Using Free Scheduling for Programming Graphic Cards

Wlodzimierz Bielecki and Marek Palkowski

Faculty of Computer Science, West Pomeranian University of Technology,
70210, Zolnierska 49, Szczecin, Poland
{bielecki,mpalkowski}@wi.zut.edu.pl
http://kio.wi.zut.edu.pl/

Abstract. An approach is presented permitting us to build free scheduling for statement instances of affine loops. Under the free schedule, loop statement instances are executed as soon as their operands are available. To describe and implement the approach, the dependence analysis by Pugh and Wonnacott was chosen where dependences are found in the form of tuple relations. The proposed algorithm has been implemented and verified by means of the Omega project software. Results of experiments with the NAS benchmark suite are discussed. Speed-up and efficiency of parallel code produced by means of the approach are studied. Problems to be resolved in order to enhance the power of the presented technique are outlined.

Keywords: fine-grained parallelism, free scheduling, parameterized affine loops, NVIDIA cards.

1 Introduction

Microprocessors with multiple execution cores on a single chip are typical computation platforms now. The lack of automated tools permitting for exposing parallelism for such systems decreases the productivity of programmers and increases the time and cost of producing a parallel program.

Because most computations are contained in program loops, automatic extraction of parallelism from loops is extremely important for multi-core systems, allowing us to produce parallel code from existing sequential applications and to create multiple threads that can be easily scheduled to achieve high program performance.

Given a loop, a schedule of loop statement instances is a function that assigns a time of execution to each loop statement instance preserving all dependences in this loop. There have been developed numerous approaches to form loop statement instance scheduling, for example [7,9,12,13,18].

Under the free schedule, loop statement instances are executed as soon as their operands are available that permits us to extract all fine-grained parallelism available in the loop, but well-known techniques based on linear or affine schedules do not guarantee finding free scheduling for non-uniform loops.

In this paper, we present a novel technique that permits for building free scheduling for both uniform and non-uniform loops. It is based on forming the

R. Keller et al. (Eds.): Facing Multicore-Challenge II 2011, LNCS 7174, pp. 72–83, 2012.

exact transitive closure of a dependence relation describing all dependences in a loop. Experimental results with the NAS benchmarks suite demonstrate that the approach can be successfully applied to produce parallel programs for NVIDIA graphic cards.

2 Background

A discussed algorithm deals with *static-control loop nests*, where lower and upper bounds as well as conditionals and array subscripts are affine functions of symbolic parameters and surrounding loop indices. A *statement instance* $s(I)$ is a particular execution of a statement s of the loop for some loop iteration I.

Two statement instances $s_1(I)$ and $s_2(I)$ are dependent if both access the same memory location and if at least one access is a write. Provided that $s_1(I)$ is executed before $s_2(I)$, $s_1(I)$ and $s_2(I)$ are called the *source* and *destination* of the dependence, respectively. The sequential execution ordering of statement instances, denoted as $s_1(I) \prec s_2(J)$, is induced by the lexicographic ordering of iteration vectors and the textual ordering of statements when the instances share the same iteration vector.

Definition [11]. The free schedule is the function that assigns statement instances (for execution) as soon as their operands are available, that is, it is mapping $\sigma{:}LD \rightarrow \mathbb{Z}$ such that

$$\sigma(p) = \begin{cases} 0 \; if \; there \; is \; no \; p' \in LD \; s.t. \; p' \prec p \\ 1 + max(\sigma(p')), p' \in LD, p' \prec p \end{cases}$$

i.e., it is the fastest schedule, where p, p' are loop statement instances, LD is the loop domain.

The approach to find free scheduling, presented in this paper, requires an exact representation of dependences. To describe the approach and carry out experiments, we have chosen the dependence analysis proposed by Pugh and Wonnacott [25] where dependences are represented with dependence relations. This analysis is implemented in Petit [14] which returns a set of dependence relations describing all dependences in a loop. A dependence relation is a tuple relation of the form $\{[input\ list] \rightarrow [output\ list] : constraints\}$, where *input list* and *output list* are the lists of variables used to describe input and output tuples and *constraints* is a Presburger formula describing the constraints imposed upon *input list* and *output list* [20].

The general form of a dependence relation is as follows [25]:

$$\{[s_1, ..., s_k] \rightarrow [t_1, ..., t_{k'}] | \bigvee_{i=1}^{n} \exists \alpha_{i1}, ..., \alpha_{im_i} s.t. F_i\},$$

where F_i, i=1, 2, ..., n, are conjunctions of affine equalities and inequalities on the input variables $s_1, ..., s_k$, the output variables $t_1, ..., t_{k'}$, the existentially quantified variables $\alpha_{i1}, ..., \alpha_{im_i}$ and symbolic constants, k, k', m, n are integers.

An *ultimate dependence source* (resp. *destination*) is a source (resp. destination) that is not the destination (resp. source) of another dependence. A set,

UDS, comprising all ultimate dependence sources can be found as domain(R)-range(R), where R represents all dependences in a loop.

Given dependence relations R_1, R_2,..., R_m, our approach requires first preprocessing these relations according to the procedure presented in [2]. Preprocessing makes the sizes of input and output tuples of dependence relations to be the same as well as inserts identifiers of loop statements in the last position of input and output tuples (this permits for applying the union, composition, and difference operations to relations describing dependences).

Positive transitive closure for a given relation R, R^+, is defined as follows $R^+ = \{[e] \rightarrow [e'] : \ e \rightarrow e' \in R \wedge \exists(e'' : \ e \rightarrow e'' \in R^+ \wedge e'' \rightarrow e' \in R)\}$.

Transitive closure, R^*, is defined as follows: $R^* = R^+ \cup I$, where I the identity relation. Details concerning these operations can be found in [16].

3 Finding Free Scheduling for Parameterized Loops

The idea of the algorithm presented in this section is as follows. Given preprocessed relations R_1, R_2, ..., R_m, we firstly calculate $R = \bigcup_{i=1}^{m} R_i$. Next we create the relation $R^{'}$ by inserting variables k and and k+1 into the first position of the input and output tuples of relation R; variable k is to present the time of a partition (a set of statement instances to be executed at time k). Next, we calculate the transitive closure of relation $R^{'}$, $R^{'}*$, and form the following relation

$$FS = \{[X] \rightarrow [k, \ Y] : X \in UDS(R) \wedge (k, \ Y) \in \text{Range}((R^{'})^{*}\backslash\{[0, \ X]\}) \wedge \neg(\exists \ k^{'}{>}k \text{ s.t. } (k^{'}, \ Y) \in \text{Range}(R^{'})^{+}\backslash\{[0, \ X]\})\},$$

where $(R^{'})^{*}\backslash\{[0, X]\}$ means that the domain of relation $R^{'}*$ is restricted to the set including ultimate dependences sources only (elements of this set belong to the first time partition); the constraint $\neg(\exists \ k^{'}{>}k$ s.t. $(k^{'}, \ Y) \in \text{Range}(R^{'})^{+}\backslash\{[0, X]\})$ guarantees that partition k includes only those statement instances whose operands are available, i.e., each statement instance will belong to one time partition only.

It is worth to note that the first element of the tuple representing the set Range(FS) points out the time of a partition while the last element of that exposes what is the statement whose instance(iteration) is defined by the tuple elements 2 to n-1, where n is the number of the tuple elements of a preprocessed relation. Taking the above consideration into account and provided that the constraints of relation FS are affine, the set Range(FS) is used to generate parallel code applying any well-known technique to scan its elements in the lexicographic order, for example the techniques presented in papers[1,26].

The outermost sequential loop of such code scans values of variable k (representing the time of partitions) while inner parallel loops scan independent instances of partition k. Techniques aimed at calculating the transitive closure of dependence relations are presented in papers [4,5,16] and are out of the scope of this paper.

Finally, we expose independent statement instances, that is, those that do not belong to any dependence and generate code enumerating them. According to the free schedule, they are to be executed at time k=0.

Below we present the algorithm that realizes the presented above idea in a formal way.

Algorithm: Finding free scheduling for statement instances of a parameterized loop
Input: Preprocessed relation R describing all dependences in a loop.
Output: Code representing free scheduling.
Method:

1. Transform relation R of the form $\{[X] \rightarrow [Y] : \mathit{constraints}\}$, where X and Y are vectors representing the input and output tuple variables, respectively, to relation $R^{'}$ of the form $\{[k,\ X] \rightarrow [k{+}1,\ Y] : \mathit{constraints} \wedge k \geqslant 0\}$.
2. Calculate $(R^{'})^{+}$ using any of known techniques, for example those presented in papers [4,5,16].
3. Form the following relation FS:
   $FS = \{[X] \rightarrow [k,\ Y] : [X] \in UDS(R) \wedge [k,\ Y] \in \mathrm{Range}((R^{'})^{*}\backslash\{[0,\ X]\}) \wedge \neg(\exists\ k^{'}{>}k \text{ s.t. } [k^{'},\ Y] \in \mathrm{Range}(R^{'})^{+}\backslash\{[0,\ X]\})\}$.
4. Generate code scanning elements of the set Range(FS). For this purpose, apply any well-known algorithm, for example that published in [14]. The outermost sequential loops of this code scan time partitions while the inner parallel loops scan instances to be executed in a particular partition.
5. Find set, IND, containing independent statement instances:
   $IND = IS$ - ($\mathrm{domain}(R) \cup \mathrm{range}(R)$),
   where IS represents the union of preprocessed iterations sets of all loop statements. Generate code scanning elements of set IND. For this purpose, apply any well-known algorithm, for example that published in [14]. This code is to be executed at time k=0.

Let us illustrate the presented algorithm by means of the following imperfectly nested parameterized loop.

Example 1

```
for(i=1; i<=n; i++){
  a[i][0] = 1;   //s1
  for(j=1; j<=n; j++){
     a[i][j] = a[i-1][j] + a[i][j-1];    //s2
  }
}
```

There are the three dependence relations returned by Petit
R1 ={[i,-1,1] → [i,1,2] : 1 ⩽ i ⩽ n};
R2 ={[i,j,2] → [i+1,j,2] : 1 ⩽ i < n && 1 ⩽ j ⩽ n};
R3 ={[i,j,2] → [i,j+1,2] : 1 ⩽ i ⩽ n && 1 ⩽ j < n}.

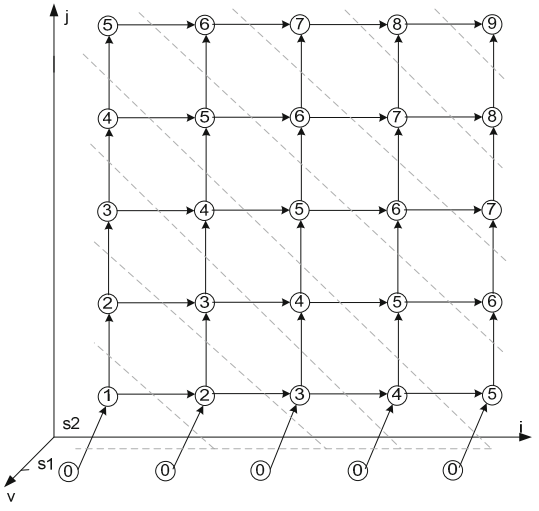

Fig. 1. The free schedule for Example 1 when n=5. The solid lines represent dependences, the dotted lines represent synchronization barriers between time partitions.

Figure 1 presents the free schedule for the loop of Example 1 when n=5.

Applying the presented algorithm, we get the following results being produced by means of the Omega calculator.

1. $R^{'} = \{[k,i,-1,1] \rightarrow [k+1,i,1,2] : 1 \leqslant i \leqslant n \;\&\&\; 0 \leqslant k\} \cup \{[k,i,j,2] \rightarrow [k+1,i+1,j,2] : 1 \leqslant i < n \;\&\&\; 1 \leqslant j \leqslant n \;\&\&\; 0 \leqslant k\} \cup \{[k,i,j,2] \rightarrow [k+1,i,j+1,2] : 1 \leqslant i \leqslant n \;\&\&\; 1 \leqslant j < n \;\&\&\; 0 \leqslant k\}$.
2. $R^{'+} = \{[k,i,j,2] \rightarrow [k',i',i-k+j-i'+k',2] : 1 \leqslant i \leqslant i' \leqslant n \;\&\&\; 0 \leqslant k < k' \;\&\&\; 1 \leqslant j \;\&\&\; k+i' \leqslant i+k' \;\&\&\; i+j+k' \leqslant n+k+i'\} \cup \{[k,i,-1,1] \rightarrow [k',i',i-k+k'-i',2] : 1 \leqslant i \leqslant i' \leqslant n \;\&\&\; k+i' < i+k' \;\&\&\; 0 \leqslant k \;\&\&\; i+k' \leqslant n+k+i'\}$.
3. $FS = \{[1,-1,1] \rightarrow [k,i',k-i'+1,2] : 1 \leqslant i' \leqslant k, n \;\&\&\; k < n+i'\} \cup \{[i,-1,1] \rightarrow [0,i,-1,1] : 1 \leqslant i \leqslant n\}$.
4. $Range(FS) = \{[k,i,k-i+1,2]: 1 \leqslant i \leqslant k, n \;\&\&\; k < n+i\} \cup \{[0,i,-1,1]: 1 \leqslant i \leqslant n\}$.
   The loop scannig elements of the set Range(FS) for $k \geq 0$ and being produced by the codegen function of the Omega library is as follows.

```
for(t2 = 1; t2 <= n; t2++) {    // parallel loop
  a[t2][0] = 1;   // s1(0,t2,-1,1);
}
for(t1 = 1; t1 <= 2*n-1; t1++) {
  for(t2 = max(-n+t1+1,1); t2 <= min(n,t1); t2++) {  //parallel loop
     a[t2][t1-t2+1] = a[t2-1][t1-t2+1] + a[t2][t1-t2];
                                          // s1(t1,t2,t1-t2+1,2);
}}
```

5. $IND = \varnothing$. There is no independent statements in the loop.

The pseudocode above was manually transformed to the parallel code for NVIDIA cards presented below. The main function of this code runs kernels of parallel loops. The value of variable *n_blocks* represents the number of threads that execute a single block of independent loop statement instances, i.e., the number of engaged CUDA cores. The value of variable *idx* defines the identifier of a block; the values of variables *lb* and *ub* indicate the lower and upper bounds of the parallel loop, respectively; variable *packet* is to represent the number of iterations in a block.

```
//Kernel definitions
__global__ void loop1_gpu(float (*a)[n])
{
   int idx = blockIdx.x, t2;
   int packet = (int)ceil(n / blockDim.x);
   int lb = idx*packet+1;
   int ub = ((idx+1)*packet < n) ? (idx+1)*packet : n;

   for(int t2 = lb; t2 <= ub; t2++)
     a[t2][0] = 1;

}

__global__ void loop2_gpu(float (*a)[n], t1)
{
   int idx = blockIdx.x, t2;
   int packet = (int)ceil((max(-n+t1+1,1) - min(n,t1)) / blockDim.x);
   int lb = idx*packet+max(-n+t1+1,1);
   int ub = ((idx+1)*packet < min(n,t1)) ? (idx+1)*packet : min(n,t1);

   for(int t2 = lb; t2 <= ub; t2++)
        a[t2][t1-t2+1] = a[t2-1][t1-t2+1] + a[t2][t1-t2];
}

int main(int argc, char * argv[]){
...
  int threads_per_block = 1;
  int n_blocks = atoi(argv[1]);  // number of CUDA cores

  // Kernel invocation
  loop1_gpu <<< n_blocks, threads_per_block>>> ((float(*)[n])d_A);
  for(t1 = 1; t1 <= 2*n-1; t1++)  {
    loop2_gpu <<< n_blocks, threads_per_block>>> ((float(*)[n])d_A, t1);
...
}
```

4 Experimental Results

The presented algorithm was implemented by us in a tool by means of the Omega library. It generates C-like pseudo-code scanning loop statement instances

according to free scheduling. Using this tool, we have experimented with loops of the NAS 3.2 benchmark suite [23].

The NAS Parallel Benchmarks (NPB) have been developed at the NASA Ames Research Centre to study performance of parallel supercomputers. Benchmarks are derived from computational fluid dynamics and include [23]:

- EP - An *embarrassingly parallel* kernel, which evaluates an integral by means of pseudorandom trials.
- MG - Simplified multigird kernel, which solves a 3D Poisson PDE.
- CG - A conjugate gradient method is used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix.
- FT - A 3-D partial differential equation solution using FFTs.
- IS - A large integer sort.
- LU - A regular-sparse, block (5x5) lower and upper triangular system solution.
- SP - Solution of multiple, independent systems of non diagonally dominant, scalar, pentadiagonal equations.
- BT - Solution of multiple, independent systems of non diagonally dominant, block tridiagonal equations with a (5x5) block size.
- UA - Unstructured Adaptive, a new kernel solving scientific problems featuring irregular, dynamic memory accesses.
- DC, DT - Data Cube operator and Data Transfer benchmarks.

From 431 loops of the NAS benchmark suite, Petit is able to carry out a dependence analysis for 257 loops only, and it discovers dependences in 133 loops only. For 133 loops qualified for experiments, the tool is able to calculate the transitive closure of dependence relations for 96 loops. Scheduling and generating code are possible for 65 ones (for the rest 31 loops the algorithm fails to produce relation FS due to the time out limitation - maximum 10 seconds to produce set FS). For 19 from those 65 loops, the algorithm does not expose any parallelism (there exists a single statement instance for each time partition). Therefore parallelism is extracted for 46 loops only.

To assess the efficiency of code produced by the proposed algorithm, the following criteria were taken into account for choosing NAS loops: (i) a loop must be computatively heavy (there are many NAS benchmarks with constant upper bounds of loop indices, hence their parallelization is not justified), (ii) code produced by the algorithm must be parallel (there are NAS loops for which there exists a single statement instance for each time partition), (iii) structures of chosen loops must be different (there are many NAS loops of a similar structure). Applying these criteria, we selected the following NAS loops: *FT_auxfnct_2* (Fast Fourier Transform Benchmark), *UA_diffuse_4* and *UA_transfer_4* (both from Unstructured Adaptive benchmark).

Codes, produced for these loops by the presented approach, were manually converted by us to parallel programs to be executed on an NVIDIA graphic card 8800 GTS, 96 CUDA Cores, 1.6 GHz, GDDR3 512 MB (in the same way as it is done for Example 1). For this purpose, we have used the NVIDIA CUDA library [24].

Table 1. Results for NAS benchmarks

Loop	Upper bounds	data transf. time, s.	time (without data transfer time) 1 GPUs	2 GPUs	8 GPUs	96 GPUs
FT_auxfnct_2	N1, N2, N3 = 100	a: 0.00034 s: 0.00626 f: 0.00750 t: 0.01411	0.4632	0.2330	0.0572	0.0088
	N1, N2, N3 = 200	a: 0.00044 s: 0.04988 f: 0.04611 t: 0.09643	3.9193	1.9588	0.4952	0.0872
UA_diffuse_4	N1, N2 = 64; N3, N4 = 10	a: 0.00032 s: 0.00319 f: 0.00167 t: 0.00518	0.1959	0.0979	0.0275	0.0011
	N1,N2 = 128; N3,N4 = 10	a: 0.00035 s: 0.00972 f: 0.00561 t: 0.01567	0.8055	0.3970	0.1064	0.0159
UA_transfer_4	N1, N2 = 1000	a: 0.00032 s: 0.00632 f: 0.00264 t: 0.00928	0.4053	0.2458	0.0717	0.0202
	N1, N2 = 2000	a: 0.00037 s: 0.02510 f: 0.00981 t: 0.03528	1.6222	0.9294	0.2456	0.0476

Table 1 presents results of time measuring (in seconds) for executing the chosen loops on the graphic card. The execution time of a loop consists of the time of data transfer to/from a graphic card and the time of calculations. Experiments were carried out for two different values of the upper bounds of loop indices (see column 2). The time of data transfer (see column 3) comprises the times of [24]: allocation (a), sending data to the graphic card (s), and fetching data memory of the graphic card (f). Column 3 presents also the sum of those times as the time of data transfer (t). It is worth to note that the time of data transfer does not depend on the number of GPU cores [24]. Columns 4-7 show the time of calculations (not including the time of data transfer) for 1, 2, 8, and 96 GPU cores.

Table 2 presents the execution time (the sum of the time of data transfer and the time of calculations), speed-up, and efficiency for different numbers of GPU cores. The results in Table 2 demonstrate that parallel loops formed on the basis of parallel code produced by the algorithm: i) permit for utilizing many GPU cores (up to 96 under our experiments); ii) speed-up increases with increasing the number of GPU cores (up to 96 under our experiments). For loop *UA_diffuse_4*, increasing values of *N1* and *N2* leads to decreasing speed-up for 96 cores. This can be explained by increasing the number of synchronization evens and not

Table 2. Time, speed-up, and efficiency

Loop	Upper bounds	1 GPU	2 GPUs			8 GPUs			96 GPUs		
		time,s	time,s	S	E	time,s	S	E	time,s	S	E
FT_auxfnct_2	N1,N2,N3= 100	0.477	0.247	1.93	0.97	0.118	6.69	0.84	0.023	20.84	0.22
	N1,N2,N3= 200	4.016	2.055	1.95	0.98	0.592	6.79	0.85	0.184	21.86	0.23
UA_diffuse_4	N1,N2=64; N3,N4=10	0.201	0.103	1.95	0.98	0.033	6.15	0.77	0.006	31.87	0.33
	N1,N2=128; N3,N4=10	0.821	0.413	1.99	0.99	0.122	6.72	0.84	0.032	25.99	0.27
UA_transfer_4	N1,N2= 1000	0.415	0.255	1.63	0.81	0.081	5.12	0.64	0.030	14.04	0.15
	N1,N2= 2000	1.657	0.965	1.72	0.86	0.281	5.90	0.74	0.083	20.01	0.21

enough increasing the work running by each core. However, with increasing the number of cores, speed-up increases for the same values of the loop upper bounds.

To evaluate the time complexity of the algorithm presented in this paper, we measured the time that takes the tool, implementing the presented algorithm, from the beginning of a dependence analysis to the end of pseudo code generation on the machine with the following features: Intel Core 2 Duo 2.34 Ghz, 2 GB RAM, Ubuntu Linux.

Table 3. Time complexity of the algorithm

loop	dep. analysis, s	R'^{+} and Range(FS), s	Final code, s	Total time, s
FT_auxfnct_2	0.01	0.07	0.01	0.09
UA_diffuse_4	0.01	0.05	0.01	0.07
UA_transfer_4	0.01	0.03	0.01	0.05

In our implementation, Petit was used as the dependence analyser. Calculating R'^{+} and the set Range(FS) was carried out by a function written by us, and final code generation was realized on the basis of the Omega library *codegen* function. Table 3 presents the results for the three examined NAS loops. The first column contains the name of a loop while the remaining columns expose time measurement results (in seconds): of dependence analysis, calculating R'^{+} and Range(FS), final code generation, and the total time.

Table 4 presents the comparison of time measurements for an Intel Core 2 Duo 2.34 Ghz computer (1 and 2 CPUs) and an NVIDIA 8800 GTS graphic card (1 and 96 CUDA cores). Loop *FT_auxfnct_2* is executed much faster on 96 CUDA cores than on two CPU cores. For the next two loops, free-scheduling introduce high volume of synchronization on two CPU cores that results in negative speed-up, while for the graphic card the speed-up is about 20 for 96 cores.

Based on the presented results, we can conclude that the presented algorithm can be successfully applied for producing parallel programs for many NAS benchmarks to be executed on graphic cards.

Table 4. CPU and GPU times comparison

Loop	Upper bounds	1 CPU	2 CPUs	1 GPU	96 GPUs
FT_auxfnct_2	N1,N2,N3= 100	0.174	0.091	0.477	0.023
	N1,N2,N3= 200	2.081	1.543	4.016	0.184
UA_diffuse_4	N1,N2=64; N3,N4=10	0.010	0.027	0.201	0.006
	N1,N2=128; N3,N4=10	0.038	0.046	0.821	0.032
UA_transfer_4	N1,N2= 1000	0.028	0.050	0.415	0.030
	N1,N2= 2000	0.071	0.101	1.657	0.083

5 Related Work

The approaches published in [8,17,22,21] build an explicit graph of a subset of the iteration space, with each node representing the instance of a statement. Free scheduling can be found by searching the graph or using the transitive closure of the graph, but dependences are restricted to uniform ones and the problem regarding boundary cases exists.

For the case of polyhedral approximations of dependences (including direction vectors), Darte and Vivien's algorithm is optimal but it does not guarantee forming free scheduling for parameterized loops [10].

For affine dependences, the most powerful algorithm for building scheduling is Feautrier's one based on multi-dimensional affine schedules [13]. But as mentioned by Feautrier, it is not optimal for all codes with affine dependences. However, the dimension of the schedules built by Feautrier's algorithm is minimal for each statement of the loop nest [27].

The approaches published in [7,15,19] present different ways of building affine partition mappings, but none of them guarantees producing free scheduling for the general case of loops with affine dependences.

Paper [3] presents a technique permitting for building free scheduling but only for non-parameterized loops.

The approach presented in [6] permits for extracting free scheduling within each synchronization-free slice but it does not produce free scheduling for all loop statement instances.

6 Conclusion and Outlook

In this paper, we presented the algorithm that permits us to build free scheduling for statement instances of parameterized arbitrarily nested loops. The necessary condition to apply it is the possibility of the calculation of the exact transitive closure of a relation describing all dependences in a loop.

There are tasks to be resolved in the future to strengthen the power of the presented algorithm and justify its application for parallelizing real-life codes: 1) developing algorithms and corresponding implementations permitting for automatic NVIDIA code generation from pseudo code produced by the presented

algorithm; 2) when a free schedule is represented by non-linear forms, techniques should be developed to generate code enumerating statement instances under such a schedule.

References

1. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: PACT 13 IEEE International Conference on Parallel Architecture and Compilation Techniques, Juan-les-Pins, pp. 7–16 (September 2004)
2. Beletska, A., Bielecki, W., Cohen, A., Palkowski, M., Siedlecki, K.: Coarse-grained loop parallelization: Iteration space slicing vs affine transformations. Parallel Computing 37, 479–497 (2011)
3. Beletskyy, V., Siedlecki, K.: Finding Free Schedules for Non-uniform Loops. In: Kosch, H., Böszörményi, L., Hellwagner, H. (eds.) Euro-Par 2003. LNCS, vol. 2790, pp. 297–302. Springer, Heidelberg (2003)
4. Bielecki, W., Klimek, T., Trifunovic, K.: Calculating exact transitive closure for a normalized affine integer tuple relation. Electronic Notes in Discrete Mathematics 33, 7–14 (2009)
5. Wlodzimierz, B., Tomasz, K., Marek, P., Beletska, A.: An Iterative Algorithm of Computing the Transitive Closure of a Union of Parameterized Affine Integer Tuple Relations. In: Wu, W., Daescu, O. (eds.) COCOA 2010, Part I. LNCS, vol. 6508, pp. 104–113. Springer, Heidelberg (2010)
6. Bielecki, W., Palkowski, M.: Extracting Both Affine and Non-linear Synchronization-Free Slices in Program Loops. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2009. LNCS, vol. 6067, pp. 196–205. Springer, Heidelberg (2010)
7. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: Conference on Programming Language Design and Implementation, pp. 101–113. ACM (2008)
8. Chen, D.K.: Compiler optimizations for parallel loops with fine-grained synchronization. Ph.D. thesis, Champaign, IL, USA (1994), uMI Order No. GAX95-12325
9. Darte, A., Robert, Y.: Constructive methods for scheduling uniform loop nests. IEEE Trans. Parallel Distrib. Syst. 5, 814–822 (1994)
10. Darte, A., Vivien, F.: Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. In: Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques, PACT 1996, pp. 281–291. IEEE Computer Society, Washington, DC, USA (1996)
11. Darte, A., Khachiyan, L., Robert, Y.: Linear scheduling is nearly optimal. Parallel Processing Letters 1(2), 73–81 (1991)
12. Feautrier, P.: Some efficient solutions to the affine scheduling problem: I. one-dimensional time. Int. J. Parallel Program. 21(5), 313–348 (1992)
13. Feautrier, P.: Some efficient solutions to the affine scheduling problem: II. multidimensional time. Int. J. Parallel Program. 21(5), 389–420 (1992)
14. Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., Wonnacott, D.: The omega library interface guide. Tech. rep., College Park, MD, USA (1995)
15. Kelly, W., Pugh, W.: A framework for unifying reordering transformations. Tech. rep., Univ. of Maryland Institute for Advanced Computer Studies Report No. UMIACS-TR-92-126.1, College Park, MD, USA (1993)

16. Kelly, W., Pugh, W., Rosser, E., Shpeisman, T.: Transitive closure of infinite graphs and its applications. Int. J. Parallel Programming 24(6), 579–598 (1996)
17. Krothapalli, V., Sadayappan, P.: Removal of redundant dependences in doacross loops with constant dependences. IEEE Transactions on Parallel and Distributed Systems 2, 281–289 (1991)
18. Le Gouëslier d'Argence, P.: Affine scheduling on bounded convex polyhedric domains is asymptotically optimal. Theor. Comput. Sci. 196, 395–415 (1998)
19. Lim, A.W., Cheong, G.I., Lam, M.S.: An affine partitioning algorithm to maximize parallelism and minimize communication. In: Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing, pp. 228–237. ACM Press (1999)
20. Surhone, L.M., Tennoe, M.T., Henssonow, S.F.: Presburger Arithmetic. VDM Verlag Dr. Mueller AG & Co. Kg (2010); ISBN: 6133083557
21. Midkiff, S.P., Padua, D.A.: Compiler algorithms for synchronization. IEEE Transactions on Computers 36, 1485–1495 (1987)
22. Midkiff, S.P., Padua, D.A.: A comparison of four synchronization optimization techniques. In: ICPP (2), pp. 9–16 (1991)
23. NAS: Parallel Benchmarks Suite, Version 3.3 (February 2008), `http://www.nas.nasa.gov`
24. NVIDIA: NVIDIA CUDA C Programming Guide 4.0 (2011), `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf`
25. Pugh, W., Wonnacott, D.: An Exact Method for Analysis of Value-Based Array Data Dependences. In: Banerjee, U., Gelernter, D., Nicolau, A., Padua, D.A. (eds.) LCPC 1993. LNCS, vol. 768, pp. 546–566. Springer, Heidelberg (1994)
26. Verdoolaege, S.: Integer set library - manual. Tech. rep. (2011), `http://www.kotnet.org/~skimo//isl/manual.pdf`
27. Vivien, F.: On the Optimality of Feautrier's Scheduling Algorithm. In: Monien, B., Feldmann, R.L. (eds.) Euro-Par 2002. LNCS, vol. 2400, pp. 299–308. Springer, Heidelberg (2002)