HAL
open science

Putting Automatic Polyhedral Compilation for GPGPU to Work

Soufiane Baghdadi, Armin Grösslinger, Albert Cohen

To cite this version:

Soufiane Baghdadi, Armin Grösslinger, Albert Cohen. Putting Automatic Polyhedral Compilation for GPGPU to Work. Proceedings of the 15th Workshop on Compilers for Parallel Computers (CPC'10), Jul 2010, Vienna, Austria. inria-00551517

HAL Id: inria-00551517

https://inria.hal.science/inria-00551517v1

Submitted on 4 Jan 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Putting Automatic Polyhedral Compilation for GPGPU to Work

Soufiane Baghdadi [1], Armin Größlinger [2,1], and Albert Cohen [1]

[1] INRIA Saclay and LRI, Paris-Sud 11 University, France
{soufiane.baghdadi,albert.cohen}@inria.fr
[2] Universität Passau, Fakultät für Informatik und Mathematik
armin.groesslinger@uni-passau.de

Abstract. Automatic parallelization is becoming more important as parallelism becomes ubiquitous. The first step for achieving automation is to develop a theoretical foundation, for example, the polyhedron model. The second step is to implement the algorithms studied in the theoretical framework and getting them to work in a compiler that can be used to parallelize real codes.
The polyhedral model is a well-established theoretical foundation for parallelizing codes with static control. In this paper, we present, from a practical point of view, the challenges to solve for getting polyhedral compilation for GPUs to work. We choose the Polyhedral Compiler Collection (PoCC) as compiler infrastructure and target CUDA as the target platform; we plan to support OpenCL in the future.

1 Introduction

In recent years, graphics processing units (GPUs) have prominently entered the parallel computing scene because they offer higher computing power than current multicore CPUs and some changes in their design have made them suitable for general-purpose computing (GPGPU computing). The peculiarities of their architecture have given rise to the new programming model *single instruction multiple threads* (SIMT, a variation of the well-known SIMD model), new languages (CUDA, OpenCL) and new tools. Research in automatic parallelization has long promised fully automatic transformation of sequential input programs to efficient parallel code. One such direction of research is the well-known polyhedron model. The model is supported by a broad theory and recent advances address the characteristics of GPUs, especially the SIMT programming model and memory hierarchies.

In this paper, we present the practical aspect of putting automatic polyhedral compilation for GPGPU computing to work. We discuss the challenges in the particular context of the automatically parallelizing compiler called the *Polyhedral Compiler Collection* (PoCC):
http://www-roc.inria.fr/ pouchet/software/pocc

PoCC has been designed as a source-to-source compiler for parallelizing static control parts (SCoPs) for multicore CPUs using OpenMP. We add modules to PoCC for dealing with GPU memory management and code generation for CUDA, each module focusing on its specific task. Both aspects, memory management and code generation,

require problems to be solved which do not occur when targeting CPUs. In addition, we have to take care that the transformation that is applied to the original code produces parallelism suitable for GPUs. In CUDA, there are two levels of parallelism: blocks and threads. Each block consists of several hundred threads that can synchronize among each other; several blocks (tens to hundreds) execute independently of each other. This fits with space-time mapping followed by tiling as this generates the right kind of two-level parallelism: outer sequential loops (time tiles) and parallel loops (space tiles) for the tiles (supersteps) and inner loops (so-called point loops) for each operation inside a tile.

In memory management, we have to deal with three levels of memory. Data has to be transferred between the host memory and the main memory of the GPU and, for efficiency, between the main memory of the GPU and the much faster scratchpad memories of the GPU's (multi)processors. We add statements (with suitable iteration domains) for memory transfers between host, GPU main memory and scratchpad memory. In addition, we have to compute the size of the memory transfers (number and size of elements) and construct a suitable layout of all the transferred data to make do with one transfer call between host and GPU for the elements of several arrays, for example.

In code generation, the first step is to apply a polyhedral code generator (e.g., CLooG). The loop nest generated by it is directly suitable for an execution using OpenMP, but for GPUs, we have to do further processing. First, we have to modify memory accesses in the computation statements to fit the chosen memory region and data layout. Second, dealing with the CUDA language, we extract each GPU part G of the loop nests (i.e., the point loops) and replace it by an invocation of a (newly synthesized) kernel function K; G becomes the body of K. To be precise, the invocation of K also does away with the space tile loops surrounding it as the iterations of space tile loops become the blocks of the corresponding kernel invocation. Third, we modify the parallel point loops in each kernel to distribute their iterations among the threads of a block according to the SIMT model.

In this paper, we detail on the different challenges mentioned and how we solve them to achieve fully automatic parallelization for GPGPU computing using the polyhedron model. In Section 2, we introduce the polyhedron model and the tool chain we use in our implementation. We discuss the specific challenges to address when auto-parallelizing for GPGPU in Section 3. We discuss the most closely related work and position the originality of our approach in Section 4. We then detail on our implementation in Section 5. Section 6 concludes the paper. Due to technical problems (cf. Section 5.3) we cannot present experimental results at this time.

2 The Polyhedron Model

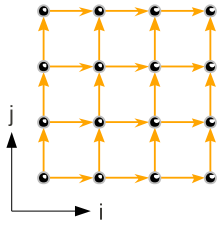
The polyhedron model is a powerful model to describe and transform certain regular loop nests. It and has been studied and extended for several decades [KMW67,Lam74,Len93]. We briefly introduce the model itself and a tool chain, the Polyhedral Compiler Collection, which we extend with modules for GPU-specific code generation.

2.1 Model

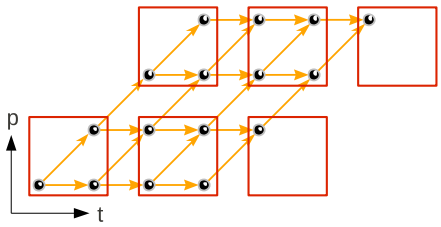
The polyhedron model (without its recent extensions for irregular codes and non-linearities) can describe codes that consist of nested `for`-loops with statements that access arrays. Both the loop bounds and the subscripts of the arrays have to be affine expressions in the surrounding loop indices and some structural parameters (e.g., the size of a matrix). Such a loop nest (or a sequence of them) is called a static control part (SCoP).

```
   for (i=1; i<=n; i++)
     for (j=1; j<=n; j++)
S:     A[i][j] = (A[i-1][j] + A[i][j-1]) * 0.5;
```

(a) Code



(b) Iteration domain



(c) Domain after space-time mapping and tiling

```
for (T=0; T<=(2*n-2)/2; T++)
  parfor (P=max(1+2*T-n,0)/2; P<=min(2*T+1,n)/2; P++)
    for (t=2*T; t<=min(2*T+1,2*n-2); t++)
        parfor (p=max(t-n,0); p<=min(t,n); p++) {
          int i = 2-p+t, j = p+1;
          A[i][j] = (A[i-1][j] + A[i][j-1]) * 0.5;
        }
```

(d) Parallel code

Fig. 1. Static control part

The loops surrounding a statement S define its iteration domain D_S, i.e., S is executed for each $i \in D_S$. For example, the statement S in the static control part shown in Figure 1(a) has iteration domain $D_S = \{(i,j) \mid 1 \leq i,j \leq n\}$. A specific execution of a statement S for a value $i \in D_S$ is called an operation of S, written $\langle i,S \rangle$.

The array accesses in the statements give rise to the dependences between the operations of the statements. When two operations $\langle i,S \rangle$ and $\langle j,T \rangle$ access the same memory cell, i.e., $f(i) = g(j)$ with array accesses $A[f(i)]$ in S and $A[g(j)]$ in T, and at least one of the two accesses is a write access, then the relative execution order of $\langle i,S \rangle$ and $\langle j,T \rangle$ may not be changed as this would, in general, change the program semantics. In

Figure 1(b), the arrows show the dependences between operations. The dependences determine which operations can be executed in parallel. The parallel execution order is given by computing a so-called schedule θ_S for each statement S which gives the execution time $\theta_S(i)$ of an operation $\langle i, S\rangle$ and a placement π_S which gives the processor $\pi_S(i)$ on which the operation is executed. Schedule and placement together are called the space-time mapping of the program model.

Since the parallelism given by schedule and placement is usually too fine-grained for the parallel target architecture (e.g., the placement describes more processors than are available), the grain of the execution is coarsened after space-time mapping by applying a tiling step, i.e., bigger blocks are formed by grouping together operations and executing them as atomic units. In the model, this is achieved by changing the enumeration of a dimension (e.g., the processors) to a two-dimensional enumeration, an outer dimension (the tile dimension) that enumerates the groups and an inner dimension that enumerates the iterations inside a given group. The size of the tiles is determined by the hardware, e.g., the number of processors available. Tiling can also be used to adjust the memory footprint of the tiles, for example, to make each tile use the cache of a processor optimally.

The final step in the transformation process is to generate executable code from the program model after space-time mapping and tiling. The state-of-the-art tool for doing this is CLooG [Bas04]. An in-depth description of all the steps of the model (targeted for distributed memory architectures) can be found in the literature [Gri04].

2.2 Polyhedral Compiler Collection

The Polyhedral Compiler Collection (PoCC) links together several tools for the polyhedron model to form a source-to-source parallelizing compiler. It uses Clan to extract a SCoP from a given source code, Candl to analyze dependences, Pluto [BHRS08] to compute schedule, placement and tiling, and CLooG to generate code. Recent development versions of PoCC include tools for vectorization and other optimizations. PoCC was designed to be used for multicore CPUs. We contribute modules to PoCC which enable a source-to-source transformation from a C program to a CUDA program where a designated SCoP is executed on a GPU.

3 GPU-Specific Challenges

Today, graphics processing units (GPUs) are not only used for rendering images but for general purpose computing, so called general purpose GPU (GPGPU) computing. This change has been driven by two factors. First, some changes to the architecture of GPUs make it possible to execute general purpose applications on GPUs, not only graphics computations. Second, the raw computing power of GPUs currently exceeds the computing power of multicore CPUs. Although one CPU core and one GPU multiprocessor (see below) have about the same computing power, GPU multiprocessors are simpler and, hence, more of them can be put on a single processor die.

The graphics heritage of GPUs is reflected in a massively parallel execution model due to computing the pixels of an image being a heavily parallel task. Apart from this

and other differences in the execution model compared to CPUs, memory hierarchies and their management and the structure of the code pose challenges that need to be addressed when targeting graphics processors for parallel applications. In the rest of this section, we present the architecture of NVidia GPUs as an example for the architecture of graphics processors. Targeting OpenCL instead of CUDA would be similar but differ in the details. We plan to support OpenCL in the future, too.

3.1 Execution Model

A GPU consists of several *multiprocessors* (analogous to the cores of a multicore CPU) that execute independently of each other. Each multiprocessor consists of 8 or, on newer hardware, 32 thread processors that execute threads. The main constraint for the execution is that the thread processors of a multiprocessor have to execute the same instruction for the computations to be parallel. The reason is that there is only one instruction decode unit for the whole multiprocessors. Each thread processor has its own arithmetic logic unit, i.e., the computations of the threads happen in parallel in a SIMD fashion.

When a *kernel* (GPU code) is called, the invocation specifies a two-level parallelism corresponding to the multiprocessor-thread hierarchy. The threads are grouped in so-called blocks. A block is mapped to a multiprocessor (as soon as a multiprocessor becomes free); hence, the execution of the blocks is independent (there cannot be communication between the blocks). In each block, several threads (mapped to the thread processors) execute. Threads in a block can synchronize and exchange data during the execution.

The blocks are addressed in one- or two-dimensional coordinates, the threads in a block in one-, two- or three-dimensional coordinates depending on the application's need.

The threads in a block are grouped in so-called *warps* of 32 threads. According to the limitations of the multiprocessors, the threads in a warp have to execute the same instruction for parallel execution to happen; otherwise, *divergence* occurs and the execution becomes sequential. Different warps can take different control paths without harming performance.

3.2 Memory Management

On CPUs, one has to deal with only one level of memory explicitly, namely main memory. Caching is handled by hardware; to profit from caching, the only challenge is to arrange the code such that accesses to main memory happen in a fashion that is suitable for caching.

On GPUs, hardware-managed caches for main memory are available (on newer architectures) but it is still necessary to exploit the scratchpad memory of a multiprocessor (called *shared memory* in CUDA) by explicit addressing. Each multiprocessor offers 16 or 64 kB of shared memory which is local to the multiprocessor and can be accessed within one clock cycle provided that certain alignment constraints (that depend on the hardware generation) are obeyed. Newer hardware generations lift the alignment restrictions.

In total, there are three levels in the memory hierarchy:

– host memory, i.e., the main memory of the CPU,
– device memory, i.e., the main memory of the GPU accessible by all multiprocessors,
– shared memory, i.e., the scratchpad memory of a multiprocessors.

To be able to execute code at all, we have to copy input data from host memory to device memory and output data from device memory to host memory. To achieve high performance, data values that are accessed multiple times have to be put in the scratchpad, i.e., at suitable points, data has to be copied to scratchpad memory before the data is used and, later, when the data is not used/updated any more, it has to be copied back to device memory.

Organizing the memory transfers places two problems. First, we have to determine which elements to copy. Second, we have to select a linearized layout for the transferred data. In case of a copy from host to device memory, we can only copy a contiguous memory region (through DMA) and for a copy between device and scratchpad memory, we do not want to waste memory in the scratchpad because of its limited size. For the scratchpad management, there is a third challenge, namely we do not need (and want) to copy all the elements each time because some elements are reused and should remain in the scratchpad longer than others.

The elements to transfer are found by computing the sets of elements that are actually used and the linearization is performed by assigning contiguous one-dimensional indices to the elements. The theoretical basis for dealing with these problems can be found in our own previous work [Grö09]. In Section 5.1, we describe the practical side of these techniques.

3.3 Code Generation

The CUDA programming language is based on C++. On the one hand, a few extensions are made, for example, to declare whether a variable (on the GPU) is in device memory or in shared memory, or a special syntax for invoking GPU code from host code specifying the number of blocks and threads.

On the other hand, kernel code is rather restricted. For example, it cannot use recursion (because there is no run-time stack) or dynamic memory allocation. Fortunately, the transformed code obtained for SCoPs does not need these features. In Section 5.2, we present the modifications we have made to the polyhedral code generation for multicores to generate CUDA code.

4 Related Work and Position Statement

Revisiting the affine transformation construction and the heuristics of Bondhugula's Pluto framework [BHRS08], Baskaran et al. developed the first polyhedral compiler optimizations for GPU targets [BBK+08a,BBK+08b]. These optimizations include locality- and access-pattern enhancing transformations for the GPU's global memory and shared memory, as well as code generation to manage the on-chip shared memory. Baskaran's C-to-CUDA is the first end-to-end, automatic source-to-source polyhedral compiler for

GPU targets, implementing and evaluating the above optimizations as well as several code generation enhancements [BRS10].

Our work is independent from from C-to-CUDA, although it shares many tools and algorithms with it. Our motivations are also slightly different, leading to the investigation of complementary optimization and code generation problems, and stressing the practical post-processing aspects of the problem.

Indeed, our first motivation was a very practical one: building a GPU code generator that would complement an arbitrary source-to-source tool-chain for polyhedral compilation (PoCC in this case). Unlike C-to-CUDA, this choice requires a careful modularization and standardization of the input parameters, polyhedral sets and relations to drive the code generation. This choice also required decoupling the code generation algorithm (CLooG) from post-processing stages dedicated to CUDA syntax generation, including the generation of the memory copying and other CUDA library calls, the declaration of the kernel's signature, and the declaration of new (local) arrays and memory management instructions. GPU-specific optimizations are also impacted by this design, and recast as independent passes working on the PoCC intermediate format. For example, the localization pass is a key component of our approach; it is an evolution of an algorithm we proposed before [Grö09]. Another advantage of this design is the ability to plug unmodified optimization heuristics, such as Pouchet's LeTSeE [PBCC08] iterative search engine, as a complement to GPU-specific tiling and parallelization passes. Eventually, we are not willing to restrict ourselves to static control loop nests, but plan to extend our optimizations and code generator to arbitrary (structured) intraprocedural control flow. This is made possible by our recent advances in polyhedral code generation and abstraction for dynamic data-dependent conditions [BPCB10].

5 Implementation

Our implementation is in a work-in-progress state as we first had to solve several technical problems to get a working source-to-source compiler. The algorithms dealing with GPU-specific challenges are still being improved but they are in a state that allows us to present our work at this stage and give an impression of where we are heading.

Parsing the input source code and computing the dependences is unmodified. For the space-time mapping and tiling steps of the transformation process, we use an unmodified version of Pluto for now because it turned out that Pluto's parallelization and optimization which are targeted at multicore CPUs are good enough for constructing a parallel program for a GPU. To fully exploit the potential of GPUs, we plan to implement parallelizing transformations that are tailored to the peculiarities of GPUs later. After the tiling phase, we insert our two modules: GPULOCALIZER and CUDAGENERATOR. The GPULOCALIZER module adds statements for the host to device memory transfers and statements for the device to shared memory transfers and reorganization. The CUDAGENERATOR module first calls CLooG to generate code for the transformed program model (including the statements added by GPULOCALIZER) and applies some post-processing to obtain CUDA code.

After space-time mapping and tiling, the principal structure of the transformed program is as shown in Figure 2.

```
for (T=...) // global time steps (time tiles)
  parfor (P=p_l(T) to p_u(T)) // global processor, i.e., space time
    for (t=...) // local time (time inside a tile)
      parfor (p=...) // local processor (thread)
        computation statements
```

Fig. 2. Principal program structure after space-time mapping and tiling

5.1 GPULOCALIZER

Both the data transfer between host and device memory and the management of the scratchpad (shared memory) is done by GPULOCALIZER. The reason for combining both tasks in one module is that the computations required are actually quite similar.

The theoretical basis for our computations can be found in our own previous work [Grö09]. We did not implement our procedure completely at that time as we had to use a library which turned out to compute incorrectly with Z-polyhedra and the deficiencies of the library could not be repaired easily. Therefore, our previous implementation could only deal for codes without tiling, for example.

Meanwhile, the integer set library[3] (ISL) has become available. This enables us to perform exact computations for integer sets defined by affine (in)equalities (including Z-polyhedra). We can now implement a sound and complete management for the scratchpad and for the data transfer between host and device. In the following, we present our technique for a single array, but the extension to multiple arrays is straightforward.

For the data transfer between host and device, we compute the set $C_A(T)$ of all indices of elements of array A needed in global time step T, i.e., $x \in C_A(T)$ means that there is a tile with global time coordinate T that accesses $A[x]$. By Ehrhart theory (as implemented in the Barvinok library[4]), we can compute an expression $\sigma_A(x,T) \in \mathbb{Z}$ such that for a given T

- the function $x \mapsto \sigma_A(x,T)$ is injective on $C_A(T)$,
- $0 \leq \sigma_A(x,T) < |C_A(T)|$ for all $x \in C_A(T)$.

In other words, σ_A maps the elements of $C_A(T)$ to a contiguous interval starting at 0. Therefore, σ_A can be used to assign positions in the transfer buffer between host and device memory to the elements of A used at time step T. In the kernel, i.e., the code that runs on the GPU, each array access $A[f(i)]$ is replaced by $A[\sigma_A(f(i),T)]$ and the kernel will operate on the right data values from the transfer buffer.

To improve performance, we have to keep relevant data values in shared memory, i.e., we actually want to replace $A[f(i)]$ by $L_A[\rho_A(f(i),t)]$ where L_A refers to shared memory reserved for elements of original array A and ρ_A is the mapping function between the original index $f(i)$ and the position in shared memory for array A. ρ_A can be computed in the same way as σ_A from the corresponding set $D_A(t)$,[5] except that ρ_A

[3] http://freshmeat.net/projects/isl, visited 2010-06-07.

[4] http://freshmeat.net/projects/barvinok, visited 2010-06-07.

[5] Actually, D_A depends not only on t but also on T and P; for the ease of notation, we only write the dependence on t.

does not assign linearized locations for a global time step (the set of all parallel tiles) but for one local time step, i.e., one iteration of the loop on t within a tile. Unlike the data transfer between host and device, we do not copy in all the data to shared memory at the beginning of each time step and copy back all the data at the end of the time step because reuse between time steps is likely and, in contrast to the transfers between host and device, we can move elements to new positions in shared memory individually. This enables us to move elements that are used in time steps t and $t+1$ with indices given by $x \in D_A(t) \cap D_A(t+1)$ from $\rho_A(x,t)$ to $\rho_A(x,t+1)$ and we need only copy in elements described by $D_A(t) - D_A(t-1)$ and copy out $D_A(t) - D_A(t+1)$.

The complication that occurs with moving elements inside shared memory is that we have to be careful not to overwrite elements prematurely, i.e., it has to be performed as a parallel assignment. Using two copies of shared memory to copy from one to the other is a possible solution requiring twice the amount of shared memory, of course. Depending on the properties of ρ_A, in-place movement is possible is some situations (see [Grö09]). The principal code after GPULOCALIZER is shown in Figure 3.

```
for (T=...) {
  for (x ∈ C_A(T)) buffer[σ_A(x,T)] = A[x];
  copy_to_device(buffer);
  parfor (P=P_l(T) to P_u(T)) {
    for (t=...) {
      parfor (x ∈ D_A(t) − D_A(t−1)) L_A[ρ_A(x,t)] = buffer[σ_A(x,T)];
      parfor (p=p_l(T,P,t); p ≤ p_u(T,P,t); p++)
        computation statements with L_A[ρ_A(f(i),t)] instead of A[f(i)]
      parfor (x ∈ D_A(t) − D_A(t+1)) buffer[σ_A(x,T)] = L_A[ρ_A(x,t)];
      parfor (x ∈ D_A(t) ∩ D_A(t+1))
        L_A[ρ_A(x,t+1)] = L_A[ρ_A(x,t)]; // parallel assignment
    }
  }
  copy_from_device(buffer);
  for (x ∈ C_A(T)) A[x] = buffer[σ_A(x,T)];
}
```

Fig. 3. Principal code after GPULOCALIZER

The drawback of the method we use is that the expressions computed for σ_A and ρ_A can be rather big compared to the original expressions for the array subscripts and can contain case distinctions on the loop iterators in general. The case distinctions can be eliminated by splitting the iteration domains of the statements according to the conditions; this increases the number of statements and puts some pressure on the code generator (CLooG) and can lead to code explosion. We are working on techniques that compute mappings σ_A and ρ_A that do away with mapping to a contiguous interval, i.e., by allowing to waste some space in the transfer buffer or scratchpad memory, the mapping gets simpler.

5.2 CUDAGENERATOR

The main challenge for CUDAGENERATOR is to output a correct CUDA program. In the model (and for code generators like CLooG), the loops are simply nested inside each other as shown in Figures 2 and 3. But in CUDA, the kernel code has to be in a separate function and a special syntax has to be used for invoking kernels. In addition, the `parfor` loops of the model have to be mapped to the blocks (in case of the loop on P) and threads (for the loop on p). The principal code (for readability without the code for managing shared memory) is shown in Figure 4. The generation of CUDA code happens in 4 steps:

(1) Kernels are extracted from the abstract syntax tree delivered by CLooG and replaced by kernel invocation statements specifying the number of blocks and threads. The number of blocks is derived from the number of tiles in the parallel dimension (i.e., the number of iterations of the loop on P).
(2) The computation statements in each kernel are modified to access shared memory (or device memory if shared memory is not used) by replacing original array identifiers (e.g., A by L_A) and modifying the subscript functions to use ρ or σ, respectively (e.g., change $A[f(i)]$ to $L_A[\rho_S(f(i),t)]$).
(3) The code for each kernel is printed in a separate CUDA file and a header file containing the prototype of the kernel function is created. In the kernel code, the values of the tile iterators are reconstructed: T is passed to the kernel as a parameter and P is computed from the block number. Parallel loops (`parfor`) are made parallel by distributing the iterations among the threads, i.e., the thread number is added to the lower bound of the loop and the loop stride is set to the number of threads.
(4) The function containing the original SCoP is augmented with declarations for the variables, initialization of CUDA, allocation of transfer buffers and, of course, the host part of the generated parallel code (i.e., the loop on T, data transfer and kernel calls).

Note that the iterations of the loop on P become the blocks of the kernel invocation and the iterations of the loop on p are distributed among the threads of a block. The required amount of shared memory and the required size of the transfer buffer can be computed using Ehrhart theory by counting the number of elements in $D_A(t)$ and $C_A(T)$, respectively. Each block is started with 512 threads (the maximum on older hardware) as, at the moment, we do not compute the maximal number of threads among all the blocks of a kernel invocation.

5.3 Technical Difficulties

Unfortunately, several technicalities prevent us from presenting benchmark results in this paper. We are aware of the fact that describing an auto-parallelizing source-to-source compiler without showing benchmarks is quite unconvincing and we are rather dissatisfied that we cannot provide any hard numbers.

The main reason for the current problems is that the tool chain (PoCC) is in a state of flux, and its interfaces, data structures and libraries are constantly evolving. Some of the recent changes are necessary for our modules to work correctly; therefore, using an

```
__global__ void kernel0(float *buffer, int T, int n) {
  int P = P_l(T) + blockIdx.x;
  for (t=...) {
    ...
    for (p=threadIdx.x+p_l(T,P,t); p ≤ p_u(T,P,t); p += blockDim.x)
      computation statements with L_A[ρ_A(f(i),t)] instead of A[f(i)]
    ...
    }
}

for (T=...) {
  for (x ∈ C_A(T)) buffer[σ_A(x,T)] = A[x];
  copy_to_device(buffer);
  dim3 blocks(P_u(T)-P_l(T)+1,1), threads(512,1,1);
  unsigned sharedSize = maxSharedSize_A(T) + ...;
  kernel0<<<blocks,threads,sharedSize>>>(buffer, T, n);
  copy_from_device(buffer);
  for (x ∈ C_A(T)) A[x] = buffer[σ_A(x,T)];
}
```

Fig. 4. Principal code after CUDAGENERATOR (shared memory management not shown)

older, more stable version of PoCC was not an option. The changes in the interfaces and data structures required to rewrite some parts of our modules several times. Changing to new libraries (e.g., ISL) exposed several bugs in all parts of the tool chain which stalled development of our modules further.

We hope to overcome the current technical difficulties in the coming weeks and to be able to run a few benchmarks when this paper is presented at CPC 2010. A few benchmarks performed with a prototype implementation of GPULOCALIZER (which did not support tiling and suffered from relying on an incorrect library) can be found in our previous work [Grö09].

6 Conclusions

With our extensions to the Polyhedral Compiler Collection (PoCC), we have shown that we can build a working source-to-source compiler which takes a sequential C program as input and produces a CUDA program that can exploit the parallelism of a modern GPU to execute certain loop nests (static control parts) in parallel. Building upon the well-established polyhedron model, the transformation from a sequential input program to a parallel output program is automatic. Our contributions are modules for the tool chain that deal with the data transfer (copy input data from the host to the GPU, copy output data from the GPU to the host), scratchpad management and post-processing of the loop code generated by a polyhedral code generator (CLooG) to be valid CUDA code. Our modules are still in development as, e.g., different codes exhibit different characteristics w.r.t. their use of the scratchpad; hence, several optimizations for common cases have to be implemented to get a fast execution. In addition, the parallelizing

transformation computed by Pluto (which has been designed for multicore CPUs) is good enough for our examples, but we are working on modifications that take the peculiarities of GPUs into account to a greater extent (for example, the restrictions on the control flow on GPUs).

References

[Bas04] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society.

[BBK+08a] Muthu M. Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J . Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic data movement and computation mapping for multi-level parall el architectures with explicitly managed memories. In *PPoPP '08: Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–10, New York, NY, USA, February 2008. ACM.

[BBK+08b] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for gpgpus. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 225–234, New York, NY, USA, 2008. ACM.

[BHRS08] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelization and locality optimization system. In *ACM SIGPLAN Conf. on Programming Languages Design and Implementation (PLDI'08)*, Tucson, AZ, USA, June 2008.

[BPCB10] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'10)*, number 6011 in Lecture Notes in Computer Science, Paphos, Cyprus, March 2010. Springer-Verlag.

[BRS10] Muthu M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In *Proceedings of the International Conference on Compiler Construction (CC 2010)*, number 6011 in Lecture Notes in Computer Science, pages 244–263. Springer-Verlag, March 2010.

[Gri04] Martin Griebl. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. University of Passau, 2004. habilitation thesis.

[Grö09] Armin Größlinger. Precise management of scratchpad memories for localising array accesses in scientific codes. In O. de Moor and M. Schwartzbach, editors, *Proceedings of the International Conference on Compiler Construction (CC 2009)*, number 5501 in Lecture Notes in Computer Science, pages 236–250. Springer-Verlag, 2009.

[KMW67] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, July 1967.

[Lam74] Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974.

[Len93] Christian Lengauer. Loop parallelization in the polytope model. In Eike Best, editor, *CONCUR'93*, LNCS 715, pages 398–416. Springer-Verlag, 1993.

[PBCC08] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *ACM Conf. on Programming Language Design and Implementation (PLDI'08)*, Tucson, Arizona, June 2008.