Evaluation of state-of-the-art polyhedral tools for automatic code generation on GPUs

J.C. Juega[1], J.I. Gómez[1], C. Tenllado[1], S. Verdoolaege[2], A. Cohen[2], F. Catthoor[3]

Abstract— **At present, multi-core and many-core platforms lead the computer industry, forcing software developers to adopt new programming paradigms, in order to fully exploit their computing capabilities. Nowadays, Graphics Processing Units (GPUs) are one of representatives of many-core architectures, and certainly the most widespread.**

This paper evaluates and compares tool frameworks that automatically generate code for GPUs, saving time and effort to programmers. These frameworks take advantage of Polyhedral Model techniques to exploit parallelism and to satisfy the specific GPU constraints. The paper shows the key features of some of these source-to-source compilers and analyzes the code that they generate. Finally we discuss the importance of some key aspects such as data mapping and code quality.

Keywords — **polyhedral model, gpu, cuda, code generation, compilers, loop transformations, C-to-CUDA, Par4All, PPCG.**

I. Introduction

GPU devices are extremely difficult to program; it is not only crucial to extract parallelism but it has to be mapped carefullyto the GPU to maximize performance. Many hardware details are exposed to the programmer, who must tune them effectively while meeting platform restrictions regarding the memory hierarchy, thread synchronization and resource management.

Thus a clear need exists for tool support assisting the programmer in this hard task. Both autotuning and *ad-hoc* parallelizing compiler techniques have been proposed to this end. An ideal tool flow would take sequential code with loops and arrays and produce efficient code to be run on a system with (several) CPUs and GPUs. This process would entail parallelism extraction, defining GPU kernels and finally efficiently mapping them onto the GPU.

Recently, several frameworks have been trying to, at least partially, fulfill this ambitious task. We are particularly interested in those frameworks based on polyhedral compilation techniques, which have shown to be very effective in more traditional parallel environments. Most of them are source-to-source tools that generate CUDA code out of sequential C code, then relying on *nvcc* to perform the final compilation steps.

For this work, we have selected three representative tools within this context: Par4All, C-to-CUDA and PPCG. We describe their common underlying approaches to generate code for GPU platforms. We will show that they all follow common steps during the mapping process, but that they significantly differ in terms of program representation and transformation heuristics to explain large performance differences.

[1]ArTeCS group, Univ. Complutense de Madrid, e-mail: cjuega@fdi.ucm.es.
[2]ENS/NRIA
[3]IMEC and K.U. Leuven

II. Related Work

With the emergence of GPUs, the Polyhedral framework has been applied to develop efficient source-to-source compilers for them. Baskaran's C-to-CUDA [1] is the first end-to-end, automatic source-to-source polyhedral compiler for GPU targets. It is based on PLUTO and, as we will explain in this work, it uses basic techniques to manage shared memory. Building on Baskaran's experience, Reservoir Labs developed its own compiler based on R-Stream [2], which introduces a more complex way to manage the memory hierarchy. *Gpuloc* is a variation of the approach proposed by Baghdadi [3], which is also based on PLUTO. It wastes GPU memory by using a dual buffer system [4] which limits the GPU computation power. Furthermore it is not in development anymore. Par4All [5] is an open source initiative developed by HPC to unify efforts concerning compilers for parallel architectures. It supports the automatic integrated compilation of applications for hybrid architectures including GPUs. CHiLL developers also extended their compiler to generate GPU code, they introduced CUDA-CHiLL [6] during 2011.

The tools mentioned above were compared with both hand-tune GPU implementations and CPU implementations when they were introduced. However they have never been compared each other. We picked out C-to-CUDA, Par4All and PPCG because they were publicly available. We also considered comparing to the approach of Baghdadi, but our preliminary implementation showed the technique for memory management on the GPU to be inappropriate. For this reason and because it is no longer in development, we rejected *gpuloc*.

III. CUDA programming model

The three tools evaluated generate code for NVIDIA GPUs using the CUDA programming model [7]. This model represents the GPU as a coprocessor that can run data-parallel kernels and provides extensions to the C language to (1) allocate data on the GPU, (2) transfer data between the GPU and the CPU and (3) launch kernels.

A CUDA kernel executes a piece of sequential code on a large number of threads in parallel. Those threads are organized into a grid of *CUDA Blocks*. Threads within a block can cooperate with each

other by (1) efficiently sharing data through a shared low latency local memory and (2) synchronizing their execution via barriers. The scheduling unit is not the individual thread but a group of threads called *warp*. Every cycle, the scheduler in each multiprocessor chooses the next warp to execute. If threads in a warp execute different code paths, only threads on the same path can be executed simultaneously and a penalty is incurred.

All the details above must be exposed to the compiler in order to efficiently generate CUDA code. However, some other aspects, like fine grained multithreading or *threadblock* to multiprocessor mapping, may be hidden. The system model assumed by the evaluated tools include: an arbitrary large number of (virtual) SIMD-like processors could be present, each with its own register file and private scratchpad memory, all sharing a common DRAM based main memory.

Given the paramount importance of the memory system, the compiler must be aware of the coalescing requirements, so it must explicitly take into account the difference between a *warp* and a *block* of threads even if they are executed on the same virtual SIMD processor.

IV. Polyhedral Model Overview

The polyhedral model is a high level mathematical representation of a piece of source code, that uses polyhedra and related mathematical abstractions to represent: all dynamic instances of the statements surrounded by loops, and all dependences between any pair of dynamic instances of any pair of statements. It is applicable to the so called Static Control Parts (SCoP) of the program [8], which are formed by loops with affine manifest boundaries and bodies with statements accessing arrays with subscript expressions affine in the surrounding loop iterators and the problem parameters.

The polyhedral model assigns to each statement an *iteration vector*, with one dimension per loop surrounding the statement. The first dimension represents the outermost loop, the next dimension the following loop, and so forth. Each valid *iteration vector* for a statement represents one of the dynamic executions of the statement, corresponding to a valid combination of values for the loop iterators in the loops surrounding the statement. Each of these dynamic executions of a statement is called an *operation*. The set of all valid *iteration vectors* for a statement is called the statement *Domain*.

In addition, the polyhedral model provides a compact representation of all the data dependences between all pairs of *operations*. For a given pair of statements S_1 and S_2, all the dependences from any *operation* of S_1 to any *operation* of S_2 form a relation between the *Domains* of S_1 and S_2. This relation is called the *Dependence Relation* between S_1 and S_2.

This representation has shown to be very useful to find valid transformed versions of the original SCoP exhibiting some interesting properties, such as exposed parallelism [9], [10] or improved locality [11], [12]. The usual way to handle code transformation in the polyhedral model is by the construction of a *Schedule* for each statement.

A *Schedule* is an affine function that assigns to each point in the original *Domain* of the statement a point in a target space. The target space is common for all statements, but several points of the original *Domains* can be mapped to the same point in the target space. Usually this target space is seen as a *date* space, in which the first dimension is more important than the second dimension and so forth (like hours, minutes, ...). In this way the *Schedule* fixes a relative and partial ordering for all the operations. The transformed code is obtained by generating code that scans all the statement *Domains* respecting the partial ordering specified by the *Schedule* [13], [14], [15]. However, it is also possible to build *Schedules* in which some dimensions have different interpretations, for instance mapping to computing units (processors) instead of time units (*dates*).

The process to build the *Schedule* depends on the goal, whether it is parallelism, locality, etc. Feautrier proposed an algorithm to find the set of all valid affine schedules for a given SCoP [9], [10].

V. Tools for GPU code generation

We can split the methodologies followed by the tools selected for the evaluation, C-to-CUDA [1], Par4All [5] and PPCG, in four main stages:

- *Exposing parallelism.* All three tools use the polyhedral model for this stage. They basically select the schedules for each statement that produce as many parallel dimensions as possible in the target space.
- *Mapping to the CUDA model.* The parallel dimensions are mapped to CUDA's *thread block* and *thread* identifiers.
- *Data mapping.* Basically, the tools decide where, in the memory hierarchy, to place all data that is being accessed. If some array is to be placed in the shared memory, specific code is added to take care of the transfers from global memory to shared memory.
- *Code generation.* For both the host and the kernels.

The first stage is similar for the three tools. However, some differences exist that affect performance. Par4All treats each original loop nest independently, generating a specific kernel for each one. On the other hand, C-to-CUDA and PPCG work on SCoPs as large as possible and apply some locality optimization criteria to generate the *Schedule*. As a result, the different loop nests can be fused, at least partially. The generation of the *Schedule* in both C-to-CUDA and PPCG is based on the algorithm proposed in Pluto [16], [17], although they currently evolve independently. Par4All uses its own algorithm.

In the second stage, the three tools look for a *parallel band* in their respective *Schedules*. In this con-

text, a *parallel band* is a set of one or more consecutive parallel dimensions (loops). If more than one *parallel band* exists all three tools select the outermost band. The selected parallel band is then mapped onto CUDA's *thread block* and *thread* identifiers. There are some important differences between the three tools in how this mapping is performed.

In Par4All at most two parallel dimensions are selected from the *parallel band*, which are then tiled. The resulting dimensions are then reordered so that all *tile dimensions* are set consecutive just before the consecutive *point dimensions*.[1] The rest of the *Schedule*'s dimensions are not modified. The *tile dimensions* are then mapped onto *thread block* identifiers. The *point dimensions* are mapped onto *thread* identifiers. Furthermore, Par4All always chooses between the same two tile sizes, 16×16 for two dimensional parallelism or 1×512 for one dimensional parallelism, i.e. it always uses 16×16 or 1×512 *thread blocks* independently of the algorithm being mapped. This leads to suboptimal solutions in many cases.

On the other hand, both C-to-CUDA and PPCG also tile the dimensions following the first two parallel ones. After that dimensions reordering is performed, putting the *tile dimensions* before the *point dimensions* independently of their parallel or sequential nature. In some cases this reordering, which leaves some sequential *tile dimensions* (that translate to loops in the kernel code) before the parallel *point loops*, increases the chances of exploiting data reuse in the shared memory at the expense of introducing some thread synchronizations. However, we should highlight that no analysis is performed to evaluate the effect on performance of these extra tilings and reorderings, for the code being mapped.

In addition, both C-to-CUDA and PPCG take the tile sizes, *thread block* sizes and *grid* sizes as user parameters. This translates to the necessity of applying extra tilings on the parallel *tile* and *point* dimensions, to wrap those dimensions on the actual *thread block* and *grid* sizes selected by the user. This has its advantages and drawbacks. On one hand, the user can fine tune these parameters for a given algorithm. However, a non expert user, without knowledge on how the tool works, would probably need an exhaustive search to find those optimal parameters. On the other hand, an extra degree of complexity is introduced for the final code generation stage, as the *Schedule* dimensions increase with these extra tilings, making it sometimes inefficient. In our opinion, the tool should select the values for these parameters, after an analysis on the code being compiled: for the designer that decision is indeed too tedious and complex to make. Using default values independently of the code being compiled, like Par4All does, is also suboptimal.

[1]Tiling a dimension translates to dividing it into consecutive *tiles*, and replacing the original dimension by a set of two new dimensions, the first identifying the tile and the second one identifying the point inside the tile. The first dimension is called *tile dimension* and the second one is called *point dimension*.

The tools also differ in the *Data mapping* stage. Par4All does not consider the exploitation of reuse in the shared memory or the register file. All accesses are performed directly on the global memory. For the most advanced Fermi architectures, it configures the memory hierarchy to use as much hardware cache as possible, relying on it to exploit data reuse.

C-to-CUDA takes a different systematic approach, every array is mapped to the shared memory. Thanks to the extra tilings on the dimensions after the parallel band, this is many times possible. However, as no analysis was performed on the code to perform these tilings, it can lead to invalid codes that use too much shared memory. In addition, mapping an array to shared memory when it does not exhibit inter-thread data reuse reduces performance, as we have the extra accesses to the shared memory without any compensation. We will see that this translates to a poor performance in many cases.

PPCG is, in this aspect, the most advanced tool. It performs the following analysis to decide if a given array is to be accessed directly from the global memory or if it is to be mapped on the shared memory or the registers file:

- If the data can be put in registers and there is reuse, then put it in registers.
- Otherwise, if the data can be put in shared memory and either there is reuse or the access is non-coalesced, then put it in shared memory.
- Otherwise, put it in global memory.

Finally, the tools also differ in the *code generation* stage. Only Par4All and PPCG generate both host and kernel codes, including all code to manage CPU-GPU transfers. In addition, Par4All and PPCG generate specific codes for specific problem sizes when the problem sizes are known at compile time. This has the advantage of generating simpler kernel codes, significantly reducing the number of dynamic instructions executed by the kernel. On the other hand C-to-CUDA takes these problem sizes as parameters, generating a code valid for different problem sizes. However, the kernel code has lot more conditions to evaluate, increasing the number of dynamic instructions for the kernel execution. On the other hand, as mentioned before, the extra tilings introduced to leave the user the control over the tiling parameters lead to very complex kernel codes, frequently inefficient. This could be solved by providing the tools with a more sophisticated algorithm in the *Mapping to the CUDA model* stage.

VI. Experimental results

In this section, we present experimental results to assess the effectiveness of the CUDA code generated by the tools introduced in Section V. We compare the performance of the automatically generated CUDA code by each tool.

The GPU device used in our experiments was a NVIDIA Tesla M2070 GPU (*Fermi* architecture). The device has 6144 MB of DRAM and has 14 mul-

tiprocessors clocked at 1.15 GHz. Each multiprocessor has 32 processor cores and has 64KB L1 memory . The L1 memory was configured to provide 16KB of software-controlled shared memory and 48KB of cache memory for the Par4All experiments and 48KB of software-controlled shared memory and 16KB of cache memory for PPCG and C-to-CUDA ones.

The CUDA code was compiled using the NVIDIA CUDA Compiler (NVCC) 4.0 to generate the device code that is launched from the CPU (host). The CPU was an Intel Xeon E5530 CPU clocked at 2.4 GHz. We used GCC 4.5 to compile CPU versions. We enabled `-O3` compiler optimizations.

We have used several benchmarks related to linear algebra, image processing and data-mining. All these benchmarks are included in *Polybench 3.1* which is a set of computation intensive programs often used in the polyhedral community.[2] Although *matrix transpose* is a small kernel, it is often used in many applications as well as it is a good example of non-coalesced access pattern, so we added it to our benchmarks suite which increases our suite to 31 benchmarks.

The analysis is divided into two sections. The first one is dedicated to *Matrix Multiply* which is included in *Polybench*. *Matrix Multiply* is probably the most tested benchmark in computation area and GPGPU. Many researchers are still working on improving the performance for different platforms (including GPUs) [18], [19]. We discuss the other benchmarks in the second section.

A. Matrix multiply (gemm)

We have picked out the platform-specific CUBLAS implementation of matrix multiply (gemm) as an ideal reference for the automatically generated code.

Figure 1 describes the scheme followed by C-to-CUDA and PPCG to perform the computations. Both tools apply tiling to each dimension so they are later able to exploit data reuse on shared memory and registers. Both map A and B to shared memory, but only PPCG promotes accesses to C to registers.

C-to-CUDA decides to map C on shared memory instead. Furthermore C-to-CUDA's schedule is slightly different, it misses the exploitation of some data reuse on shared memory because the schedule computes an entire partial value of C before starts with the others. These decisions make the difference between PPCG and C-to-CUDA and their effects are reflected in performance results.

Each threadblock has $BX * BY$ threads and computes a C tile of $TN * TM$. Each thread can compute several elements of C depending on TN/BY and TM/BX ratios. In our case each thread computes 8 elements.[3] This means some parallelism is sacrificed in order to make a better exploitation of data reuse on shared memory.

[2] `http://www.cse.ohio-state.edu/~pouchet/software/polybench/`

[3] Parameters are manually fixed according to the optimal values found in our previous work [20].

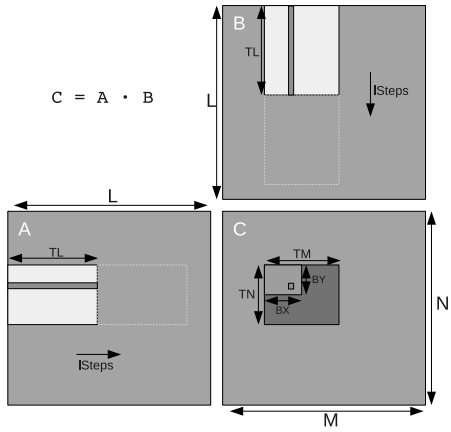

Fig. 1. Matrix Multiply diagram (C-to-CUDA and PPCG).

On the other hand Par4All generates simpler code that avoids tiling and exploits the maximum parallelism possible. It only exploits reuse on C by using one register per thread, whereas A and B are always read from global memory.

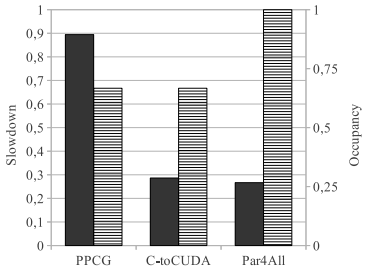

Fig. 2. Matrix Multiply - slowdown (baseline, CUBLAS)

Figure 2 compares the performance of the generated code for gemm by the different parallelizing compilers and CUBLAS.

The gpu provides a software-managed L1 memory layer, favoring tools with more elaborated Data Mapping strategy (PPCG and C-to-CUDA in the present experiments). Maximizing the use of shared memory makes the platform occupancy drop to 66% for PPCG and C-to-CUDA. Nevertheless, they both outperform Par4All stressing the paramount importance of data locality exploitation. PPCG clearly outperforms C-to-CUDA due to the more efficient compilation of the accesses to the C matrix, which prevents unneeded writes to the shared memory. Indeed, PPCG get very close to the hand-tuned CUBLAS implementation.

B. Other benchmarks

We now conduct an overall analysis of the remaining benchmarks. Unlike the Matrix Multiply case, we do not have a hand-tuned implementation for each benchmark, so we compare each compiler against the others. In addition we compare PPCG and Par4All with sequential CPU implementations.

Problem sizes were defined by default in Polybench, while input parameters for PPCG and C-to-CUDA were found based on our GPU knowledge and our knowledge of the tools themselves. All results presented here are the average of 100 executions and

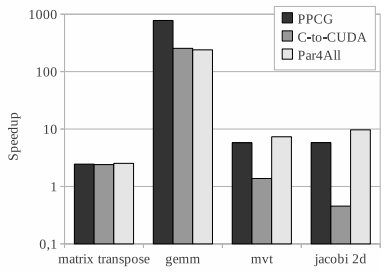

Fig. 3. Comparison between the three tools on Tesla M2070: speedup, CPU sequential version as baseline.

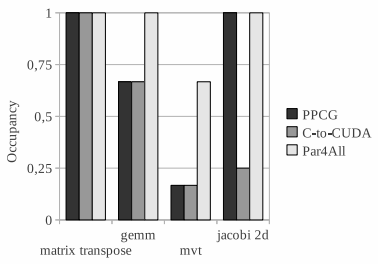

Fig. 4. Comparison between the three tools on Tesla M2070: occupancy of the generated code.

include GPU initialization, data transfers and kernel execution time.

Unfortunately the three tools are not robust enough to deal with all benchmarks. PPCG generates code for 29 out of 31 polybenchs. We use only 20 benchmarks with Par4All because the other 11 benchmarks do not result in the extraction of a CUDA kernel. Finally, C-to-CUDA generates code for 11, but only 4 of 11 benchmarks compiled with C-to-CUDA were correctly executed. So, in order to show a wide range of results, we first show in Figure 3 and Figure 4 the comparison of the three tools, and then we will analyze the codes generated by Par4All and PPCG for all the benchmarks.

PPCG shows the best performance for two benchmarks running on the Tesla M2070 as it is shown in Figure 3. With the exception of gemm, differences in performance are small among all the tools, which is easily explained by the poor data reuse exposed by the benchmarks. Both C-to-CUDA and PPCG can not always achieve high occupancy values because of the shared memory requirements. Although the effect of shared memory is noticeable, it is less remarkable due to the poor data reuse. On the other hand Par4All shows the highest concurrent resources utilization since it does not use shared memory.

The effect of the cache memory speeds up to a greater extent the codes compiled with Par4All than the other ones. So Par4All shows the best performance on the other two benchmarks, in which the poor data reuse does not justify the loose of parallelism due to the high shared memory requirements.

Figure 5 shows a global comparison for default problem sizes on Tesla M2070. The results do not include C-to-CUDA as it fails to produce any code for most of the benchmarks. However we include CPU parallel code generated by Pluto. We remove some benchmarks from the figure because 9 of the algorithms perform worse than their sequential CPU version. These are all algorithms with reduced exposed parallelism, and thus not good candidates for a massive parallel platforms like the GPU. However, some of them could become more parallel after some transformations, like for instance scalar expansion (privatization). Unfortunately neither PPCG nor Par4All implement this transformation automatically. Anyhow we leave gramschmidt as a representative of this group.

The figure shows that the algorithms that do not exhibit enough parallelism are better mapped to OpenMP by Pluto. PPCG performs better than Par4All in the 7 out 10 remaining benchmarks, where the software-controlled shared memory is used by PPCG to exploit data reuse. However, as in Figure 3, Par4All appears to be the most favored by the effects of cache memory for the cases in which the explotation of shared memory incurs a loss of effective parallelism as it is reflected in the Occupancy in Figure 6. The figure shows that Par4All exploits all the computational resources in almost all cases, while PPCG often sacrifices computational potential to reuse large amount of data in shared memory.

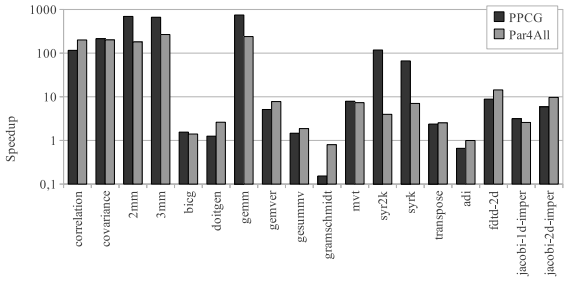

Fig. 5. PPCG and Par4All speedup on Tesla M2070 and Pluto (OpenMP) speedup on CPU (baseline sequential CPU).

Although most benchmarks analyzed so far are relative simple, the codes generated by the tools present important differences. Both PPCG and C-to-CUDA must include the code to manage the software-controlled shared memory which may involve bigger control code. In addition the control code generated by C-to-CUDA usually uses a lot of registers. On the other hand Par4All tries to generate always the simplest kernel code, even if the host code may then incur thousands of invocations of the kernel, which may lead to slowdowns.

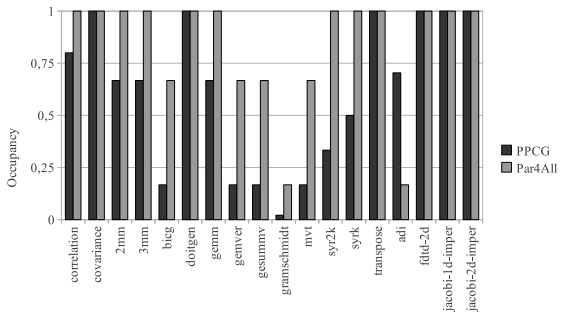

Fig. 6. PPCG and Par4All occupancy on Tesla M2070.

On more complex benchmarks like *covariance*, *2mm*, *3mm* and *doitgen*. PPCG finds different schedules based on the fusion strategy. The more ambitious ones, maximal fusion and maximize band

depth, try to bring data closer to its consumption; and sometimes, these strategies incur a loss of parallelism and excessive resource requirements. C-to-CUDA also provides options to change the scheduling strategy, however it is not able to deal with these complex benchmarks. On the other hand, Par4All tries to extract parallelism within the input schedule, so if the source is clear enough, Par4All will generate the simplest code; so it is easier for Par4All to deal with more complex algorithms.

VII. Conclusions

With the increasing use of GPUs for general purpose development, automatic or semi-automatic tools take on an ever greater importance to assist GPU-unfamiliar programmers in their daily work.

In this paper we have evaluated the current state of the art of the polyhedral model based tools for automatic code generation for GPUs, giving a detailed functional description of three of them: PPCG, Par4All, C-to-CUDA. The results show that such tools are already mature enough to deal with most of the numerical kernels in the PolyBench 3.1 suite.

To gain some perspective on the quality of the code generated by these tools, we selected some of the benchmarks to compare with hand-tuned versions for GPUs, with automatically generated parallel versions for regular SMPs and with sequential versions executed on high performance CPUs. The performance achieved by the automatically generated GPU code is, in some cases close the best known manual implementation.

Regarding the comparison of the three tools under study, we found that the efficiency in the exploitation of the memory hierarchy largely determines the performance of the code generated. On GPUs that only incorporate software controlled shared memories, PPCG has generally shown the best performance. This tool incorporates the most complex memory mapping strategy among the three, and in some cases sacrifices some parallelism to enable a better exploitation of the memory hierarchy. However, on the Fermi architecture that also incorporates hardware managed caches, the differences in performance are significantly reduced. Par4All fully relies on the efficiency of the hardware caches, and this allows it to generates a very simple code. When locality is efficiently captured by the cache this strategy performs better than the strategy followed by PPCG, that exploits all the locality through the shared memory. The reason is that the simple code has less instructions and warp divergence, and some times also translates to a larger occupancy.

References

[1] M. M. Baskaran, J. Ramanujam, and P. Sadayappan, "Automatic c-to-cuda code generation for affine programs," *Compiler Construction, 19th International Conference, CC 2010, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings, Volume 6011 of Lecture Notes in Computer Science, pp. 244-263. Springer.*, 2010.

[2] Allen Leung, Nicolas Vasilache, Benoît Meister, Muthu Baskaran, David Wohlford, Cédric Bastoul, and Richard Lethin, "A mapping path for multi-gpgpu accelerated computers from a portable high level programming abstraction," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, New York, NY, USA, 2010, GPGPU '10, pp. 51–61, ACM.

[3] S. Baghdadi, A. Grösslinger, and A. Cohen, "Putting automatic polyhedral compilation for gpgpu to work," *Proc. Compilers for Parallel Computer (CPC)*, 2010.

[4] A. Grösslinger, "Preciese management of scratchpad memories for localising array accesses in scientific codes," *Proceedings of the 18th International Conference on Compiler Construction, ETAPS 2009*, 2009.

[5] HPC Project, "Par4all automatic parallelization," http://www.par4all.org.

[6] Gabe Rudy, Malik Murtaza Khan, Mary Hall, Chun Chen, and Cha Jacqueline, "A programming language interface to describe transformations and code generation," in *Proceedings of the 23rd international conference on Languages and compilers for parallel computing*, Berlin, Heidelberg, 2011, LCPC'10, pp. 136–150, Springer-Verlag.

[7] NVIDIA Corporation, *NVIDIA CUDA Programming guide 4.0*, 2011.

[8] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam, "Putting polyhedral loop transformations to work," in *(LCPC'03)*, College Station, Texas, Oct. 2003, LNCS, pp. 23–30, Springer-Verlag.

[9] Paul Feautrier, "Some efficient solutions to the affine scheduling problem. i. one-dimensional time," *International Journal of Parallel Programming*, vol. 21, pp. 313–347, 1992, 10.1007/BF01407835.

[10] Paul Feautrier, "Some efficient solutions to the affine scheduling problem. part ii. multidimensional time," *International Journal of Parallel Programming*, vol. 21, pp. 389–420, 1992, 10.1007/BF01379404.

[11] Cédric Bastoul and Paul Feautrier, "Improving data locality by chunking," in *CC'12 International Conference on Compiler Construction, LNCS 2622*, Warsaw, Poland, Apr. 2003, pp. 320–335.

[12] Cédric Bastoul and Paul Feautrier, "Reordering methods for data locality improvement," in *CPC'10 Compilers for Parallel Computers*, Amsterdam, The Netherlands, Jan. 2003, pp. 187–196.

[13] Cédric Bastoul, "Efficient code generation for automatic parallelization and optimization," in *ISPDC'2 IEEE*, Ljubljana, Slovenia, Oct. 2003, pp. 23–30.

[14] Cédric Bastoul, "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, France, September 2004, pp. 7–16.

[15] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen, "Polyhedral code generation in the real world," in *Proceedings of the International Conference on Compiler Construction (ETAPS CC'06)*, Vienna, Austria, Mar. 2006, LNCS 3923, pp. 185–201, Springer-Verlag, Classement CORE : A, nombre de papiers acceptés : 20, soumis : 71.

[16] Uday Bondhugula, J. Ramanujam, and et al., "Pluto: A practical and fully automatic polyhedral program optimization system," in *Proceedings of the ACM SIGPLAN 2008 Conference on PLDI 08*, 2008.

[17] Uday Bondhugula, "Pluto - an automatic parallelizer and locality optimizer for multicores," http://pluto-compiler.sourceforge.net/.

[18] V. Volkov and J. W. Demmel, "Benchmarking gpus to tune dense linear algebra," *ACM/IEEE Conference on Supercomputing (SC08)*, 2008.

[19] Rajib Nath, Stanimire Tomov, and Jack Dongarra, "An improved magma gemm for fermi gpus," 2010.

[20] J. C. Juega, "Gpu performance study," M.S. thesis, Complutense University of Madrid, Madrid, Spain, 2010.