason is that each iteration of the base computing loop (calls to the `copy_to_shared_mem` function) starts storing data to the bank `0` of the shared memory. This causes all threads requesting from the same memory bank when retrieving data.

To solve this problem, we use the *padding technique*. More specifically, if `Number` is a multiple of 32, we store one invalid data in the shared memory by altering the `copy_to_shared_mem` function. This can be done by changing line 4 of Fig. 4 to `bufOff=(i-teamLb)`×`(Number+1)+Tid`. Also, to ignore the invalid location, we add `1` to the `Number` when retrieving data in the parallel region function.

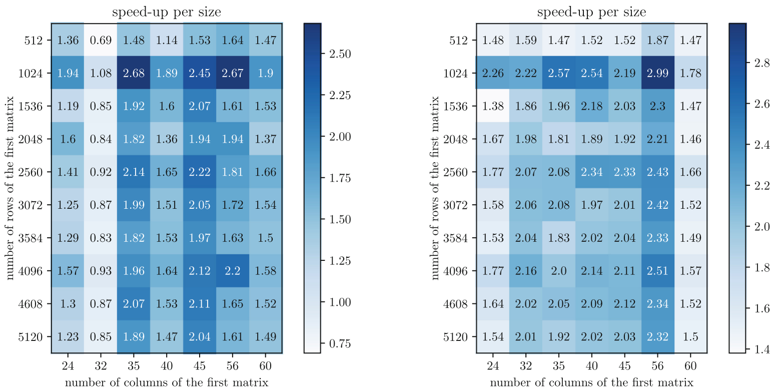
5 Performance Evaluation

In this section, we evaluate the method explained in Sect. 3 and the optimization techniques proposed in Sect. 4 in terms of running time improvements. We compile all the programs with the Clang compiler, along with `-O3` and `-openmp-opt-inline-device` options. We run the experiments on an NVIDIA `GeForce MX150` GPU of `sm=6.0`. For the shared memory experiments, we allocate enough space of dynamic shared memory for the kernel, by setting the environment variable `LIBOMPTARGET_SHARED_MEMORY_SIZE` to the appropriate number. This number varies based on the size of the experiment.

For evaluation, we use the rectangular matrix multiplication kernel, and consider multiplying matrices with different number of rows and columns. We compare the running times of the kernel (reported by NVIDIA profiler, `nvprof`) with and without prefetching and report the speedups based on the multiplier's size. Figure 5 shows the speedup we get when multiplying two matrices by prefetching *rows* of the multiplier, and Fig. 6 shows the speedup we get when we multiply transposes of two matrices and prefetch *columns* of the multiplier.

In both of these sets of experiments, the outermost loop is considered the work-sharing loop with `distribute parallel for` pragma, the number of threads we use is 32, and the number of teams is the number of iterations of the work-sharing loop (the number of rows in the first case, and the number of columns in the second case) divided by 32. Also, locations read by each thread in each team are different and we cannot apply the space optimization from Sect. 4.1 on these kernels (compile them with the default (`different`) option).

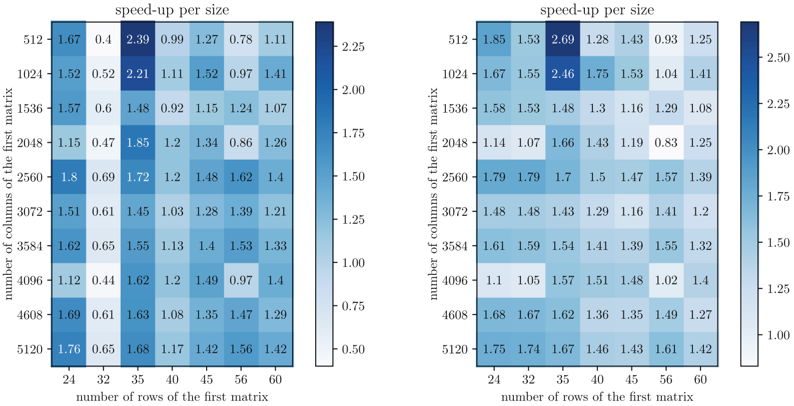
To examine the effect of bank conflict and padding method's effectiveness explained in Sect. 4.2, Figs. 5a and 6a show the speedup we get without applying the padding method, and Figs. 5b and 6b shows the speedup when we apply the padding method when prefetching data.



(a) Without padding.

(b) With padding applied.

Fig. 5. Prefetching speedup of the matrix multiplication when applying prefetching.



(a) Without padding.

(b) With padding applied.

Fig. 6. Prefetching speedup of the matrix transpose multiplication by applying prefetching.

To test the space optimization of Sect. 4.1, we again consider the matrix multiplication kernel and we add the `collapse(2)` construct to the outermost loop. We set the number of teams and the number of threads per team equal to the number of rows of the multiplier. For these examples, the locations used by all threads in a team are similar and we compile them with the `same` option. Figure 7 shows the speedups we get by prefetching.

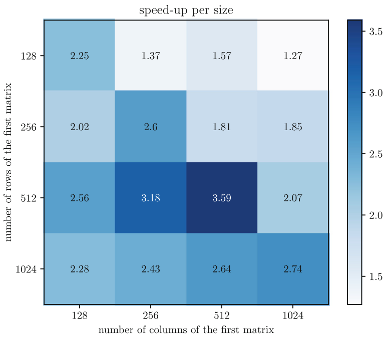


Fig. 7. Prefetching speedup of matrix multiplication with collapsed loops.

In the final evaluation, we consider the XSBench [17] with small size. For different grid types (`Nuclide, Unionized, Hash`) the input data structure has three small size arrays that we can prefetch to the static shared memory, as explained in Sect. 3.5. Figure 8 shows the speedup and also the total number of global memory load requests (ld_req) with and without prefetching, reported by `nvprof`. The maximum speedup is 5%, and the number of load requests from the global memory decreased by prefetching.

grid type	speedup	ld_req	ld_req with prefetching
`Nuclide`	1.05	1 885 268 743	1 754 280 336
`Unionized`	1.03	1 066 897 410	935 422 655
`Hash`	1.05	1 331 110 977	1 199 607 982

Fig. 8. Prefetching speedup and comparing number of global memory load requests in XSBench.

5.1 Analysis of the Experiments

In the experiments represented in Figs. 5 and 6, there are reuses of the multiplier's rows and columns in each team of threads, respectively. As a result, by applying the prefetching technique, we reduce the number of global memory accesses, which improves the performance of the kernels in most cases. However, as shown in Figs. 5a and 6a, the kernels get slowdown when the number of rows in Fig. 5a and the number of columns in Fig. 6a is 32, because of shared memory bank conflict. We can improve the performance in these cases by applying the padding method, as explained in Sect. 4.2. The effectiveness of the padding method is shown in Figs. 5b and 6b. Similarly, in the experiment represented in Fig. 7, there are reuses of the same row of the multiplier in each team of threads. By prefetching and applying the space optimization explained in Sect. 4.1 the kernels gain speedup in all cases.

In our experiment with XSBench, represented in Fig. 8, accessing the prefetched arrays is not time-consuming compared to the other steps of the algorithms. Although prefetching works as expected and reduces the number of load requests from the global memory, the kernels do not gain significant speedup.

6 Conclusion

In this paper, we used the infrastructure of the `OpenMPOpt` pass to develop an LLVM pass to optimize offloaded regions of OpenMP when targeting GPUs by prefetching data to the shared memory. The method can be applied on the kernels with some properties (explained in Sect. 3.1) and we show that it improves the performance of these kernels. We also propose solutions for more efficient use of shared space and for avoiding bank conflicts.

For future works, we plan to improve the applicability of the method by supporting more general kernels and by relaxing the limitations explained before. For instance, we want to improve our process to handle functions that use other OpenMP constructs. Moreover, in the current version, we only prefetch one of the read-only arrays. An interesting idea is to improve the algorithm to prefetch more than one array or choose the best one for prefetching.

Acknowledgements. First and second authors thank NSERC of Canada (Grant RGPIN-2018-06534) for their support. Also, part of this research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative. Part of this research was supported by the Lawrence Livermore National Security, LLC ("LLNS") via MPO No. B642066.

References

1. CUDA programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide
2. LLVM version 11. https://releases.llvm.org/download.html#11.0.0
3. OpenMP application programming interface version 4.0. https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf
4. OpenMP application programming interface version 5.0. https://www.openmp.org/spec-html/5.0/openmp.html
5. SCEV Class Reference. https://llvm.org/doxygen/classllvm_1_1SCEV.html
6. SCEVAddRecExpr Class Reference. https://llvm.org/doxygen/classllvm_1_1SCEVAddRecExpr.html
7. SCEVExpander Class Reference. https://llvm.org/doxygen/classllvm_1_1SCEVExpander.html
8. Using Shared Memory in CUDA C/C++. https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/
9. Value Class Reference. https://llvm.org/doxygen/classllvm_1_1Value.html

10. Antao, S.F., et al.: Offloading support for OpenMP in Clang and LLVM. In: 2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC), pp. 1–11. IEEE (2016)
11. Bataev, A., Bokhanko, A., Cownie, J.: Towards OpenMP support in LLVM. In: 2013 European LLVM Conference (2013)
12. Bertolli, C., et al.: Integrating GPU support for OpenMP offloading directives into Clang. In: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, pp. 1–11 (2015)
13. Hayashi, A., Shirako, J., Tiotto, E., Ho, R., Sarkar, V.: Performance evaluation of OpenMP's target construct on GPUs-exploring compiler optimisations. Int. J. High Perform. Comput. Networking **13**(1), 54–69 (2019)
14. Huber, J., et al.: Efficient execution of OpenMP on GPUs. In: 2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 41–52. IEEE (2022)
15. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis and transformation. In: International Symposium on Code Generation and Optimization, CGO 2004, pp. 75–86. IEEE (2004)
16. Tian, S., Chesterfield, J., Doerfert, J., Chapman, B.: Experience report: writing a portable GPU runtime with OpenMP 5.1. In: McIntosh-Smith, S., de Supinski, B.R., Klinkenberg, J. (eds.) IWOMP 2021. LNCS, vol. 12870, pp. 159–169. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85262-7_11
17. Tramm, J.R., Siegel, A.R., Islam, T., Schulz, M.: XSBench - the development and verification of a performance abstraction for Monte Carlo reactor analysis. The Role of Reactor Physics toward a Sustainable Future (PHYSOR) (2014)

Feasibility Studies in Multi-GPU Target Offloading

Anton Rydahl(✉), Mathias Gammelmark, and Sven Karlsson

Technical University of Denmark, Anker Engelunds Vej 1,
2800 Kongens Lyngby, Denmark
`rydahlanton@gmail.com`, `{magam,svea}@dtu.dk`

Abstract. Many of the largest supercomputers are based on heterogeneous architectures with multiple general-purpose graphics processing units (GPGPUs) per compute node. While many APIs for GPU programming are vendor-specific, OpenMP offers a portable alternative. Therefore OpenMP target offloading is advantageous in terms of long-term code sustainability. Further, many applications have already been parallelized with OpenMP. Hence the amount of work needed to port the code to GPUs may be limited. However, the support for the OpenMP 5.x specification is not equally mature across different compilers. Additionally, the multi-GPU support in the OpenMP 5.x specification is limited. We explore what is possible with the Nvidia NVC compiler.

We present a case study of solving the Poisson equation on multiple GPGPUs to outline which approaches for multi-target offloading give good results. We find that a task-based multi-GPU implementation leads to better performance than generating deferrable tasks with the `nowait` clause.

We demonstrate that data transfers and computations can be fully overlapped by using only the subset of the OpenMP specifications, which is supported in the 22.3 release of the Nvidia NVC compiler. For compute nodes with multiple Nvidia A100 or V100, we obtain close to ideal strong scaling when increasing the number of accelerators.

Keywords: Heterogeneous computing · GPGPU programming · Tasks · Target offloading

1 Introduction

Many supercomputers are now based on heterogeneous architectures with multiple accelerators per compute node. An important example is the world's first publicly known exascale supercomputer, Oak Ridge National Laboratory's Frontier [2]. The compute nodes at Frontier come with four AMD Instinct 250X GPUs [3]. Many other leading supercomputers are based on heterogeneous architectures [2]. To name a few, Summit at the Oak Ridge National Laboratory comes with six Nvidia Volta V100s per node [4], and Lawrence Livermore National Laboratory's Sierra has four Nvidia Tesla P100 GPUs per compute node [5].

M. Klemm et al. (Eds.): IWOMP 2022, LNCS 13527, pp. 81–93, 2022.
https://doi.org/10.1007/978-3-031-15922-0_6

The recent evolution in supercomputing has brought powerful GPGPUs to the compute nodes. Thus it is crucial to study how to make performant applications utilizing multiple accelerators. The compiler support for OpenMP target offloading is not yet mature across all compilers, which limits the number of clauses and runtime functions that can be used if one wants to write portable multi-GPU applications. Also, some programmers are accustomed to using inter-device address mapping from other GPU programming APIs that OpenMP does not support. However, it is possible to write code that scales well across multiple GPUs with only a subset of the OpenMP 5.x specification.

In some literature, a basic approach for programming multiple GPUs with OpenMP is to create deferrable tasks in a loop by using the `nowait` and `device()` clauses [6]. While this simple approach works for some problems, we were unable to fine-tune it to obtain satisfactory results across more than two accelerators. Instead, we found that using tasks and task groups gave better strong scaling across multiple targets. It was found that there are two main benefits of making a task-based implementation. First, it allows the task scheduler to start kernels on multiple targets without delay between the launch time. Next, embedding data transfers in tasks allowed us to avoid unnecessary synchronization. Both aforementioned advantages result in an application with significantly less idle time on the accelerators.

2 Background

In this case study, we will consider solving the Poisson equation on a hyper-rectangle Ω with a finite difference scheme. The Poisson equation can readily and efficiently be solved by direct methods such as Sparse LU decomposition [7]. However, when the system size grows, the matrix decompositions become computationally infeasible [8]. That motivates using an iterative method such as multigrid methods or the simpler finite difference methods. The Jacobi method is an embarrassingly parallelizable lower order method but proves to be a good example of what one can achieve on a single compute node.

We will consider the Poisson equation on the form

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = -f, (x, y, z) \in \Omega. \tag{1}$$

A research team at the Technical University of Denmark needed a Poisson solver with Dirichlet boundary conditions on two of the boundary planes and Neumann-type boundary conditions on the remaining four. Hence these are the boundary conditions that we will consider in this paper,

$$\begin{aligned} \left.\frac{\partial u}{\partial x}\right|_{x=x_{min}} &= \eta_{min}, & \left.\frac{\partial u}{\partial x}\right|_{x=x_{max}} &= \eta_{max}, & \left.\frac{\partial u}{\partial y}\right|_{y=y_{min}} &= \xi_{min} \\ \left.\frac{\partial u}{\partial y}\right|_{y=y_{max}} &= \xi_{max}, & u\big|_{z=z_{min}} &= g_{min}, & u\big|_{z=z_{max}} &= g_{max}. \end{aligned} \tag{2}$$

Due to the Neumann boundary conditions, it is necessary to derive different stencils for interior, boundary, and corner points. The stencils for boundary walls and corners can be derived with the *ghost point* approach. In total, this yields nine different kernels.

For a univariate function, the second order central difference operator is given by

$$\frac{\partial^2 u}{\partial x^2} \simeq \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} \tag{3}$$

where h is the uniform spacing between the lattices in the grid. From the separability of the Laplacian operator (1) it follows that the second-order accurate finite difference approximation is given by

$$\frac{u_{i+1,j,k} + u_{i,j+1,k} + u_{i,j,k+1} - 6u_{i,j,k} + u_{i-1,j,k} + u_{i,j-1,k} + u_{i,j,k-1}}{h^2} \simeq -f \tag{4}$$

where i, j and k are indices in the x, y and z dimensions respectively. By reordering the approximation (4) we arrive at the Jacobi update for interior points

$$u^{(l+1)}_{i,j,k} = \frac{1}{6}\big(u^{(l)}_{i-1,j,k} + u^{(l)}_{i+1,j,k} + u^{(l)}_{i,j-1,k} + u^{(l)}_{i,j+1,k} + u^{(l)}_{i,j,k-1} + u^{(l)}_{i,j,k+1} + h^2 f\big) \tag{5}$$

where l indicates the iteration number. Listing 1.1 exemplifies how the Jacobi update (5) for interior points can be implemented in C. We derived the remaining eight lattice update equations for boundaries and corners in the same manner as L. Cheng [10] but for a 3D domain.

Figure 1 shows a slab partitioning of the hyperrectangular domain. We will now outline how multiple GPUs can be used to work on each slab and how to exchange intermediate results from each GPU to its neighbor GPUs.

```c
#pragma omp declare target
void interior(double * u1, double * u2, double * f,
              int i, int j, int k, int ny, int nz)
{
    size_t idx = i*ny*nz+j*nz+k;
    u1[idx] = FRAC*(u2[idx-ny*nz] + u2[idx+ny*nz] + u2[idx-nz]
            + u2[idx+nz] + u2[idx-1] + u2[idx+1] + f[idx]);
}
#pragma omp end declare target
```

Listing 1.1. The stencil to update interior points. For this stencil to work, the right-hand side f must have been multiplied with h^2 in advance, and `FRAC` should hold the value $\frac{1}{6}$. The arguments i, j, k are the indices in the three dimensions, and ny and nz are the numbers of lattices in the y and z dimensions respectively.

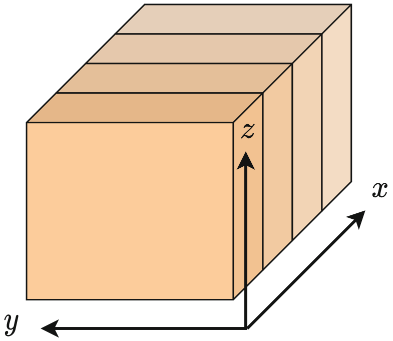


Fig. 1. Illustration of the partitioning of the domain. It was chosen to split the domain across the x-axis but it could in principle have been partitioned along any axis.

3 Design Patterns

The OpenMP specification provides many options for utilizing accelerators. However, the Nvidia HPC SDK release 22.3 [9] supports only a subset [1] of the OpenMP 5.0 API. Therefore we will pursue what is possible under these limitations.

3.1 Initial Target Version

The nowait clause makes the target task a deferrable task allowing the host thread to continue execution after launching a kernel on the device [6]. Thereby multiple kernels can be run in parallel, potentially on multiple devices.

Therefore a simple approach for implementing a multi-GPU version of the code is to use the nowait clause for data transfers and kernel execution. All target constructs yield a new task, and thereby a combination of nowait and taskwait provides a straightforward yet powerful way of using accelerators in parallel. However, this approach has its limitations compared to tasking since more synchronization points are needed.

Parallelizing the Jacobi Scheme on the Target. We have to consider multiple levels of parallelism to achieve good performance on the target. First, the target construct creates a thread on the target device. Then teams is used to start a league of initial threads. The distribute construct can then be applied to distribute the loop iterations across the initial threads. By adding the parallel construct, each initial thread becomes the master of a new team of threads.

The collapse clause must also be applied to fully distribute the iterations if high performance should be achieved for multiple nested loops. Listing 1.2 shows how to use the combined constructs to loop over interior points in the domain.

```
#pragma omp target teams distribute parallel for collapse(3) \
    schedule(static) device(node->id) nowait
for (int i=1; i<nx-1; i++)
    for (int j=1; j<ny-1; j++)
        for (int k=1; k<nz-1; k++)
            interior(u1,u2,f,i,j,k,ny,nz);
```

Listing 1.2. Kernel to loop over all interior points in target region.

```
// Copy boundary layers from devices to host
for (int i=0;i<num_devices;i++){
    #pragma omp target update from(...) \
        device(node->id) nowait
}
#pragma omp taskwait
// Copy boundary layers from host to devices
for (int i=0;i<num_devices;i++){
    #pragma omp target update to(...) device(node->id)
    nowait
}
#pragma omp taskwait
```

Listing 1.3. The nowait approach for communicating the boundary layers from each device to the neighbor devices in the grid.

Transferring Boundary Points. As of the OpenMP 5.1 specification, omp_target_memcpy() should be able to copy data between any combination of host and device pointers. However, it is one of the features currently not supported in the Nvidia HPC SDK. An alternative approach is to transfer the data from a device to a buffer located on the host and subsequently transfer it to another device. Listing 1.3 provides an example of how boundary points can be exchanged via the host using target update and taskwait clauses. In the example, we assume that the ids of devices are stored in a linked list.

Overlapping Computations and Communication. To overlap computations and data transfer the interior kernel must first be applied to the boundary lattices. Then the results on the boundaries can be exchanged with neighbor GPUs while the remaining lattices are updated. With this strategy, it is possible to overlap computations and communication. At least to a certain degree.

```
#pragma omp parallel
#pragma omp single nowait
#pragma omp taskgroup
for (int i=0;i<num_devices;i++){
    ...
    if (node->prev != NULL){
        #pragma omp task firstprivate(node)
        {
            #pragma omp target update from(...) \
                device(node->id)
            #pragma omp target update to(...) \
                device(node->prev->id)
        }
    }
    ...
}
```

Listing 1.4. Example task to move data from one device to another. One could also have used two calls to omp_target_memcpy() instead of the compiler directives.

3.2 Tasking-Based Strategy

While using the nowait and taskwait clauses in combination provides a simple framework for using multiple accelerators in parallel, this approach does not scale well when using more accelerators. The primary reason for this is that the kernels are not launched at the exact same time on each device. That leads to a high fraction of idle time. Using tasks and task groups is a powerful alternative that scales better with the number of accelerators for our type of problem.

Transferring Boundary Points with Tasking. Assuming that the devices are stored in a linked list data structure, Listing 1.4 gives an example of how a task can be used to transfer boundary points from one device to a neighbor device via the host. A similar approach could be to use the task depend clause, but we once again found that the required clauses are still not well supported in the NVC compiler. However, this simplistic approach of including two blocking data transfers in the same task performs well. Note that since the target update clause generates a task itself, we now need to consider task groups to wait for all tasks and their descendants generated in the relevant code region.

Overlapping Computations and Communication with Tasking. We found that Nvidia's compilers do not support embedding a target teams construct in a task. Therefore the stencil kernels were still launched with the nowait clause in this version. Listing 1.4 gives a simple example of how the data transfers can be embedded in tasks and how the tasks can be created inside a task group. Note that it is generally not recommended to open parallel regions inside a loop due to the overhead of opening and closing parallel regions. In our implementation, the entire iteration loop was embedded in a single region inside a parallel region and therefore differed a bit from the example.

4 Results

In this section, the implementation based on generating a deferrable task in a loop with the nowait clause will be compared to the slightly more complex tasking-based implementation. We will refer to them as the *nowait* strategy and the *tasking* strategy, respectively.

4.1 Experimental Setup

Two types of nodes were used to test the multi-GPU implementations of the Jacobi scheme. One with an AMD EPYC CPU and four Nvidia A100 GPUs. Most of the results stated in the next section were obtained on this node. We also had access to a node with an Intel Xeon Gold 6142 CPU and four Nvidia V100 GPUs. We chose to focus on the Ampere node in the figures. Mainly for brevity

Table 1. System information for the two types of nodes used for the benchmarks. The two tables under (a) states highlighted information about the node with Nvidia A100 GPUs which was used for most figures in this paper. Table (b) lists the equivalent information for the node with Nvidia V100 GPUs.

(a) Ampere Node

CPU Information

Model name	AMD EPYC 7F72
Sockets	2
Cores per socket	24
Threads per core	2
L1d cache	32K
L1i cache	32K
L2 cache	512K
L3 cache	16384K

Accelerator Information

Number of GPGPUs	4
Model name	Tesla A100
RAM	40 GB
FP64 Cores	3456
L1 cache	20736 KB
L2 cache	40960 KB

(b) Volta Node

CPU Information

Model name Intel Xeon Gold 6142	
Sockets	2
Cores per socket	16
Threads per core	1
L1d cache	32K
L1i cache	32K
L2 cache	1024K
L3 cache	22528K

Accelerator Information

Number of GPGPUs	4
Model name	Tesla V100
RAM	32 GB
FP64 Cores	2560
L1 cache	10240 KB
L2 cache	6144 KB

but also because the Ampere architecture is newer than the Volta architecture. However, we will highlight when the results differ. Table 1 states technical details about the two compute nodes.

4.2 Experimental Results

The experimental results were collected by running 5000 iterations of the applications ten times and averaging the results. Note that it required significantly more than 5000 iterations to converge on the high-resolution domains. The performance of the multi-GPU versions was compared to an equivalent SIMD-optimized and parallelized CPU version. When the GPU versions are compared to the CPU version, we compare them to using all cores per socket for all sockets on the compute node.

Figure 2 shows that both the strategies we tested led to significant speedups when utilizing one or more GPUs. We obtained similar results on both the Volta and Ampere nodes, but only the results for the latter are shown. However, it is noticeable that even on the largest domain size that was tested, the nowait strategy led to worse results when using four GPUs compared to using only three. With the tasking strategy, we obtained 48.62 times speedup compared to the average running time of the parallel CPU version on a domain with 10^9 lattices. With the nowait strategy, using four GPUs only led to 22.46 times speedup.

By using the nsys and nv−nsight−cu−cli profilers, we found an explanation for why the nowait approach does not scale well. The profiling results made it clear that even though the kernels and data transfers execute in parallel, there is a delay in the launch time for each GPU. That leads to a significant amount of idle time when the number of accelerators increases. Another reason is that this version has more synchronization points due to the lack of compiler support for task dependencies on target regions. Hence we must wait for all the device to host transfers to terminate before we can initiate the data transfers

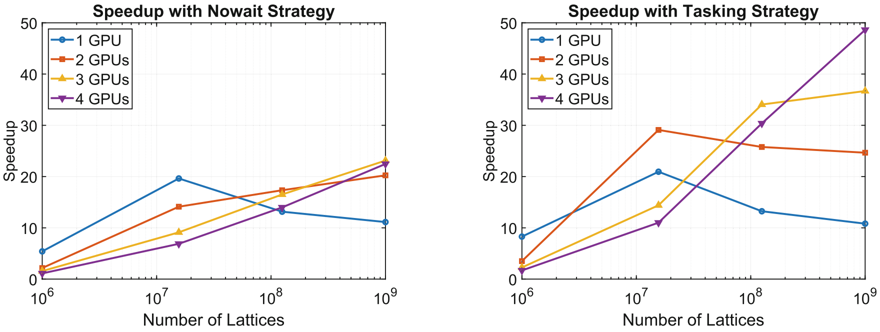
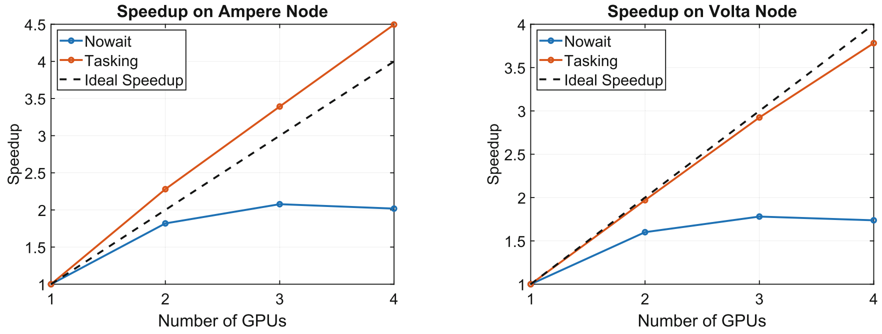


Fig. 2. Speedup from using an entire CPU to using multiple GPUs. This figure is made with results from the Ampere node Table 1a. The speedup is measured in comparison to using all 48 physical cores on the Ampere node Table 1b



(a) (b)

Fig. 3. Speedup as function of the number of accelerators utilized on the two types of nodes. Figure (a) shows the speedup for the Ampere node and (b) for the Volta node. In the test the number of lattices in the domain was fixed at 10^9.

from the host to the devices. That results in only partially overlapping data transfers and kernel executions.

Figure 3 shows how the number of lattice updates per second scales with the number of GPUs used. This time we would like to address both the results on the Nvidia A100 and V100 GPUs. On the node with Volta GPUs, we observe that the task-based approach gives close to ideal speedup for a domain with 10^9 lattices. With this approach, we obtain 3.78 times speedup compared to using only one of the V100 GPUs.

However, for the same domain resolution, the tasking approach is more than four times faster when using four Nvidia A100 GPUs. Figure 4 can explain this result. We can explain the behavior on the Ampere node by investigating the cache performance, which is plotted as a function of resolution in Fig. 4a. When the resolution of the domain becomes higher, the level two cache hit rate drops to approximately 50%. When we go from a domain size of $250^3 \simeq 1.56 \cdot 10^7$ to $500^3 = 1.25 \cdot 10^8$ lattices, we see a simultaneous decrease in level two cache hit rate and streaming multi-processor throughput, Fig. 4d. So when we partition the domain on more GPUs, the interior point kernel is more efficient. It turns out that this performance gain more than cancels out the synchronization that is introduced when working with multiple accelerators for our particular problem.

The main reason that we do not see this behavior on the Volta node is that the level two cache is approximately seven times smaller compared to the Ampere GPU. However, for an even larger domain, it can be expected that the scaling on the Ampere node will be similar to the scaling on the Volta node.

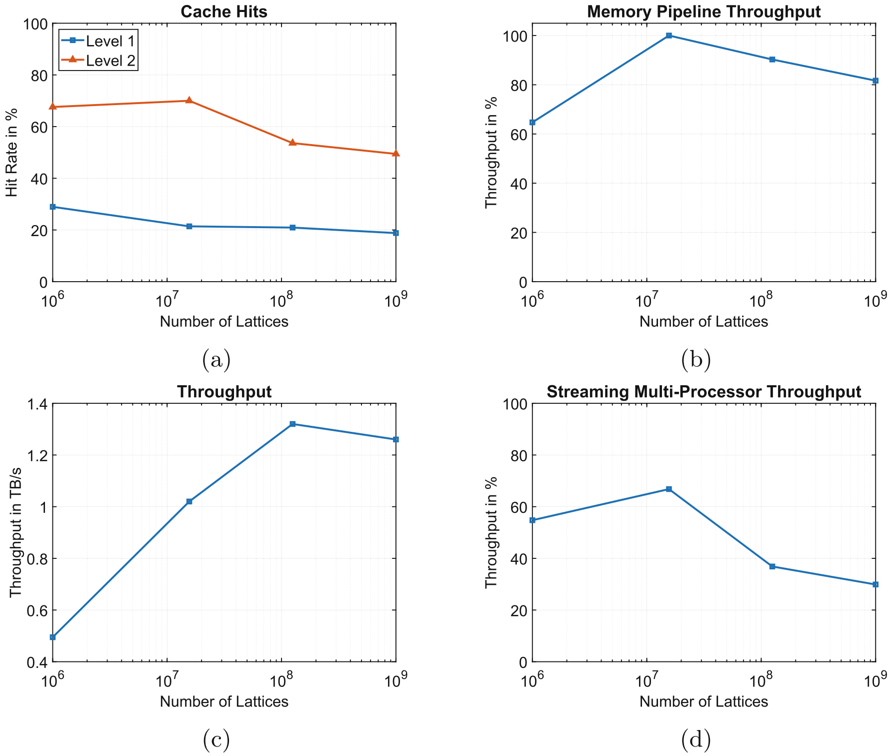


Fig. 4. Performance counters for the interior point kernel Listing 1.2 that takes up the vast majority of the running time. Figure (a) shows the cache hit rate for the level one and level two cache on a Nvidia A100 GPU. (b) shows the memory pipeline throughput, (c) shows the throughput in TB/s, and (d) shows the SM throughput in percent.

5 Related Work

There are many examples of directive-based multi-GPU programming on a single compute node. An early example, Xu et al. [11] use an OpenMP and OpenACC hybrid model to solve the heat equation in 2D using two GPUs. Here OpenMP is used for host parallelism and OpenACC for device parallelism. The lattice update procedure in this work is similar to the Jacobi stencil, but due to the evolution of OpenMP, we had the option to use OpenMP for offloading to GPUs as well.

Jaber et al. [12] have developed a far more complex open-source Poisson solver, which is based on an MPI and OpenACC hybrid model. This solver relies on fast Fourier transform (FFT) and parallel cyclic reduction (PCR). Further, the weak scaling analysis of this solver was run for up to 16384 GPUs at Oak Ridge National Laboratory's Titan supercomputer. In comparison to this approach, the solver we present is, first of all, limited to one single node. Second, the method we use has worse time and storage complexity than an FFT-based method [7]. Yet our work shows that OpenMP can be used for multi-target offloading, potentially in a more complicated Poisson solver.

Kale et al. [13] use OpenMP target offloading to utilize multiple GPUs. Here the taskloop construct is used to distribute the work and data among the accelerators. In this way, blocks of work are dynamically distributed among the accelerators to handle work imbalance. Since the problem considered in this paper is not work imbalanced, it was chosen not to pursue the same parallelization strategy. Instead, letting the data reside on a specific GPU helped us minimize the total amount of intra-node communication.

Patel et al. [14] demonstrate how OpenMP can be used to offload to remote accelerators. They show how accelerators efficiently can be utilized across a distributed system without MPI. Their work has been included in the LLVM project. That work is interesting because it shows that an application like the Poisson solver outlined in this paper could, in principle, run on remote accelerators without using other APIs such as MPI. However, this would require moving to an LLVM-based compiler.

In terms of the choice of compiler, we are not the first to address the lack of OpenMP support for target offloading and tasking in Nvidia's HPC SDK. Chapman et al. [15] report that their program did not compile or resulted in a crashing executable when compiled with NVC. In comparison, the program worked when compiled with Clang, CCE, and, to some extent, GCC. For their application, release 21.3 of Nvidia's HPC SDK was used. While we got our application to work with release 22.3, we also faced challenges. During the development of this application, we reported a compiler bug related to data scoping for tasks [16] and two bugs [17] related to embedding a target region in a task. The data scoping bug was fixed in release 22.5, but the two latter have not been fixed at the time of writing.

6 Conclusion

In this paper, we compared different strategies for utilizing multiple GPGPUs with the OpenMP API. We showed that for sufficiently large problems, it is possible to fully overlap computations and data transfers with the subset of the OpenMP specification, which is currently supported in the Nvidia NVC compiler.

We found that when embedding data transfers in tasks, our application achieved close to ideal, strong scaling across multiple accelerators. We got superlinear, strong scaling on a compute node with four Nvidia A100 GPUs. This was explained by the fact that the performance of the interior lattice update kernel was influenced by the level two cache hit rate. We expect this effect to be less influential for larger domains which will probably lead to sublinear, strong scaling.

The approach of making deferrable tasks with the nowait clause did make it possible to use multiple accelerators. However, for the large domains, we got a twice as large speedup by using tasks to transfer boundary points when utilizing all four accelerators. That holds for both types of compute nodes that we had access to.

By using the nsys−ui profiler, we could explain this behavior by the fact that the kernels and data transfers launch with some offset in time when using the nowait clause to generate deferrable tasks. That results in more and more idle time when increasing the number of accelerators. Therefore the nowait approach was less performant. We found that all data transfers and kernels launched in parallel without delay for the tasking approach. The tasking strategy also provided much greater flexibility in terms of synchronization. This is also part of the explanation for why embedding data transfers in tasks led to substantially better performance.

The application considered in this case study is embarrassingly parallelizable, yet it demonstrates how a finite difference method, or a similar method with lattice updates, can be implemented in OpenMP. It would not be difficult to extend this implementation to a higher-order finite difference approximation.

We would have liked to pursue an implementation based solely on tasks and task dependencies, but this was not possible due to bugs in the NVC compiler. However, when these bugs have been fixed, we would like to try to follow this path. It would perhaps be a way to eliminate the synchronization that we were forced to have between each iteration of the Jacobi updates. For instance, it would be straightforward to use task dependencies to improve Listing 1.4.

Acknowledgments. This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 951732. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Germany, Bulgaria, Austria, Croatia, Cyprus, Czech Republic, Denmark, Estonia, Finland, Greece, Hungary, Ireland, Italy, Lithuania, Latvia, Poland, Portugal, Romania, Slovenia, Spain, Sweden, United Kingdom, France, Netherlands, Belgium, Luxembourg, Slovakia, Norway, Switzerland, Turkey, Republic of North Macedonia, Iceland, Montenegro.

References

1. NVIDIA Corporation: NVIDIA HPC Compilers User's Guide, section 7. Using OpenMP. https://docs.nvidia.com/hpc-sdk/archive/22.3/compilers/hpc-compilers-user-guide/index.html#openmp-use. Accessed 12 July 2022
2. Strohmaier, E., Dongarra, J., Horst, S., Meuer, M.: TOP500 List - June 2022. https://top500.org/lists/top500/list/2022/06/. Accessed 7 July 2022
3. Oak Ridge Leadership Computing Facility: Frontier - Direction of Discovery. https://www.olcf.ornl.gov/frontier/. Accessed 7 July 2022
4. Oak Ridge Leadership Computing Facility: Summit - Oak Ridge National Laboratory's 200 petaflop supercomputer. https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/. Accessed 25 May 2022
5. Lawrence Livermore National Laboratory: Using LC's Sierra Systems. https://hpc.llnl.gov/documentation/tutorials/using-lc-s-sierra-systems. Accessed 25 May 2022
6. van der Pas, R., Stotzer, E., Terboven, C.: Using OpenMP-The Next Step, 1st edn. The MIT Press, Cambridge (2017)
7. Demmel, J.: Solving the Discrete Poisson Equation using Jacobi, SOR, Conjugate Gradients, and the FFT. https://people.eecs.berkeley.edu/~demmel/cs267/lecture24/lecture24.html. Accessed 27 May 2022

8. Burkardt, J.: Jacobi Iterative Solution of Poisson's Equation in 1D. https://people.sc.fsu.edu/~jburkardt/presentations/jacobi_poisson_1d.pdf. Accessed 15 July 2022
9. NVIDIA Corporation: NVIDIA HPC Compilers User's Guide. https://docs.nvidia.com/hpc-sdk/compilers/hpc-compilers-user-guide/index.html. Accessed 11 May 2022
10. Cheng, L.: Finite Difference Methods for Poisson Equation. https://www.math.uci.edu/~chenlong//226/FDM.pdf. Accessed 15 July 2022
11. Xu, R., Tian, X., Chandrasekaran, S., Chapman, B.: Multi-GPU Support on Single Node Using Directive-Based Programming Model. Sci. Program. **2015** (2015). https://doi.org/10.1155/2015/621730
12. Hasbestan, J.J., Xiao, C., Senocak, I.: PittPack: an open-source Poisson's equation solver for extreme-scale computing with accelerators. Comput. Phys. Commun. **254**, 107272 (2020). https://doi.org/10.1016/j.cpc.2020.107272
13. Kale, V., Lu, W., Curtis, A., Malik, A.M., Chapman, B., Hernandez, O.: Toward supporting multi-GPU targets via taskloop and user-defined schedules. In: Milfeld, K., de Supinski, B.R., Koesterke, L., Klinkenberg, J. (eds.) IWOMP 2020. LNCS, vol. 12295, pp. 295–309. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58144-2_19
14. Patel, A., Doerfert, J.: Remote OpenMP offloading. In: Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP, pp. 441–442. Association for Computing Machinery, New York (2022). https://doi.org/10.1145/3503221.3508416
15. Chapman, B., et al.: Outcomes of OpenMP hackathon: OpenMP application experiences with the offloading model (Part I). In: McIntosh-Smith, S., de Supinski, B.R., Klinkenberg, J. (eds.) IWOMP 2021. LNCS, vol. 12870, pp. 67–80. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85262-7_5
16. Rydahl, A.: Why does 'default(none)' not always take effect in OpenMP? https://forums.developer.nvidia.com/t/why-does-default-none-not-always-take-effect-in-openmp/213247. Accessed 26 May 2022
17. Rydahl, A.: OpenMP Target Offloading Bug - Making Target Region in Task. https://forums.developer.nvidia.com/t/openmp-target-offloading-bug-making-target-region-in-task/216308. Accessed 12 July 2022

Reducing OpenMP to FPGA Round-Trip Times with Predictive Modelling

Julian Brandner(✉), Florian Mayer, and Michael Philippsen

Friedrich-Alexander Universität Erlangen-Nürnberg (FAU),
Programming Systems Group, Erlangen, Germany
{julian.brandner,florian.andrefranc.mayer,michael.philippsen}@fau.de

Abstract. Recent works aimed at expanding the target offloading capabilities of OpenMP to FPGA platforms. While enabling the easy construction of heterogeneous systems, the approach has to face a major hurdle: by blurring the line between software and hardware development, it forces software developers to consider hardware limitations. This can be difficult through the abstractions that OpenMP introduces over the generated hardware. The high level synthesis tools used by OpenMP compilers to generate hardware already offer predictions on hardware usage. Their value for OpenMP offloading however is questionable. This paper is based on the data mining we conducted on thousands of kernel variations. It demonstrates and proves under which circumstances these predictions can be trusted in the context of OpenMP to FPGA offloading and concludes by showing how to derive runtime performance predictions from them. The model we present can be used without experience in hardware development and quickly predicts runtime on our benchmarks with an average Pearson correlation of 0.897. This knowledge allows developers to make fast, informed design decisions.

Keywords: FPGA · OpenMP · target offloading · data mining

1 Introduction

FPGAs are already widely used for a broad variety of computational tasks. Depending on the use case they can outmatch CPUs and GPUs in performance and energy efficiency given the right domain [2,26]. However there is an obstacle to the use of FPGAs: while basic software development is a widespread skill, knowledge of hardware development is far less common. Hence, there is a need to make FPGA development more accessible for software programmers.

One way to bridge this gap is OpenMP. OpenMP already offers a model and syntax for target offloading, that conceptually can be applied to FPGA offloading. Previous works presented tools that already generate hardware for offloaded regions and to construct heterogeneous systems from OpenMP code [14,18,19,23].

Figure 1 shows the general workflow of these tools. First, the OpenMP compiler identifies the annotated regions that are to be executed on the FPGA platform.

M. Klemm et al. (Eds.): IWOMP 2022, LNCS 13527, pp. 94–108, 2022.
https://doi.org/10.1007/978-3-031-15922-0_7

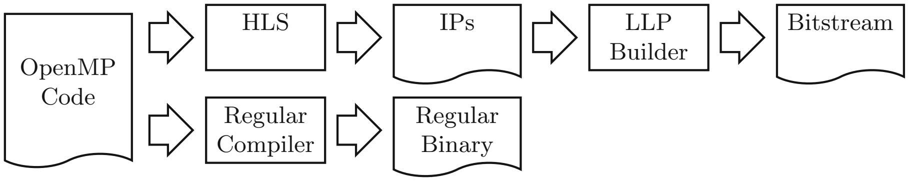


Fig. 1. Workflow of a typical OpenMP to FPGA compiler.

For each of these regions it identifies the data that has to be transmitted at runtime between host and FPGA and vice versa. This step is called target outlining. For the code that later runs on the host and that is fed into a regular CPU-compiler the outlined region is cut from the host side code and is replaced by routines that handle data transfer and launch the computation on the FPGA.

In order to program the FPGA, a so-called high level synthesis tool (HLS) translates the outlined code into a hardware description, yielding a so-called intellectual property (IP) kernel. Typically, the HLS is an of-the-shelf building block provided by FPGA vendors [4,15,18,29]. The kernel is then embedded into a low-level platform (LLP) that, apart from the actual calculation, provides the necessary infrastructure, like FPGA memory, bus system, and kernel control [19]. In general the OpenMP-to-FPGA tools use fixed LLPs since the OpenMP code itself does not provide details on the underlying hardware architecture. Finally, a Bitstream is generated from the hardware description. The resulting Bitstream and the generated host code form the heterogeneous system.

Unfortunately, this workflow cannot entirely bridge the gap between hardware and software development, as often OpenMP code developed without FPGAs in mind turns into hardware with unsatisfactory runtime performance [18]. HLS tools struggle with offloaded OpenMP code. There are two main reasons. First, HLS tools are targeted at translating C/C++ code written specifically for an FPGA, which often is different from the C/C++ code cut from an OpenMP source code. Second, HLS tools expect their users to be FPGA experts who manually tune, annotate, and adjust their codes [5,10,11,17]. Expert knowledge is crucial as a single hardware generation can often take hours or even days, hence only experts can avoid extensive experimentation and soon get to a well-performing FPGA that fits into and best exploits the available hardware space.

As the typical OpenMP developer is not an FPGA expert, the offloaded code is neither written with an FPGA mindset nor does it include HLS pragmas. The OpenMP developer needs help as due to the excessive HLS generation times compiling and testing multiple solutions is not an option. As a remedy the HLS-tools offer early estimates (on both area use and on runtime) that are available within minutes instead of hours. But even these estimates are of no help in this setting. This is because to produce reasonable and reliable numbers the HLS needs annotations and specifications on the exact path of execution. To interpret the runtime estimates a detailed understanding of the floorplan of the generated hardware and its circuits is needed. One that an average OpenMP developer without extensive hardware knowledge does not have.

This is where the contributions of this paper come in. We provide the OpenMP developer first with crucial information on the reliability of the early HLS area estimates and second with a model to predict the runtime performance of OpenMP code within minutes instead of hours, more specifically to predict the effect that HLS pragmas have on the runtime time performance when developers add them to the target regions of their OpenMP codes. Note that in contrast to the vendors' runtime performance estimates, our prediction model does not require any additional (path) specification or other user input to achieve reliable numbers.

To understand how different combinations of HLS pragmas affect the resulting FPGA runtime performance, we explored the extensive search space. We automatically annotated many OpenMP codes with various HLS pragmas and their combinations. We then measured both final hardware usage of the FPGA and the resulting runtime. We trained regression models on a part of this dataset to predict the final runtime purely from early HLS area estimates. We validated our predictions with another part of this dataset. Section 2 explains our approach and the conducted and evaluated experiments. Section 3 presents the statistics and results and discusses their impact of runtime performance predictions for OpenMP programmers that are no hardware experts. Section 4 covers the limitations and the threats to the validity of our work. Before we conclude, Sect. 5 sketches related work on hardware performance prediction.

2 Data Mining

We took a benchmark set (see Table 1) written in C + OpenMP and randomly annotated the codes with various parameterizations of a set of HLS pragmas (see Table 2). We synthesized and ran the resulting random samples, yielding an extensive dataset. On this set we evaluated the reliability of early HLS area estimates with respect to the four main FPGA resources, namely lookup tables (LUT), block ram (BRAM), flipflops (FF), and digital signal processors (DSP), and trained regression models that use these early area estimates to quickly predict the runtime more reliably and without any additional input from the developers. Let us explain the details below.

Table 1. Benchmarks.

Name	**Description**	**Performance Bottleneck**	***Floating Point***	**Functions**	**Loops**	**Samples**
SHA256	Hash	compute		15	9	542
SHA3	Hash	compute		8	13	432
MD5	Hash	compute		2	7	551
Filter	2D Convolution	memory	X	3	4	297
OE-Sort	Odd-even-transposition sort	memory		3	3	238
Mandelbrot	Mandelbrot set	compute	X	1	2	335

Table 2. Inserted HLS pragmas.

Pragma	applicable to	parameters
`HLS unroll`	loops	`factor` [1;1024]
`HLS pipeline`	loops	initiation interval (`II`) [1;32], `rewind`
`HLS inline`	functions	`region`

2.1 Benchmark Set

The benchmark set holds hash algorithms, a 2D convolution (filter), an (odd-even-transposition) sort, and Mandelbrot. The reason for this choice is that fast hashing and fast filter operations on FPGAs have been an ongoing topic for decades [1,3,6,12,20,21,25]. We included a sort as a building block of many FPGA applications [13,16]. The Mandelbrot computation is in the set as it is a widely used benchmark [9,15,31]. The six C implementations are from textbooks or taken from open source code. They are not specifically optimized for the use in HLS/FPGA. Table 1 loosely classifies them on whether their performance is compute bound or memory bound. Some benchmarks require a significant proportion of floating point arithmetics. The table also lists the number of functions and loops in each benchmark's target region (which itself forms the top level function after the outlining). These are the hooks where we later add the HLS pragmas. We postpone the discussion of the last column (samples) to Sect. 2.3. For each benchmark we generate a random input set that we later use to measure the runtime performance of the generated Bitstream.

2.2 Inserted HLS Pragmas

HLS pragmas manipulate the way code is synthesized, control the degree of parallelism that the hardware achieves, and thus have a huge impact on both the area demand and the runtime performance of the FPGA. We chose the pragmas from Table 2 for their ability to control the degree of parallelism in FPGA logic (in contrast to memory). All these pragmas keep the semantic correctness of the program intact. They are mere hints to the HLS. For example, if an HLS pragma suggests to parallelize a loop but existing data dependencies disallow that parallelization, no parallel hardware is generated.

At each loop in a given benchmark code we inserted (at most) one HLS loop pragma (`unroll` or `pipeline`) plus pragma-specific parameter values and optional flags. In total there are up to $1090 = 1024 + 1 + 64 + 1$ variants that can be the result of pragma insertions per loop.

The `unroll` pragma unrolls the annotated loop, enabling the synthesis of parallel hardware for the loop bodies where possible. The pragma can supplied with an explicit unroll `factor`. We considered the range [1;1024] for it. If the `factor` is omitted, the HLS will try to fully unroll the loop where possible.

The `pipeline` pragma instructs the HLS to construct a pipeline from the annotated loop. We only applied the pragma to loops. While conceptually it is feasible to ask the HLS to figure out the minimal initiation interval (`II`) of such a pipeline, i.e., the latency between the completion of two consecutive iterations, our pragma insertion process only considers explicitly specified `II` values in the range [1;32]. Moreover, if there is only one single pipelined loop within an OpenMP target region and if there is neither a conditional statement nor function call outside of it, then an optional `rewind` flag can be added to further decrease latency. For codes where `rewind` is feasible we considered two versions per `II` value, i.e., with and without the `rewind` flag.

At each function in the given benchmark code we can insert the HLS `inline` pragma, with or without an optional `region` flag. This yields 3 potential variants as a result of the pragma insertion per function. The `inline` pragma controls the inlining of functions, allowing their content to be executed in parallel with their caller. The annotated function is inlined upwards unless the optional parameter `region` is set. Then the function calls within the function are instead inlined into the function body (downwards). In theory these two options are not exclusive to each other, but we treat them as such, since applying both to the same function tends to lead to excessive hardware bloat.

2.3 Search Space Exploration

The mentioned set of pragmas and parameters leads to a vast parameter space on our benchmarks. For instance, for the SHA3 benchmark there are in the order of 10^{28} combinations, even after we remove meaningless ones (e.g., we do not unroll a statically bound loop beyond its number of iterations, we do not inline a function downwards if it does not have an inner function call, etc.). Since it is infeasible to fully explore the whole parameter space, we only generate random samples to acquire a representative dataset.

For each benchmark we construct samples in two phases. The initialization creates a set of samples, say 500. The refining inspects the set and adds more samples that hopefully inhabit undiscovered areas of the search space. The refining is repeated until a given time budget, say a week, is up.

To construct an initial sample, we randomly select 25% of the hooks in a benchmark code to insert one of the applicable pragmas. If that pragma has a parameter we use a geometric distribution to pick its value. The `factor` parameter of the `unroll` pragma was omitted with a probability of 20%. If feasible we add `rewind` or `region` with a chance of 50%. After construction of the sample, the HLS either provides an early area estimation or indicates unsynthesizability, for instance if a large unroll factor is combined with heavy inlining into the respective loop because this may need more space than the target FPGA can offer. We purge unsynthesizable samples from the set.

The second phase refines the set of samples. Often there are samples whose pragmas or whose parameter values have little or no impact on the area estimates. The HLS estimates of those samples are close to each other; the samples form clusters around similar area estimates. For example a pragma that suggests parallelism if there are data dependencies does not affect the FPGA area estimates at all. The idea of the refining phase is to take representatives from such

Table 3. Fraction of unsynthesizable codes. HLS and Bitstream generation.

Name	HLS	Bitstream
SHA256	17%	4%
SHA3	29%	7%
MD5	19%	1%
Filter	9%	7%
OE-Sort	31%	1%
Mandelbrot	4%	0%

clusters of samples with similar area estimates and to modify, extend, and alter the inserted pragmas, hoping to end up with new samples, whose area estimates are outside of already discovered clusters, i.e., cover previously unknown areas of the search space. Alteration of a sample is again randomized. When we generate a new pragma for a hook, we either replace the preexisting pragma (50%) or remove it completely (50%).

To find the clusters and to pick the samples that we use to generate new samples we rank all samples by their distance to their nearest neighbor based on the utilized FPGA area (low to high). We then randomly pick 25% of the samples with a probability that is proportional to their rank (rank 1 being the most unlikely) and generate a variation as described above. The new sample is again fed to the HLS for area estimates. An unsynthesizable sample is purged. If the time budget is not up, there is another round of refining. The “samples” column Table 1 shows how many samples we generated for each of the benchmark codes (in a week for the hash benchmarks and in five days for the others). The HLS-column of Table 3 shows the fraction of generated samples that are unsynthesizable. Our benchmarks as well as the samples generated from them are available from https://github.com/FAU-Inf2/OpenMP2FPGASamples.git.

2.4 Performance Measurements

After completing the HLS we build Bitstreams from the (non-purged) samples. This step can again fail due to the hardware getting too complicated, see the Bitstream-column of Table 3. We again purge failing samples from the set.

For each of the successfully generated Bitstreams we measured its runtime performance on our test system: an Intel Core i7-4770 CPU running Ubuntu 20.04.4 LTS and communicating with a Xilinx VCU118 FPGA board via PCI express (speed 5GT/s, width ×4). We invoked the heterogeneous system with the benchmark’s input data and measured the runtime that is spent on computations on the FPGA. These runtime measurements ignore the runtime of the host-side code as well as the delays caused by data transfers between host and FPGA. In total we have three different sets of data for each sample: the HLS area estimates, that are available after the HLS is completed, the final area reports, that are generated together with the Bitstream, and the runtime of the respective sample.

2.5 Predictive Modeling and Interpretation

To quickly predict the runtime performance from early area estimates, i.e., without having to await the Bitstream generation and without needing additional annotations, we trained regression models on our datasets. We set aside 30% of each dataset for the evaluation. We use the remaining 70% to train three regressors:

- **Linear:** We fit a linear regression model using the method of least squares.
- **Tree:** We train regression trees using the analysis of variance method. We set the minimum split size to 20 and the cost complexity parameter to 0.001.
- **Random Forest:** We conduct a random forest regression with 9 trees per model and consider all input axes per tree.

We measure and interpret the prediction quality of the models by calculating the correlation between the predicted value and the measurement. To do so we utilize the Pearson correlation coefficient r_p as well as the Spearman rank correlation coefficient r_s. Depending on the use case either coefficient can quantify desirable qualities in a prediction. While an automatic optimization procedure (e.g. generic optimization, simulated annealing) may only be interested in the rank of measurements to gradually improve the result, a human manually optimizing the code may be better represented by Pearson's r_p.

3 Results

3.1 Reliability of Early HLS Area Estimates

When offloading onto an FPGA, OpenMP developer have to consider both the runtime performance as well as the resource consumption of their codes. As this is usually done with the vendors' HLS area estimates, we examine their reliability. For all our samples, Table 4 compares these area estimates to the reported actual area usage of the generated Bitstream and correlates these measures.

Table 4. Reliability of early area estimates measured by correlation.

All	LUT		BRAM		FF		DSP	
Samples	r_p	r_s	r_p	r_s	r_p	r_s	r_p	r_s
SHA256	0.890	0.850	0.622	0.678	0.903	0.891	-[2]	-[2]
SHA3	0.957	0.895	0.707	0.665	0.903	0.890	-[2]	-[2]
MD5	0.997	0.893	0.404	0.678	0.998	0.907	-[2]	-[2]
Filter	0.591	0.636	-[1]	-[1]	0.518	0.676	0.346	0.382
OE-Sort	0.995	0.679	-[1]	-[1]	0.982	0.616	-[2]	-[2]
Mandelbrot	0.454	0.501	-[1]	-[1]	0.375	0.533	0.096	0.484

[1] The BRAM usage is reported in 36 kb tiles. If a benchmark uses so few BRAM tiles that the number of reported tiles barely fluctuates, the correlation is undefined.

[2] Only floating point kernels use DSPs.

Table 5. Correlation of area estimates with FPGA runtimes.

All	LUT		BRAM		FF		DSP	
Samples	r_p	r_s	r_p	r_s	r_p	r_s	r_p	r_s
SHA256	−0.568	−0.153	0.472	0.390	−0.608	−0.150	-	-
SHA3	−0.515	−0.407	0.587	0.527	−0.394	−0.390	-	-
MD5	0.013	−0.556	0.346	0.568	0.044	−0.460	-	-
Filter	0.013	−0.006	−0.213	−0.184	−0.005	−0.008	−0.151	−0.118
OE-Sort	−0.384	−0.169	0.182	0.212	−0.369	−0.195	-	-
Mandelbrot	−0.161	−0.455	0.055	0.004	−0.176	−0.462	−0.067	−0.398

There are some main insights: For benchmarks that require a large proportion of *floating point* arithmetics the early area estimates are not very reliable, especially in terms of their usage of digital signal processors (DSPs, $r_p < 0.5$) but also with regard to the other measures. OpenMP developers should use these estimates with care.

For benchmarks that do not need floating point arithmetics and hence do not need DSPs in their IPs, the area estimates for the lookup table (LUT) and for flipflops (FF) often are fairly reliable (correlation above 85%). For BRAM cells there still is a strong correlation (62%–69%). This is crucial for OpenMP developers as it allows them to optimize the target regions of their codes, i.e., they can add/modify HLS pragmas and quickly and reliably check the effects with the early HLS estimates. But there is a caveat: If the OpenMP developer modifies the HLS pragma (or its parameters) only slightly and subsequently sees only a small effect on area consumption, this estimated small effect is not reliable, only large effects are. In Table 4 this can be seen in particular for OE-Sort. On this benchmark many pragma parameterizations only have a small impact on the final hardware usage. This causes the rank of the samples (and thereby r_s) to be distorted by noise (while r_p is still strong).

3.2 Correlating Area and Runtime Performance

Predicting the runtime performance of an FPGA solution from its area estimates sounds like a reasonable approach as FPGAs often derive their performance from highly parallel hardware, that correspondingly requires a lot of resources.

We checked the validity of this assumption with quantitative measurements on all samples, i.e., on the entire dataset, see Table 5. For the usage of LUTs and FFs the assumption holds, although the correlation fluctuates between the benchmarks. There is the expected negative correlation between resource usage and runtime, but in some cases it is purely by rank (MD5, LUT, $r_s = -0.556$) or purely linear by value (SHA256, LUT, $r_p = -0.568$). For digital signal processor usage there is also a negative correlation, but it less pronounced, as predictions for *floating point* benchmarks are again less reliable (see Sect. 3.1). In contrast, for the BRAM numbers more hardware usage does not cause better runtime performance. More BRAM usage even seems to indicate slower hardware.

Table 6. Correlation of regression models with FPGA runtimes.

Test Samples	Linear		Tree		Random Forest	
	r_p	r_s	r_p	r_s	r_p	r_s
SHA256	0.674	0.344	0.774	0.691	0.847	0.728
SHA3	0.648	0.629	0.863	0.725	0.876	0.735
MD5	0.384	0.587	0.882	0.787	0.932	0.863
Filter	0.530	0.563	0.963	0.660	0.960	0.729
OE-Sort	0.255	0.275	0.679	0.444	0.798	0.538
Mandelbrot	0.812	0.703	0.956	0.711	0.969	0.826

For the OpenMP developer this means that naive attempts to improve the performance simply by increasing hardware usage are bound to fail. While there is a clear correlation it is too inconsistent to be of practical use for the purpose of optimization by means of iteratively fiddling with HLS pragmas.

3.3 Predicting Runtime Performance

Regression models excel at combining multiple varying relationships (like those shown in Sect. 3.2) into a strong prediction. We explore three predictors for the runtime performance of an FPGA that take the four early area estimates as their inputs. These models were trained on the training data. Table 6 holds the results as computed on the evaluation data, i.e., the samples set aside.

The Linear model already generates a meaningful prediction (averages: $r_p = 0.546$, $r_s = 0.517$) that works well, especially for *Mandelbrot*, but fails for OE-Sort in the evaluation dataset. Due to the non-linear nature of the prediction task the Linear model still leaves room for improvement.

The Tree based predictor (averages: $r_p = 0.853$, $r_s = 0.670$) and especially the Random Forest regression (averages: $r_p = 0.897$, $r_s = 0.736$) nicely predict FPGA runtimes for all benchmarks. Only the comparatively small impact of pragmas on OE-Sort again hurts the rank correlation on this benchmark.

Hence, the OpenMP developer has a highly reliable runtime prediction at his/her fingertips within minutes. The model can provide fast and meaningful feedback on the runtime effect of added or modified HLS pragmas, while being trivial and quick to acquire.

4 Threats to Validity

We see three possible threats to the validity of our results.

While we strictly separate the training data from the test data during model training, both sets contained samples constructed from the same benchmarks. In practice, however, an ideal regression would deliver reliable predictions for arbitrary OpenMP programs with target regions. We still think our work is relevant. First, as discussed in Sect. 1, our benchmarks are typical examples of FPGA use cases and many OpenMP target regions are similar to the ones we cover. Second, the benchmark set and the diverse set of samples that we constructed from them by adding parameterized HLS pragmas are both publicly available, so the methodology we use to add pragmas to a benchmark set and to construct a diverse set of samples for the search space exploration can be applied to a larger and more diverse dataset. This would yield a more universally applicable prediction model. A larger benchmark set also allows for future research that checks if more advanced machine learning approaches can achieve even better predictions.

We conducted our experiments using only one FPGA platform and with the affiliated vendor tool. Our work is still relevant to a wide range of users since this platform is based on the common UltraScale FPGA architecture and the vendor Xilinx has more than a 50% share of the FPGA market. Our methodology, the benchmark codes, and the generated samples can be used to construct predictors for other platforms as other vendors offer similar estimates.

Finally, our models only learn from synthesizable samples that generated a working Bitstream. They cannot predict unsynthesizability. Which samples are unsynthesizable depends on particular FPGA board used. It even fluctuates between FPGAs from the same architecture. For a different board, the predictors hence have to be re-trained (which can be done with the available benchmark set and the samples). But even without such a re-training the results form Sect. 3.1 indicate that the early HLS area estimates suffice for the OpenMP developer to reason about the feasibility, as there is strong correlation to the actual hardware demands (at least for non-floating point codes).

5 Related Work

Our work relates to HLS resource and performance prediction, to HLS design space exploration (DSE), and to OpenMP-to-FPGA compilers. We organize the discussion accordingly.

In the first research field, several groups found accuracy gaps between the early HLS estimates on both resource usage and runtime performance and the actual values that the generated hardware achieves. We look at some of the projects below. They all explore data mining, machine learning, or other approaches to improve the HLS estimates. They have in common that they focus on pure C/C++ code, that they mostly use static analysis to derive resource and runtime data, and that they are not predicting the effect that HLS pragmas have on the resource usage and on the runtime performance of the resulting FPGA. In contrast, we employ OpenMP programs, help the developer in picking suitable HLS pragmas, and collect the area and runtime data on a fully generated Bitstream on a real FPGA board. Let us sketch some representative projects now.

Ustun et al. [30] notice that the HLS often mispredicts the actual usage of DSPs, because the FPGA toolchain only later decides where to use a DSP. The authors use Graph Neural Networks (GNN) to better predict the DSP usage for a given code segment and also to also improve the HLS's delay estimates. As their training codes mainly are randomly generated basic blocks, it is unknown whether the approach also works on benchmark codes with more complex control flows. In contrast to our work, they do neither benchmark nor predict the effect of HLS pragmas. They also did not measure on a real FPGA board.

Makrani et al. [22] use Machine Learning to improve the HLS estimates on throughput and throughput-to-area for a given code segment (instead of area and runtime). In addition to the HLS estimates, the authors use the maximal clock frequency (F_{max}) of their benchmark codes (found with Minverva [7]) to train the model. We may look into adding F_{max} to our models in the future. They do not explore the effect of HLS pragmas and also do not measure on a real FPGA board.

HlsPredict [24] by O'Neal et al. predicts an FPGA's runtime performance and power consumption. The authors profile each benchmark code with predefined workloads both on a regular CPU and on an FPGA board. They train their model with the measured data. While the authors manually optimize their benchmarks with HLS pragmas, their tool does not help in gauging the effect of such pragmas on the FPGA area or its runtime performance.

HLS design space exploration (DSE) also shares common ground with our work, as researchers try to automatically find the best set of HLS pragmas (or other means) to generate the best possible hardware. A survey by Schafer et al. [28] identifies three general approaches. *Synthesis-based* works use unmodified HLS resource usage and runtime performance estimates to guide the DSE [8,27]. Because it is time consuming to run the HLS in each DSE iteration, *supervised learning* approaches first build a custom model from previously collected HLS estimates and use that model instead of the HLS to guide the DSE [32,35]. Finally, *graph analysis based* DSE bypasses the HLS and builds a graph representation for each input. Resource and performance metrics to guide the DSE [33,34] are extracted from these graphs.

We exclude from the discussion the many publications mentioned in the survey that do not target FPGAs or that do not use C/C++ as input language. We exclude the remaining synthesis-based approaches as they neither predict HLS metrics nor aid the programmer with these predictions. The few remaining graph-based methods that bypass the HLS are far away from our work. As far as we know, only the *supervised learning* approach by Zhong et al. [35] is closely related to our work as it predicts the area and performance to accelerate the DSE. But in contrast to our work it only focuses on code that solely consists of loops, it only covers the `unroll` pragma, and it does not deal with OpenMP.

Finally, to the best of our knowledge, **OpenMP-to-FPGA compilers** do not yet use the early HLS estimates on hardware usage to automatically improve the runtime performance, nor do they employ them to aid the programmer in picking HLS pragmas [14,15,19].

6 Conclusion

We generated a dataset by randomly annotating the OpenMP target regions of benchmark codes with HLS pragmas. The benchmarks and the generated annotated samples are available online. We measured the runtime performance of these samples on fully generated Bitstreams on a real FPGA board and trained regression models to predict the runtime performance from the quickly available early HLS area estimates. The output of our fast prediction models reaches average Pearson and Spearman correlations to the runtime of $r_p = 0.897$, $r_s = 0.736$ and is trivial to interpret, so that the OpenMP developer does not need hardware expertise to use it as feedback. These insights can help the developer to make informed decisions on the selection of HLS pragmas to add to her/his code. This decreases development round-trip times and thereby increases productivity.

Future work will utilize our prediction model to fully automatize the pragma selection process. The idea is that some meta heuristics will optimize the model output and thereby reliably increase performance of a given OpenMP code with target regions for an FPGA.

Acknowledgments. The authors acknowledge the financial support by the Federal Ministry of Education and Research of Germany in the framework of ORKA-HPC (project number 01IH17003A).

References

1. Al-Odat, Z.A., Ali, M., Abbas, A., Khan, S.U.: Secure hash algorithms and the corresponding FPGA optimization techniques. ACM Comput. Surv. **53**(5), 1–36 (2020). https://doi.org/10.1145/3311724. Accessed 19 May 2022
2. Asano, S., Maruyama, T., Yamaguchi, Y.: Performance comparison of FPGA, GPU and CPU in image processing. In: Proceedings of the International Conference on Field Programmable Logic and Applications, pp. 126–131. Prague, Czech Republic, September 2009. https://doi.org/10.1109/FPL.2009.5272532. Accessed 19 May 2022
3. Baldwin, B., et al.: FPGA implementations of the round two SHA-3 candidates. In: : Proceedings of the International Conference on Field Programmable Logic and Applications, pp. 400–407. Milan, Italy, September 2010. https://doi.org/10.1109/FPL.2010.84. Accessed 19 May 2022
4. Bosch, J., et al.: Application acceleration on FPGAs with OmpSs@FPGA. In: Proceedings of the International Conference on Field-Programmable Technology (FPT 2018), pp. 70–77. Naha, Okinawa, Japan, December 2018. https://doi.org/10.1109/FPT.2018.00021. Accessed 19 May 2022
5. Cho, M., Kim, Y.: Implementation of data-optimized FPGA-based accelerator for convolutional neural network. In: Proceedings of the International Conference on Electronics, Information, and Communication (ICEIC 2020), pp. 1–2. Barcelona, Spain, January 2020. https://doi.org/10.1109/ICEIC49074.2020.9050993. Accessed 19 May 2022
6. Evans, J.B.: Efficient FIR filter architectures suitable for FPGA implementation. IEEE Trans. Circ. Syst. II: Analog Digit. Sig. Process. **41**(7), 490–493 (1994). https://doi.org/10.1109/82.298385. Accessed 19 May 2022

7. Farahmand, F., Ferozpuri, A., Diehl, W., Gaj, K.: Minerva: automated hardware optimization tool. In: Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig 2017), pp. 1–8. Cancun, Mexico, December 2017. https://doi.org/10.1109/RECONFIG.2017.8279804. Accessed 19 May 2022
8. Ferretti, L., Ansaloni, G., Pozzi, L.: Lattice-traversing design space exploration for high level synthesis. In: Proceedings of the International Conference on Computer Design (ICCD 2018), pp. 210–217. Orlando, FL, October 2018. https://doi.org/10.1109/ICCD.2018.00040. Accessed 19 May 2022
9. Färber, C., Schwemmer, R., Machen, J., Neufeld, N.: Particle identification on an FPGA accelerated compute platform for the LHCb upgrade. IEEE Trans. Nuclear Sci. **64**(7), 1994–1999 (2017). https://doi.org/10.1109/TNS.2017.2715900. Accessed on May 19, 2022
10. Georgopoulos, K., et al.: An evaluation of Vivado HLS for efficient system design. In: Proceedings of the International Conference Symposium on Electronics in Marine (ELMAR 2016), pp. 195–199. Zadar, Croatia, September 2016. https://doi.org/10.1109/ELMAR.2016.7731785. Accessed 19 May 2022
11. González, R., Sutter, G., Patiño, H.D.: Optimized ud filtering algorithm for floating-point hardware execution. In: Proceedings of the International Conference on Information Fusion (FUSION 2014), pp. 1–6. Salamanca, Spain July 2014
12. Gramata, P., Trebatickỳ, P., Gramatová, E.: The MD5 message-digest algorithm in the XILINX FPGA. In: Proceedings of the International Workshop on Field Programmable Logic and Applications, pp. 126–128. Prague, Czech Republic, September 1994. https://doi.org/10.1007/3-540-58419-6_79. Accessed 19 May 2022
13. Hematian, A., Chuprat, S., Manaf, A.A., Parsazadeh, N.: Zero-delay FPGA-based odd-even sorting network. In: Proceedings of IEEE Symposium on Computers Informatics (ISCI 2013), pp. 128–131. Langkawi, Malaysia, April 2013. https://doi.org/10.1109/ISCI.2013.6612389. Accessed 19 May 2022
14. Huthmann, J., Sommer, L., Podobas, A., Koch, A., Sano, K.: OpenMP device offloading to FPGAs using the Nymble infrastructure. In: Proceedings of the International Workshop on OpenMP (IWOMP 2020), pp. 265–279. Austin, TX. September 2020. https://doi.org/10.1007/978-3-030-58144-2_17. Accessed 19 May 2022
15. Knaust, M., Mayer, F., Steinke, T.: OpenMP to FPGA Offloading Prototype using OpenCL SDK. In: Proceedings of the International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2019), pp. 387–390. Rio de Janeiro, Brazil, May 2019. https://doi.org/10.1109/IPDPSW.2019.00072. Accessed 19 May 2022
16. Lipu, A.R., Amin, R., Islam Mondal, M.N., Mamun, M.A.: Exploiting parallelism for faster implementation of bubble sort algorithm using FPGA. In: Proceedings of the International Conference on Electrical, Computer Telecommunication Engineering (ICECTE 2016), pp. 1–4. Rajshahi, Bangladesh, December 2016. https://doi.org/10.1109/ICECTE.2016.7879576. Accessed 19 May 2022
17. Martinez Vallina, F.: Implementing memory structures for video processing in the Vivado HLS tool. Xilinx Appl. Notes **793**, 1–8 (2012)
18. Mayer, F., Brandner, J., Hellmann, M., Schwarzer, J., Philippsen, M.: The ORKA-HPC compiler–practical OpenMP for FPGAs. In: Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (LCPC 2021). LNCS, vol. 13181, pp. 83–97. Springer, Newark (2022). https://doi.org/10.1007/978-3-030-99372-6_6. Accessed 19 May 2022

19. Mayer, F., Knaust, M., Philippsen, M.: OpenMP on FPGAs-a survey. In: Proceedings of the International Workshop on OpenMP (IWOMP 2019), pp. 94–108. Auckland, New Zealand, August 2019. https://doi.org/10.1007/978-3-030-28596-8_7. Accessed 19 May 2022
20. McEvoy, R.P., Crowe, F.M., Murphy, C.C., Marnane, W.P.: Optimisation of the SHA-2 family of hash functions on FPGAs. In: Proceedings of IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI 2006), pp. 317–322. Karlsruhe, Germany, March 2006. https://doi.org/10.1109/ISVLSI.2006.70. Accessed 19 May 2022
21. Meher, P.K., Chandrasekaran, S., Amira, A.: FPGA realization of FIR filters by efficient and flexible systolization using distributed arithmetic. IEEE Trans. Signal Process. **56**(7), 3009–3017 (2008). https://doi.org/10.1109/TSP.2007.914926. Accessed on May 19, 2022
22. Mohammadi Makrani, H., et al.: Pyramid: machine learning framework to estimate the optimal timing and resource usage of a high-level synthesis design. In: 2019 29th International Conference on Field Programmable Logic and Applications (FPL 2019), pp. 397–403. Barcelona, Spain, September 2019. https://doi.org/10.1109/FPL.2019.00069. Accessed 19 May 2022
23. Nepomuceno, R., Sterle, R., Valarini, G., Pereira, M., Yviquel, H., Araujo, G.: Enabling OpenMP task parallelism on multi-FPGAs. arXiv:2103.10573 [cs.DC] (March 2021). https://doi.org/10.1109/FCCM51124.2021.00047. Accessed 19 May 2022
24. O'Neal, K., Liu, M., Tang, H., Kalantar, A., DeRenard, K., Brisk, P.: HLSPredict: cross platform performance prediction for FPGA high-level synthesis. In: Proceedings of the International Conference on Computer-Aided Design (ICCAD 2018), pp. 1–8. San Diego, CA, November 2018. https://doi.org/10.1145/3240765.3264635. Accessed 19 May 2022
25. Park, S.Y., Meher, P.K.: Efficient FPGA and ASIC realizations of a DA-based reconfigurable FIR digital filter. IEEE Trans. Circ. Syst. II: Exp. Briefs **61**(7), 511–515 (2014). https://doi.org/10.1109/TCSII.2014.2324418. Accessed 19 May 2022
26. Qasaimeh, M., Denolf, K., Lo, J., Vissers, K., Zambreno, J., Jones, P.H.: Comparing energy efficiency of CPU, GPU and FPGA implementations for vision kernels. In: Proceedings IEEE International Conference on Embedded Software and Systems (ICESS 2019), pp. 1–8. Las Vegas, NV, June 2019. https://doi.org/10.1109/ICESS.2019.8782524. Accessed 19 May 2022
27. Schafer, B.C., Wakabayashi, K.: Divide and conquer high-level synthesis design space exploration. ACM Trans. Des. Autom. Electron. Syst. **17**(3), 1–19 (2012). https://doi.org/10.1145/2209291.2209302. Accessed on May 19, 2022
28. Schafer, B.C., Wang, Z.: High-level synthesis design space exploration: past, present, and future. IEEE Trans. Comput. Aid. Des. Integr. Circ. Syst. **39**(10), 2628–2639 (2020). https://doi.org/10.1109/TCAD.2019.2943570. Accessed 19 May 2022
29. Sommer, L., Korinth, J., Koch, A.: OpenMP device offloading to FPGA accelerators. In: Proceedings of the International Conference on Application-specific Systems, Architectures and Processors (ASAP 2017), pp. 201–205. Seattle, WA, July 2017. https://doi.org/10.1109/ASAP.2017.7995280. Accessed 19 May 2022
30. Ustun, E., Deng, C., Pal, D., Li, Z., Zhang, Z.: Accurate operation delay prediction for FPGA HLS using graph neural networks. In: Proceedings of the International Conference on Computer Aided Design (ICCAD 2020), pp. 1–9. San Diego, CA, November 2020. https://doi.org/10.1145/3400302.3415657. Accessed 19 May 2022

31. Wang, K., Nurmi, J.: Using OpenCL to rapidly prototype FPGA designs. In: Proceedings of the IEEE Nordic Circuits and Systems Conference (NORCAS 2016, pp. 1–6. Copenhagen, Denmark, November 2016. https://doi.org/10.1109/NORCHIP.2016.7792907. Accessed 19 May 2022
32. Zacharopoulos, G., Barbon, A., Ansaloni, G., Pozzi, L.: Machine learning approach for loop unrolling factor prediction in high level synthesis. In: Proceedings of the International Conference on High Performance Computing Simulation (HPCS 2018), pp. 91–97. Orleans, France, July 2018. https://doi.org/10.1109/HPCS.2018.00030. Accessed 19 May 2022
33. Zhao, J., Feng, L., Sinha, S., Zhang, W., Liang, Y., He, B.: COMBA: a comprehensive model-based analysis framework for high level synthesis of real applications. In: Proceedings of the International Conference on Computer-Aided Design (ICCAD 2017), pp. 430–437. Irvine, CA, November 2017. https://doi.org/10.1109/ICCAD.2017.8203809. Accessed 19 May 2022
34. Zhong, G., Prakash, A., Liang, Y., Mitra, T., Niar, S.: Lin-analyzer: a high-level performance analysis tool for FPGA-based accelerators. In: Proceedings of the International Conference on Design Automation (DAC 2016), pp. 1–6. Austin, TX, June 2016. https://doi.org/10.1145/2897937.2898040. Accessed 19 May 2022
35. Zhong, G., Venkataramani, V., Liang, Y., Mitra, T., Niar, S.: Design space exploration of multiple loops on FPGAs using high level synthesis. In: Proceedings of the International Conference on Computer Design (ICCD 2014), pp. 456–463. Seoul, South Korea, October 2014. https://doi.org/10.1109/ICCD.2014.6974719. Accessed 19 May 2022

OpenMP Tool Support

Improving Tool Support for Nested Parallel Regions with Introspection Consistency

Vladimir Indic[1](✉) and John Mellor-Crummey[2]

[1] Faculty of Technical Sciences, University of Novi Sad, Novi Sad, Serbia
vladaindjic@uns.ac.rs

[2] Department of Computer Science, Rice University, Houston, TX 77251-1892, USA
johnmc@rice.edu

Abstract. The OpenMP 5 standard defines OMPT—an application programming interface for tools that includes a set of introspection routines. At any point in time, a sampling-based performance tool may invoke these introspection routines from a signal handler to inquire about the nesting of parallel and task regions. Unfortunately, the OpenMP 5 standard doesn't precisely specify what one may observe with these routines when monitoring a program as it executes nested parallel regions. To address this shortcoming, we propose that the OpenMP standard require that an OpenMP implementation supports *introspection consistency*. This paper defines introspection consistency, describes why tools need it, and explains a novel strategy for implementing it using wait-free coordination between an OpenMP implementation and its OMPT introspection routines. We describe an implementation of this technique in the LLVM OpenMP runtime and evaluate the runtime overhead of supporting introspection consistency in LLVM OpenMP using a microbenchmark for nested parallel regions and SPEC OMP2012.

Keywords: OpenMP · OMPT · wait-free coordination

1 Introduction

The OpenMP Application Programming Interface (API) defines a directive-based programming model for harnessing parallelism within nodes that employ one or more multicore processors and sometimes accelerators. Since the inception of OpenMP, providing tools that support two or more OpenMP implementations has been a challenge. The substantial semantic gap between an OpenMP program and its implementation, compounded by differences between OpenMP implementations, makes it difficult for tools to attribute performance metrics

This work was supported in part by Faculty of Technical Sciences, Novi Sad, Serbia and the Exascale Computing Project (17-SC-20-SC)—a collaborative effort of the U.S. DOE Office of Science and the National Nuclear Security Administration.

M. Klemm et al. (Eds.): IWOMP 2022, LNCS 13527, pp. 111–126, 2022.
https://doi.org/10.1007/978-3-031-15922-0_8

to a source-level calling contexts. To bridge this gap, Eichenberger et al. [5] proposed OMPT—an API for performance analysis and correctness tools. In 2018, OMPT became an integral part of the OpenMP 5.0 standard [10].

The OMPT API defines a set of runtime entry points designed to support asynchronous introspection of the state of an executing OpenMP program by a sampling-based performance tool. To retrieve information about the current parallel (respectively, task) region, a sampling-based performance tool asynchronously invokes the `ompt_get_parallel_info` (`ompt_get_task_info`) runtime entry point passing `ancestor_level` $= 0$; information about enclosing parallel (task) regions can be obtained by specifying values of `ancestor_level` > 0. To date, not enough attention has been paid to the semantics of these routines. The standard simply states that `ompt_get_parallel_info` (respectively, `ompt_get_task_info`) returns

- 2 if a parallel (task) region exists at the specified ancestor level and information about the parallel (task) region is available,
- 1 if a parallel (task) region exists at the specified ancestor level but information is currently unavailable, and
- 0 otherwise.

While integrating support for the OMPT interface into Rice University's HPCToolkit performance tools [1,13], which use asynchronous sampling to collect call path profiles and traces [12] for CPU threads as a program executes, it became clear that this definition of the semantics for the `ompt_get_parallel_info` (`ompt_get_task_info`) introspection routines is too weak. There are several problems.

- After an OpenMP program (1) enters a parallel (respectively, task) region and (2) the OpenMP implementation provides information about that region to a tool by returning 2 to an introspection query, there is no requirement that the OpenMP introspection routine must continue to return information about the region until the program begins to exit the region.
- If a tool stores into the `ompt_data_t parallel_data` (respectively, `ompt_task_data`) word maintained by an OpenMP implementation for a parallel (task) region, a tool will not be reliably able to use an OpenMP introspection routine to retrieve this data throughout the lifetime of the region.

These problems are not just theoretical. We have observed both issues in practice while using HPCToolkit to profile an OpenMP program linked againts the LLVM OpenMP runtime. When a thread receives a sample, it executes HPCToolkit's signal handler code and unwinds the call stack to determine the context that incurs cost. Worker threads of OpenMP parallel regions can only determine partial call paths because they are unaware of the region's invocation context known only to the primary thread. To compute a full, user-level calling context, a worker thread subscribes to receive the region context from the primary thread by updating a tool data structure whose pointer is stored in the active region's `parallel_data` word. At the end of the parallel region, the primary thread reads `parallel_data` to obtain the pointer to the tool data structure, unwinds the call stack to determine the region's context, and shares it with subscribed worker threads that then asynchronously assemble full call paths for samples received while executing the region.

In the LLVM OpenMP runtime, we noticed timing windows while creating nested parallel regions during which HPCToolkit receives wrong information about enclosing parallel regions, causing a sampled thread to be unable to determine if it is a worker thread that should subscribe to receive the region's context. Furthermore, runtime might override the content of an enclosing region's `parallel_data` while creating a nested parallel region; this causes the loss of the worker threads' subscriptions, which causes the profiler to fail.

Such issues can be avoided by improving the OpenMP specification and tightly integrating the implementation of OpenMP parallel and task regions with their OMPT introspection routines. This paper makes the following contributions:

– We propose that the OpenMP specification requires that an OpenMP implementation supports *introspection consistency*; in brief, this requires that when a program is executing in an OpenMP parallel or task region, the region must be visible to OMPT introspection routines.
– We explain why introspection consistency is needed to support reliable sampling-based performance tools for call path profiling.
– We describe a novel approach for implementing introspection consistency using wait-free coordination between an OpenMP implementation and its OMPT introspection routines.
– We overview an implementation of introspection consistency in LLVM OpenMP.
– We evaluate the runtime overhead of supporting introspection consistency in LLVM OpenMP using a microbenchmark for nested parallel regions and SPEC OMP2012.

Section 2 briefly describes the implementation of parallel regions, tasks[1], and OpenMP introspection routines in LLVM OpenMP to provide context for understanding our contributions. The remaining sections motivate introspection consistency, describe a high-level approach to providing it, describe our implementation in LLVM OpenMP, evaluate its cost, discuss related work, and present our conclusions.

2 Background

This section briefly describes the implementation of parallel regions, serialized parallel regions, implicit and explicit tasks, and OMPT introspection support for nested parallel and task regions in LLVM OpenMP runtime.

Parallel Regions and Implicit Tasks. When a thread encounters a parallel region construct that at least two threads should execute, it must form a team of threads

[1] We use the word "task" as a synonym for a task region for brevity.

that will execute that region. The encountering thread becomes the primary thread of the region's team. Prior to entering this region, the primary thread initializes the team descriptor, records the number of threads in the team, sets the descriptor's parent pointer to the enclosing parallel region (if any), assembles a team of threads for the region, shares the descriptor with the team of threads, and sets its own current team descriptor to the region's team descriptor. Then, each thread of the region's team begins an implicit task that executes the code for the region. Before scheduling the implicit task for execution, a thread assembles the task descriptor, updates its team and scheduling parent pointers to link the parallel region's team descriptor and the enclosing task, respectively, and finally sets its own current task descriptor to the assembled task descriptor. After every thread completes the work of its implicit task and synchronizes with a final barrier, it destroys the current implicit task descriptor. Afterward, the primary thread destroys the parallel region's team descriptor and resumes work in the context of the region's parent.

Explicit Tasks. An explicit task may suspend its execution after the creation. Furthermore, multiple threads may schedule and suspend the execution of an untied explicit task, causing the runtime to update the task's scheduling parent pointer in its task descriptor when the untied task was rescheduled.

Serialized Parallel Regions. A serialized parallel region is executed by a single thread. Serialized parallel regions occur frequently enough that OpenMP implementations often provide a tailored implementation for high performance. Here, we discuss the implementation of nested serialized parallel regions in the LLVM OpenMP runtime.

When a thread executing a serialized parallel region R_1 encounters a parallel region construct that yields another serialized region R_2, it reuses the current team descriptor associated with R_1 for R_2 rather than allocating and initializing a separate descriptor for R_2. Similarly, the thread reuses the current task descriptor associated with the R_1's implicit task to represent the R_2's implicit task. This approach is roughly 80% faster than using separate descriptors; however, it leads to problematic behaviors for both `ompt_get_parallel_info` and `ompt_get_task_info`.

The lack of separate team and task descriptors leads to losing important OMPT information about nested serialized parallel regions and corresponding implicit tasks. To overcome this problem, LLVM runtime developers introduced a separate lightweight team descriptor (lwt) to store OMPT information about nested serialized parallel regions and associated implicit tasks. When creating nested serialized parallel region R_2, the current team and task descriptors contain OMPT information about R_1 and its implicit task. The thread allocates a new lwt descriptor, fills it with OMPT information associated with R_1 and the corresponding task from the current team and task descriptors, and then overwrites the OMPT information in those descriptors with information about R_2 and its implicit task. The lwt descriptor for R_1 is then pushed into a linked list, known hereafter as the lwt list. After executing R_2's implicit task, the process is reversed. The thread removes R_1's lwt descriptor from the lwt list and copies OMPT information about R_1 and its implicit task back to the current team and task descriptors, overwriting the OMPT information about region R_2 and the corresponding task which recently completed.

Tool Support for Parallel and Task Regions. For each `ancestor_level` level at which information is available, `ompt_get_parallel_info` will return the team size for the parallel region and a pointer to a `parallel_data` word provided for the region by an OpenMP implementation for use by a tool. Similarly, a tool might inspect the OMPT state of an active task region by invoking `ompt_get_task_info`. This routine reveals the type of the task, procedure frame information for that task, and the number of the thread in the parallel region executing the task. `ompt_get_task_info` provides the `task_data` word associated with the task and maintained by the runtime for use by a tool as well as `parallel_data` word for the region that contains the task.

While a thread executes a parallel region, `ompt_get_parallel_info` reads OMPT information from the current team descriptor and either follows the parent pointers of region team descriptors or a chain of lightweight team descriptors that contain information about enclosing serialized parallel regions. The routine `ompt_get_task_info` is similar, returning information about the nesting of explicit tasks, implicit tasks, and their associated parallel regions. It reads the information from the current task and team descriptor and eventually follows the chain of task and team (lwt) descriptors in pairs.

3 Approach

As described in the Introduction, having tool data associated with parallel and/or task regions be lost or unavailable as a program executes is unacceptable for tools. To address these issues, we propose introspection consistency to avoid having tool data become unavailable while a parallel region or task is active.

An OpenMP implementation provides *introspection consistency* if it obeys the following principles:

- A thread that is part of a team for a parallel region must provide information about the region and its implicit task to the OMPT introspection routines `ompt_get_parallel_info` and/or `ompt_get_task_info` upon request from the time of its *implicit-task-begin* event until the *implicit-task-end* event of the primary thread in the region.
- A thread must provide information about a tied explicit task to the OMPT introspection routine `ompt_get_task_info` upon request from the time the task is scheduled on a thread until the task completes.
- A thread must provide information about an untied explicit task to the OMPT introspection routine `ompt_get_task_info` upon request from the time the task is scheduled for execution on a thread until it suspends.

When creating a parallel region, one must associate a thread with descriptors for both a team and an implicit task. One approach for entering a parallel region is to atomically push a (team, task) pair of descriptors representing a parallel region and its implicit task. However, that approach will require that the runtime always accesses the descriptors with an additional level of indirection. The LLVM OpenMP runtime maintains the current task and team descriptors separately to avoid this extra level of indirection. When creating a new parallel region, the runtime first updates the current team descriptor to the new region and then the current task descriptor to the region's implicit task. With this approach, one must be careful, or `ompt_get_task_info` might, for instance, read OMPT information from the current team descriptor matching the new innermost parallel region and the current task descriptor corresponding to an implicit task of an enclosing parallel region (if any). In the upstream LLVM OpenMP, this causes inconsistent results from `ompt_get_task_info`.

To avoid such inconsistencies, in our improved implementation, `ompt_get_task_info` first reads OMPT information from the current task descriptor and then accesses the current team descriptor. Each task descriptor links the descriptor of the team to which it belongs. Suppose `ompt_get_task_info` finds that the current task descriptor does not reference the current team descriptor, meaning that the current task descriptor corresponds to the implicit task of an enclosing parallel region. In that case, `ompt_get_task_info` will report the presence of the inner parallel region but indicate that it cannot provide information about the region. `ompt_get_task_info` will not provide information about a parallel region and its implicit task until the runtime updates both the team and task descriptors for the region and its task. A tool can still access the information about the inner parallel region by invoking `ompt_get_parallel_info`, and thus detect the creation/destruction of the region.

We encountered a different obstacle to providing introspection consistency for nested serialized parallel regions. As described in Sect. 2, the implementation of nested serialized parallel regions is optimized to avoid allocation and full initialization of a descriptors for nested serialized parallel regions and corresponding implicit tasks. However, the runtime fails to preserve introspection consistency for an enclosing serialized parallel region R_1 and its implicit task during the creation (and respectively, destruction) of a nested serialized parallel region R_2. The signal handler might interrupt the runtime during R_2's creation and store a value for `parallel_data` for R_1 (for `task_data` for R_1's implicit task) after the runtime has captured `parallel_data` for R_1 (`task_data` for R_1's implicit task) to move it into a lightweight team (lwt) descriptor, causing a value of `parallel_data` (`task_data`) to be lost.

The runtime can trivially overcome this problem by blocking all signals during the creation/destruction of a nested serialized parallel region to prevent signal delivery and tool introspection while information about a serialized parallel region and its implicit task is being updated. However, in the case of frequent short nested serialized parallel regions, blocking and unblocking signals can significantly interfere with sampling and cause a tool to collect unrepresentative data. Instead, we have the runtime and an introspection routine invoked from a signal handler employ wait-free coordination [7] to achieve consensus about the information associated with R_1 and R_2. The next section describes a protocol that enables a call to `ompt_get_parallel_info` or `ompt_get_task_info` from a signal handler to recognize when the runtime is in the process of moving data to prepare for a nested parallel region. In that case, the introspection routine finishes preparing the nested region so it can return information about both the nested and enclosing regions. It atomically writes this information into the runtime's current team, task, and lwt descriptors to ensure that the runtime will observe it after the introspection routine finishes.

4 Implementation

In this section, we describe the implementation of a novel wait-free protocol for coordinating the LLVM OpenMP runtime with its OMPT introspection routines to maintain introspection consistency for nested serialized parallel regions. For this purpose, we extended each of the team, task, and lwt descriptors with a pair of OMPT descriptors (depicted and numbered with 1 and 2 in Fig. 1) and a pointer (shown as `ptr` in Fig. 1) that indicates which descriptor of the pair contains valid OMPT information about the OpenMP parallel/task construct. The runtime and an introspection routine called from a tool's signal handler use wait-free coordination to achieve consensus about which OMPT descriptor associated with a nested serialized parallel region (and its implicit tasks) contains valid state by atomically updating the value of the `ptr` pointer to reference one of the pair's descriptors.

Figure 1a depicts the descriptors containing information associated with two serialized parallel regions, R1 and R2, and their corresponding implicit tasks when the primary thread executes the nested region R2. As the `ptr` pointer (shown in blue) of the heap-allocated team descriptor indicates, the R2's OMPT information resides in the first OMPT descriptor of the pair. The `ptr` (shown in red) of the task descriptor allocated on heap indicates that the second OMPT descriptor of the corresponding pair contains the information associated with R2's implicit task. The first OMPT descriptor of the pair belonging to the stack-allocated lwt descriptor depicted in the right half of Fig. 1a contains information about the enclosing parallel region R1 and its implicit task. This lwt descriptor is the only element of the lwt list at the moment, meaning that the list's head pointer (`lwt_list`) references this descriptor.

When a primary thread encounters a parallel construct yielding another serialized region R3 while executing region R2, it starts executing runtime code responsible for assembling a new serialized parallel region. The thread allocates a new lwt descriptor (bottommost in the Fig. 1b) for storing the OMPT information associated with region R2 and its implicit task. As Fig. 1b depicts, the thread sets the lwt's `ptr` to 0, meaning that consensus is not achieved yet. Similarly, this thread marks the `ptr` pointers of the current team and task descriptors by setting their least significant bits to 1 to announce that the information about

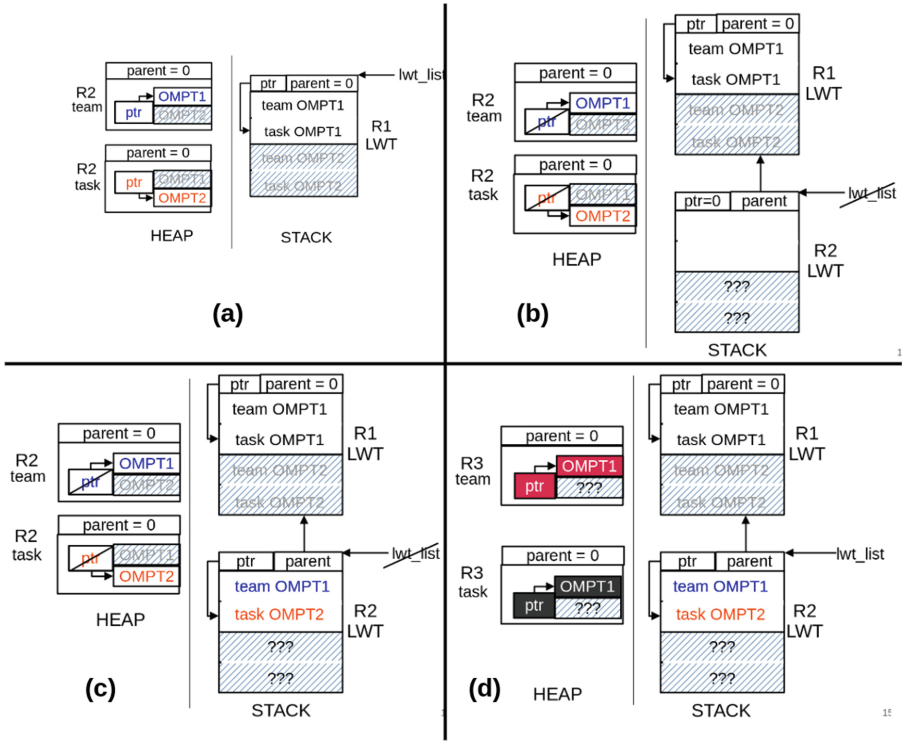


Fig. 1. Migrating data from the current team and task descriptors to a newly allocated lwt descriptor while creating a nested serialized region. OMPT descriptors belonging to the current team, task and lwt descriptors are numbered with the corresponding index in the pair (1 or 2). A slash over a pointer indicates the pointer has been marked by setting its least significant bit to 1. Shaded fields do not contain information of interest. (Color figure online)

the R2 region and its implicit task will be moved shortly after. Suppose a tool receives a sample at this moment. In that case, the signal handler invokes the introspection routine to inspect the information about R2 and its implicit task. The routine masks team and task `ptr` pointers by removing marks and reads the contents of the addresses referenced by masked pointers.

Afterward, the primary thread sets the new lwt's parent pointer to reference the `lwt_list` head pointer, marks the address of the lwt, and updates the `lwt_list` pointer with the marked address. As a result, the lwt is inserted at the end of the lwt list. The updated marked `lwt_list` head pointer indicates that the process of copying OMPT information requiring wait-free coordination is in progress, meaning that the introspection routine is responsible for finishing it if called from the signal handler.

As Fig. 1c depicts, the primary thread copies the OMPT information associated with region R2 and its implicit task from the locations indicated by the `ptr` pointers of the team and task descriptors to the first OMPT descriptor of the pair belonging to the recently allocated lwt. Subsequently, the thread tries to atomically update the value of the lwt's `ptr` with the address of the previously updated OMPT descriptor by using compare-and-swap. As shown in Fig. 1c, the primary thread decides that the first OMPT descriptor of the pair contains valid information associated with R2 and its implicit task. Otherwise, suppose a tool receives a sample before the primary thread updates the lwt's `ptr`. In that case, the introspection routine invoked by the tool's signal handler and executed by the same primary thread decides that the second OMPT descriptor of lwt's pair contains the valid information, and the compare-and-swap executed by the runtime fails.

Afterward, the primary thread tries to reuse the current team and task descriptors depicted on the left in Fig. 1d for storing the information associated with the new R3 region and corresponding implicit task. To do so, it initializes the content of the first OMPT descriptors of team and task descriptor pairs (solid red and dark grey, respectively) to store the desired OMPT information. Subsequently, it tries to update the values of the corresponding `ptr` pointers to reference the initialized OMPT descriptors using compare-and-swap. As in the case of copying the OMPT information about the R2 and its task to the lwt descriptor, the update succeeds if no call to the introspection routine by a tool's signal handler interrupts the runtime execution. Otherwise, the first call to an introspection routine finishes the update by deciding that the second OMPT descriptor of the team (task) descriptor's pair contains the valid information about the region R2 (R2's implicit task). Finally, the primary thread clears the mark bit of `lwt_list`, indicating that the preparation of the new serialized parallel region R3 is complete.

If an introspection routine sees that `lwt_list` is marked, it recognizes that the runtime is in the process of updating and initializing OMPT information about the two innermost serialized parallel regions (and associated tasks). The introspection routine finishes the update. Since the primary thread executes the introspection routine atomically with respect to the runtime code, the introspection routine need not use compare-and-swap to update the `ptr` pointers shared with the runtime. The introspection routine examines whether the lwt's `ptr` is equal 0 to observe if the runtime finished moving OMPT information from the current team and task descriptors to the lwt. Similarly, it examines if the `ptr` pointers of team and task descriptors are marked, meaning the runtime still has not reused the team and task descriptors for storing OMPT information about the innermost parallel region and its task. The introspection routine uses the second OMPT descriptors of the pairs that belong to the team, task, and lwt descriptors while moving and initializing OMPT information associated with the innermost serialized regions. After finishing the process of migrating the OMPT information for these regions, the introspection routine provides OMPT information about all active parallel regions and tasks to the tool.

5 Evaluation

This section evaluates four standard-compliant versions of the LLVM OpenMP runtime. The versions differ in how they implement nested serialized parallel regions and support OMPT introspection. Consider how they handle a program executing serialized parallel region R_1 that enters another nested serialized parallel region R_2.

- The U version uses a lightweight implementation of serialized parallel regions for speed but lacks special support for introspection consistency. This version is basically upstream LLVM OpenMP[2] adjusted to not provide incorrect information about enclosing serialized parallel regions during creation of an inner region. As a result, it may report that information about R_1 is Unavailable while R_2 is being created.
- The W version, described in the previous section, uses a lightweight implementation of serialized parallel regions and supports introspection consistency by using Wait-free coordination between the LLVM OpenMP runtime and its OMPT introspection routines.
- The F version is modified from the U version to implement serialized parallel regions using Full team descriptors. With our changes to avoid mismatched team and task information for parallel regions described in the Sect. 3, this version supports introspection consistency.
- The B version provides introspection consistency by Blocking all signals during the creation/destruction of a nested serialized region, preventing introspection while state is inconsistent. We modified the U version to block signals using the Linux `sigprocmask` routine while a serialized region is created/destroyed.

We compared the performance of the runtime versions on a system with one Intel Xeon Phi 7250 processor with 68 4-way SMT cores with 115 GB of DRAM running CentOS Linux 7.2.1511. The system was chosen principally because of its availability for isolated experiments. To avoid performance variability caused by different code and data layout in our experiments, we disabled Linux Address Space Layout Randomization. On the system's `x86_64` processors, we implemented the `compare-and-swap` used for wait-free coordination in the W runtime version using the `cmpxchg` instruction. No `lock` prefix is necessary since the `compare-and-swap` coordinates between the OpenMP runtime and a signal handler making introspection calls executed by the same thread.

We conducted two groups of experiments. The first group uses one synthetic microbenchmark, S (Listing 1), designed to measure the worst-case overhead of maintaining nested serialized parallel regions. S contains a serialized parallel region that spawns 16 million trivial nested serialized parallel regions. For the second group of experiments, we used the SPEC OMP 2012 [9] benchmark suite. We compiled all runtime implementations U, W, F, and B with Clang

[2] Forked from the commit with the hash b552adf8b388a4fbdaa6fb46bdedc83fc738fc2b on March 11th 2021.

```
#pragma omp parallel num_threads(1)
  for (int i = 0; i < 16000000; i++)
    #pragma omp parallel num_threads(1)
      volatile int x = 0;
```

Listing 1. Microbenchmark S measures the overhead of creating nested serialized parallel regions.

12.0.0 into four shared libraries in Release mode with -O3 optimization and `OMPT_SUPPORT=on` and used them in both groups of experiments.

5.1 Stress Testing of OpenMP Runtime Variants

The first group of experiments represents a stress testing of the runtime implementations overhead. For this purpose, we compiled S microbenchmark with Clang 12.0.0 using -O3 optimization and -g to provide line maps for tools. We created four executables by dynamically linking S to each of the U, W, F, and B runtime shared libraries. In our experiments, we measured each microbenchmark 30 times, computing its average execution time and standard deviation. For more representative results, the 30 runs of each microbenchmark are performed by three processes, each measuring ten runs of the microbenchmark after a warmup run.

Table 1 presents measurements of the S microbenchmark linked to each of the U, W, F and B runtime versions under three conditions: (a) with no tool present, (b) with a trivial OMPT tool that performs no measurement but causes the runtime maintain OMPT state, and (c) a basic OMPT sampling tool that periodically interrupts the program and invokes the OMPT `ompt_get_task_info` introspection routine to inspect every active parallel region and implicit task.[3] We discuss the performance of our microbenchmark under each of these three conditions to assess the cost of providing introspection consistency. The U variant, which does not support introspection consistency, serves as the baseline for the other three runtime implementations.

No Tool. Running a program without a tool is the common case, so high performance is important. Table 1(a) compares the cost of executing the S microbenchmark for serialized parallel regions using each of the U, W, F, and B runtime variants with no tool present. Overhead for S_W, S_F, and S_B are relative to S_U. The W version, which uses a wait-free protocol to provide introspection consistency was 1.49% faster than the U version which does not support introspection consistency. Our claim here is not that W is inherently faster than U but rather that they have comparable performance with no tool. Although blocking signals does not happen when the tool is not attached, introducing calls to `sigprocmask` changed the code layout resulting in 7% more overhead. One may not observe

[3] HPCToolkit uses `ompt_get_task_info` to assemble user-level calling contexts for all OpenMP work.

Table 1. Performance of benchmark S (Listing 1), which repeatedly executes a trivial nested serialized parallel region using four runtimes: U may report region information Unavailable; W uses a Wait-free strategy to implement introspection consistency; F implements nested parallelism using Full team descriptors; B may Block signals to support introspection consistency.

	(a) No tool		(b) Trivial tool		(c) Sampling Tool	
Code	Time(s)	Ovhd(%)	Time(s)	Ovhd(%)	Time(s)	Ovhd(%)
S_U	8.9846 ± 0.0006	-	10.926 ± 0.004	-	10.959 ± 0.001	0.30
S_W	8.8508 ± 0.0009	-1.49	12.477 ± 0.007	14.20	12.513 ± 0.008	0.29
S_F	16.077 ± 0.007	78.94	16.241 ± 0.012	48.65	16.280 ± 0.002	0.24
S_B	9.656 ± 0.002	7.47	43.965 ± 0.056	302.40	44.062 ± 0.024	0.22

this cost on other machines with more cache per core. In contrast, F, which always allocates and initializes full team descriptors, is almost 80% slower than the others. This shows that the complexity of W's wait-free coordination needed for introspection consistency with optimized serialized parallel regions is a good alternative to F's simpler protocol based on full region descriptors.

Trivial Tool. We developed a simple tool that uses the `ompt_start_tool` function, an initializer, and a finalizer to inform the runtime that a tool is present, but that's all; it doesn't use register for OMPT callbacks or enable sampling. Table 1(b) compares the cost of executing the S microbenchmark for serialized parallel regions using each of the U, W, F, and B runtime variants with a trivial tool, calculating overhead relative to S_U. Compared with the no tool version in column (a), S_U with a tool, which must maintain lwt descriptors, is 21.6% slower. On top of that, the wait-free coordination in the W version adds a 14.2% overhead. This shows that the cost of maintaining the lwt descriptors with a wait-free protocol is about 2/3 more costly than introducing lwt descriptors in the first place. The F version, which does not maintain lwt descriptors, has an overhead of almost 49%, which is more than 3×- higher than the overhead of the W version. Again, W delivers introspection consistency much cheaper than F, which maintains full descriptors for nested serialized regions. Furthermore, blocking signals while profiling short nested parallel regions is extremely costly. Namely, the B version is almost 3.5×- slower than the W version, meaning that providing introspection consistency using wait-free coordination is suitable for short nested parallel regions.

Sampling Tool. To assess the performance of nested serialized parallel regions for the four runtime variants while being observed with a sampling-based OMPT tool, we developed a simple proxy tool for benchmarking. We extended the trivial tool from the previous experiments with a simple signal handler and configured each thread to receive 200 samples per second from a Linux CPUTIME timer. The signal handler calls `ompt_get_task_info` for each available enclosing parallel and task region. Unlike the previous experiments in Table 1(a) and Table 1(b), for the sampling-based measurements in Table 1(c), we calculate the overhead of sampling for each code relative to the trivial tool times in Table 1(b). Each of the runtime versions have similar overhead from sampling, even though for the W version, invoking `ompt_get_task_info` might require additional work to finish assembly of a nested serialized parallel region that was in progress when the runtime was interrupted. Less than 10% of the asynchronous samples were received while the runtime was executing the code that requires wait-free synchronization between the runtime and the introspection routine invoked from a signal handler, which explains why the difference in time between Table 1(b) and Table 1(c) is similar for the U and W versions.

5.2 OpenMP Runtime Performance in Real-World Scenarios

To test the performance of U, W, F, and B runtime implementations in real-world scenarios, we used the SPEC OMP 2012 benchmark suite [9]. We created a configuration file for each runtime shared library to be used by the runspec [9] running tool. All configuration files specify the usage of intel compilers icc/icpc and ifort (version 16.0.3) with -O3 optimization for compiling C/C++ and Fortran benchmarks, respectively. We observed that thread binding sometimes slows a benchmark's execution, so we disabled it by setting `OMP_PROC_BIND` to false. We supply the configuration files to runspec. By default, runspec runs each benchmark three times with the reference workload with no profiling tool attached, meaning OMPT support is compiled but not used. Runspec finds each benchmark's median run time and divides it by the reference system's run time to calculate the normalized ratio. Finally, runspec calculates the geometric mean of all fourteen benchmark normalized ratios.

For brevity, Table 2 provides only the geometric mean of normalized ratios for each runspec invocation supplied with configuration files corresponding to the U, W, F, and B, respectively. The W runtime version employing wait-free coordination shows a negligible drop in performance compared to the others. We observed an overhead of about 3% when running the 376-tree benchmark, which spawns many recursive explicit tasks. Measurements using Linux `perf` showed that W causes more branch and instruction cache misses than U. Although the code we introduced for the wait-free coordination protocol is not executed when creating explicit tasks, it changes the code and data layout, which has a surprising effect on the cache performance on the Xeon Phi.

The W implementation outperforms the F and B versions while providing introspection consistency for short nested serialized parallel regions. However, the results presented in Table 2 show that nested serialized parallel regions are not widespread in real-world applications.

6 Related Work

The OMPT interface has been widely adopted by open-source performance tools including Caliper [4], HPCToolkit [12], Tau [11], and Score-P [8] as well as data race detection tools such as ARCHER [2], ROMP [6], and SWORD [3].

Table 2. The final measurements reports of SPEC OMP 2012 benchmark suite run four times with each of the U, W, F, and B runtime implementation. One run of suite assumes running all 14 benchmarks link to the same runtime version, determining the median run time, dividing it by the reference time to calculate normalized ratio, and calculating the geometric mean of all 14 ratios.

Runtime	Geomean
$SPEC_U$	4.23
$SPEC_W$	4.22
$SPEC_F$	4.23
$SPEC_B$	4.23

With the exception of Tau and HPCToolkit, which support asynchronous sampling, the remainder of these tools use OMPT callbacks and synchronous calls to introspection routines. Only tools that monitor OpenMP programs with asynchronous sampling are affected when OpenMP implementations lack support for introspection consistency.

7 Conclusions

An OpenMP implementation that supports introspection consistency for parallel and task regions is necessary for sampling-based performance tools to provide accurate information about nested regions. We have described strategies for supporting introspection consistency, including how to coordinate entry to regular parallel regions and a wait-free coordination protocol for nested serialized parallel regions, which efficiently handles a corner case that was an impediment to introspection consistency.

Our experiments with the microbenchmark that stresses the runtime implementation to its limits have shown that the cost of providing introspection consistency is negligible without a tool. When sampling is enabled, our wait-free implementation of optimized serialized parallel regions delivers introspection consistency at a significantly lower cost than allocating and initializing full region team descriptors or blocking signals to provide introspection consistency. We found that the runtime overhead for providing introspection consistency in a representative set of HPC benchmarks is negligible. The drawback of our approach is the complexity introduced to handle a corner case not commonly encountered in OpenMP applications. This might encourage runtime developers to prefer an alternative strategy such as blocking signals while entering or leaving a nested serialized parallel region.

In our view, the benefit of introspection consistency for tools greatly outweighs its cost. As a result, we believe that the next OpenMP standard should specify that OpenMP implementations and their OMPT introspection routines must support introspection consistency to be standard-conforming. It is worth noting that the OpenMP Debugging API [10] also needs introspection consistency. Since one can interrupt a program execution at any time in a debugger, the OMPD interface would benefit from being able to determine the nesting of parallel and task regions at arbitrary points in time using mechanisms described in this paper.

Although we developed a wait-free coordination protocol to solve a problem specific to the LLVM OpenMP runtime implementation, our approach is more broadly applicable. Namely, whenever a program manipulates data that a signal handler can inspect and change at any time, our wait-free coordination approach handles the data race between the runtime and a signal handler.

References

1. Adhianto, L., et al.: HPCToolkit: tools for performance analysis of optimized parallel programs. Concurr. Comput. Pract. Exp. **22**(6), 685–701 (2010)
2. Atzeni, S., et al.: ARCHER: effectively spotting data races in large OpenMP applications. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 53–62 (2016)
3. Atzeni, S., et al.: SWORD: a bounded memory-overhead detector of OpenMP data races in production runs. In: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 845–854 (2018)
4. Boehme, D., et al.: Caliper: performance introspection for HPC software stacks. In: SC 2016: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 550–560 (2016)
5. Eichenberger, A.E., et al.: OMPT: an OpenMP tools application programming interface for performance analysis. In: Rendell, A.P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2013. LNCS, vol. 8122, pp. 171–185. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40698-0_13
6. Gu, Y., Mellor-Crummey, J.: Dynamic data race detection for OpenMP programs. In: SC 2018: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 767–778 (2018)
7. Herlihy, M.: Wait-free synchronization. ACM Trans. Programm. Lang. Syst. (TOPLAS) **13**(1), 124–149 (1991)
8. Knüpfer, A., et al.: Score-P: a joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In: Brunst, H., Müller, M., Nagel, W., Resch, M. (eds.) Tools for High Performance Computing, pp. 79–91. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-31476-6_7
9. Müller, M.S., et al.: SPEC OMP2012—an application benchmark suite for parallel systems using OpenMP. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 223–236. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30961-8_17
10. OpenMP Architecture Review Board: OpenMP application programming interface, version 5.0 (2018)

11. Shende, S.S., Malony, A.D.: The TAU parallel performance system. Intl. J. High Perform. Comput. Appl. **20**(2), 287–311 (2006)
12. Tallent, N.R., et al.: Scalable fine-grained call path tracing. In: Proceedings of the International Conference on Supercomputing, ICS 2011, pp. 63–74. Association for Computing Machinery, New York (2011)
13. Zhou, K., et al.: Measurement and analysis of GPU-accelerated applications with HPCToolkit. Parallel Comput. **108**, 102837 (2021)

On the Migration of OpenACC-Based Applications into OpenMP 5+

Harald Servat(✉), Giacomo Rossi, Alejandro Duran,
and Ravi Narayanaswamy

Intel Corporation, Santa Clara, USA
{harald.servat,giacomo.rossi,alejandro.duran,
ravi.narayanaswamy}@intel.com

Abstract. OpenACC* and OpenMP* have been supporting offloading to accelerators for almost a decade now, with OpenACC leading the adoption rate for the usage in the context of accelerated computing. However, as OpenACC only supports limited device vendors, there has been a growing interest in adopting OpenMP for offloading to accelerators, especially now with an increasing number of accelerator vendors supporting OpenMP offload.

Motivated by the recent additions into the OpenMP 5+ specifications and the wider compiler adoption of these versions, we have developed and open-sourced the Intel® Application Migration Tool for OpenACC to OpenMP API. We present the tool in this paper and discuss the singularities and commonalities of both programming models, and then we present the migration tool, discuss its implementation and its limitations, and present successful stories when migrating applications from OpenACC to OpenMP 5+.

Keywords: Accelerated computing · Application portability · Tools

1 Introduction

Accelerators are nowadays present in many supercomputers as a way to achieve higher computer density at a lower cost and power envelope. As their availability has increased, so has increased the number of application developers interested in harnessing their computational power. While initially these programmers could only use proprietary languages, libraries or language extensions, the OpenMP* ARB released its first Technical Report (TR1) [17] describing a set of extensions that would make accelerators programmable in an standard way. OpenMP version 4.0 consolidated the proposed directives from TR1. This initial support was lacking in several aspects which have been slowly improved in subsequent specification versions.

In the time-frame between the initial discussions of the TR1 [7] and the 4.0 specification, OpenACC* [5] emerged as an alternative to use a directive based language for accelerators instead of OpenMP. OpenACC focused on

M. Klemm et al. (Eds.): IWOMP 2022, LNCS 13527, pp. 127–141, 2022.
https://doi.org/10.1007/978-3-031-15922-0_9

providing support particularly for GPU accelerators and being available before the OpenMP extensions and an initial focus to introduce advanced features at a faster rate than OpenMP made it appealing to many programmers. However, although it was supposed to be portable standard, in practice, mature implementations of OpenACC are limited and support few hardware vendors.

As OpenMP has closed the feature gap (and even gone beyond OpenACC in some aspects) its appeal has increased as it allows OpenMP to be used more portably across a wider range of compiler and hardware vendors. Furthermore, the migration path from one language to the other has also become much simpler and many (but not all) of the directives of one language can be systematically translated to an equivalent one in the other language. Still, for developers of large scale applications that decided to use OpenACC in the past, moving away might not be a negligible cost.

In this paper we present the open-source Intel® Application Migration Tool for OpenACC* to OpenMP*[1] (currently released under BSD 3-clause) which aims precisely to help making the migration from OpenACC to OpenMP simpler by automating as much of the process as possible. Unfortunately, because of a number of reasons discussed later, the tool cannot perform a perfect translation, and the user is required to validate the final translation as well as comments that the tool inserts where it can not translate the original with full certainty.

2 Related Work

There are several research projects related to the translation of OpenACC into OpenMP (up to 4.5). These projects only focus on one application source code language (either C/C++ or Fortran).

Hernandez *et al.* describe in [11,13] a mechanical approach to translate OpenACC 2.0 applications into OpenMP 4.0. Their approach consists of application developers following five steps: 1) modify OpenACC constructs that do not have an OpenMP counterpart, 2) translate data regions, 3) translate data update operations, 4) translate accelerator parallel regions through some general hardware mapping indications, and 5) adjust attribute specifiers for routines. While this is a manual approach, it can be considered the inception for many of the subsequent translation tools.

Sultana *et al.* describe an algorithm to convert a subset of OpenACC constructs into OpenMP [18]. Authors implemented it and validated against EPCC Level1 OpenACC benchmarks [2] but supported C only. In essence, their approach is similar to the one of Hernandez but they extend step 4) with some steps to leverage loop nests and explicit hardware mapping through binding clauses.

Clacc [10] is an open-source production OpenACC C/C++ compiler ecosystem which by design features a translation of OpenACC into OpenMP to use Clang's existing OpenMP compiler and runtime support. Given the differences between OpenACC and OpenMP, the translation applied is effectively a lowering of the representation and mapping it to the compiler phases that are not

[1] https://github.com/intel/intel-application-migration-tool-for-openacc-to-openmp.

normally exposed to the compiler user. This approach also allows it to reuse the existing runtime infrastructure (including debugging and development). While the tool is capable of transparently supporting OpenACC, it also enables a source-to-source translation. Flacc [9] is related to this project but supports Fortran instead. To the best of our knowledge, the research around Flacc has focused on the support for the OpenACC compiler tool-chain to run on top of the OpenMP LLVM runtime, mimicking Clacc; however, Flacc does not offer a source-to-source translator.

ACC2OMP [1] is a source-to-source translation from OpenACC to OpenMP (4.5) for Fortran applications written by Nichols Romero. It is a Python-based tool that parses Fortran source files looking for OpenACC directives and translates them into OpenMP through a dictionary.

GPUFORT [3] is an AMD project aiming at providing a source-to-source translation tool for Fortran with OpenACC or CUDA to OpenMP (4.5). As in ACC2OMP, GPUFORT is written in Python but aims at creating a Fortran2003 parser to identify the OpenACC offloaded regions, and OpenACC API and CUDA calls to convert them into OpenMP or HIP calls [6].

We finally refer to CCAMP [14], an OpenMP 4+ and OpenACC interoperable framework built on top of OpenARC [15]. It provides two functionalities to help programmers transplant non-portable code into new architectures: language translation between the two standards, and device-specific directive optimization within each standard.

The tool we describe in this paper differs from the aforementioned on several fronts. First, it generates OpenMP 5+ compliant code which results on a more natural (almost one-to-one) clause mapping between the two programming models as described in Sect. 4. Second, compared to the previous efforts, our tool supports both C/C++ and Fortran applications. Last, while previous tools focused on providing an optimized translation (under certain assumptions), our tool focuses on providing a semantically equivalent translation with a number of knobs that allow the user to specify more performant translations if she knows they are correct.

3 OpenACC vs OpenMP 5+

There are several works that discuss the differences between OpenACC and OpenMP (up to 4.5) (see [10,13,14,18]). Among the differences, authors highlight differences related to:

- the OpenACC `kernels` construct, which delimits a program region and lets the compiler to identify the enclosed loops, and among those which are safe to parallelize and accelerate,
- the work distribution for single and nested loops because the leeway that the OpenACC compilers have compared to the explicit OpenMP requirement to distribute the work, and
- the `cache` construct does not have an OpenMP counterpart, and any caching decision will be done by the compiler based upon its optimization guidelines.

The advent of OpenMP 5.0 (and, later, 5.1 and 5.2) have introduced features that has simplified the mapping with OpenACC, most notably with the `loop` and `present` constructs but also the interoperability directives can help matching some of the OpenACC's API. The `loop` in OpenMP behaves similarly to the one in OpenACC, it specifies that the logical iterations of the associated loop may execute concurrently and the compiler has leeway to apply some optimizations and map bindings. One important difference between the two refers to the hardware map binding because OpenACC allows the vector binding but this is not possible in OpenMP, where this may occur implicitly (due to compiler optimizations) or explicitly through the `simd` construct. With respect to the `present` construct, OpenMP offers the `present` map type modifier or argument in `defaultmap`. The `present` map type modifier alters the `map` behavior by checking if a variable exists on the device address space on entry to the region. If it does not exists, execution results in runtime error. If it does exist, then the `map` statement is applied. Still, some differences remain even when comparing OpenACC to OpenMP 5+: OpenMP still lacks a construct that resembles to the OpenACC `kernels` construct and the inability to nest a `loop` construct inside a `simd` construct.

4 Description of the Tool

The Intel Migration Tool for OpenACC to OpenMP API is a tool that aims to help migrate valid OpenACC Fortran and/or C/C++ based applications into OpenMP 5+ and it focuses on providing a semantically equivalent translation. The tool currently converts a commonly used subset of the OpenACC directives into OpenMP, an approach taken by other tools like [14,18]. In contrast to the tools mentioned in the related work which generate OpenMP 4+, our tool generates OpenMP 5+ compliant code. There are two reasons: first, the recent OpenMP specifications include a number of directives that map naturally with OpenACC; and second, the growing compiler support for OpenMP 5+[2]. Regarding the directives, we highlight the `loop` directive (introduced in OpenMP 5.0) and the `present` map type modifier or argument for `defaultmap` (introduced in OpenMP 5.1). Still, there are a number of differences between OpenACC and OpenMP, and given that the OpenACC runtime calls are not currently translated, it is imperative that the user supervises the translation.

We want to stress that the tool does not focus on guaranteeing the best achievable performance but at generating a semantically equivalent translation. The rationale is threefold:

1. many accelerators achieve high performance when using the vendor-compiler tied to that device;
2. there are a number of OpenACC performance-related clauses that are unlikely to be performance-portable to different hardware accelerators (even from the same vendor); and,
3. some of these clauses require greater (compiler-like) effort that is currently beyond the scope of this tool.

[2] https://www.openmp.org/resources/openmp-compilers-tools.

This is why, once the application has been successfully ported, we encourage developers and users to use performance tools for fine-tuning their application performance.

4.1 Tool Overview

The Intel Migration Tool for OpenACC to OpenMP API is written in Python because of the parsing flexibility it provides and, as of the writing of this paper, the tool does not use any compiler technology underneath. In short, the tool translates OpenACC statements into OpenMP statements with little or no application source-code context. This approach simplifies the translation mechanism but also imposes certain restrictions that we discuss later. Notwithstanding these limitations, we believe that our approach is still valid because most OpenACC applications tend to use a limited set of OpenACC constructs.

The tool generates a translation for a valid OpenACC application source code that adheres to the latest OpenACC (3.2) specification, and also emits a report with some translation details and issues found during the translation. The resulting translation consists of the input source code, where OpenACC statements are followed by the translated statements. The translation tool supports a number of knobs to alter the translation, among others: alter the translation to support OpenMP 5.0 or 5.1, alter the asynchronous clauses translation, specify whether to honor the binding clauses, or add C preprocessor conditional guards into the pre-existing OpenACC and OpenMP constructs. With respect to the report, it contains a list of identified OpenACC statements, and for each of those there is the OpenMP translation as well as some comments and/or warnings related to the conversion.

The tool is composed by the following components, which are executed in a sequential manner:

parser. module, responsible for parsing C/C++ and Fortran (in either fixed or free-form formats) files. It extracts each OpenACC statement as a single string without superfluous blank characters and in lower-case to simplify the subsequent phase. Then this statement is added together with the statement as-is and the delimiting code line numbers into a data-object that is inserted into a hash-map indexed by the code-line.

migration. module traverses the hash-map by line number and converts the extracted OpenACC statements into OpenMP statements as described in the following subsection. This module also annotates potential translation issues.

code-generation. module, which dumps the input file but extended with the translated construct. Upon request, this step also adds preprocessor guards into the OpenACC or freshly generated OpenMP statements. It is also responsible for creating the translation report.

As a side note, the repository hosting the tool includes many validation tests. These tests can be validated at three levels: 1) against an OpenACC compiler to ensure that the input is legitimate, 2) a reference output file to ensure that the translation is correct, and 3) against an OpenMP compiler that ensures that the translation can be compiled.

Table 1. Concise translation mappings for OpenACC constructs to OpenMP 5+.

OpenACC		**OpenMP 5+**	**OpenACC**		**OpenMP 5+**
`atomic`	⇒	`atomic`	`kernels`	⇒	`target` / `target teams`
`cache`	⇒	*untranslated*	`loop`	⇒	`loop`
`data`	⇒	`target data`	`kernels loop`	⇒	`target teams loop`
`declare`	⇒	`declare target`	`parallel`	⇒	`target teams`
`enter data`	⇒	`target enter data`	`routine`	⇒	`declare target`
`exit data`	⇒	`target exit data`	`serial`	⇒	`target`
`host_data`	⇒	`target data` / `target update`	`update`	⇒	`target update`

Table 2. Concise translation mappings for OpenACC clauses to OpenMP 5+.

OpenACC		**OpenMP 5+**
`async`	⇒	`nowait`
`delete(X)`	⇒	`map(delete:X)`
`detach(X)`	⇒	`map(release:X)`
`deviceptr(X)`	⇒	`is_device_ptr(X)`
`default(present)`	⇒	`defaultmap(present:aggregate)` and `defaultmap(present:pointer)` *in OpenMP 5.1*
	⇒	*implicit mapping rules in OpenMP 5.0*
`present(X)`	⇒	`map(present,alloc:X)` *in OpenMP 5.1*
	⇒	`map(alloc:X)` or `map(tofrom:X)` *in OpenMP 5.0*
`(p|present_or_)copy(X)`	⇒	`map(tofrom:X)`
`(p|present_or_)copyin(X)`	⇒	`map(to:X)`
`(p|present_or_)copyout(X)`	⇒	`map(from:X)`
`(p|present_or_)create(X)`	⇒	`map(alloc:X)`
`wait`	⇒	`taskwait`

4.2 OpenACC to OpenMP 5+ Migration

The tool generates OpenMP 5+ compliant code because it allows a simpler and more natural translation. Tables 1 and 2 summarize the translations applied by our migration tool. From these, we highlight some translation details regarding the OpenACC constructs and clauses:

loop. This construct is translated into the OpenMP construct with the same name. The OpenMP construct has similar behavior to OpenACC (although not equal as we describe in the following section) while offering the compiler some leeway to apply optimizations.

present. The translation of this clause depends on the target OpenMP version which can be tuned through a knob. OpenMP 5.1+ added the `present` map-type modifier that checks for the availability of the data in the device address space. Since the `present` clause in OpenACC does not imply any transfer, we translate it to `map(present,alloc)`. The `present` modifier allows to check for the presence of the data in the device address space while the `alloc` map-type prevents any memory transfer. However, OpenMP 5.0 does support the present modifier and thus the user has to choose between map(alloc) or map(tofrom) mappings and although they are not identical to present they are likely to behave similarly in most situations.

default. This clause in OpenACC applies to all variables except scalars but the OpenMP counterpart (defaultmap) applies to all variables except if a variable category is given. The translation to OpenMP combines defaultmap constructs so that they apply to all variable categories except scalars. For the particular case of default(present), the behavior is similar to the present clause. The same knob used to control the present clause dictates how to translate defaultmap(present).

host_data. This construct makes the address of data in device memory available to the host (useful, for example, in GPU-enabled MPI implementations). Since this functionality may not be present on the target system or its software stack, our tool can translate this construct to target data or to target update. While the former has similar semantics to OpenACC, the second approach uses the host memory as a staging buffer for the operation to be executed. Users should choose the former if the enclosed statements can make use of the address of data in device memory, and choose the latter otherwise.

async, wait. These two clauses can be mapped, on trivial cases, to the nowait clause and taskwait construct, respectively. However, OpenACC supports an optional argument that specifies an activity queue to which the clause refers, potentially leading to non-trivial dependencies. We are currently working to support this functionality using OpenMP dependencies (*i.e.* depend clause). As of now, the user can disable the translation of asynchronous statements when those statements define non-trivial dependencies.

4.3 Limitations and Potential Work-Arounds

Given the simple nature of the migration tool, it has a number of limitations although some of these can be circumvented in particular cases. The following sections describe known limitations and potential work-arounds.

OpenACC Kernels Construct and Loop Auto Clause. The first, and possibly most notable, limitation refers to the lack of a direct translation of the kernels construct. This construct instructs the compiler to identify parallelization opportunities within the enclosed construct, and if so, parallelize and offload the identified regions. OpenMP, however, requires explicit identification of parallel and offload regions. If the kernels construct is combined with the loop construct, then the tool generates a parallel target teams loop offload parallel region, otherwise the tool offloads the code region serially through the target construct. Since the loop construct may appear nested in target regions, we have implemented an experimental feature for Fortran applications that identifies these loops and generates target teams loop rather than target loop.

In a similar direction, the auto clause from the loop construct tells the compiler to determine whether the iterations have data dependencies. If not,

Listing 1. Sample OpenACC code with loop vector.

```
void mmul (int SIZE, float ** restrict c, float **a, float **b)
{
  int i,j,k;
  #pragma acc data copyin(a[0:SIZE][0:SIZE],b[0:SIZE][0:SIZE]) \
                   copy(c[0:SIZE][0:SIZE])
  {
    #pragma acc parallel loop
    for (i = 0; i < SIZE; ++i)
      #pragma acc loop vector
      for (j = 0; j < SIZE; ++j)
      {
        float tmp = 0.f;
        #pragma acc loop reduction(+:tmp)
        for (k = 0; k < SIZE; ++k)
          tmp += a[i][k] * b[k][j];
        c[i][j] = tmp;
      }
  }
}
```

the code within the loop can be executed in parallel. As stated earlier, the migration tool cannot extract parallelism from the source code and the code within this clause is executed sequentially. Since the OpenMP `loop` expresses that the associated loop is to be executed in parallel, the tool simply ignores the `loop` construct when the `auto` clause is given so that the associated loop is executed serially.

Hardware Binding Clauses. The OpenACC hardware binding clauses relate to the `gang`, `worker` and `vector` clauses. Despite performance being beyond the current goals of the tool, because of the difficulty of performance portability across vendors and specific compiler optimizations, we support translating these clauses into the corresponding OpenMP binding clauses, if appropriate knobs are provided. This allows users to test the potential performance of maintaining these clauses. However, there is a caveat regarding the `vector` clause. Its translation is the `simd` construct but this construct does not allow nested `loop` constructs. Consequently, the translation of the valid OpenACC code shown in Listing 1 would need further modifications than those in the scope of the tool so that the `simd` construct and its associated loop appear as the inner-most loop.

Multi-dimensional and Non-contiguous Data Mapping. In our tests, OpenACC has shown to be somewhat more flexible than OpenMP when dealing with multi-dimensional non-contiguous data as in the example shown in Listing 2. Depending on the definition of variable `v3`, if the data referenced by the aforementioned statement is contiguous the user could accept the ordinary OpenMP translation (see Listing 3). For C/C++ applications, the migration

Listing 2. Multi-dimensional OpenACC construct.

```
#pragma acc enter data copyin(v3[:3][:7][:8])
```

Listing 3. Multi-dimensional OpenACC construct translated into OpenMP.

```
#pragma omp target enter data map(to:v3[:3][:7][:8])
```

tool proposes an OpenMP translation that uses nested mappings, as shown in Listing 4; we are exploring similar alternatives for Fortran applications.

Seq Clauses. OpenACC supports the `seq` clause on the `loop` and `routine` constructs prevents any automatic parallelization or vectorization, ultimately to allow composability with invoking routines and/or loops. To our knowledge, there is no such a similar OpenMP counterpart and this clause is currently ignored by the migration tool.

Runtime Calls. OpenACC offers a number of runtime routines that the application developer can use to interact with the runtime library and/or device. Currently, our tool does not convert any of these and this implies that the user has to manually translate these calls into their OpenMP counterparts, if any. In the future, we will consider migrating OpenACC API calls into OpenMP API calls. To this end, the interoperability mechanisms added in OpenMP 5.1 seem helpful.

Other Differences. There exist other OpenACC and OpenMP specific differences that may hinder a fully automated translation. For instance, OpenACC supports the `read-only` modifier when transferring data to the device (*i.e.* `copyin`) and the `zero` modifier when allocating a variable in the device memory address space. Also, the OpenACC `declare copyin` directive would be translated into `declare target` but the OpenMP translation is more restrictive when applied to variables. In a similar direction, the OpenACC `declare`

Listing 4. Multi-dimensional OpenACC construct translated into OpenMP using nested of allocations.

```
#pragma omp target enter data map(to:v3[0:3])
for (int _idx0 = 0; _idx0 < 3; ++_idx0)
{
  #pragma omp target enter data map(to:v3[0+_idx0][0:7])
  for (int _idx1 = 0; _idx1 < 7; ++_idx1)
  {
    #pragma omp target enter data map(to:v3[0+_idx0][0+_idx1][:8])
  }
}
```

`create` directive does not have a OpenMP counterpart. To our understanding, these OpenACC directives are not widely used and thus it is not urgent to convert them, but we are exploring migration alternatives for them.

5 Experiences on Application Migrations.

In this section we describe our experiences migrating OpenACC benchmarks and mini-applications using the Intel Application Migration Tool for OpenACC to OpenMP API[3]. Table 3 provides some characteristics of these applications. Unless explicitly stated, we have used the migration tool with the default settings, except disabling the translation of the asynchronous clauses (*i.e.* knob `-async=ignore`) thus not allowing explicit overlapping. Despite the aim of the translation tool being portability, we have done an initial performance study on a few of the applications.

We have compiled and evaluated the translated sources in two systems. The first system is equipped with 2 Intel® Xeon® Platinum 8360Y processors and a discrete NVIDIA® A100 80 GB PCIe GPU. Applications on this system have been compiled with the NVIDIA HPC SDK 22.2 using the flags: `-fast -Mstack_arrays -Mnouniform -Mfprelaxed -acc -Minfo -gpu=fastmath`. The second system is equipped with an Intel® Core™ i7-1165G7 chip, which features an integrated GPU: this GPU does not support FP64 operations natively and these operations are emulated. We have used an engineering version of the Intel C++/Fortran Compilers that supports a subset of the OpenMP 5.1 specification. The features used in the context of this paper are likely to be available as part of the next major release of the compiler. The OpenMP offload compilation flags for the C++ applications are `-fiopenmp -fopenmp-version =51 -fopenmp-targets=spir64` whereas the flags for the Fortran applications are `-fiopenmp -fopenmp-targets=spir64`.

Table 3. Application characteristics.

			OpenACC constructs / clauses				
Application name	**Language**	**#lines**	*#compute*[a]	*#data*[b]	*#atomic*[c]	*#async*[d]	*#API* calls
SpecACCEL-370-BT	C	7765	52	9	0	0	0
SpecACCEL-354-CG	C	1418	18	2	0	0	0
POT3D	Fortran	11698	33	36	2	19	0
CloverLeaf	Fortran	10134	628	562	0	0	0

[a] Compute constructs cover: `kernels`, `loop`, `parallel`, `serial` and `loop`.

[b] Data constructs cover: `data`, `host_data` and `update`.

[c] Atomic constructs cover: `atomic`.

[d] Asynchronous clauses and constructs cover: `async` and `wait`.

[3] Commit-id be6c54f11564b8cfa185c0e1c5390be57d78a8f1.

SpecACCEL-370-BT and SpecACCEL-354-CG. The SPEC ACCEL® benchmark suite [12] tests performance with computationally intensive parallel applications using both OpenACC and OpenMP APIs. We have chosen the 370-BT and 354-CG benchmarks from SPEC ACCEL 1.4 which are a C+ OpenACC re-implementation of the NAS benchmarks [4]. BT solves a 3D discretization of Navier-Stockes equation, while CG solves an unstructured sparse linear system using the conjugate gradient method.

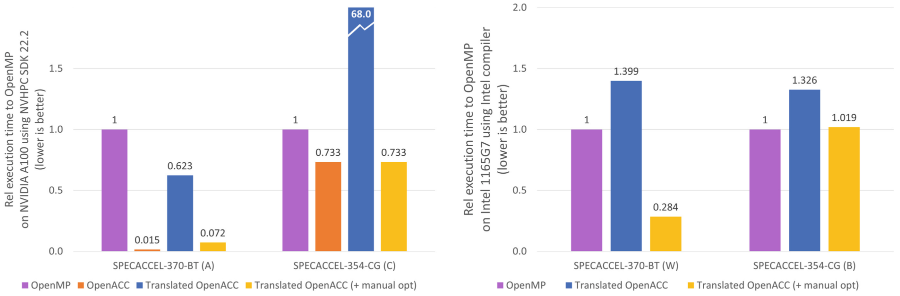
Figure 1 shows the estimated[4] relative performance on the different variants when compared to the OpenMP executions on the same system. We want to note that, given the different purpose of the selected systems, we have used different inputs so that the benchmarks finish in a reasonable amount of time. The BT results on the NVIDIA machine show that the OpenACC version (in orange) is about 66× faster than the OpenMP version (in purple). An inspection of the source code shows that 370.bt and 570.pbt differ in the loop ordering on the most-time consuming kernels and this is likely to explain part of this difference. The OpenACC translation (in blue) is 1.6× faster than the OpenMP version. If we manually apply loop collapsing to the most time-consuming offloaded loops (which is applied automatically by the OpenACC compiler but not in OpenMP) then this version (shown in yellow) is 13.8× faster than the OpenMP variant. On the Intel machine, the translated OpenACC version is 1.4× slower than the OpenMP version, but manually applying the previous loop collapse modifications improves the translated OpenACC version to be 3.5× faster than the OpenMP version. Regarding the CG benchmark, on the NVIDIA-based system, the translated OpenACC version is much slower (68×) than the OpenMP version because the OpenACC variant uses the `kernels` construct without combining with the `loop` construct and the migration tool does not parallelize it. If we explicitly offload and parallelize the loops within the `kernels` constructs, then the performance gets in line with the OpenACC version on the NVIDIA system and the performance on the Intel system gets in line with the OpenMP version.

CloverLeaf. [16][5] is a Fortran mini-application that solves the compressible Euler equations on a Cartesian grid, using an explicit, second-order accurate method. Regarding the translation, we have mapped the `present` clause to the `alloc` map-type because the `present` map type modifier is not supported on the nvfortran version we have used. We also have disabled the experimental feature that parallelizes `loop` constructs within `kernels` on two files[6] because the `kernels` construct is immediately followed by control flow statements and this is not supported by this experimental feature. We have focused a comparison in the NVIDIA-based machine because the application uses double precision FP heavily and we cannot obtain performance results on the system with the integrate Intel GPU in a reasonable amount of time.

[4] Estimated results are those not audited by SpecACCEL.

[5] Commit-id d957cef81c5e5765a2e9045432845eada980cc79 from https://github.com/UK-MAC/CloverLeaf_OpenACC.

[6] `advec_cell_kernel.f90` and `advec_mom_kernel.f90`.



(a) Performance obtained on NVIDIA hardware.

(b) Performance comparison obtained on Intel hardware.

Fig. 1. Translation performance achieved on the two systems. (see Sect. 5 for workloads and configurations and Appendix for additional usage details. Results may vary.)

Also, despite there exists an initial implementation of the application using OpenMP offload[7], it is an older version tagged as work-in-progress and does not compile as-is, and consequently we cannot use it for comparison purposes. If we limit ourselves to the OpenACC and its translated version, we observe that the translated version on the NVIDIA system is approximately 9% slower than the original version on the same system. Interestingly, if we manually add the `teams` construct into the serial `target` offloaded regions to run the regions in parallel in the aforementioned files there is no performance impact on the translated version.

POT3D. [8][8] is a Fortran code that computes potential field solutions to approximate the solar coronal magnetic field using observed photospheric magnetic fields as a boundary condition. The code is parallelized using MPI and is GPU-accelerated using Fortran standard parallelism (do concurrent) and OpenACC. The application exhibits a variety of OpenACC constructs, including `host_data` for GPU-enabled MPI environments. The application translates without issues and compiles successfully on the Intel system but fails to generate a binary using the NVIDIA HPC SDK 22.2, so we cannot fairly compare the performance.

6 Conclusions and Future Work

We have introduced and described the open-source Intel Application Migration Tool for OpenACC to OpenMP API, which helps migrating OpenACC applications to OpenMP with user supervision. In contrast to previous works in this field, this Python-based tool generates OpenMP 5+ compliant code and supports

[7] https://github.com/UK-MAC/CloverLeaf_OpenMP4.

[8] Commit-id 42984a8ce428d54036d6b8a0732f05a046e8f840 (3.1.0r) from https://github.com/predsci/POT3D.

C/C++ and Fortran. The tool is still in development but it already translates many of the OpenACC clauses and it has allowed us to convert several benchmarks and mini-applications.

There are a number of topics that we are interested in pursuing in the future. First, we envision the tool translating the asynchronous mechanisms offered by OpenACC, so that applications can effectively overlap CPU and GPU computations and memory transfers. We are exploring how to extend the support of the `kernels` construct so that the tool, with potential user guidance, can offload and parallelize the enclosed code regions. Finally, it may be convenient to support the translation of the OpenACC API routines into their OpenMP counterparts. Furthermore, the same API migration process could be reused to translate standard APIs, such as BLAS routines.

Acknowledgments. We would like to thank Mikko Byckling, Jeongnim Kim, Rakesh Krishnaiyer, Geoff Lowney and Christopher Dahnken for the fruitful discussions while developing this tool.

Notices & Disclaimers. Performance varies by use, configuration and other factors.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See appendix for configuration details. No product or component can be absolutely secure.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Your costs and results may vary.

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.

Intel technologies may require enabled hardware, software or service activation.

A Using the Migration Tool on POT3D

A.1 Downloading the Migration Tool

```
$ git clone https://github.com/intel/intel-application-migration-tool-for-openacc-to-openmp
Cloning into 'intel-application-migration-tool-for-openacc-to-openmp'...
remote: Enumerating objects: 307, done.
remote: Counting objects: 100% (307/307), done.
remote: Compressing objects: 100% (203/203), done.
remote: Total 307 (delta 132), reused 275 (delta 102), pack-reused 0
Receiving objects: 100% (307/307), 86.24 KiB | 9.58 MiB/s, done.
Resolving deltas: 100% (132/132), done.
$ cd intel-application-migration-tool-for-openacc-to-openmp/
$ git rev-parse HEAD
be6c54f11564b8cfa185c0e1c5390be57d78a8f1
```

A.2 Downloading POT3D

```
~/apps $ git clone https://github.com/predsci/POT3D
Cloning into 'POT3D'...
remote: Enumerating objects: 211, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 211 (delta 1), reused 0 (delta 0), pack-reused 208
Receiving objects: 100% (211/211), 24.56 MiB | 12.56 MiB/s, done.
Resolving deltas: 100% (102/102), done.
$ git checkout 42984a8ce428d54036d6b8a0732f05a046e8f840
Note: switching to '42984a8ce428d54036d6b8a0732f05a046e8f840'.
..
HEAD is now at 42984a8 RMC: POT3D v3.1.0 - Cleaned up code, added analytic validation run mode, updated
      documentation and scripts.
```

A.3 Applying the Migration Tool to POT3D

```
~/apps/POT3D/src $ ~/intel-application-migration-tool-for-openacc-to-openmp/src/intel-application-migration-
      tool-for-openacc-to-openmp -async=ignore -overwrite-input *.f
Processing file number_types.f
Processing file pot3d.f
Processing file zm_parse.f
Processing file zm_parse_modules.f
Processing file zm_sds.f
Processing file zm_sds_modules.f
```

A.4 Makefile Example Using Intel Compiler

```
~/apps/POT3D/src $ cat Makefile.ifx
FC = mpiifort -fc=ifx
FFLAGS = -fiopenmp -fopenmp-targets=spir64 -mllvm -vpo-paropt-atomic-free-reduction=true -O -g -mcmodel=
      medium -I$(HDF5_HOME)/include
OBJS = number_types.o \
 zm_parse_modules.o \
 zm_parse.o \
 zm_sds_modules.o \
 zm_sds.o \
 pot3d.o
LDFLAGS = -L$(HDF5_HOME)/lib -Wl,-rpath -Wl,$(HDF5_HOME)/lib -lhdf5_fortran -lhdf5_hl_fortran -lhdf5 -
      lhdf5_hl

all:    $(OBJS)
        $(FC) $(FFLAGS) $(OBJS) $(LDFLAGS) -o pot3d
        rm *.mod *.o 2>/dev/null
clean:
        rm pot3d 2>/dev/null
        rm -f *.mod *.o 2>/dev/null
number_types.o: number_types.f
        $(FC) -c $(FFLAGS) $<
zm_parse_modules.o: zm_parse_modules.f
        $(FC) -c $(FFLAGS) $<
zm_parse.o: zm_parse.f zm_parse_modules.f number_types.f
        $(FC) -c $(FFLAGS) $<
zm_sds_modules.o: zm_sds_modules.f
        $(FC) -c $(FFLAGS) $<
zm_sds.o: zm_sds.f zm_sds_modules.f number_types.f
        $(FC) -c $(FFLAGS) $<
pot3d.o: pot3d.f
        $(FC) -c $(FFLAGS) $<
```

References

1. ACC2OMP. https://github.com/naromero77/ACC2OMP. Accessed 27 Apr 2022
2. EPCC OpenACC benchmarks. https://github.com/EPCCed/epcc-openacc-benchmarks. Accessed 11 May 2020
3. GPUFORT. https://github.com/ROCmSoftwarePlatform/gpufort. Accessed 27 Apr 2022
4. NAS Parallel Benchmarks. https://www.nas.nasa.gov/software/npb.html. Accessed 11 May 2022
5. OpenACC specification (2012). https://www.openacc.org/specification. Accessed 16 May 2022

6. HIP Programming Guide (2022). https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-GUIDE.html. Accessed 16 May 2022
7. Beyer, J.C., Stotzer, E.J., Hart, A., de Supinski, B.R.: OpenMP for accelerators. In: Chapman, B.M., Gropp, W.D., Kumaran, K., Müller, M.S. (eds.) IWOMP 2011. LNCS, vol. 6665, pp. 108–121. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21487-5_9
8. Caplan, R.M., Mikic, Z., Linker, J.A.: From MPI to mpi+openacc: Conversion of a legacy FORTRAN PCG solver for the spherical laplace equation. CoRR abs/1709.01126 (2017). http://arxiv.org/abs/1709.01126
9. Clement, V., Vetter, J.S.: Flacc: Towards OpenACC support for Fortran in the LLVM Ecosystem. In: 2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC), pp. 12–19 (2021). https://doi.org/10.1109/LLVMHPC54804.2021.00007
10. Denny, J.E., Lee, S., Vetter, J.S.: CLACC: translating OpenACC to OpenMP in clang. In: 2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC), pp. 18–29 (2018). https://doi.org/10.1109/LLVM-HPC.2018.8639349
11. Hernandez, O., Ding, W., Joubert, W., Bernholdt, D., Eisenbach, M., Kartsaklis, C.: Porting OpenACC 2.0 to OpenMP 4.0: key similarities and differences (2016). https://openmpcon.org/wp-content/uploads/openmpcon2015-oscar-hernandez-portingacc.pdf. Accessed 27 Apr 2017
12. Juckeland, G., et al.: SPEC ACCEL: a standard application suite for measuring hardware accelerator performance. In: Jarvis, S.A., Wright, S.A., Hammond, S.D. (eds.) PMBS 2014. LNCS, vol. 8966, pp. 46–67. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17248-4_3
13. Juckeland, G., et al.: From describing to prescribing parallelism: translating the SPEC ACCEL OpenACC suite to openMP target directives. In: Taufer, M., Mohr, B., Kunkel, J.M. (eds.) ISC High Performance 2016. LNCS, vol. 9945, pp. 470–488. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46079-6_33
14. Lambert, J., Lee, S., Malony, A., Vetter, J.S.: CCAMP: OpenMP and OpenACC Interoperable Framework. In: Schwardmann, U., et al. (eds.) Euro-Par 2019. LNCS, vol. 11997, pp. 357–369. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-48340-1_28
15. Lee, S., Vetter, J.S.: OpenARC: open accelerator research compiler for directive-based, efficient heterogeneous computing, In: HPDC 2014, Association for Computing Machinery, New York, NY, USA, pp. 115–120 (2014). https://doi.org/10.1145/2600212.2600704
16. Mallinson, A., Beckingsale, D.A., Gaudin, W., Herdman, J., Levesque, J., Jarvis, S.A.: Cloverleaf: Preparing hydrodynamics codes for exascale. In: Proceedings of the Cray User Group 2013 (2013)
17. Stotzer, E., et al. OpenMP Technical Report 1 on Directives for Attached Accelerators (2012). https://www.openmp.org/wp-content/uploads/TR1_167.pdf. Accessed 16 May 2022
18. Sultana, N., Calvert, A., Overbey, J.L., Arnold, G.: From OpenACC to OpenMP 4: toward automatic translation. In: Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale. XSEDE16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2949550.2949654

OpenMP and Multiple Translation Units

Just-in-Time Compilation and Link-Time Optimization for OpenMP Target Offloading

Shilei Tian[1](✉), Joseph Huber[2], John Tramm[3], Barbara Chapman[1], and Johannes Doerfert[3](✉)

[1] Stony Brook University, Stony Brook, USA
{shilei.tian,barbara.chapman}@stonybrook.edu
[2] Oak Ridge National Laboratory, Oak Ridge, USA
huberjn@ornl.gov
[3] Argonne National Laboratory, Lemont, USA
{jtramm,jdoerfert}@anl.gov

Abstract. Following the mass adoption of external accelerators for high performance computing, the overall performance of many applications has become increasingly dependent on relatively small accelerated kernels. As static analysis is fundamentally limited by dynamic values and external definitions, standard ahead-of-time compilation is not always sufficient to achieve the best performance. Furthermore, many users looking to port an existing application to run on an external accelerator will not want to fundamentally restructure their programs. These and other problems can be addressed through both link-time optimization (LTO) and just-in-time (JIT) compilation, but until now had sparse and inconsistent support from the compiler.

In this work, we present a new compilation method that enables device-side LTO as well as a transparent JIT compilation tool-chain for OpenMP target offloading. Our contributions include an entirely new device linking and embedding scheme to enable LTO as well as a novel JIT engine to efficiently optimize OpenMP offloading regions at run-time. We also introduce a persistent caching system to improve end-to-end runtime using the JIT engine and minimize kernel launching overheads. We measure the performance of our LTO and JIT implementation via several real-world scientific applications. With our optimizations we observe significant improvements through LTO on large applications as well as significant end-to-end execution time improvement using JIT.

Keywords: OpenMP · GPU · LTO · JIT

1 Introduction

The dominance of massively-parallel GPGPU based accelerators in high performance computing systems has resulted many applications being highly dependent on small accelerated kernels executed on the device. This poses a challenge

M. Klemm et al. (Eds.): IWOMP 2022, LNCS 13527, pp. 145–158, 2022.
https://doi.org/10.1007/978-3-031-15922-0_10

for compilers looking to optimize applications targeting heterogeneous systems, especially through generic programming models, such as OpenMP target offloading. The massively parallel nature of these systems means that any missed optimizations or overhead can result in considerably large performance losses. Furthermore, the compiler's ability to optimize these program is fundamentally limited by external definitions or dynamic values only known at runtime. OpenMP especially makes heavy use of environment variables whose values can only be known at runtime. This means that ahead-of-time (AoT) optimizations alone are not sufficient to determine important constants, such as the number of teams and threads in a region.

In this work we present a transparent implementation of link-time optimization (LTO) and just-in-time (JIT) compilation for OpenMP offloading for the LLVM/Clang compiler infrastructure. We first show an overhauled driver for compiling OpenMP offloading programs in LLVM/Clang that allows transparent embedding and linking of device LLVM IR. The JIT engine uses the linked bitcode to perform further optimizations and code generation at runtime with the knowledge of runtime values, e.g., environment variables. Finally, we present the performance improvements of the LTO and JIT compilation on several benchmarks and proxy-applications.

In the following we first briefly explain the necessary background on OpenMP offloading compilation via LLVM/Clang, LTO, and JIT. Sections 3 and 4 describe our LTO and JIT contributions in detail. The evaluation of our approach is given in Sect. 5, and finally, before the conclusion in Sect. 7, we discuss related works in Sect. 6.

2 Background

In this section, we will briefly introduce the current compilation pipeline used to create OpenMP target offloading applications and support LTO and JIT.

The LLVM/Clang compiler driver is responsible for creating the necessary actions to produce the compiled output. Compilation for LLVM/OpenMP offloading is more complex than a standard compilation job because the compiler must compile and link for multiple architectures at once. In order to maintain standard compilation semantics, the compiler driver will create a compilation for each target architecture and then embed the result into a single fat binary. Device linking occurs by extracting the device code inside the fat binary and first running the appropriate device linking job on it. After linking, the device image still needs to be registered with offloading runtime on the host. To register the device image we create a new module containing the necessary registration code and link it with the rest of the application.

Standard LTO in LLVM is performed by emitting LLVM IR bitcode instead of an object file. During linking, if the linker supports LTO, all the identified LLVM IR bitcode files will be merged and optimized together using symbol resolution information from the linker. The linked and optimized bitcode is then compiled and linked with the other input files. Similarly, JIT compilation uses LLVM IR bitcode to compile for the target architecture as-needed at runtime.

3 Link Time Optimization Support

As mentioned in Sect. 2, creating offloading binaries is more challenging than standard compilation. In this section, we first introduce our new offloading driver for LLVM/Clang. Then, we describe our new augmented linker to support device linking with LTO.

3.1 Offload Driver

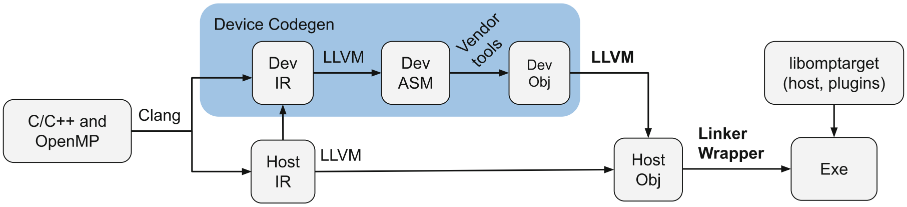


Fig. 1. The main phases of the new LLVM/Clang offloading driver. Source file compilation is done for every target architecture using the device tools. This is then embedded into the host to create a fat-binary and linked with the new augmented linker.

Our new driver supports OpenMP offloading compilation is a unified manner by utilizing a common embedding and linking scheme for each target architecture. The first step is to compile each input OpenMP offloading program to an object file using LLVM/Clang. If we are performing LTO, we instead emit LLVM-IR instead of a standard ELF object file

We then complete the host compilation and create a fat-binary by storing each output device object file in a special section in the output ELF file. This special section contains both the embedded image and a binary blob containing necessary metadata to link the device image, such as its target and architecture. This section will be named `llvm.offloading` and is identified in the ELF using the new `SHT_LLVM_OFFLOADING` section type. Furthermore, we use the `SHF_EXCLUDE` section flag to indicate that this section should be dropped by the linker when creating the final executable. The final compilation step is to pass the new fat binary to the linker, which will use the embedded device code to create an executable device image. These steps are roughly outlined in Fig. 1.

3.2 Offload Linker

We created a new augmented linker supporting offloading device linking and registration. This new augmented linker works as a thin wrapper over the original host linking job called the linker wrapper. First, the linker wrapper searches every input file or library for embedded device code stored in the `SHT_LLVM_OFFLOADING` sections. Once the files are located and extracted we sort each input object using its target architecture that was extracted from the metadata stored previously. We then identify all the input files containing LLVM-IR and use LLVM's existing LTO library to create an object file output. All device objects for a single target architecture are then linked together using the vendor linker to create an executable.

The device executable is useless on its own, so first we need to register each linked executable with the vendor's runtime library. We perform this final step by creating a new module containing the executable data and the runtime calls necessary to register it. This module is then compiled to an ELF object file and added to the linker input. Finally, we run the original host linking job and obtain an executable containing offloading code.

4 Just-in-Time Compilation Support

In this section, we will introduce our support for JIT compilation of OpenMP offloading applications in LLVM. We will first talk about the necessary compiler support, followed by the code generation and sub-architecture portability support. Next, we propose three specializations to improve optimizations using JIT. Finally, we discuss a multi-level caching implementation used to mitigate host overhead caused by JIT compilation and optimization.

4.1 Compilation Flow

We utilized the LTO support shown in Sect. 3 to create linked LLVM-IR necessary for JIT compilation. The only change required to the compilation flow for JIT is to skip the LTO back-end in the linker and register the linked LLVM-IR directly. Now when the runtime attempts to register the device code it will use JIT if it encounters LLVM-IR instead of a device executable.

4.2 JIT Kernel Invocation

When the OpenMP runtime attempts to execute a kernel when performing JIT we first need to compile and register it as before. We will also use LLVM's LTO support for the code generation to call the same back-end we skipped during ahead-of-time linking. The previous LTO optimization pass is augmented with JIT-specific optimizations that will be described later as well as aggressive pruning of global definitions unused by the current kernel. After the LTO backend is run, we then need to register the kernel with the device runtime and proceed to the kernel launch.

4.3 Sub-architecture Portability

AoT compilation does not support sub-architecture portability because device images are not usually compatible with different compute capabilities. For example, if the program is compiled for `sm_35`, it can only run on `sm_35` GPUs. Programs instead must be recompiled for the desired target device.

With our JIT support, device images are generated at runtime using target information collected from the target device, thus we only need to compile programs once (AoT) and they can be executed on different target devices with varying sub-architectures. However, it is worth noting that this does not support the portability across different vendors. As we mentioned before, we embed LLVM IR which is still inherently vendor dependent. In addition, for AMD GPUs, an extra pass is set up to update all target features attached to functions to make sure its backend works properly.

4.4 Specialization

We propose three specializations with information only available at runtime. They are all enabled by default and can be configured via environment variables.

Scalar Kernel Arguments. One of the most important pieces of runtime information is kernel arguments. There are two kinds of kernel arguments: pointer values and scalar values. We do not specialize pointer values because it can easily invalid caching, which will be discussed later, incurring more host side overhead. Scalar values are however specialized, hence replaced with their runtime value prior to optimizations. If the scalar values are loop bounds, it can make more aggressive loop optimization possible.

Pointer Alignments. An important characteristic of a pointer is its alignment, which plays an important role in vectorization and instruction selection. Although each target has a default pointer alignment, the actual alignment of a pointer can be more strict. For each pointer value p, we iterate a list of predefined alignments $a \in \{128, 64, 32, 16, 8\}$ in decreasing order and find the first a that p is aligned to. If a is greater than the default alignment, an attribute `align` with value a is added to the pointer kernel argument.

Launch Parameters. Two kernel launch parameters, grid size and block size, are provided to the driver API when launching a kernel. If the `num_teams` clause or `thread_limit` clause is present and a compile time constant value is specified, the corresponding runtime functions to query the size are optimized away at compile time [1]. If the clause is not specified, the runtime will choose a default value. Given that these launch parameters are known to the JIT, specialization is performed as if the user provided constant values via the respective clauses.

4.5 Internalization

In the device runtime, there are global variables listed in `@llvm.used` to prevent to be optimized out when building the device runtime. At JIT time[1], since the module has already been linked, all feasible global variables, except those that should be exposed to users, can be optimized. We mark all global variables, except those exposed to users, as `internal` and remove them from `@llvm.used`.

4.6 Caching

JIT compilation requires constructing an `LTOModule`, going through kernel arguments, modifying the module, generating a device image, and loading it to the target device. This can have significant overheads and can potentially cancel the benefits of our runtime optimizations. To mitigate the cost we need to reuse the generated device image for multiple kernel launches. To this end, we implement a novel classification system for kernel launches together with a persistent, two-level cache system that keeps specialized images in the host memory (L2) as well as in the device memory (L1) for future reuse. JIT compilation is only invoked if there is no compatible cached image available at launch time.

Kernel Launch Identification. In order to reuse an image, we need an efficient way to determine if it is compatible with an incoming kernel launch request. A kernel launch is effectively defined by the kernel (function) name, the kernel arguments, and the number of teams and threads (= grid dimensions). However, kernel launches do not require identical values to share/reuse the same optimized image. For example, pointers `0x1230` and `0x4560` are not exactly same, but they are both 16-byte aligned. If that is the only difference between two kernel launches, they are compatible. Additionally, if parameters are not involved in specialization their values do not impact kernel launch compatibility.

In order to efficiently query the cache we employ a *kernel launch descriptor*, which includes: the kernel name, kernel arguments, architecture, and a list of *specializations* applied to the kernel when the image was compiled. An existing optimized image with a kernel launch descriptor is compatible with a kernel launch, and hence can be reused, if (1) the kernel name and architecture match and (2) the specializations applied to obtain the image match what would have been applied for the new launch in question.

L1: Target Table Cache. A *target table* stores information about offloading entries, such as entry size, host pointer and its corresponding device pointer. It is constructed when an image is loaded to the device. Therefore, it is per device and every execution starts with an empty L1 cache that is filled on-demand.

We set up a target table cache, indexed by kernel entry name, for each target device. Each entry is a list of target tables for the same kernel entry but with different kernel launch descriptors.

[1] Technically, this does not have to be limited to JIT time but LTO time is sufficient.

L2: Image Cache. An *image* is a memory buffer that can be loaded to a target device. It can be used for all target devices with same sub-architecture. More importantly, it does not contain (dynamic) device pointers, which gives us the ability to reuse it across executions. Images are reused within and across program runs whenever a compatible kernel launch is encountered.

We set up an image cache for each sub-architecture. An image cache is organized similar to target table cache. In addition, during the runtime shutdown, the image cache writes all cached images and metadata to a file. When the runtime is loaded, it reads all images from the file and construct the image cache to be used by the application. Hence, prior runs with compatible kernel launch parameters can effectively eliminate most overheads of just-in-time compilation.

Cache Lookup. When a kernel `k` is launched, the cache lookup works as follows:

1. Check if there is a compatible entry in the target table cache (L1) for the target device. If yes, move to Step 2; otherwise, move to Step 3.
2. Iterate over the list of target tables. If there is a *match*, it is a L1 cache *hit*, and the target table can be used directly; otherwise, it is a L1 cache *miss* and we proceed with Step 3.
3. Check if there is a compatible entry in the image cache (L2) for the sub-architecture of the target device. If yes, move to Step 4; otherwise, it is a L2 cache *miss* and we proceed to Step 5.
4. Iterate over the list of images. If there is a *match*, it is a L2 cache *hit*. The image will be loaded to the target device, a new target table will be constructed and added to the L1 cache. If no *match* was found it is a cache *miss*. Move to Step 5.
5. JIT the device image, add it to the L2 cache (for cross-execution persistence), load it to the target device, and add it to the L1 cache.

4.7 Specialization Tracker

In spite of the multi-level caching system, it is still possible that scalar kernel arguments for a kernel vary in every (or many) different kernel launches. For example, `552.pep` in SPEC ACCEL [2] has one scalar argument that changes in every kernel launch. As consequence, we have to compile a new image and load it to the device for every launch. This situation can cause significant overheads and device resource waste.

We set up a specialization tracker for each kernel entry which records the total number of specializations, denoted by N, and the number of specializations for each kernel argument, including the launching parameters, represented by n_i. Before we apply any specialization, we check if $N > T$ and $n_i/N > R$, where T is a threshold to always allow a certain amount of specialization, and R is an argument specialization control ratio. If both conditions are true we have exceeded the specialization quota for an argument and it is not specialized in the future. For any subsequent kernel launch, no matter whether the argument value change, there has to be a match as no argument specializations has been applied to one image. Both T and R can be configured via environment variables.

5 Evaluation

For our performance evaluation we used a Nvidia A100 GPU system with an AMD EPYC 7532 CPU and 256 GB DDR4 RAM. We used CUDA 11.4.0 for all experiments and collected kernel times with `nsys`. In addition to the Nvidia system, an AMD MI100 GPU system with two AMD EPYC 7532 CPUs and 512 GB DDR4 RAM is used for portability evaluation. Our prototype version (Ᵽ) is based on git `3723868d`.

5.1 Benchmarks

We looked at seven scientific proxy applications for our performance study for LTO and JIT compilation and evaluated both the end-to-end execution time and the performance of their main GPU kernels. Our results are presented relative to the performance of AoT compilation without LTO. We also test four of the seven proxy applications for sub-architecture portability with JIT.

OpenMC is a continuous-energy Monte Carlo particle transport application [3] that has recently been ported to the OpenMP target offloading programming model for use on GPU-based systems [4]. In addition to being an open source application, OpenMC also provides a host of advanced modeling and simulation capabilities including depletion, advanced geometry representations, on-the-fly Doppler broadening, and multigroup cross section generation.

XSBench and RSBench are two proxy applications for the Open Monte Carlo (OpenMC) project. Both proxies compute the continuous energy macroscopic neutron cross-section lookup when studying neutron transport and both are available in multiple programming languages and frameworks. While XSBench [5] extracts one of the main kernels in OpenMC, which is in memory bound, RSBench [6] provides a compute bound alternative implementation.

MiniFMM is a proxy application developed by the University of Bristol for Fast Multipole Method (FMM) [7]. It solves the Laplace equation in a three-dimensional polar coordinate plane by applying the FMM, which uses a dualtree traversal method.

SU3 is a Lattice QCD SU(3) matrix-matrix multiply microbenchmark. The kernel is based on the `mult_su3_nn()` SU(3) matrix-matrix multiply routine in the MILC Lattice Quantum Chromodynamics(LQCD) code.

Thermo4PFM is a software library used for Phase-Field modeling of solidification in metallic alloys [8]. Given the thermodynamic properties of a materials in its various phases, it solves a small system of non-linear equations to compute the force that drives phase changes.

miniMDock is a GPU-accelerated performance portable particle-grid based protein ligand molecular docking tool. It is used for virtual drug discovery compound screens based on a molecular recognition model, that analysis a three-dimensional model of an interaction between a protein and a small molecule.

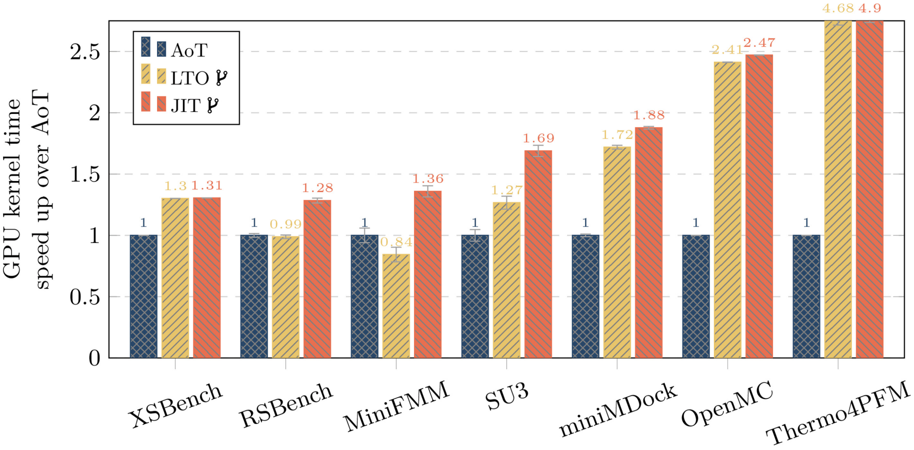


Fig. 2. Kernel execution time relative to the base AoT case without LTO.

5.2 Performance Results

Figure 2 shows the relative improvement in kernel time for LTO and JIT. Our results show large improvements for miniMDock, OpenMC, and Thermo4PFM when LTO is used because these applications have code split between many files and benefit most from LTO. Although the other cases do not benefit from cross-file optimizations, LTO can still affect performance due to additional optimizations and internalization of external symbols. The performance evaluation for JIT uses only the kernel timings and does not include the overhead necessary to first compile the image. Hence, we assume a perfect pre-filled cache.

Figure 3 shows the relative improvement in end-to-end time for JIT with and without the offline cache. We can see that in most cases, JIT compilation can improve the end-to-end performance, except for miniMDock and Thermo4PFM. For miniMDock, the total kernel time is optimized to about 1.4 s from 2.6 s. However, because miniMDock uses random inputs the cache is easily invalidated which results in overall slower execution. In the worst case, the overall JIT overhead is more than 4.5 s, leading to the performance regression shown in Fig. 3. This demonstrates that for applications similar to miniMDock, specialization should be disabled (adaptively). Thermo4PFM also shows an end-to-end performance regression, but the offline cache allows it to retain performance.

5.3 Portability Results

Although performance can change when using JIT, another advantage is providing sub-architecture portability. The sub-architectures of our GPU systems are SM80 (Nvidia A100) and GFX908 (AMD MI100) respectively. Figure 4 shows the results of different benchmarks compiled with different sub-architectures on the two GPU systems, where ✓ means the benchmark runs without any issue.

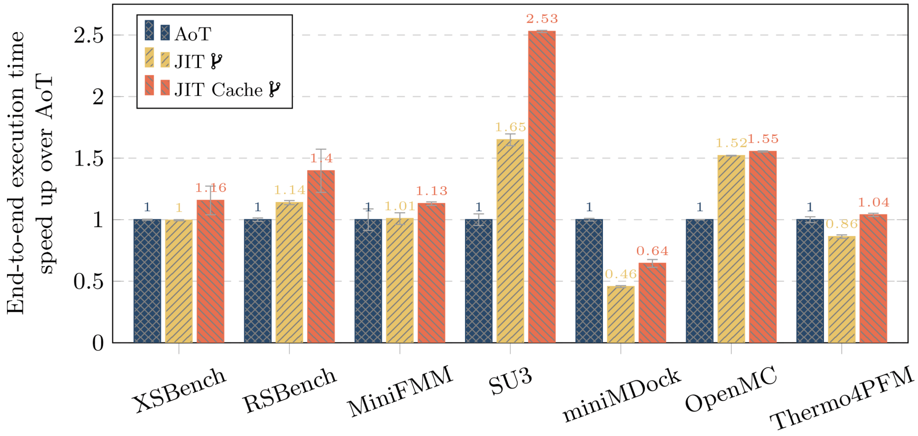


Fig. 3. End-to-end execution time relative to the base AoT case without LTO.

Benchmark	SM35	SM53	SM60	SM75	GFX701	GFX803	GFX902
XSBench	✓	✓	✓	✓	✓	✓	✓
RSBench	✓	✓	✓	✓	✓	✓	✓
MiniFMM	✓	✓	✓	✓	✓	✓	✓
SU3	✓	✓	✓	✓	✓	✓	✓

Fig. 4. Portability of benchmarks compiled with different SM versions for Nvidia A100 GPU and different GFX versions for AMD MI100 GPU.

6 Related Works

6.1 OpenMP Target Offloading

OpenMP 4.0 introduced target offloading. In LLVM/Clang, OpenMP offloading support for GPUs was first presented by [9,10]. The (PGI) Fortran front-end, known as Flang, supports OpenMP offloading via the LLVM/OpenMP runtimes [11]. GCC 5 first supports OpenMP target offloading on Intel MIC architecture. Starting from GCC 7, Nvidia platforms support was added. All existing implementation feature ahead-of-time compilation of device code.

6.2 Just-in-Time Compilation

Just-in-time compilation has been used in software systems for decades [12]. However, the support in programming languages vary. For parallel programming models, OpenCL naturally employs JIT compilation for parallel code execution via an intermediate representation SPIR-V [13]. [14] implemented automatic translation of OpenACC to LLVM IR with SPIR kernels, optimization of the IR code by LLVM optimizer, and execution of the host LLVM IR by LLVM JIT. For OpenMP, [15] proposed an on-the-fly technique on top of the Pin binary instrumentation [16] to detect data races in OpenMP programs. [17] presented support for parallel programs written in OpenMP executing on JVM using LLVM IR.

6.3 Link Time Optimization

Link-time optimization is not a new concept and has been supported by various compilers, including GCC and LLVM/Clang [18]. Nvidia has supported device-side LTO for CUDA following the CUDA 11.2 release [19], however the Nvidia compilers do not support LTO for OpenMP offloading. The AMDGPU toolchain uses LLVM IR bitcode as its relocatable object file format and used bitcode linking and LLVM optimizations as a part of its compilation.

6.4 Compiler Optimization for OpenMP

Regarding compiler-based optimizations on OpenMP, [20] introduced the first front-end based optimizations for Nvidia GPUs in LLVM/Clang, related to choosing the number of teams and threads for parallel loops to avoid idle threads and reduce register usage. [21] presented the TRegion interface which delayed the discovery of SPMD regions into LLVM, by contrast to the Clang-based approach, which enabled more kernels to execute in SPMD mode. [22] introduce in the IBM XL C/C++ compiler a lowering of OpenMP that executes without the control loop state machine in a mode where all threads execute in parallel, deemed SPMD mode of execution, when the target offloaded region encloses a single parallel construct. [1] presented OpenMP-aware program analyses and optimizations that allow efficient execution of the generic, CPU-centric parallelism model provided by OpenMP on GPUs. [23] presented a co-design methodology for optimizing applications using a specifically crafted OpenMP GPU runtime inducing near-zero overhead in most cases. Recent advances in architecture porting have made it feasible to extend our work of sub-architecture specialization to retargeting across different vendors [24,25].

To our best knowledge, this is the first work to present JIT compilation and LTO for OpenMP target offloading.

7 Conclusion and Future Works

In this paper, we proposed just-in-time compilation and link-time optimization for LLVM/OpenMP target offloading. We showed a new compiler driver to embed and link device bitcode, and a novel JIT engine that features optimization, caching, and sub-architecture portability. The evaluation results show that link-time optimization can provide large performance benefits for certain applications and we can further optimize applications and offer sub-architecture portability using JIT compilation. In the future, we plan to further improved JIT compilation and specialization in LLVM/Clang as a default for better interoperability between architectures.

Acknowledgement. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative. We also gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The publisher acknowledges the US government license to provide public access under the DOE Public Access Plan (https://energy.gov/downloads/doe-public-access-plan).

References

1. Huber, J., et al.: Efficient Execution of OpenMP on GPUs. In: IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2022, Seoul, Republic of Korea, 2–6 April 2022, pp. 41–52 (2022)
2. Juckeland, G., et al.: SPEC ACCEL: A standard application suite for measuring hardware accelerator performance. In: High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation - 5th International Workshop, PMBS 2014, New Orleans, LA, USA, 16 November 2014. Revised Selected Papers. vol. 8966, pp. 46–67 (2014)
3. Romano, P.K., Horelik, N.E., Herman, B.R., Nelson, A.G., Forget, B.: OpenMC: a state-of-the-art Monte Carlo code for research and development. Ann. Nucl. Energy **82**, 90–97 (2015). https://doi.org/10.1016/j.anucene.2014.07.048, https://doi.org/10.1016/j.anucene.2014.07.048
4. Tramm, J., et al.: Toward portable GPU acceleration of the OpenMC Monte Carlo particle transport code. In: International Conference on Physics of Reactors (PHYSOR 2022). Pittsburgh, USA (2022)
5. Tramm, J.R., Siegel, A.R., Islam, T., Schulz, M.: XSBench - the development and verification of a performance abstraction for Monte Carlo reactor analysis. In: PHYSOR (2014)
6. Tramm, J.R., Siegel, A.R., Forget, B., Josey, C.: Performance analysis of a reduced data movement algorithm for Neutron cross Section data in Monte Carlo simulations. In: Solving Software Challenges for Exascale - International Conference on Exascale Applications and Software, EASC 2014, Stockholm, Sweden, 2–3 April 2014, Revised Selected Papers. vol. 8759, pp. 39–56 (2014)
7. Atkinson, P., McIntosh-Smith, S.: On the performance of parallel tasking runtimes for an irregular fast multipole method application. In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2017. LNCS, vol. 10468, pp. 92–106. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65578-9_7
8. Fattebert, J.L., Wickett, M., Turchi, P.: Phase-field modeling of coring during solidification of au-ni alloy using quaternions and calphad input. Acta Materialia **62**, 89–104 (2014). https://doi.org/10.1016/j.actamat.2013.09.036

9. Bertolli, C., et al.: Coordinating GPU threads for OpenMP 4.0 in LLVM. In: Proceedings of the 2014 LLVM Compiler Infrastructure in HPC, LLVM 2014, New Orleans, LA, USA, 17 November 2014, pp. 12–21 (2014)
10. Bertolli, C., et al.: Integrating GPU support for OpenMP offloading directives into clang. In: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM 2015, Austin, Texas, USA, 15 November 2015. pp. 5:1–5:11 (2015)
11. Özen, G., Atzeni, S., Wolfe, M., Southwell, A., Klimowicz, G.: OpenMP GPU Offload in Flang and LLVM. In: 2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC), pp. 1–9 (2018)
12. Aycock, J.: A brief history of just-in-time. ACM Comput. Surv. **35**(2), 97–113 (2003)
13. The Khronos Group Inc.: SPIR Overview (2022). https://www.khronos.org/spir/
14. Peng, H., Shann, J.J.: Translating OpenACC to LLVM IR with SPIR kernels. In: 15th IEEE/ACIS International Conference on Computer and Information Science, ICIS 2016, Okayama, Japan, 26–29 June 2016. pp. 1–6 (2016)
15. Ha, O., Kuh, I., Tchamgoue, G.M., Jun, Y.: On-the-fly detection of data races in OpenMP programs. In: Proceedings of the 10th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, PADTAD 2012, Minneapolis, MN, USA, 16 July 2012. pp. 1–10 (2012)
16. Luk, C., Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, 12–15 June 2005. pp. 190–200 (2005)
17. Gaikwad, S., Nisbet, A., Luján, M.: Hosting OpenMP programs on Java virtual machines. In: Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, MPLR 2019, Athens, Greece, 21–22 October 2019. pp. 63–71 (2019)
18. Glek, T., Hubicka, J.: Optimizing real world applications with GCC link time optimization. arXiv preprint arXiv:1010.2196 (2010)
19. Murphy, M., Sundaram, A.: Improving GPU application performance with NVIDIA CUDA 11.2 device link time optimization, February 2021. https://developer.nvidia.com/blog/improving-gpu-app-performance-with-cuda-11-2-device-lto/
20. Antão, S.F., et al.: Offloading support for OpenMP in clang and LLVM. In: Third Workshop on the LLVM Compiler Infrastructure in HPC, LLVM-HPC@SC 2016, Salt Lake City, UT, USA, 14 November 2016. pp. 1–11 (2016)
21. Doerfert, J., Diaz, J.M.M., Finkel, H.: The TRegion interface and compiler optimizations for OpenMP target regions. In: Fan, X., de Supinski, B.R., Sinnen, O., Giacaman, N. (eds.) IWOMP 2019. LNCS, vol. 11718, pp. 153–167. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-28596-8_11
22. Tiotto, E., Mahjour, B., Tsang, W., Xue, X., Islam, T., Chen, W.: OpenMP 4.5 Compiler optimization for GPU offloading. IBM J. Res. Dev. **64**(3/4), 14:1–14:11 (2020)
23. Doerfert, J., Patel, A., Huber, J., Tian, S., Diaz, J.M.M., Chapman, B., Georgakoudis, G.: Co-Designing an OpenMP GPU runtime and optimizations for near-zero overhead execution. In: 36th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2022, St. Petersburg, FL USA, 15–19 May 2023. IEEE (2022)

24. Doerfert, J., et al.: Breaking the vendor lock – performance portable programming through OpenMP as target independent runtime layer. In: International Conference on Parallel Architectures and Compilation Techniques, PACT (2022, to appear)
25. Moses, W.S., Ivanov, I.R., Domke, J., Endo, T., Doerfert, J., Zinenko, O.: High-performance GPU-to-CPU transpilation and optimization via high-level parallel constructs (2022). https://doi.org/10.48550/ARXIV.2207.00257, https://arxiv.org/abs/2207.00257

Extending OpenMP to Support Automated Function Specialization Across Translation Units

Giorgis Georgakoudis(✉), Thomas R. W. Scogland, Chunhua Liao, and Bronis R. de Supinski

Lawrence Livermore National Laboratory, Livermore, CA 94550, USA
{georgakoudis1,scogland1,liao6,desupinski1}@llnl.gov

Abstract. OpenMP's variant directives support specialization at compilation time using OpenMP context for portability or performance. This specialization is confined either to variants explicitly written by the code author, which can cross translation units, or to implicit context passed by the compiler. The implicit context allows a `metadirective` directive to choose a directive variant based on context or the compiler to optimize out runtime interactions. However, that implicit context only exists in a single translation unit (TU), either in a single compilation or with link-time optimization linking the set of TUs. In order to enable more optimization opportunities, we propose the `metavariant` directive, a new variant directive to define possible function specializations over a set of specified OpenMP contexts that are available across different translation units. The compiler lowers the definition of a `metavariant`-annotated function to different instances for each specified context. Calls of the function in different translation units use local OpenMP context for specialization, relying on the fact that the compiler will have generated appropriately mangled symbol names for those instances. Using a prototype source-to-source tool and a set of use cases, we evaluate our approach to observe a speedup of up to 30× with inter-procedural specialization versus no specialization, while simplifying and enhancing modular adaptation with modest user effort.

Keywords: OpenMP · Function Specialization · Translation Units

1 Introduction

Recent versions of OpenMP have added several directives that enable the application programmer to specify context-specific optimization hints and specializations as well as application-specific requirements. However, these mechanisms currently provide limited support across translation units and often include restrictions that the programmer must ensure that they are used consistently across them. Mechanisms to extend optimization opportunities across translation units and to enforce requirements across them would enable greater modularity in application source code while also preserving and, perhaps, simplifying implementation-specific optimizations.

M. Klemm et al. (Eds.): IWOMP 2022, LNCS 13527, pp. 159–173, 2022.
https://doi.org/10.1007/978-3-031-15922-0_11

For example, the metadirectives and declare variant directives allow the application programmer to specify context-specific specialization. The use of OpenMP contexts allows these mechanisms to incorporate aspects of the OpenMP (and, through the `user` selector, application-specific) context. These contexts may arise in a different translation unit. However, metadirectives do not provide any mechanism for the application programmer to specify that the context-specific specialization is available to the calling context. Alternatively, declare variant directives allow specialized functions to be called, possibly from a different translation unit, based on the calling context. However, the programmer must fully specify the specialized function and the OpenMP implementation is not expected to exploit the information about the calling context that the directive provides.

Application programmers would benefit from OpenMP extensions that better support optimization and specialization across translation units. In this paper, we define the `metavariant` directive, a mechanism that enables the programmer to specify that the implementation should generate function variants that can be safely called from a specific context. When combined with the use of metadirectives and declare variant directives, the programmer can ensure that the function conforms to any requirements of the calling context and that it provides information about assumptions that may hold in it.

We evaluate the `metavariant` directive using a prototype source-to-source translation tool. We explore potential use cases of the directive that illustrate the benefits that it can provide in terms of programmability and in making optimization hints more useful across translation units. Our preliminary experiments demonstrate that it can yield substantial performance benefits.

2 Background

Variant directives, including `metadirectives` and `declare variant` directives, are major new features introduced in OpenMP 5.0 [3] to improve performance portability by enabling adaptation of OpenMP pragmas and user code at compile time. The basic idea of variant directives is to allow programmers to suggest a suitable code variant for a given OpenMP context, which includes traits that describe active OpenMP constructs, execution devices, functionality of an implementation, and user-defined conditions. While OpenMP 5.0 supported only matching compile-time conditions on traits, OpenMP 5.1 [4] extended that to include user-defined conditions resolved at runtime. A metadirective is a declarative directive that conditionally resolves to another executable or declarative directive by selecting from multiple directive variants based on traits that define an OpenMP context. The declare variant directive has similar functionality as the metadirective but selects a function variant at the call site based on context or user-defined conditions.

However, a major limitation of the current variant directives is that programmers need to specify both context and variant information (i.e., the mapping of context and variants) within the same translation unit. Currently, OpenMP does not include any direct language support to propagate context-variant mapping information easily across multiple translation units for either specialization or facilitating compiler optimizations. Existing context-aware directives, such as `declare simd` and `declare target` have a narrow definition of context and provide limited specialization across translation units. The `declare variant` directive supports OpenMP traits for expanding the possible context specification but requires explicit user specification of the correspondence of variant function symbols to the matching context that specializes the base function symbol. Further, this correspondence is visible to callers of `declare variant` functions, to specialize their call sites, while function variants themselves are not required to include a notion of their corresponding context, thus lacking in using and propagating context in their definitions.

The following code snippet shows an example use of `metadirective`. An OpenMP compiler can readily generate two specializations (or variants) of the for loop based on the context-variant mapping information explicitly specified by the `metadirective`.

```
void foo (int* v1, int* v2, int*v3, size_t N)
{
  #pragma omp metadirective \
      when(target_device={arch(nvptx)}: \
        target teams distribute parallel for \
       map(to:v1[0:N],v2[0:N]) map(from:v3[0:N]) ) \
      otherwise(parallel for)
  for (int i= 0; i< N; i++)
    v3[i] = v1[i] * v2[i];
}
```

The function `foo()` may be defined in one source file while many of its call sites are located in other source files. A compiler may better optimize the function (such as generating more or less specializations) if it knows all possible contexts within which the function will be called, when compiling the source file with the function definition. Similarly, a compiler may invoke the right function variant at each call site, if it knows all available variants of the function definition when compiling a source file containing the call site.

For example, if a compiler knows that all call sites of `foo()` will be within a parallel region, it may specialize the function definition to implement only `omp for` and to avoid generating code for nested parallelism. A CPU implementation involves relatively little difference between these choices other than the fork and join overhead. However, the difference on a GPU though can be between running the entire region directly on the native parallel threads, and being forced to use a heavy-weight concurrent state machine to implement varying levels of parallelism as the kernel progresses. The difference in performance between the "lightweight" native runtime and the state-machine can be orders of magnitude in some cases, even for otherwise identical code.

While in some cases, the compiler can perform LTO (Link-Time-Optimization) to propagate the OpenMP context and code variant information across translation units, LTO for most heterogeneous platforms is currently prohibitively expensive. Enabling LTO may entirely serialize the process of compilation of large-scale code bases. This cost is too high in practice as many large scientific applications already take hours to build while being compiled in parallel. Another limitation of compiler-based solutions is that statically computing the caller-callee relationship (such as the one used by Interprocedural Optimization or IPO) is still a challenging problem when facing complex uses of function pointers and dynamic dispatch. Yet another approach is to detect activated context at runtime and to trigger runtime code specialization. However, this approach requires the implementation to generate code variants at runtime, with potentially significant runtime overhead.

Therefore, OpenMP needs to allow programmers to communicate the context-variant mapping information across multiple translation units explicitly. An implementation could then automatically exploit such semantics to optimize code generation at compile time, without relying on sophisticated program analyses or incurring runtime overhead. While we could introduce such functionality through significant re-definition of the semantics and specification of `declare variant`, we choose to introduce a new directive, named `metavariant`, both for exposition and because it is cleaner and more concise than retrofitting it to `declare variant`.

3 The `metavariant` Directive

We introduce a new directive, the `metavariant` directive, to enable compile-time function specializations across translation units, by matching the OpenMP context that propagates at call sites of those functions. For the translation unit that contains the definition of a metavariant-annotated function, compilers generate OpenMP context specializations of the function by specializing `metadirective` directives in the function's body, assuming a specific OpenMP context is matched. Correspondingly, at the translation units containing metavariant function callers, OpenMP compilation tracks the OpenMP context at call sites of the metavariant function to call the function specialization that matches the context, if any, otherwise falling back to the function specialization that does not assume a specific context. Since the metavariant function and its caller functions may be in different translation units, we propose *semantic function symbol naming* to encode the different function specializations, so that OpenMP compilation infers the function specialization symbol to call at call sites by using the OpenMP context and the original function symbol.

In more detail, the syntax of the `metavariant` directive is:

```
#pragma omp metavariant [clause,[[,] clause]...] new-line
  function definition or declaration
```

where *clause* is the following:

```
#pragma omp metavariant \
    match(construct={parallel})
void foo(int* v1, int *v2, int* v3, size_t N)
{
  #pragma omp metadirective \
    when(construct={parallel}: taskloop) \
    otherwise(parallel for)
  for (int i= 0; i< N; i++)
    v3[i] = v1[i] * v2[i];
}
```

Fig. 1. An example translation unit with the definition of a metavariant function

```
match(context-selector-specification)
```

Semantically, a `metavariant`-annotated base function has one specialization associated with each `match` clause and this specialization assumes the context specified in the clause is in effect. Thus, the specified context of this specialization will forward to `metadirective` directives in the function definition or calls to other `metavariant` functions, which will be specialized by this propagating context. Also, the base function without specialization remains available without assuming any specific OpenMP context. The symbol name of each function specialization is determined from base language rules extended by a string determined by the effective context selector of its associated `match` clause. The symbol name of the function without specialization is determined from base language rules without any string extension.

The function specialization variant is determined at the call site depending on its OpenMP context. If the OpenMP context at the call site matches the context of a specialization, the call is replaced with a call to the function variant of this specialization. Otherwise the call is not replaced, thus resolving to the original function, which is the no-specialization variant.

To make use of a `metavariant` function across translation units, a function prototype with its corresponding `metavariant` directive can be put into a header file, which is then included by other translation units. Symbol names of specializations provide the ABI contract that supports this functionality.

In summary, on a metavariant function definition, metavariant-enabled OpenMP compilation generates different functions for the different context specification in each `match` clause, specializing any metadirectives, calls declare variant functions, or to other metavariant functions in the function body by forwarding the matching context. Each different function variant follows an ABI convention that encodes context-awareness across translation units. On metavariant function declarations, metavariant-enabled OpenMP compilation generates the variant function declarations following the same ABI convention and transforms call sites of the metavariant function to use the function variant that matches the context at the call site.

```
// Variant matching execution in a parallel context.
void foo_construct_parallel(int* v1, int *v2, int* v3, size_t N)
{
#pragma omp taskloop
  for (int i= 0; i< N; i++)
    v3[i] = v1[i] * v2[i];
}
// Variant matching execution in sequential context.
void foo(int* v1, int *v2, int* v3, size_t N)
{
#pragma omp parallel for
  for (int i= 0; i< N; i++)
    v3[i] = v1[i] * v2[i];
}
```

Fig. 2. Lowering of the translation unit with the metavariant function definition

Figure 1 illustrates the definition of a `metavariant` function named `foo`, performing a simple vector addition that specializes the execution of its loop depending on whether it is called within a parallel or a sequential context. Specifically, `foo` uses a metadirective to use a `taskloop` construct for parallel execution of the loop when it is called from a parallel context, whereas the metadirective specifies `parallel for` for parallel work-sharing execution when it is called from a sequential context. Figure 2 presents a source-to-source lowering of the metavariant function, using symbol naming that uniquely identifies different specializations corresponding to different metavariant contexts, including flattening of metadirectives in the function's body.

Figure 3 also shows how the metavariant function is declared and called in a different translation unit. The source code must include the prototype of the metavariant-annotated function declaration to declare its possible specializations. The caller function `bar` calls the metavariant function using its declaration symbol (`foo`). OpenMP compilation replaces this symbol at each call site with the symbol name that corresponds to the metavariant function specialization that matches the enclosing OpenMP context of the call site. Figure 4 shows a source-to-source lowering of the translation of a caller to a metavariant function. The metavariant function declaration resolves to two function declarations, one matches the parallel context specialization assuming the same function symbol in the callee's translation unit in Fig. 2, the other matches the non-parallel context specialization with the function symbol unchanged.

3.1 Discussion

In this section we discuss several aspects of metavariant-enabled compilation and its relation to other compilation and specialization approaches.

JIT/LTO Context Propagation and Specialization. Using LTO for context propagation across translation units is an interesting proposition assuming some

```
#pragma omp metavariant \
    match(construct={parallel})
void foo(int* v1, int *v2, int* v3, size_t N);

void bar(int* v1, int *v2, int* v3, size_t N)
{
  // Calling foo() in a parallel context.
  #pragma omp parallel
  {
    // do other parallel work.
    // Call foo() by main thread.
    if (omp_get_thread_num() == 0)
        foo(v1, v2, v3, N);
  }

  // Calling foo() in a sequential context.
  foo(v1, v2, v3, N);
}
```

Fig. 3. An example translation unit with a call site of a metavariant function

mechanism conveys context information from existing context-aware directives, such as `declare variant`, to both callers and callees. This alternative requires that LTO completely re-constructs the call graph to couple it with context information. However, this alternative can be problematic since LTO compilation is time consuming and also necessitates shared libraries to be amenable to it. Context-aware JIT compilation is also a possibility, however the cost of runtime compilation and tracking context may be prohibitive. The metavariant approach avoids those issues by providing an elegant way to specify and to propagate context at compile time, to generate specializations through automatic code generation, and proposing a context-dependent ABI convention for cross-translation unit specialization. It does so without relying on LTO or JIT compilation, which may be unavailable or undesirable due to their limitations.

Consumers of Context for Specialization. In this formulation of the metavariant directive possible consumers of context for specialization include metadirectives and calls to declare variant or other metavariant functions, propagating context, in the metavariant function's definition. While this approach gives explicit control to the user, a specification-based rule set for automatic transformations of OpenMP directives can avoid pathological use-cases, such as nesting parallel regions shown in the example of Fig. 3. Such rule sets can also be used for error checking to emit warning/errors when incompatibilities or performance degradation occurs between the caller and the assumed callee context of a metavariant function. We leave that as interesting future work.

Differences with `declare variant` Specialization. The `declare variant` directive explicitly specifies the symbol of function variant specializations of a

```
void foo_construct_parallel(int* v1, int *v2, int* v3, size_t N);
void foo(int* v1, int *v2, int* v3, size_t N);

void bar(int* v1, int *v2, int* v3, size_t N)
{
  // Calling foo() in a parallel context.
  #pragma omp parallel
  {
    // do other parallel work.
    // Call the parallel context specialization of foo().
    if (omp_get_thread_num() == 0)
        foo_construct_parallel(v1, v2, v3, N);
  }
  // Call the non-parallel specialization of foo().
  foo(v1, v2, v3, N);
}
```

Fig. 4. Lowering of the translation unit with a call site of a metavariant function

base function that correspond to different matching contexts. Calls to the base function symbol are specialized to the function variant that matches the caller's context. By the specification, context cannot be not assumed to propagate to the function variant. Thus, the user must explicitly implement any specialization in the definition of the function variant symbol without assuming any context is propagated during compilation for compiler-based specialization, e.g., through metadirectives. By contrast, metavariant function variants are context-aware, auto-generated during compilation using compiler-based code generation for specialization by propagating context to existing variant directives (metadirectives, declare variant) or to other metavariant functions called by the variant. The base function definition of a metavariant function is a fallback that does not assume any OpenMP context. We purposefully avoid providing user-visible naming of function variants in the metavariant directive, relying instead on an ABI convention. The functionality of calling a function variant without requiring explicitly naming the function symbol is possible through a metadirective, using a `when` clause with the matching context. Also, exposing function variants to users is error prone, since users could call them within incompatible caller contexts.

Tracking Context. The example specialization of Fig. 1 uses `taskloop` for parallelizing loop execution when called within a parallel context instead of a work-sharing construct. This choice is necessary to avoid possibly incompatible nesting of work-sharing constructs. Implementors of metavariant functions should be aware of such limitations to implement compatible specializations or context specification should be enhanced to include extended traits, such as work-sharing execution. The metavariant directive proposal opens up this discussion, which we leave as future work.

4 Evaluation

4.1 Experimentation Setup

We experimented on a computing node of the Lassen cluster at Lawrence Livermore National Laboratory. The node is equipped with IBM Power9 processors (2 sockets × 20 user-level usable cores), 256 GB of main memory, and four NVIDIA Tesla V100 GPUs with 16 GB of device memory each. We build a source-to-source transformation tool in Python that parses `metavariant` annotations and lowers metavariant function definitions, declarations and call sites, tracking the OpenMP context, similarly to the way presented in the examples of the previous section. The compiler used for generating executables from the lowered sources is Clang/LLVM version 13.0.1.

Our experimentation includes three use cases: (1) a use case that avoids nested parallelism; (2) a use case that specializes a metavariant function for concurrent parallel host execution and offloading execution; and (3) a use case that avoids nesting target regions with unspecified behavior.

The example programs with which we experiment invoke a reasonably optimized blocked GEMM kernel in single precision, using square matrix inputs. The kernel is implemented as a metavariant function, specialized depending on the propagated caller's OpenMP context. For each experiment we perform 10 trials of each configuration to present the mean execution time. Any confidence intervals shown correspond to 95% confidence level. We also ensure that experiments run on an exclusively allocated node, using threads pinned to the physical cores of the machine to reduce variability.

4.2 Using Metavariant to Avoid Nested Parallelism

Figure 5 presents this use case in pseudo-code. The `main` function in the driver translation unit (Fig. 5a) is the caller of the `gemm` function, which implements the matrix multiplication in another translation unit. The driver emulates calling `gemm` within a parallel context, masked to execute by the main thread. There are three versions of the `gemm` function: (1) `gemm` is a `metavariant` function (Fig. 5b) that specializes at compile time to `taskloop` execution, when called in a parallel context, or to a `parallel for` work-sharing construct when called in a sequential context; (2) `gemm` is specialized at runtime (Fig. 5c) to use either `taskloop` or `parallel for` depending on the result of the runtime call `omp_in_parallel()` which dynamically detects a parallel context; (3) `gemm` is not specialized (Fig. 5d), nesting instead a parallel region within masked execution that results in sequential execution within the main thread. We omit the metavariant function declaration in the driver (when applicable) since the function prototype is the same as in the metavariant function definition. The runtime mode replicates the functionality of the metavariant mode using a dynamic context selector in a metadirective by checking the result of `omp_in_parallel`. However, this specialization is limited to the context information available through runtime calls, in contrast to the much more widely available context specification available through OpenMP

```
int main() {
// Init
#pragma omp parallel
  {
  #pragma omp masked
    gemm(···)
  }
}
```

(a) Caller

```
#pragma omp metavariant \
 match(construct={parallel})
void gemm(···) {
#pragma omp metadirective \
  when(construct={parallel} : \
    taskloop collapse(2)) \
  otherwise (parallel for collapse(2))
  for(···) {}
}
```

(b) Metavariant

```
void gemm(···) {
  if (omp_in_parallel())
    #pragma omp taskloop \
       collapse(2)
    for(···) {}
  else
    #pragma omp parallel for \
      collapse(2)
    for(···) {}
}
```

(c) Runtime specialization

```
void gemm(···) {
  #pragma omp parallel for \
    collapse(2)
  for(···) {}
}
```

(d) Nested parallel region (no specialization)

Fig. 5. A use case that avoids nested parallelism

traits. Further, dynamic context selectors require extra conditionals generated at compile time to select the specialization variant, which add to the overhead of the runtime call. Metavariant avoids those overheads while also providing a more powerful specialization mechanism through context forwarding that propagates context to both metadirectives and calls to other metavariant functions.

Figure 6 shows the execution time of the GEMM kernel for different matrix dimensions in the different modes of execution. The metavariant mode, denoted as *meta*, and the runtime mode, denoted as *runtime*, both avoid the nested parallel region. The serialized, nested parallel region execution mode, denoted as *nometa*, is the slowest. The performance of *meta* and *runtime* is comparable, with *meta* being slightly faster by avoiding the runtime call to `omp_in_parallel()`, around 5% on average.

4.3 Using Metavariant for Concurrent CPU or GPU Execution

Figure 7 shows the pseudo-code of the second case where the program performs a batch of matrix multiplications of different sizes. The main program, in Fig. 7a, processes the batch within a parallel work-sharing loop. Based on a threshold of the matrix sizes, the loop issues a matrix multiplication to execute on the

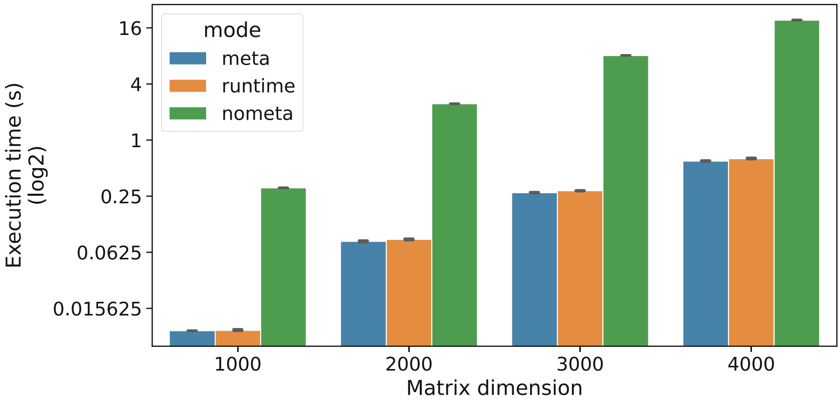


Fig. 6. Execution time for GEMM over various matrix dimensions using the metavariant or runtime to avoid serialized nested parallel regions

CPU or to the GPU, through the metavariant function `gemm` specialization for different contexts (shown in Fig. 7b) including a parallel context, a target context, or defaulting to a parallel work-sharing construct (does not apply in this use case). Adding a `nowait` clause to the target region of Fig. 7a could enable more concurrency through asynchronous execution. However, we did not observe a noticeable performance difference when we used it.

Besides concurrent execution on both CPU and GPU, we experiment with executing on the GPU only, setting the threshold to 0, or the CPU only (by setting the threshold above the largest dimension in the batch, i.e., 4000. Figure 8 shows the results. CPU-only execution is the slowest, as expected, since for larger matrices the speedup from CPU parallelization reaches its limits. GPU-only execution is much faster, about 5×. Concurrent CPU and GPU execution performs even better, by about 3%, compared to GPU-only execution by utilizing both processing elements.

4.4 Using Metavariant to Avoid Nested Target Regions

Figure 9 presents a use case in which the metavariant is used to avoid nesting target regions. Nesting a target construct within a target region, without providing the `ancestor` modifier for reverse offloading to the host, results in unspecified behavior [4]. The metavariant definition of `gemm()` avoids this issue, by specializing to use the `distribute` construct, within a target region, otherwise defaulting to using a target construct for offloading.

We experiment by running the GEMM kernel for various matrix dimensions, using either the target context specialization by executing within a target region, or by using the default execution through a target construct. Figure 10 shows the resulting execution times. Execution through the target context specialization is denoted as *target*, while execution outside a target region using a target construct is denoted as *default*. Execution times are not significantly different.

```
int main() {
 int size[BATCH_SIZE] = {
   100, 200, 400, 800,
   1000, 2000, 3000, 4000 };
 #pragma omp parallel for
 for(i=0; i<BATCH_SIZE; ++i) {
   int N = size[i];
   if (N < /*THRESHOLD=*/1000)
      // CPU taskloop variant
      gemm(batch[i])
    else
    #pragma omp target teams \
      map(to:A[i][0:N*N]) \
      map(to:B[i][0:N*N) \
      map(tofrom:C[i][0:N*N])
      // GPU distribute variant
      gemm(batch[i]);
  }
}
```

(a) Caller

```
#pragma omp metavariant \
  match(construct={parallel}) \
  match(construct={target})
void gemm(···) {
  #pragma omp metadirective \
    when(construct={parallel}:\
      taskloop collapse(2)) \
    when(construct={target}: \
      distribute collapse(2)) \
    otherwise (parallel for \
      collapse(2))
  for(···) {}
}
```

(b) Metavariant

Fig. 7. A use case of concurrent CPU and GPU execution

However, the combined construct of *default* shows higher performance as the matrix dimension grows, ranging between 10% to 18%, compared to the nested `distribute` construct within the target context. Observing the generated LLVM IR, we notice more aggressive optimization in the combined construct case. We plan to investigate further compilation differences due to specialization and especially how to integrate context information in the metavariant specification for additional inter-procedural optimization during compilation.

5 Related Work

Many related research efforts extend OpenMP for better productivity, portability, and performance, especially in the context of programming heterogeneous architectures. We only name a few examples in this section.

A popular approach to achieving portable performance of OpenMP is through autotuning. One early study [1] leveraged source code outlining to extract OpenMP loops from large-scale scientific applications and subsequently enable the tuning of different OpenMP execution parameters. Similarly, Sreenivasan et al. [7] proposed a lightweight autotuner of OpenMP pragmas to optimize OpenMP execution parameters such as scheduling policies, chunk sizes, and thread counts.

To study the benefits of specialization mechanisms introduced in OpenMP 5.0, Pennycook et al. [6] used the miniMD benchmark from the Mantevo suite to investigate how `metadirective` and `declare variant` may impact real-life

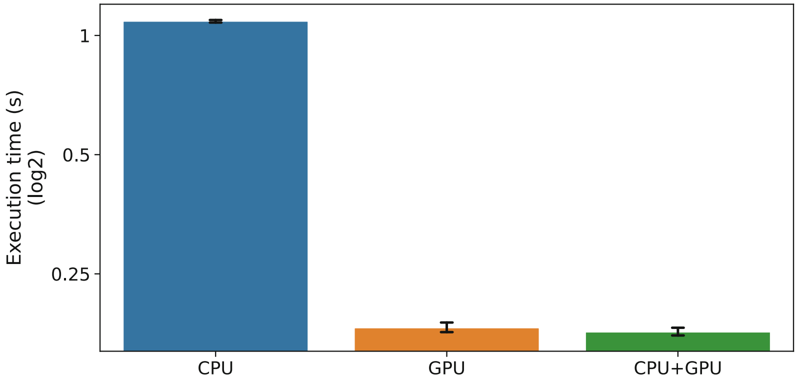


Fig. 8. Execution time of the matrix multiplication batch.

```
int main() {
 #pragma omp target teams \
   map(to: A[0:N*N]) \
   map(to: B[0:N*N) \
   map(tofrom: C[0:N*N])
   gemm(···);
}
```

(a) Calling metavariant function within a target region

```
#pragma omp metavariant \
  match(construct={target})
void gemm(···) {
#pragma omp metadirective \
 when(construct={target} : \
 distribute collapse(2)) \
 otherwise (target teams distribute \
 collapse(2) map(tofrom: C[0:M*N])) \
 map(to: A[0:M*K], B[0:K*N])
  for(···) {}
}
```

(b) Metavariant

Fig. 9. A use case for avoiding nested `target` regions

codes. They reported that these features allowed a more compact source code form to express code variants, resulting in a performance portability of 59.35% across CPU and GPU hardware. While `metadirective` allows user-guided runtime adaptation as proposed by Yan et al. [8], having users define portable conditions across different hardware platforms is impractical. To address this limitation, Liao et al. [2] proposed a `declare adaptation` directive that enables automatic model-driven runtime adaptation, by integrating machine learning techniques into OpenMP compiler and runtime systems. The directive allows programmers to express semantics related to the desired type of machine learning model, the input parameters to the model and the ranges of the parameters.

In order to avoid writing different directives for different devices, Ozen and Wolfe [5] proposed a descriptive model with a new OpenMP loop directive to demonstrate how a compiler implementation could automatically decide target-specific parallelization for multiple devices based on a single directive. In their work, they exploited the parallelism semantics associated with the loop construct

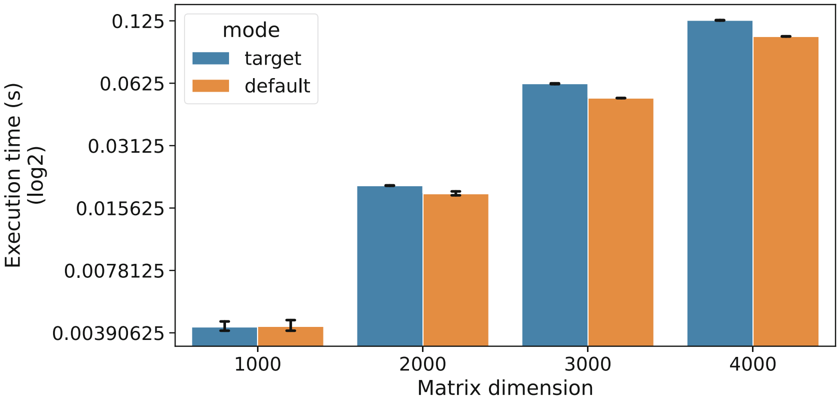


Fig. 10. Executing on GPU directly or using target context specialization

and also used dependence analysis to discover more parallelism. Their evaluation showed that 60% fewer directives were required for the SPEC ACCEL benchmark suite while yielding competitive performance compared to other compilers.

While it is not a significant point in that paper, the solution proposed is heavily influenced by OpenACC, which provides the `acc routine` directive to provide context information on parallelism across translation units. That feature is highly related to ours, but differs in that the `acc routine` directive states the levels of parallelism that a function intends to consume where metavariant states the set of possible contexts from which a function may be called and, thus, for which it should be specialized. The context specification of metavariant is a superset of the `acc routine` parallelism-only context specification. Also, metavariant supports specialization through metadirectives or calls to other metavariant functions in the metavariant function's body by propagating context, whereas `acc routine` designates functions for device compilation using parallelism-level information for error checking. Context matching is explicit in metavariant, by contrast, `acc routine` defines an implicit rule set to determine whether parallelism levels are composable. Interesting future work for the metavariant specification includes exploring specification-based implicit rule sets both for automated transformations and error checking.

6 Conclusion

In this paper, we propose to extend OpenMP to support automated function specialization across translation units by providing the `metavariant` directive. The `metavariant` directive explicitly communicates information about calling context and specializations between call sites and function definitions residing in different source files. Using a source-to-source prototype tool and a set of use cases, we have shown the feasibility and benefits of this extension.

In the future, we plan to extend other directives (such as `assume` and `requires` directives) to support optimization across translation units further. We will explore fully automated generation of function specialization without relying on `metadirective`. We will also expand the evaluation by using a production quality implementation based on LLVM and comparing it to alternative approaches such as LTO and runtime specialization, combined with more use cases running on more platforms. Further, we will propose this extension for upcoming versions of the OpenMP specification by drafting a more rigorous functional specification and expanding the use cases using our robust LLVM implementation.

Acknowledgments. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-837685), partially supported by the LLNL-LDRD Program under Project No. 21-ERD-018 and by the U.S. Dept. of Energy, Office of Science, Advanced Scientific Computing Program (ASCR SC-21).

References

1. Liao, C., Quinlan, D.J., Vuduc, R., Panas, T.: Effective source-to-source outlining to support whole program empirical optimization. In: Gao, G.R., Pollock, L.L., Cavazos, J., Li, X. (eds.) LCPC 2009. LNCS, vol. 5898, pp. 308–322. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13374-9_21
2. Liao, C., et al.: Extending OpenMP for machine learning-driven adaptation. In: Bhalachandra, S., Daley, C., Melesse Vergara, V. (eds.) WACCPD 2021. LNPSE, vol. 13194, pp. 49–69. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-97759-7_3
3. OpenMP Architecture Review Board: OpenMP Application Programming Interface 5.0, November 2018. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf
4. OpenMP Architecture Review Board: OpenMP Application Programming Interface 5.1, November 2020. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf
5. Ozen, G., Wolfe, M.: Performant portable OpenMP. In: Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction, pp. 156–168 (2022)
6. Pennycook, S.J., Sewall, J.D., Hammond, J.R.: Evaluating the impact of proposed OpenMP 5.0 features on performance, portability and productivity. In: 2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), pp. 37–46, November 2018. https://doi.org/10.1109/P3HPC.2018.00007
7. Sreenivasan, V., Javali, R., Hall, M., Balaprakash, P., Scogland, T.R.W., de Supinski, B.R.: A framework for enabling OpenMP autotuning. In: Fan, X., de Supinski, B.R., Sinnen, O., Giacaman, N. (eds.) IWOMP 2019. LNCS, vol. 11718, pp. 50–60. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-28596-8_4
8. Yan, Y., Wang, A., Liao, C., Scogland, T.R.W., de Supinski, B.R.: Extending OpenMP Metadirective semantics for runtime adaptation. In: Fan, X., de Supinski, B.R., Sinnen, O., Giacaman, N. (eds.) IWOMP 2019. LNCS, vol. 11718, pp. 201–214. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-28596-8_14

Author Index

Araya-Polo, Mauricio 17

Baldassin, Alexandro 50
Brandner, Julian 94

Carribault, Patrick 3
Chapman, Barbara 17, 145
Ciesko, Jan 35

de Supinski, Bronis R. 159
Doerfert, Johannes 17, 67, 145
Duran, Alejandro 127

Ferat, Manuel 3

Gammelmark, Mathias 81
Gautier, Thierry 3
Georgakoudis, Giorgis 159

Huber, Joseph 145

Indic, Vladimir 111

Karlsson, Sven 81

Liao, Chunhua 159
Lu, Wenbin 17

Malik, Abid M. 17
Mayer, Florian 94
Maza, Marc Moreno 67
Mellor-Crummey, John 111
Meng, Jie 17

Narayanaswamy, Ravi 127

Olivier, Stephen L. 35

Pereira, Romain 3
Philippsen, Michael 94

Raut, Eric 17
Rossi, Giacomo 127
Roussel, Adrien 3
Rydahl, Anton 81

Salamanca, Juan 50
Scogland, Thomas R. W. 159
Servat, Harald 127
Shan, Baodi 17
Steffenel, Luiz-Angelo 3

Talaashrafi, Delaram 67
Tian, Shilei 145
Tramm, John 145