Polyhedral Parallel Code Generation for CUDA

Article *in* ACM Transactions on Architecture and Code Optimization · January 2013

DOI: 10.1145/2400682.2400713

6 authors, including:

Sven Verdoolaege
National Institute for Research in Computer Sci…
63 PUBLICATIONS **936** CITATIONS

SEE PROFILE

J.I. Gomez
Complutense University of Madrid
47 PUBLICATIONS **239** CITATIONS

SEE PROFILE

Christian Tenllado
Complutense University of Madrid
37 PUBLICATIONS **419** CITATIONS

SEE PROFILE

Polyhedral Parallel Code Generation for CUDA

Sven Verdoolaege, INRIA and École Normale Supérieure
Juan Carlos Juega, Universidad Complutense de Madrid
Albert Cohen, INRIA and École Normale Supérieure
José Ignacio Gómez, Universidad Complutense de Madrid
Christian Tenllado, Universidad Complutense de Madrid
Francky Catthoor, IMEC

This paper addresses the compilation of a sequential program for parallel execution on a modern GPU. To this end, we present a novel source-to-source compiler called PPCG. PPCG singles out for its ability to accelerate computations from any static control loop nest, generating multiple CUDA kernels when necessary. We introduce a multilevel tiling strategy and a code generation scheme for the parallelization and locality optimization of imperfectly nested loops, managing memory and exposing concurrency according to the constraints of modern GPUs. We evaluate our algorithms and tool on the entire PolyBench suite.

Categories and Subject Descriptors: D.3.4 [**Programming languages**]: Processor — Compilers, Optimization

General Terms: Compiler, Algorithms, Performance

Additional Key Words and Phrases: Polyhedral model, GPU, CUDA, code generation, compilers, loop transformations, C-to-CUDA, Par4All, PPCG.

ACM Reference Format:
Verdoolaege, S., Juega, J. C., Cohen, A., Gómez, J. I., Tenllado, C., and Catthoor, F. 2012. Polyhedral Parallel Code Generation for CUDA. ACM Trans. Architec. Code Optim. 9, 4, Article 54 (January 2013), 24 pages.
DOI = 10.1145/2400682.2400713 http://doi.acm.org/10.1145/2400682.2400713

1. INTRODUCTION

This paper addresses the compilation of a sequential program for parallel execution on a modern GPU. PPCG (Polyhedral Parallel Code Generator) (`http://freecode.com/projects/ppcg`) is a source-to-source compiler based on polyhedral compilation techniques. It combines affine transformations to extract data-parallelism with a code generator to orchestrate it over multiple levels of parallelism and memory. PPCG narrows the gap between parallelizing compilation for shared-memory multiprocessors and code generators for manycore accelerators with strict memory and concurrency constraints. For the first time, we provide a set of algorithms and a robust implementation capable of generating correct host and GPU code for any static control loop nest with affine loop bounds and affine subscripts.

Our priority is to offload any data-parallel computation to a GPU, as long as the process can be expressed using affine transformations. We do not address the selection of tile sizes, unroll factors, or any profitability consideration. We defer to complementary

This work is supported by the European Commission through the FP7 project CARP id. 287767 and by the Spanish government through the project TIN 2012-32180.

DOI 10.1145/2400682.2400713 http://doi.acm.org/10.1145/2400682.2400713

research to handle these issues for specific CPU-GPU configurations, but we of course strive to generate the best performing GPU code.

Beyond generality and robustness, PPCG makes significant algorithmic and experimental contributions:

— It implements a multilevel tiling strategy tailored to the multiple levels of parallelism and to the memory hierarchy of GPU accelerators. In particular, our method decouples multilevel parallelization from locality optimization, allowing to select the block and thread count independently from the array blocks allocated to on-chip shared memory and registers.
— Imperfect loop nests may be broken down into multiple GPU kernels. Data transfers are handled automatically, but in the current implementation, only happen at the beginning and end of the static control loop nest. Every computational statement can be mapped to the GPU, leaving only control code on the host processor (outer sequential loops). This may not be the optimal strategy, but it does offer maximal flexibility to complementary heuristics to later decide what part of the code is profitable to run on which architecture.
— We also introduce affine partitioning heuristics and code generation algorithms for locality enhancement specific to the registers and shared memory. In particular, we take advantage of inter-thread data, and we guarantee the consistency of local memory allocation even in presence of multiple overlapping references to a given array.
— We evaluate our algorithms and tool on the entire PolyBench suite version 3.1, targeting an NVIDIA Fermi GPU, and compare our results with state-of-the-art parallelization frameworks for GPUs.

2. RELATED WORK

So far, the practice of accelerator programming has been dominated by application-specific studies. Code generation and optimization tools have been slower to emerge, and this section focuses on the state-of-the-art in this area. We survey different aspects of the compilation challenge for GPUs: performance modeling and tuning, programming models, polyhedral approaches, and optimizations for short-vector SIMD architectures.

2.1. Polyhedral compilation

The polyhedral model has been the basis for major advances in automatic optimization and parallelization of programs [Boulet et al. 1998; Feautrier 1992a; Feautrier 1992b; Lim 2001; Bondhugula et al. 2008a]. Many efforts have been invested to develop source-to-source compilers such as PoCC [PoCC 2012], Pluto [Bondhugula et al. 2008b] and CHiLL [Chen et al. 2008], which use a polyhedral framework to perform loop transformations. Nowadays, traditional compilers such as GCC, LLVM [Grosser et al. 2011] and IBM XL make use of polyhedral frameworks to compile for multicore architectures. Progress is also underway to extend the applicability of polyhedral techniques to dynamic, data dependent control flow [Benabderrahmane et al. 2010].

2.2. Loop transformations and code generation for GPUs

With the emergence of GPUs, the polyhedral model has been applied to develop efficient source-to-source compilers for them. Baskaran's C-to-CUDA [Baskaran et al. 2010] is the first end-to-end, automatic source-to-source polyhedral compiler for GPU targets. It is based on Pluto's algorithms and explicitly manages the software controllable part of the memory system. However, it remains a prototype and handles only a small set of benchmarks. Building on Baskaran's experience, Reservoir Labs developed its own compiler based on R-Stream [Leung et al. 2010], which introduces a more

advanced algorithm to exploit the memory hierarchy. R-Stream appears to be using a more advanced scheduling algorithm than PPCG and automatically determines the required tile sizes, whereas PPCG currently expects these to be provided by the user. However, the paper glosses over many of the details, and no source code is available to inspect. For example, although R-Stream is presented as performing all the mapping phases at the level of the polyhedral model, most of them are explained in terms of ASTs. Unrolling is presented as a heuristic, while it is actually required for privatization. We could not find any details on how the copying to shared memory is performed, while in our experience, this can have a significant impact on the performance. A recent publication [Vasilache et al. 2012] provides more details about the scheduling algorithm, but not enough to enable a reimplementation; e.g., it is not clear whether a wavefront transformation is involved. Also, it does not explain how it ties in with the CUDA code generation. R-Stream appears to be using the same global memory synchronization as C-to-CUDA. Finally, the tool only appears to have been validated on two kernels (matrix multiplication and Gauss Seidel).

Gpuloc is a variation of the algorithm proposed by Baghdadi et al. [Baghdadi et al. 2010], which is also based on Pluto. It uses a ranking based technique [Größlinger 2009] to transfer data to and from shared memory, which results in inefficient accesses to memory that limit the overall performance. Moreover, at least one of the proposed implementations wastes memory and limits the GPU computation power. In addition, it is not in development anymore. The CHiLL developers also extended their compiler to generate GPU code — they introduced CUDA-CHiLL [Rudy et al. 2011], which does not perform an automatic parallelization and mapping to CUDA but instead offers high-level constructs that allow a user or search engine to perform such a transformation.

Unfortunately we were unable to obtain a copy of Reservoir Labs' R-Stream [Leung et al. 2010; Vasilache et al. 2012] or CUDA-CHiLL [Rudy et al. 2011] to provide a deeper assessment of these tools' capabilities.

Non-polyhedral tools have also seen major developments. Par4All [HPC Project 2012; Amini et al. 2011] is an open source initiative developed by Silkan to unify efforts concerning compilers for parallel architectures. It supports the automatic integrated compilation of applications for hybrid architectures including GPUs. The compiler is not based on the polyhedral model, but uses abstract interpretation for array regions, which also involves polyhedra; this allows Par4All to perform powerful interprocedural analysis on the input code.

2.3. Programming models

CUDA-lite [Ueng et al. 2008] started a trend to achieve better performance portability on CUDA programs. The user provides basic annotations about parallelism exploitation and deals with the offloading of live-in and live-out data. Based on these, the compiler performs loop tiling and unrolling to exploit temporal locality, local (shared) memory management, and attempts to help memory coalescing. In the same spirit, the PGI Accelerator language and framework [The Portland Group 2010] focuses on abstracting the performance details of the target and automating much of the tuning, aiming for the high-performance computing domain. In parallel, in the OpenMPC project, the Cetus compiler framework was used to translate OpenMP to CUDA [Lee et al. 2009]; unlike current polyhedral compilers, it can handle dynamic, data-dependent control flow. There has also been independent efforts to port OpenMP to heterogeneous architectures [Ferrer et al. 2010]. As a complementary approach, CAPS has been pushing for a more explicit programming model called HMPP, emphasizing the need for expert programmers to guide the optimization [HMPP 2010]. Influenced by all these trends,

the OpenACC language has recently been proposed as a unifying direction [OpenACC 2011]. OpenACC is now supported by PGI and CAPS in their commercial compilers.

Short vector SIMD architectures have also been the subject of active programming model research and development. Offload [Cooper et al. 2010] is a set of high-level C++ extensions, a compiler and a runtime system to accelerate kernels on the Cell BE. Intel designed the similar, and contemporary LEO programming model (Language Extensions for Offload), targeting its MIC accelerator architecture. Microsoft's AMP is also based on the same ideas [AMP 2011].

2.4. Performance modeling and tuning

GPGPU programming models such as CUDA and OpenCL expose application developers to a relatively abstract model of an SPMD many-core architecture. However, platform-specific details remain largely exposed: it is critical to utilize the hardware resources efficiently in order to achieve high performance.

Porting applications is complicated by the heterogeneity of GPU-accelerated platforms with distributed memory, and the restrictions of the GPGPU programming models. Ryoo et al. have shown that the optimization space for CUDA programs can be highly non-linear, and highly challenging for model-based heuristics, performance modeling, or feedback-directed methods [Ryoo et al. 2008b; Ryoo et al. 2008a]. G-ADAPT is a compiler framework to search and predict the best configuration for different input sizes for GPGPU programs [Liu et al. 2009]; it starts from already optimized code and aims to adapt the code to different input sizes.

The previous methods are not fully automatic or integrated into a GPGPU compiler. Progress towards fully automatic performance tuning have been achieved by the Crest [Unkule et al. 2012] and CHiLL [Chen et al. 2008] projects.

Rather than targeting a low-level programming model like OpenCL or CUDA, Apricot [Ravi et al. 2012] focuses on the automatic insertion of offload constructs, targeting LEO. It rounds some of the performance tuning difficulties by performing selective offloading at runtime. As the hardware and tool flows mature, we believe the performance tuning aspects will become more and more important and challenging.

2.5. Automatic vectorization

Fine-grain data level parallelism is one of the most effective ways of achieving scalable performance of numerical computations. It is pervasive in graphical accelerators, although some GPUs use different schemes to expose vector computations: current NVIDIA GPUs use multiple levels of fine-grain threads, while others use explicit short-vector instructions, and AMD has been using a combination of both. It is thus interesting to compare code generation for GPUs with traditional short-vector vectorization techniques.

Automatic vectorization for modern short-SIMD instructions has been a popular topic, with target-specific as well as retargetable compilers, for the ARM Neon, Intel AVX an SSE, IBM Altivec and Cell SPU. All of these works had a successful impact on production compilers [Wu et al. 2005; Bik 2004; Nuzman et al. 2006; Nuzman and Zaks 2008]. Exploiting subword parallelism in modern SIMD architectures, however, suffers from several limitations and overheads involving alignment, redundant loads and stores, support for reductions and more. These overheads complicate the optimization dramatically. Automatic vectorization was also extended to handle more sophisticated control-flow restructuring including if-conversion [Shin et al. 2005] and outer-loop vectorization [Nuzman and Zaks 2008]. Classical techniques of loop distribution and loop interchange [Wolfe 1996; Allen and Kennedy 2001; Trifunović et al. 2009] can dramatically impact the profitability of vectorization.

Leading optimizing compilers recognize the importance of devising a cost model for vectorization, but have so far provided only partial solutions [Wu et al. 2005; Bik 2004]. On the other hand, opportunities may be missed due to overly conservative heuristics. These state-of-the-art vectorizing compilers incorporate a cost model to decide whether vectorization is expected to be profitable. These models however typically apply to a single loop or basic-block, and do not consider alternatives combined with other transformations at the loop-nest level.

Combined loop-nest auto-vectorization and loop-interchange has been addressed in the context of vector supercomputers [Allen and Kennedy 1987; Wolfe 1996; Allen and Kennedy 2001]. Overheads related to short-SIMD architectures (such as alignment and fine-grained reuse) or GPUs were not considered until recently. Realignment and data-reuse were considered together with loop-unrolling [Shin et al. 2002] in the context of straight-line code vectorization. A cost model for vectorization of accesses with non-unit strides was proposed in [Nuzman et al. 2006], but it does not consider other overheads or loop transformations. The most advanced cost model for loop transformations-enabled vectorization was proposed by Trifunovic et al. [Trifunović et al. 2009]. It is based on polyhedral compilation, capturing the main factors contributing to the profitability of vectorization in the polyhedral representation itself.

These works indicate ways to improve our algorithms, taking into account profitability metrics to select loop transformations trading the exposition of fine-grain data parallelism and the exploitation of temporal and spatial locality. The algorithms in PPCG address these important problems, but do not yet include a clever profitability heuristic. This is left for future work.

3. GPU ARCHITECTURE MODEL

This work focuses on the NVIDIA's GPUs.

In order to expose the challenges that compilers have to face to produce efficient codes for these devices, and since code generation for accelerators tends to be highly target specific, we present an overview of the CUDA GPU architecture and programming model [NVIDIA Corporation 2011].

A GPU is a device that can be used as a high performance coprocessor, suitable for accelerating data parallel codes. The program running on the CPU (the host) must explicitly manage data transfers from host memory to device memory and vice versa, and can control the execution of programs on the device.

The architecture of a GPU is composed of several multiprocessors. Each multiprocessor is composed of several scalar processors that share a single instruction unit. The processors within a multiprocessor execute in lock-step, all executing the same instruction each cycle, but on different data. Each multiprocessor can maintain hundreds of threads in execution. These threads are organized in sets, called *warps*, the size of which is equal to the number of scalar processors that are in a multiprocessor. Every cycle, the hardware scheduler of each multiprocessor chooses the next warp to execute (i.e., no individual threads but warps are swapped in and out), using fine grain simultaneous multithreading to hide memory access latencies. This execution model is called Single Instruction Multiple Thread (SIMT) by Nvidia.

Regarding the memory hierarchy, all multiprocessors can access the same on-board DRAM memory (global memory in CUDA parlance) through a high bandwidth bus. This global memory is banked, which allows the hardware to coalesce several simultaneous memory accesses to adjacent positions into a single memory transaction. In addition, each multiprocessor contains an SRAM scratch pad, i.e., a software controlled local memory. In more recent GPUs (starting from the Fermi architecture) the SRAM can be configured as scratch pad or cache memory. The user can decide, with certain restrictions, the amount of cache and scratch pad needed. These newer GPUs also in-

corporate a L2 cache common to all multiprocessors. Finally, GPU multiprocessors can also access the global memory through a special read-only two level hierarchy of so called *texture* caches, that is optimized to capture 2D access patterns.

This model is exposed to the programmer or compiler by the CUDA driver. It allows to control the execution of a *kernel* on the device. A *kernel* consists of a sequential piece of code that has to be executed by a large set of threads on the GPU multiprocessors. Those threads are grouped into warps.[1] Threads within a warp are simultaneously executed on the scalar processors of a single multiprocessor in lock step. If the threads in a warp execute different code paths, only those that follow the same path can be executed simultaneously and a penalty is incurred.

Warps are further organized into a grid of *CUDA Blocks*: threads within a block are all executed on the same multiprocessor, and are then able to cooperate with each other by (1) efficiently sharing data through the shared low latency local memory and by, (2) synchronizing their execution via barriers. In contrast, threads from different blocks can be scheduled on different multiprocessors and thus they can only coordinate their execution via accesses to the high latency global memory. Within certain restrictions, the programmer specifies how many blocks and how many threads per block are assigned to the execution of a given kernel. When a kernel is launched, threads are created by hardware and dispatched to the GPU cores.

According to NVIDIA the most significant factor affecting performance is the bandwidth usage. Although the GPU takes advantage of multithreading to hide memory access latencies, having hundreds of threads simultaneously accessing the global memory introduces a high pressure on the memory bus bandwidth. Therefore, reducing global memory accesses, by using local shared memory to exploit inter thread locality and data reuse, largely improves kernel execution time. In addition, improving memory access patterns is important to allow coalescing of warp-wise memory accesses and to avoid bank conflicts on shared memory accesses.

To summarize, we can draw some conclusions on the challenges being faced when mapping code to this kind of devices. First, the compiler needs to partition the code into host and GPU code, knowing that not all parallel codes are suitable for the GPU. The parallel code pieces for the GPU must be mapped onto the CUDA model of blocks and threads. Here we have two different levels of parallelism, independent threads that are assigned to different blocks and cooperating threads, that are assigned to the same block forming warps. The latter should exhibit regular SIMD parallelism to avoid warp divergence and to minimize the number of non-coalesced memory accesses (threads in the same warp should access adjacent memory addresses). In addition, to reduce bandwidth requirements, data locality should be efficiently exploited in the register file and the local shared memories. This implies that compilers need to explicitly consider, schedule, and control data transfers between the different memory spaces. Doing so, it must aim to reduce the accesses to the global memory and unnecessary transfers. The existence of caches in some GPUs introduces an additional variable that should be considered. Overall, compiling for GPUs raise a number of program transformation and code generation challenges, and once these have been successfully addressed, compiler heuristics are faced with many degrees of freedom.

4. ILLUSTRATIVE EXAMPLE AND BACKGROUND

We will illustrate the functionality of PPCG on the classical matrix multiplication algorithm (reproduced in Figure 1), introducing the required background knowledge along the way. We will map this application onto a CUDA device using a mathematical abstraction of the program known as the polyhedral model. On a CUDA device, both the

[1]Currently, there are 32 threads per warp

```
void matmul(int M, int N, int K,
            float A[M][K], float B[K][N], float C[M][N])
{
    for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++) {
S1:         C[i][j] = 0;
            for (int k = 0; k < K; k++)
S2:             C[i][j] = C[i][j] + A[i][k] * B[k][j];
        }
}
```

Fig. 1. Matrix multiplication

threads in a block and the blocks in a grid are (at least conceptually) run in parallel. We therefore need to find two levels of parallelism, which can be obtained by applying the classical technique called *tiling*. This transformation is also instrumental in limiting the amount of memory used inside a block, to better exploit local memory resources.

Our first step is to extract a polyhedral model using `pet` [Verdoolaege and Grosser 2012]. Such a model mainly consists of an *iteration domain*, *access relation*s and a *schedule*, each of which described using affine constraints. The iteration domain contains the dynamic instances of the statements in the input program, where each instance is identified by the values of the enclosing loop iterators. The iteration domain of the example program is

$$(K, N, M) \to \{\, \mathrm{S2}(i, j, k) \mid 0 \le i < M \wedge 0 \le j < N \wedge 0 \le k < K \,\} \cup$$
$$(K, N, M) \to \{\, \mathrm{S1}(i, j) \mid 0 \le i < M \wedge 0 \le j < N \,\},$$

where K, N and M are parameters. The access relations map statement instances to the array elements accessed by those instances. In the example, the write access relation (simplified with respect to the iteration domain constraints) is

$$\{\, \mathrm{S2}(i, j, k) \to \mathrm{C}(i, j) \,\} \cup \{\, \mathrm{S1}(i, j) \to \mathrm{C}(i, j) \,\},$$

while the read access relation is

$$\{\, \mathrm{S2}(i, j, k) \to \mathrm{A}(i, k) \,\} \cup \{\, \mathrm{S2}(i, j, k) \to \mathrm{B}(k, j) \,\} \cup \{\, \mathrm{S2}(i, j, k) \to \mathrm{C}(i, j) \,\}. \tag{1}$$

Finally, the schedule specifies the order in which the statement instances are executed. An integer tuple is associated to each instance and the instances are executed in the lexicographic order of these tuples. The following schedule reflects the original execution order:

$$\{\, \mathrm{S2}(i, j, k) \to (i, j, k, 1) \,\} \cup \{\, \mathrm{S1}(i, j) \to (i, j, 0, 0) \,\}. \tag{2}$$

The next step is to look for parallel loops, so that they can be tiled and mapped to the blocks and threads. In general, as explained in Section 6, we may have to reorder the executions to obtain parallel and/or tilable loops. In this example, it is clear that the outermost loops can be executed in parallel because their iterations update disjoint parts of the `C` array. Furthermore, the three loops can be permuted without changing the semantics of the program, which is a sufficient condition for tiling. We can therefore tile the three loops, meaning that we execute the iterations of the loops in chunks. Only after executing all iterations in one chunk do we move on to the next chunk. A tiling where the size of a tile is 16 in each direction can be obtained by combining the original schedule (2) with the tiling map

$$\{\, (i, j, k, s) \to (\lfloor i/16 \rfloor, \lfloor j/16 \rfloor, \lfloor k/16 \rfloor, i \bmod 16, j \bmod 16, k \bmod 16, s) \,\}, \tag{3}$$

where $\lfloor x \rfloor$ represents the greatest integer part of x. After tiling, we have 6 loops, four of which are derived from the two parallel loops in the original program and are therefore also parallel. In principle, we could map the first two of these loops directly to the blocks in the grid and the remaining two to the threads in a block. However, this would require the tile sizes to match the number of threads in a block exactly. We prefer to keep these choices separate and instead map the remainder of division by the grid or block size to the block or thread identifiers. For example, if we choose a grid of 16×16 blocks, each with 8×16 threads, then we would intersect the range of the tiling map (3) with the set

$$(b_0, b_1, t_0, t_1) \to \{ (I, J, K, i, j, k, s) \mid \exists \alpha_0, \alpha_1, \beta_0, \beta_1 : I = 16\alpha_0 + b_0 \wedge J = 16\alpha_1 + b_1 \wedge i = 8\beta_0 + t_0 \wedge j = 16\beta_1 + t_1 \}, \quad (4)$$

where $0 \leq b_0 < 16$, $0 \leq b_1 < 16$, $0 \leq t_0 < 8$ and $0 \leq t_1 < 16$ are parameters referring to the block and thread identifiers.

Finally, we need to determine where to place the data. Recall from Section 3, that a CUDA device has several memories. Here, we will concentrate on global memory, shared memory and registers. In this example, the same elements of the `A` and `B` arrays are used by different threads in the same block, so we choose to place them in shared memory. An element of `C`, on the other hand, is only ever accessed from a single thread, where it is accessed several times. We therefore place the element in a register. The final code, generated using `isl` [Verdoolaege 2010], is shown in Figure 2. Note that if the parameters are fixed or constrained to values smaller than 4096, then the code generator will automatically prune the outer two loops. Similarly, if the values of the parameters are multiples of the block or tile sizes, then many of the conditions disappear.

5. OVERVIEW

In this section we present an overview of the inner workings of PPCG, followed by a comparison with the most closely related tools.

5.1. A bird's eye view of PPCG

Several of the substeps in this overview have already been illustrated in Section 4.

(1) Model extraction. This step takes C code as input and produces a polyhedral model consisting of an iteration domain, access relations and a schedule. Note that this schedule is only used during the dependence analysis of Step 2. In particular, it is not used during the construction of the optimized schedule in Step 3. The extraction of a polyhedral model is fairly standard and in PPCG it is performed using `pet`.
(2) Dependence analysis. This step takes an iteration domain, access relations and a schedule as input and determines which statement iterations depend on which other statement iterations. Dependence analysis is also fairly standard and in PPCG it is performed using `isl`. It is explained in more detail in Section 6.1.
(3) Scheduling. In this step a new schedule is constructed.
    (a) Exposing parallelism and more tiling opportunities. This is necessary to exploit the multiple levels of parallelism and of the memory hierarchy of GPU accelerators. In PPCG, this step is performed using `isl` based on the "Pluto" algorithm. The differences with the standard Pluto algorithm and, in particular, the modifications that we applied to our implementation for use in PPCG are explained in Section 6.2
    (b) Map to host and GPU. In this step, we decide which part of the schedule is executed on the host and which part is executed on the GPU. See Section 6.3.

```
__global__ void kernel0(float *A, float *B, float *C,
                        int N, int K, int M)
{
  int b0 = blockIdx.y, b1 = blockIdx.x;
  int t0 = threadIdx.y, t1 = threadIdx.x;
  __shared__ float s_A[16][16];
  __shared__ float s_B[16][16];
  float p_C[2][1];

  #define min(x,y)    ((x) < (y) ? (x) : (y))
  for (int g1 = 16 * b0; g1 < M; g1 += 4096)
    for (int g3 = 16 * b1; g3 < N; g3 += 4096) {
      if (K >= 1)
        if (N >= t1 + g3 + 1 && M >= t0 + g1 + 1) {
          p_C[0][0] = C[(t0 + g1) * (N) + t1 + g3];
          if (M >= t0 + g1 + 9)
            p_C[1][0] = C[(t0 + g1 + 8) * (N) + t1 + g3];
        }
      for (int g9 = 0; g9 <= (K >= 1 ? K - 1 : 0); g9 += 16) {
        if (K >= g9 + 1) {
          if (N >= t1 + g3 + 1)
            for (int c0 = t0; c0 <= min(K - g9 - 1, 15); c0 += 8)
              s_B[c0][t1] = B[(g9 + c0) * (N) + t1 + g3];
          if (K >= t1 + g9 + 1)
            for (int c0 = t0; c0 <= min(15, M - g1 - 1); c0 += 8)
              s_A[c0][t1] = A[(g1 + c0) * (K) + t1 + g9];
        }
        __syncthreads();
        if (g9 == 0 && M >= t0 + g1 + 1 && N >= t1 + g3 + 1) {
          p_C[0][0] = (0);
          if (M >= t0 + g1 + 9)
            p_C[1][0] = (0);
        }
        if (M >= t0 + g1 + 1 && N >= t1 + g3 + 1)
          for (int c2 = 0; c2 <= min(K - g9 - 1, 15); c2 += 1) {
            p_C[0][0] = (p_C[0][0] + (s_A[t0][c2] * s_B[c2][t1]));
            if (M >= t0 + g1 + 9)
              p_C[1][0] = (p_C[1][0] +
                           (s_A[t0 + 8][c2] * s_B[c2][t1]));
          }
        __syncthreads();
      }
      if (N >= t1 + g3 + 1 && M >= t0 + g1 + 1) {
        C[(t0 + g1) * (N) + t1 + g3] = p_C[0][0];
        if (M >= t0 + g1 + 9)
          C[(t0 + g1 + 8) * (N) + t1 + g3] = p_C[1][0];
      }
      __syncthreads();
    }
}
```

Fig. 2. Generated CUDA code for matrix multiplication with minor editing

   (c) Tiling and map to blocks and threads. In this step, we exploit the parallelism and tiling opportunities exposed in Step 3a. See Section 6.4.
(4) Memory management.
   (a) Transfer to/from GPU. Section 7.1 explains how we transfer data to the GPU and back to the host.
   (b) Detection of array reference groups. In the next step, we will try to allocate some array elements to registers and shared memory, but we first need to determine which array references should be handled as one atomic group to make sure a consistent allocation is selected for reads and writes to the same element. This is explained in Section 7.2.
   (c) Allocation to registers and shared memory. The actual allocation is explained in Section 7.3.
(5) Code generation. Code generation takes a schedule and generates (C or CUDA) code that visits each element in the iteration domain in the order specified by the schedule. In PPCG, we use our own code generator, implemented in `isl` to perform this step. For the purpose of the present paper, our code generator is similar to `CLooG` [Bastoul 2004], except that it has support for nested code generations. In particular, `CLooG` was designed to take a complete schedule and generate a single piece of code. In our application, however, we need to generate several pieces of code, one for the host, one piece of top-level GPU code for each kernel and several pieces for the cores of the kernels and for transferring data to/from registers and shared memory. Each of these pieces (except the host code) depends on choices that have been made during the generation of the code at outer levels.

5.2. Comparison

We can complement this overview with a comparison of the design of PPCG with state-of-the-art parallelizers targeting the CUDA programming model. We focus on C-to-CUDA [Baskaran et al. 2010] and Par4All [HPC Project 2012] because of the availability of their source code. These will be used for the most systematic evaluation of PPCG.

We distinguish 4 stages in the parallelization and code generation problem:

— *Exposing parallelism*. Both C-to-CUDA and PPCG 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. On the other hand, Par4All uses a more restricted combination of loop fusion to improve locality and loop distribution to expose parallelism.
— *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 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 the largest possible region of static control code. As a result, successive 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 [Bondhugula et al. 2008b; Bondhugula 2012], although they currently evolve independently. Par4All uses the schedule provided in the input source.

In the second stage, the three tools look for a *parallel band* in their respective schedules. In this context, 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. Some important differences exist between the three tools in how this mapping is performed.

In Par4All at most three parallel dimensions are selected from the *parallel band*, which are then tiled. Tiling a dimension translates to strip-mining + interchange. Strip-mining divides the dimension into bands, or strips of iterations [Allen and Kennedy 2001]. It replaces the original dimension by two new dimensions, the first identifying the strip and the second one identifying the point inside the strip. Combined with interchange, these strips form multidimensional tiles. The outer dimensions on strips are called *tile dimensions* and the inner ones on points inside tiles are called *point dimensions*. 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. Unlike C-to-CUDA and PPCG, Par4All is capable of selecting tile sizes automatically at runtime, alleviating some of the limitations of static heuristics.

On the other hand, both C-to-CUDA and PPCG also tile the dimensions following the first two parallel ones. In some cases, the additional sequential *tile dimensions* (that translate to loops in the kernel code) before the parallel *point loops*, may increase 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 levels of tiling, making it sometimes inefficient.

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. As explained in Section 7.3, it decides 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, based on the following analysis:

— 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, while all three tools are capable of generating parametric code, Par4All is much better at detecting that a seemingly parametric input program is not actually parametric due to its powerful interprocedural analysis. This has the advantage of generating simpler kernel codes, significantly reducing the number of dynamic instructions executed by the kernel. 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, frequently inefficient, kernel codes. This could be solved by providing the tools with a more sophisticated algorithm in the *Mapping to the CUDA model* stage.

Let us now present our new algorithms and the design of PPCG in greater detail.

6. SCHEDULING

Scheduling consists of two main parts. We first need to expose parallel and tilable sequences of loops and then we need to tile those loops and map the parallel ones to blocks and threads. In the example of Section 4, the first step was not needed as the original schedule already exposed parallel and tilable loops.

6.1. Dependence analysis

Before we describe how to transform the program to expose parallel and tilable loops, let us first explain how to detect such loops using the polyhedral model. The desired properties are determined by the dependences in the program, which can be represented using a *dependence relation*. This relation maps statement instances $\mathbf{i}$, to statements instances that need to be executed after $\mathbf{i}$, either because they read a value written by $\mathbf{i}$ or because they overwrite a value read or written by $\mathbf{i}$. The dependence relation can be obtained using standard techniques [Feautrier 1991] and in the example of Section 4, we obtain

$$\{ \mathrm{S1}(i,j) \to \mathrm{S2}(i,j,0) \} \cup \{ \mathrm{S2}(i,j,k) \to \mathrm{S2}(i,j,1+k) \},$$

again simplified with respect to the domain constraints. Applying the schedule (2) to both sides of the relation and taking the difference between image and domain elements results in the *dependence distance*s $\{ (0,0,0,1), (0,0,1,0) \}$. Since we applied the original schedule to arrive at these dependence distances, they are lexicographically positive. That is, statement instances only depend on earlier statement instances. The initial two zeros in these two tuples indicate that the first two schedule dimensions (the `i` and `j` loops in the original program) are parallel as there is no dependence between distinct `i` or `j` iterations. Finally, since all the entries are non-negative, we can freely permute them and still arrive at lexicographically positive dependence distances. This means that we can permute the loops without changing the semantics and in particular that we can tile the loops.

6.2. Exposing parallelism and tiling opportunities

We resort to affine scheduling, transforming the loop nest into a tilable, parallel one thanks to a suitable single-valued multidimensional affine function.

Affine schedules map statement iterations from the iteration domain

$$S = (S_i)_{1 \le i \le d}$$

to logical dates in multidimensional time. More precisely, the schedule is affine on each part of the iteration domain corresponding to the iterations of a given statement. Since the schedule is single-valued, the dimension d of this multidimensional affine function, i.e., the number of affine functions, should be at least as large as the maximal loop depth in the original program. The affine functions S_i are constructed one by one in such a way that the corresponding dependence distances are non-negative, i.e.,

$$\Delta(S_i \circ F \circ S_i^{-1}) \geq 0, \tag{5}$$

with F the dependence relation and Δ an operator that maps a relation to the differences between image and domain elements. This property ensures the validity of the schedule and also the ability to freely permute the loops enclosing S_i. Moreover, S_i is constructed in such a way that it is linearly independent of previously computed parts of the schedule and such that the dependence distance is minimized. In particular, the dependence relation and the previously computed parts are used to construct an ILP problem over the schedule coefficients, which is then solved for a minimal dependence distance. In the optimal case, the dependence distance is zero and the loop corresponding to this schedule dimension is parallel. The sequence of S_is that are constructed in this way form what is known as a tilable band, or simply *band*.

It may, however, not always be possible to satisfy the condition in (5). In such a situation, we need to either split F into smaller pieces or remove those dependences from F that are already carried by the latest band. In both cases, the subsequently computed affine functions form one or more new bands that are separate from the band that was computed up to that point. In particular, if carried dependences are removed from F, a single new band is constructed, while if F is split, several new bands are constructed. A repeated application therefore results in a tree of bands.

Let us consider the two cases in more detail. Consider the graph with as nodes the statements in the original program and with an edge between two nodes if F maps some iterations of the source statement to some iterations of the target statement. This graph is called the dependence graph. The strongly connected components (SCCs) of this graph can be handled separately. We merely need to insert extra schedule functions that force a topological order of the SCCs. In the second case, we remove those dependences that are already covered by the current band, i.e., those for which the dependence distance is strictly greater than zero ($\Delta(S_i \circ F \circ S_i^{-1}) > 0$).

For details about the above "Pluto" algorithm for computing a tree of tilable bands, i.e., for computing the affine functions S_i, we refer to [Bondhugula et al. 2008a]. In PPCG, we use a variation of this algorithm, implemented in `isl` [Verdoolaege 2010]. First, it should be noted that the Pluto tool does not compute a tree of bands, but simply a sequence of bands. As to the actual algorithm, the main differences are that we allow negative coefficients and that we insist that at least one of the loops in a band is parallel. Due to the properties of the Pluto algorithm, the parallel loops will always appear first in a band. Since it may not always be possible to find a parallel loop, we use the classical Feautrier algorithm [Feautrier 1992b] as a back-up to construct a band with a single loop. Such a band will not be considered during tiling as it does not contain a parallel loop. We consider three strategies for breaking up the tree into SCCs. The first ("minimal fusion") splits the tree into SCCs as soon as possible. The other two ("maximal fusion") split the tree only when absolutely necessary. The difference between the two remaining strategies is that one will simply move on to next band, while the other ("maximize band depth") discards the current band and starts over in the split tree.

6.3. Map to host and GPU

After having computed a tree of bands, we look for the outermost bands containing at least one parallel loop. Note that the positions of these bands may be different for different groups of statements. The part of the schedule before these bands is run on the host and a separate kernel is created for each band. Each of these kernels runs the partial schedule consisting of the given band and all its descendants. In the example of Section 4 there is only a single band and it starts at the very beginning. No loops are therefore generated on the host and a single kernel is created.

6.4. Tiling and map to blocks and threads

Within each kernel, we tile the band as explained in Section 4, based on user-specified tile sizes. This tiling splits the parallel loops into a pair of loops, one "tile" loop iterating over tiles and one "point" loop iterating inside tiles. The outermost (up to) two parallel tile loops are mapped to the blocks in the grid, while the innermost (up to) three parallel point loops are mapped to the threads in a block. This mapping process is performed as explained in Section 4. Again, the block and grid sizes are specified by the user. In CUDA, the thread identifiers are called `x`, `y` and `z`. The `x` identifier determines coalescing opportunities. Since it is easiest to analyze coalescing on the innermost dimension, we map the innermost of the selected parallel loops to this `x` identifier.

7. MEMORY ALLOCATION

7.1. Transfer to/from GPU

Since the memory space of a CUDA device is separate from the memory space of the host, we first need to allocate space on the device for the accessed arrays and copy data to and from the device. We currently follow a fairly simplistic but effective approach. Any accessed array is allocated in its entirety on the device. Any array that contains elements that are used without having been defined in the input program fragment is copied to the device. Any array that contains an updated element is copied from the device after all kernels have finished. Scalars that are updated by a kernel are treated as zero-dimensional arrays. Read-only scalars are passed as arguments to the kernels that read them.

7.2. Detection of array reference groups

Within a kernel, we may additionally want to copy parts of the global memory to shared memory or registers. We first group the array references that appear in the input code. In the example of Section 4, each array is only accessed through a single array reference (or two identical array references), but in general, different parts of an array may be accessed through different references and we may only want to place some of them in shared memory or registers. If, however, two references access some common elements and if at least one of the two references is a write, then the two references need to be considered together since placing only one of them in shared memory or registers could lead to inconsistencies. The initial reference groups are then the connected components in the graph with as nodes the array references and an edge between two nodes if the access relations (restricted to the tile loops) intersect and if at least one of the two references is a write.

7.3. Allocation to registers and shared memory

Recall that during scheduling we have tiled a tilable band, resulting in tile loops and point loops. Once the above initial grouping of array references has been performed, we consider the array elements accessed through a given group for each iteration of the tile loops. These elements are represented as a relation between the tile loop iterators

(together with all outer loop iterators that are mapped to the host) and the array indices. For example, for the single `A` reference in the program in Section 4, we obtain the relation

$$\{\,(I,J,K) \to \mathrm{A}(a_0,a_1) \mid 16I \le a_0 \le 15+16I \wedge 16K \le a_1 \le 15+16K \wedge \\ 0 \le J \le 15 \wedge 0 \le a_1 \le 255 \wedge 0 \le a_0 \le 255\,\}.$$

This relation can be obtained from the access relation (1), by applying the schedule (2) followed by the tiling (3) to its domain and subsequently projecting the domain onto its first three coordinates. Based on this relation we now need to find a constant-size block in the `A` array with an offset that may depend on the tile (and outer) loop iterators that covers the set of accessed elements, i.e., the image of the relation. If such a tile can be found, it can be copied to shared memory. The expression of the offset should be as simple as possible, to preserve the benefits of shared memory. In practice, we consider each lower bound on an index and take the one that leads to the smallest tile size. In the example, there are two lower bounds on a_0: $16I \le a_0$ and $0 \le a_0$. The first bound leads to a tile size of 16 (due to the upper bound $a_0 \le 15+16I$), while the second bound leads to a tile size of 256 (due to the upper bound $a_0 \le 255$). We therefore choose the first bound in this example. After computing such tiles for each reference group separately, we also compute them for pairs of reference groups for the same array, combining the groups into one if the resulting tile size is smaller than the sum of the individual tile sizes.

For mapping to registers, we consider the relation between the point loops (i.e., those that will be mapped to the threads) and the array elements, with the outer loops appearing as parameters. Abstracting away the constraints on these outer loops, we obtain the following relation for the `C` references in the program of Section 4:

$$(I,J,K) \to \{\,(i,j) \to C(c_0,c_1) \mid c_0 = 16I+i \wedge c_1 = 16J+j\,\}.$$

We first need to check that each element is only accessed by a single thread and we do so by checking that this mapping is injective. Note that this test is strictly speaking too strong because some of the domain elements may still be mapped to the same thread. Secondly, we check that the accessed element does not depend on any inner loops, i.e., those loops that have been projected out from this mapping. Again, we apply a slightly stricter test and check that the mapping is single-valued. The reason for this second check is that it ensures that the accessed element only depends on parallel loops. These loops can then be permuted innermost and completely unrolled. In particular, after having been permuted innermost, the loops are marked for unrolling and then effectively unrolled during the construction of the AST in the code generation phase. This unrolling ensures that the index expression is constant, a requirement for registers. Combining the conditions of the mapping being injective and single-valued, we see that we need to require the mapping to be a bijection.

If the access relation satisfies all the requirements, we compute a register tile size as in the case of shared memory. However, we first apply the mapping to threads, intersecting the domain with

$$(t_0,t_1) \to \{\,(i,j) \mid \exists \beta_0,\beta_1 : i = 8\beta_0 + t_0 \wedge j = 16\beta_1 + t_1\,\},$$

and projecting out the parallel loops. In the example, this results in the set (simplified with respect to the constraints on the parameters)

$$(t_0,t_1,I,J,K) \to \{\,C(c,t_1+16J) \mid 16I \le c \le 16I+15 \wedge (c-t_0) \bmod 8 = 0\,\}.$$

Naively applying the tile size computation would result in a 16×1 register tile. Instead we take the stride constraint ($(c-t_0) \bmod 8 = 0$) into account to obtain a 2×1 register tile.

ALGORITHM 1: Computing shared memory and register tiles

```
Compute initial reference groups;
for each reference group do
    Compute elements accessed by group per code tile;
    Compute shared memory array tile (rectangular overapproximation);
end
for each pair of reference groups do
    Compute elements accessed by pair per code tile;
    Compute shared memory array tile (rectangular overapproximation);
    if size of combined array tile is smaller than sum of individual array tiles then
        Merge the two reference groups into a single group;
    end
end
for each reference group do
    Compute elements accessed by code tile element;
    if mapping is a bijection then
        Compute register array tile (rectangular overapproximation);
    end
end
```

So far, we have explained how to compute the regions of global memory that can be mapped to shared memory or registers. A schematic overview of this computation is shown in Algorithm 1. We now describe how we decide whether or not to use these local memories. An important criterion is whether or not there is any reuse of the memory elements. In particular, we currently consider the reuse within a fixed iteration of the tile loops. A second criterion is coalescing. If threads with consecutive values for the `x` dimension access elements in global memory, then coalescing occurs and the accesses are optimized. If no coalescing occurs, then it can be advantageous to first copy the data in a coalesced way from global memory to shared memory. The decision on where to place data is then taken as follows.

— If we are able to compute register tiles and there is any reuse, then the data is placed in registers.
— Otherwise, if we are able to compute shared memory tiles and there is any reuse or the original accesses were not coalesced, then we place the data in shared memory.
— Otherwise, the data is kept in global memory.

Copying to and from shared memory is performed in such a way that consecutive threads (in the `x` dimension) access consecutive elements. Furthermore, copying is performed inside the innermost loop that affects the offset within the shared memory or register tile. For example, in Figure 2, the offsets of the shared memory tiles of `A` and `B` depend on the `g4` loop; hence copies are performed inside this loop, whereas the offset into the register tile of `C` does not depend on any loop and is therefore performed outside all loops. Finally, when data is copied from global memory to shared memory, we do not only copy the data that is actually needed, but instead copy the entire tile. This usually results in much simpler code, at the expense of copying a few extra elements.

8. EXPERIMENTS

We conducted extensive experiments to validate the generality of the approach, and to evaluate the quality of the generated code.

8.1. Methodology

We compare the performance of CUDA code generated by PPCG (ID d23b96b8 from PPCG's git repository) to OpenMP code generated by Pluto 0.7 and CUDA code generated by C-to-CUDA (Pluto 0.6.2) and Par4All 1.3. We were unable to obtain a copy of R-Stream for the purpose of an experimental comparison. The experiments were performed on a host with two 2.4 GHz Intel Xeon E5530 chips, each with four cores and HyperThreading, resulting in 16 virtual cores. Our main GPU is a Tesla M2070 (based on the Fermi architecture) and includes 14 multiprocessors @ 1.15 GHz, each with 32 cores, 32768 registers and 64KB of L1 memory. Since Par4All does not exploit the shared memory, we configured the L1 to favor the cache. The L1 memory was configured to provide 16KB of shared memory and 48KB of cache for the Par4All experiments and 48KB shared memory and 16KB cache for the C-to-CUDA and PPCG experiments. The GPU code was compiled using NVCC 4.0, whereas the CPU code was generated using gcc version 4.6 `-O3`. As test bed we used the matrix transpose algorithm and the PolyBench 3.1 suite.[2] The latter includes many common algorithms in fields such as linear algebra, data mining and image processing, areas where GPUs have been intensively used over the last few years. We configured PolyBench to use single precision for all our comparisons. The *symm* benchmark was slightly modified, expanding the `acc` variable into a two-dimensional array to expose coarse-grain data parallelism. Finally, all performance results were obtained from the mean execution time of 100 executions, taking into account the GPU initialization data transfer, and kernel execution time. We did not observe significant execution time variability across these different measurements.

Both Par4All and PPCG provide several compilation options to guide their operation. In this work we show the different options provided by PPCG. However, an exhaustive comparison of all possible values for Par4All would become unaffordably large and clearly out of the scope of this text. Thus, we chose the command line options that behave the best on average for all benchmarks. More precisely, we compiled all benchmarks with Par4All using the `--cuda` and `--com-optimization` options.

8.2. Results

We first compare the three PPCG scheduling strategies in Figure 3. It is important to remark that PPCG is the only tool under evaluation that can handle the complete PolyBench suite. Overall, most algorithms take advantage of the GPU architecture, showing relevant performance speedups when compared with sequential implementations. On the other hand, 10 of the benchmarks perform worse than their sequential CPU version. PPCG always map the entire benchmark onto the GPU (as it is instructed to do), so if the final schedule does not unveil GPU-like parallelism, performance degrades substantially. Most of those problematic benchmarks, listed in Figure 4, required skewing to unveil parallelism. Indeed, C-to-CUDA and Par4All were not able to find any parallelism in those benchmarks, except for *gramschmidt* and *adi* in the case of Par4All. Figure 4 includes a brief description of the main reason behind their poor performance. These are all algorithms with reduced exposed parallelism, and thus not good candidates for massively parallel platforms like the GPU. Wavefront transformations expose the existing parallelism but, in these specific benchmarks, lead to occasionally launching kernels with very few threads (even one thread in some cases) and unaffordable CPU-GPU synchronization due to embedding kernel calls into large sequential outer loops.

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

Regarding the PPCG scheduling strategies, we may conclude that *Maximal Fusion* is sometimes too aggressive, incurring large performance losses compared to the other scheduling strategies. A more in depth analysis of the code generated by PPCG for *2mm* shows that the maximal fusion strategy translates in a fusion of a parallel loop with a sequential loop, reducing the total amount of loop-level parallelism exposed. *Minimal Fusion* splits the loops as much as possible, which usually leads to a single CUDA kernel per loop nest and may therefore result in unnecessary accesses to/from global memory to pass data across different kernel calls. Nevertheless, this strategy shows to be more robust and in some particular cases, (e.g., *doitgen*) it exhibits the best performance. Finally, *Maximize Band Depth* shares some of the advantages of *Minimal Fusion* and usually outperforms *Maximal Fusion*. The results in Figure 3 were obtained using the default tile, grid and block sizes, and can therefore be considered as fully automatic. Taking advantage of the ability in PPCG for an expert user to specify these sizes manually, we were able to obtain an additional speedup of $2\times$ on average and $3.8\times$ in the extreme case. These values were selected based after an exhaustive search among a narrow set of sizes for each benchmark (typically multiples of the warp size). This study was based on our GPU and PPCG knowledge. The results in the remainder of this section include this customization.

To start with the inter-tool comparison, Figure 5 shows the performance obtained for the matrix multiplication kernel (*gemm*) by different parallelizing compilers for varying problem sizes. As reference GPU implementation, we include the performance of the latest CUBLAS *gemm* implementation.[3] ATLAS [Whaley and Petitet 2005] is a heavily optimized parallel implementation of the BLAS library. We have run its *sgemm* routine on the 16 core machine mentioned above. Finally, as baseline, we executed a sequential version of the matrix multiplication compiled with Intel *icc* compiler. For PPCG, we consider a parametric version where the problem size is unknown at compile time and one where the problem size is known.

With the exception of the *mini* size, PPCG shows by far the best performance among all the tools under analysis, and matches the performance of manually optimized

[3]CUBLAS is a GPU-accelerated version of the complete standard BLAS library

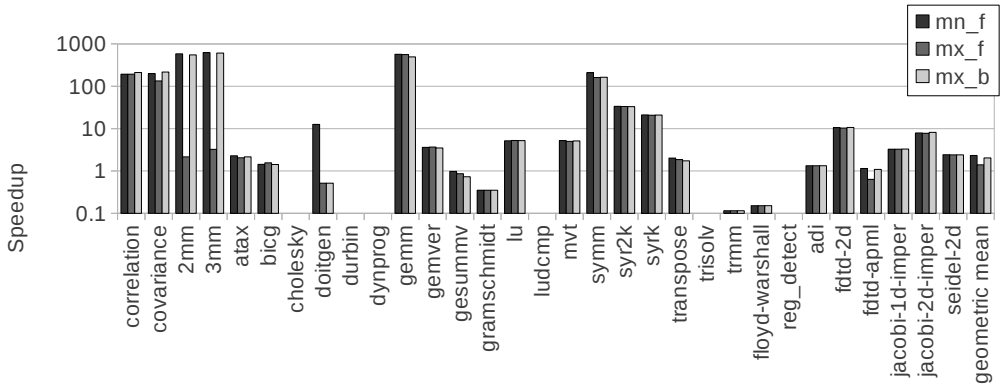

Fig. 3. Speedup over sequential CPU execution on Tesla M2070 of PolyBench using PPCG with default options and the three scheduling strategies: (mx_f) maximal fusion, (mx_b) maximize band depth and (mn_f) minimal fusion. Problem size: standard.

Name	Reason
cholesky	Requires scalar expansion to expose parallelism
durbin	PPCG does not find any parallelism, although some parallelism is present. However, this parallelism cannot be profitably exploited on a GPU
dynprog	Parallelism obtained through skewing leads to excessive CPU-GPU interaction
gramschmidt	Schedule does not unveil much parallelism and the generated code is extremely complex
ludcmp	No parallelism detected at all. Requires scalar expansion to expose parallelism
trisolv	Parallelism obtained through skewing leads to excessive CPU-GPU interaction. Computation too lightweight compared to data transfer
trmm	Parallelism obtained through skewing leads to excessive CPU-GPU interaction
floyd-warshall	Parallelism obtained through skewing leads to excessive CPU-GPU interaction. Irregular code induces highly divergent branches
reg_detect	Complicated generated code
adi	Complicated generated code

Fig. 4. Benchmarks performing worse than sequential after parallelization with PPCG

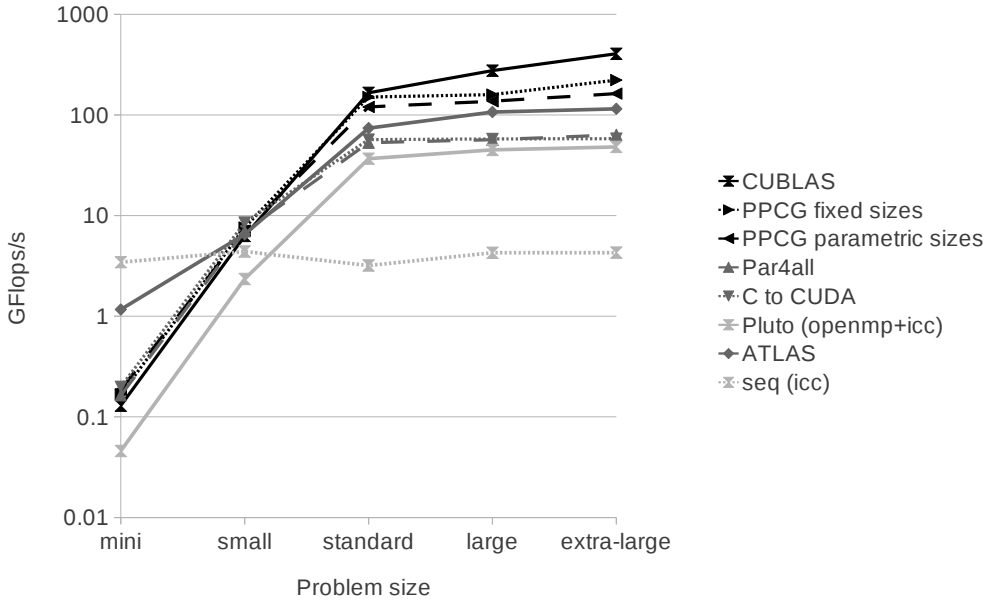

Fig. 5. Gemm performance comparison on Tesla M2070. Sizes as defined by the PolyBench suite: mini=32, small=128, standard=1024, large=2000 and extra-large=4000

CUBLAS implementation in several cases. In some cases however, code generated by PPCG can be two to three time slower than CUBLAS: this happens when the problem

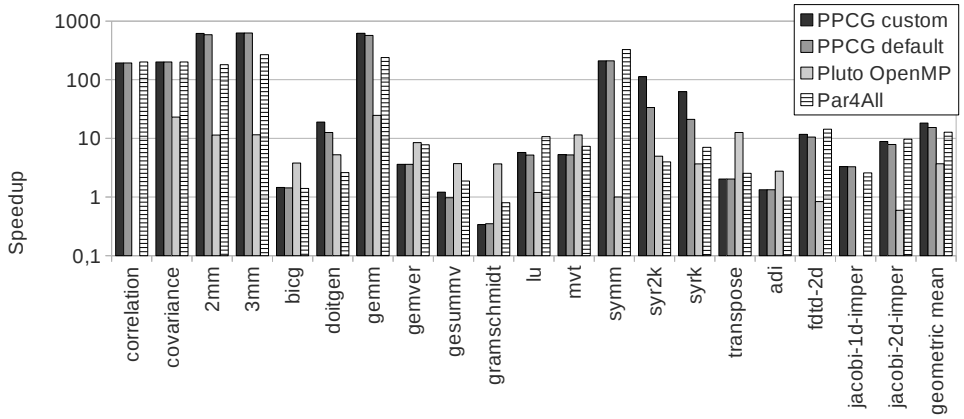

Fig. 6. Speedup over sequential CPU execution on Tesla M2070 of PolyBench for different compilers, using minimal fusion and both customization and default parameters for PPCG. Problem size: PolyBench standard.

size is not a multiple of the tile size. This is the result of control flow overhead introduced by the code generator and explains the performance hit of the parametric size experiment, as well as the scaling problem for large and extra-large matrices which are not multiples of the tile sizes in PolyBench. We expect this overhead can be eliminated with the implementation of full-tile/partial-tile separation on the code generator [Renganarayanan et al. 2007]. Although C-to-CUDA uses a scheduling strategy similar to PPCG, it makes a less efficient use of the memory hierarchy since it does not consider the register file. This induces severe performance losses compared to PPCG. Notice that Par4All and C-to-CUDA perform very similarly. Even if Par4All follows a much simpler mapping strategy with every thread computing a single element of the result matrix and it does not perform any explicit data management, the large L1 hardware cache introduced in Fermi architectures boosts its performance significantly.

Figure 6 shows an overall comparison of a subset of the benchmarks for default problem sizes. The results do not include C-to-CUDA as it fails to produce any code for most of the benchmarks. Moreover, Figure 6 does not show any of the 11 benchmarks that Par4All is not able to deal with (mainly because Par4All does not find any parallelism there). The figure shows that the algorithms that do not exhibit enough parallelism (*gramschmidt*), or those that show a low ratio between parallelism and memory footprint requirements (*bicg*, *gemver*, *gesumm* and *mvt*), are better mapped to OpenMP by Pluto. The *transpose* benchmark deserves a special mention: due to the computational simplicity of that benchmark, most of the time measured in the GPU implementations is spent on copying data to and from the GPU, while the OpenMP implementation does not require such a step. The graph shows two bars for PPCG results on each benchmark: the first one corresponds to the default tile sizes, and the second one to the custom, hand-tuned sizes.

Although Par4all does not exploit shared memory, it performs better than PPCG in 4 cases, taking advantage of the hardware L1 cache included in the Fermi architecture. Of course, performance of Par4all-generated code is dramatically lower for previous generations of GPUs which did not include L1 cache memory. On the other hand, PPCG performs better than Par4All in 7 cases where the hardware cache is not performing as well as the software-controlled shared memory exploited by PPCG.

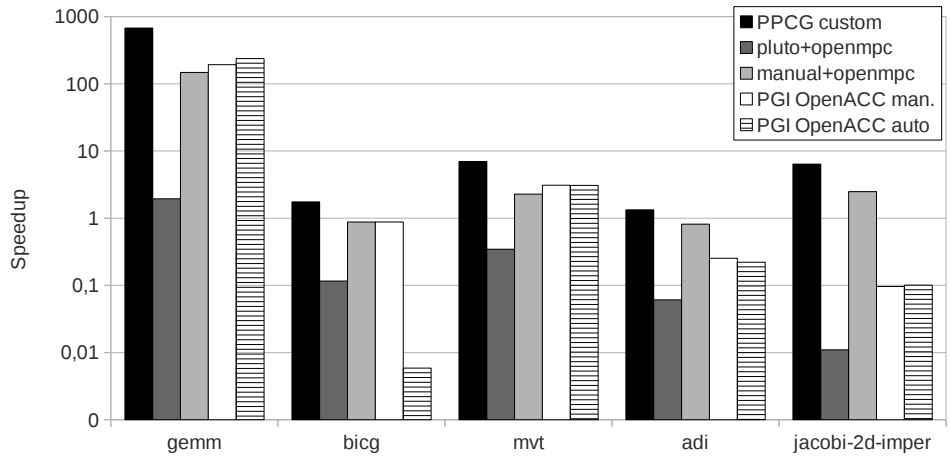

Fig. 7. Performance comparison between PPCG and two semi-automatic mapping tools: OpenACC and OpenMPC.

Finally, and for the sake of completeness, we compare PPCG with two of the most representative semi-automatic tools: the PGI compiler with support for OpenACC [OpenACC 2011] and OpenMPC [Lee and Eigenmann 2010]. When using these tools the programmer has to annotate manually the source code with some pragmas, that expose parallelism and might steer some data mapping. The tool then generates CUDA code for the GPU. The performance of the resulting mapping depends both on the quality of the code generated by the tool and the ability of the programmer in setting all the required pragmas. Of course, this makes these tools much more difficult to use compared to PPCG, C-to-CUDA or Par4All. For these reasons, we have selected as benchmarks only those PolyBench codes where no extra loop transformations were needed to uncover parallelism. These are, by coincidence, those codes where PPCG does not excel, with the exception of *gemm*. Furthermore, we have evaluated two different strategies: one in which the pragmas are placed manually by the programmer and an other where pragmas are placed automatically by a third application. For the later we chose Pluto, which was designed for automatic parallelization on SMPs. The aim for this is clear, it provides OpenACC and OpenMPC of an automatic parallel extraction stage and allows for a fair comparison with PPCG. Figure 7 shows the results. With few exceptions the strategy selected for pragma placement significantly influences the performance obtained. For OpenMPC manual placement was always better, while for OpenACC Pluto generally generates a slightly better input code, with one distinct exception, bicg, for which the manually annotated code was significantly better. These results illustrate the limitations of these semi-automatic tools, that heavily rely on the effectiveness of a previous pragma placement stage. On the other hand, PPCG showed the best performance in all cases compared to the full-automatic strategies (Pluto + OpenACC or OpenMPC). Only in one case, *adi*, the OpenMPC with manual pragma placement performed better than PPCG. The main reason for this is the complex code generated by PPCG for this particular benchmark.

9. CONCLUSION AND FUTURE WORK

We presented new algorithms to generate CPU and GPU code with one or more CUDA kernels, for any static affine loop nest. These algorithms include multilevel paralleliza-

tion methods, multilevel locality optimizations, and a code generator to manage memory and expose concurrency according to the constraints of modern GPUs. PPCG is the first polyhedral tool to compile the complete PolyBench suite to CUDA, outperforming SMP (OpenMP) execution in most cases and performing very well against the state-of-the-art GPU parallelizers.

Much optimization potential remains. Tiling is currently only applied to the outermost band of permutable loops. Enabling tiling of additional dimensions may further improve our ability to expose parallelism while simultaneously enhancing memory locality. Tile size selection is still not automatic; we are considering feedback-directed and analytical methods. Kernels are still offloaded sequentially to the GPU; we wish to hide memory transfers with asynchronous computations on the CPU and GPU, and we also wish to consider hybrid execution schemes combining CUDA with OpenMP on the CPU. We believe PPCG provides an ideal basis for exploring these important optimizations.

Acknowledgments. This work was partly supported by the European FP7 project CARP id. 287767.

REFERENCES

ALLEN, R. AND KENNEDY, K. 1987. Automatic translation of fortran programs to vector form. *ACM Tr. on Programming Languages and Systems 9,* 4, 491–542.

ALLEN, R. AND KENNEDY, K. 2001. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers.

AMINI, M., COELHO, F., IRIGOIN, F., AND KERYELL, R. 2011. Static compilation analysis for host-accelerator communication optimization. In *Workshop on Languages and Compilers for Parallel Computing (LCPC'11)*. LNCS. Springer-Verlag.

AMP 2011. C++ accelerated massive parallelism `http://msdn.microsoft.com/en-us/library/hh265137`.

BAGHDADI, S., GRÖSSLINGER, A., AND COHEN, A. 2010. Putting automatic polyhedral compilation for GPGPU to work. In *Proc. Compilers for Parallel Computer (CPC)*.

BASKARAN, M., RAMANUJAM, J., AND SADAYAPPAN, P. 2010. Automatic C-to-CUDA code generation for affine programs. In *Compiler Construction (CC 2010), Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings, Volume 6011 of Lecture Notes in Computer Science, pp. 244-263. Springer.*

BASTOUL, C. 2004. Code generation in the polyhedral model is easier than you think. In *PACT'04*. IEEE Computer Society, Washington, DC, USA, 7–16.

BENABDERRAHMANE, M.-W., POUCHET, L.-N., COHEN, A., AND BASTOUL, C. 2010. The polyhedral model is more widely applicable than you think. In *Proceedings of the International Conference on Compiler Construction (CC'10)*. Number 6011 in LNCS. Springer-Verlag, Paphos, Cyprus.

BIK, A. J. C. 2004. *The Software Vectorization Handbook. Applying Multimedia Extensions for Maximum Performance*. Intel Press.

BONDHUGULA, U. 2012. PLuTo: An automatic parallelizer and locality optimizer for multicores, version 0.7. `http://pluto-compiler.sourceforge.net/`.

BONDHUGULA, U., HARTONO, A., RAMANUJAM, J., AND SADAYAPPAN, P. 2008a. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Not. 43,* 6, 101–113.

BONDHUGULA, U., RAMANUJAM, J., AND ET AL. 2008b. PLuTo: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI) 08*.

BOULET, P., DARTE, A., SILBER, G.-A., AND VIVIEN, F. 1998. Loop parallelization algorithms: from parallelism extraction to code generation. *Parallel Comput. 24*, 421–444.

CHEN, C., CHAME, J., AND HALL, M. 2008. A framework for composing high-level loop transformations. Tech. rep., USC Computer Science.

COOPER, P., DOLINSKY, U., DONALDSON, A. F., RICHARDS, A., RILEY, C., AND RUSSELL, G. 2010. Offload - automating code migration to heterogeneous multicore systems. In *HiPEAC*. 337–352.

FEAUTRIER, P. 1991. Dataflow analysis of array and scalar references. *Int. J. Parallel Prog. 20,* 1, 23–53.

FEAUTRIER, P. 1992a. Some efficient solutions to the affine scheduling problem. Part I. one-dimensional time. *International Journal of Parallel Programming 21*, 313–347.

FEAUTRIER, P. 1992b. Some efficient solutions to the affine scheduling problem. Part II. multidimensional time. *International Journal of Parallel Programming 21*, 389–420.

FERRER, R., BELTRAN, V., GONZÁLEZ, M., MARTORELL, X., AND AYGUADÉ, E. 2010. Analysis of task offloading for accelerators. In *HiPEAC*. 322–336.

GROSSER, T., ZHENG, H., A, R., SIMBÜRGER, A., GRÖSSLINGER, A., AND POUCHET, L.-N. 2011. Polly: Polyhedral optimization in llvm. In *First Intl. Workshop on Polyhedral Compilation Techniques (IMPACT'11)*. Chamonix, France.

GRÖSSLINGER, A. 2009. Precise management of scratchpad memories for localising array accesses in scientific codes. In *CC'09*. Springer-Verlag, Berlin, Heidelberg, 236–250.

HMPP 2010. HMPP workbench: a directive-based multi-language and multi-target hybrid programming model `http://www.caps-entreprise.com/hmpp.html`.

HPC PROJECT. 2012. Par4All automatic parallelization version 1.3. `http://www.par4all.org`.

LEE, S. AND EIGENMANN, R. 2010. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '10. IEEE Computer Society, Washington, DC, USA, 1–11.

LEE, S., MIN, S.-J., , AND EIGENMANN, R. 2009. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In *Proc. Symp. on Principles and Practice of Parallel Programming*.

LEUNG, A., VASILACHE, N., MEISTER, B., BASKARAN, M., WOHLFORD, D., BASTOUL, C., AND LETHIN, R. 2010. 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*. GPGPU '10. ACM, New York, NY, USA, 51–61.

LIM, A. 2001. Improving parallelism and data locality with affine partitioning. M.S. thesis, Stanford University.

LIU, Y., ZHANG, E. Z., AND SHEN, X. 2009. A cross-input adaptive framework for gpu programs optimization. In *Proc. IEEE International Parallel & Distributed Processing Symp.*

NUZMAN, D., ROSEN, I., AND ZAKS, A. 2006. Auto-vectorization of interleaved data for SIMD. In *Proc. Conf. on Programming Language Design and Implementation (PLDI'06)*.

NUZMAN, D. AND ZAKS, A. 2008. Outer-loop vectorization - revisited for short SIMD architectures. In *Intl. Conf. on Parallel Architecture and Compilation Techniques (PACT'08)*.

NVIDIA Corporation 2011. *NVIDIA CUDA Programming guide 4.0*. NVIDIA Corporation.

OpenACC 2011. OpenACC: Directives for accelerators. `http://www.openacc-standard.org`.

PoCC 2012. PoCC: the polyhedral compiler collection version 1.1. `http://www.cse.ohio-state.edu/~pouchet/software/pocc/`.

RAVI, N., YANG, Y., BAO, T., AND CHAKRADHAR, S. 2012. Apricot: An optimizing compiler and productivity tool for x86-compatible many-core coprocessors. In *Intl. Conf. on Supercomputing (ICS'12)*.

RENGANARAYANAN, L., KIM, D., RAJOPADHYE, S., AND STROUT, M. 2007. Parameterized tiled loops for free. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*.

RUDY, G., KHAN, M. M., HALL, M., CHEN, C., AND JACQUELINE, C. 2011. A programming language interface to describe transformations and code generation. In *Proceedings of the 23rd international conference on Languages and compilers for parallel computing*. LCPC'10. Springer-Verlag, Berlin, Heidelberg, 136–150.

RYOO, S., RODRIGUES, C. I., BAGHSORKHI, S. S., STONE, S. S., KIRK, D. B., AND HWU, W.-M. 2008a. Optimization principles and application perform- ance evaluation of a multithreaded GPU using CUDA. In *Proc. Symp. on Principles and Practice of Parallel Programming (PPoPP'08)*.

RYOO, S., RODRIGUES, C. I., STONE, S. S., BAGHSORKHI, S. S., S. UENG, J. A. S., AND HWU, W.-M. 2008b. Optimization space pruning for a multithreaded GPU. In *Proc. Intl. Symp. on Code Generation and Optimization (CGO'08)*.

SHIN, J., CHAME, J., AND HALL, M. W. 2002. Compiler-controlled caching in superword register files for multimedia extension architectures. In *Intl. Conf. on Parallel Architecture and Compilation Techniques (PACT'02)*.

SHIN, J., HALL, M., AND CHAME, J. 2005. Superword-level parallelism in the presence of control flow. In *Proc. Intl. Symp. on Code Generation and Optimization (CGO'05)*.

The Portland Group 2010. *PGI Accelerator Programming Model for Fortran & C* v1.3 Ed. The Portland Group.

TRIFUNOVIĆ, K., NUZMAN, D., COHEN, A., ZAKS, A., AND ROSEN, I. 2009. Polyhedral-model guided loop-nest auto-vectorization. In *Intl. Conf. on Parallel Architecture and Compilation Techniques (PACT'09)*. Raleigh, North Carolina.

UENG, S., LATHARA, M., BAGHSORKHI, S. S., AND HWU, W.-M. 2008. CUDA-lite: Reducing GPU programming complexity. In *Proc. Workshop on Languages and Compilers for Parallel Computing (LCPC'08)*.

UNKULE, S., SHALTZ, C., AND QASEM, A. 2012. Automatic restructuring of gpu kernels for exploiting inter-thread data locality. In *International Conference on Compiler Construction (CC'12)*. Number 7210 in LNCS. Springer-Verlag.

VASILACHE, N., MEISTER, B., BASKARAN, M., AND LETHIN, R. 2012. Joint scheduling and layout optimization to enable multi-level vectorization. In *IMPACT'12*. Paris, France.

VERDOOLAEGE, S. 2010. isl: An integer set library for the polyhedral model. In *Mathematical Software - ICMS 2010*, K. Fukuda, J. Hoeven, M. Joswig, and N. Takayama, Eds. Lecture Notes in Computer Science Series, vol. 6327. Springer, 299–302.

VERDOOLAEGE, S. AND GROSSER, T. 2012. Polyhedral extraction tool. In *IMPACT'12*. Paris, France.

WHALEY, R. C. AND PETITET, A. 2005. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience 35,* 2, 101–121. `http://www.cs.utsa.edu/~whaley/papers/spercw04.ps`.

WOLFE, M. 1996. *High Performance Compilers for Parallel Computing*. Addison Wesley.

WU, P., EICHENBERGER, A. E., WANG, A., AND ZHAO, P. 2005. An integrated Simdization framework using virtual vectors. In *Intl. Conf. on Supercomputing (ICS'05)*.

Received June 2012; revised September 2012; accepted October 2012