Split Tiling for GPUs: Automatic Parallelization Using Trapezoidal Tiles to Reconcile Parallelism and Locality, avoiding Divergence and Load Imbalance

Albert Cohen, Tobias Grosser, Paul H. J. Kelly, J. Ramanujam, Ponnuswamy Sadayappan, Sven Verdoolaege

Split Tiling for GPUs: Automatic Parallelization Using Trapezoidal Tiles

Tobias Grosser
École Normale Supérieure
tobias.grosser@inria.fr

Albert Cohen
École Normale Supérieure
albert.cohen@inria.fr

Paul H J Kelly
Imperial College London
p.kelly@doc.ic.ac.uk

J. Ramanujam
Louisiana State University
jxr@ece.lsu.edu

P. Sadayappan
Ohio State University
sadayappan.1@osu.edu

Sven Verdoolaege
École Normale Supérieure
sven.verdoolaege@inria.fr

ABSTRACT

Tiling is a key technique to enhance data reuse. For computations structured as one sequential outer "time" loop enclosing a set of parallel inner loops, tiling only the parallel inner loops may not enable enough data reuse in the cache. Tiling the inner loops along with the outer time loop enhances data locality but may require other transformations like loop skewing that inhibit inter-tile parallelism.

One approach to tiling that enhances data locality without inhibiting inter-tile parallelism is split tiling, where tiles are subdivided into a sequence of trapezoidal computation steps. In this paper, we develop an approach to generate split tiled code for GPUs in the PPCG polyhedral code generator. We propose a generic algorithm to calculate index-set splitting that enables us to perform tiling for locality and synchronization avoidance, while simultaneously maintaining parallelism, without the need for skewing or redundant computations. Our algorithm performs split tiling for an arbitrary number of dimensions and without the need to construct any large integer linear program. The method and its implementation are evaluated on standard stencil kernels and compared with a state-of-the-art polyhedral compiler and with a domain-specific stencil compiler, both targeting CUDA GPUs.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processor—*Compilers, Optimization*

General Terms

Algorithms, Performance

Keywords

Polyhedral model, GPGPU, CUDA, code generation, compilers, loop transformations, index set splitting, time tiling, stencil

GPGPU-6 March 16 2013, Houston, TX, USA

1. INTRODUCTION AND RELATED WORK

Advances in technology over the last few decades have yielded significantly different rates of improvement in the computational performance of processors relative to the cost of a memory access. Because of the significant mismatch between computational latency and throughput when compared to main memory latency and bandwidth, the use of hierarchical memory systems and the exploitation of significant data reuse in the higher levels (i.e., closest to the processor) of the memory hierarchy are critical for high performance. Tiling is a key loop-level compiler transformation technique to enable high data reuse in higher levels of the memory hierarchy.

The effective use of cache and scratchpad memory in GPUs is critical for high performance, since the computational performance (in GFlops) on current GPUs is well over an order of magnitude higher than the aggregate bandwidth to global memory (in Gigawords per second). While a sizable number of applications have been developed for GPUs by application developers using CUDA and OpenCL, the expertise and effort needed to develop GPU applications that reach high performance is limiting its potential use. Therefore, there is a growing recognition of the need to provide higher level programming abstractions for GPUs if their use is to be broadened.

The recent development of the OpenACC standard [16] is an outcome of this recognition. With OpenACC, just as with the popular OpenMP model for shared-memory multi-core programming, the developer uses parallelization directives, retaining the control flow of a sequential program. The OpenACC compiler is responsible for (i) automatically off-loading the execution of the marked regions of code for execution on a GPU, (ii) handling the management of memory on the GPU, (iii) data transfers between the host and GPU memory, and (iv) the generation of effective GPU code. With current OpenACC compilers, especially when supported with user control over data regions, the compiler can be very effective in managing data transfers between CPU and GPU. However, many challenges remain in the automatic generation of effective GPU code for the marked OpenACC regions of the program.

Several recent efforts address the automatic parallelization of sequential code for GPUs [1, 2, 4, 23] and many of the techniques developed in these projects can be expected to be incorporated into OpenACC compilers. In this paper, we focus on a class of computations for which effective tiled code generation for GPUs remains a challenge. The computational pattern we focus on is that of *stencil* computations, characterized by an outer sequential "time" loop that iterates over a sequence of inner parallel loops that sweep

multi-dimensional grids with neighbor-based updates. When the multi-dimensional data grids are larger than cache capacity, untiled implementations of such codes result in repeated cache misses during each "time" loop iteration leadind to low performance that is limited by the available bandwidth to main memory. Thus, tiling also along the time dimension is critical to achieving high performance with such codes. Recent advances in polyhedral compilation and tools provide an effective solution to this problem for multi-core CPUs [3, 5]. However, the effective schemes for time-tiling on multi-core CPUs do not translate well to GPUs. Generation of high-performance time-tiled code for GPUs remains a challenge due to many issues, including the need to exploit high degrees of SIMD parallelism, limited synchronization capabilities in GPUs, the criticality of avoiding thread divergence, and the limited amount of cache and scratch-pad memory. Several recent studies focus on optimizing stencil computations for multicore CPUs and GPUs [3, 6–8, 10, 11, 14, 18–20]

Christen et al. [6] develop the PATUS stencil compiler that uses a stencil description and a machine mapping description to generate efficient CPU and GPU code. The GPU support of PATUS is reported to be still preliminary and, in contrast to our approach, it does not perform any time tiling. Strzodka et al. [18, 19] use time skewing with cache-size oblivious parallelograms to reduce memory system pressure and improve parallelism in stencils on CPUs, but do not address GPU-specific challenges. Han et al. [10] develop pattern-based optimization of stencil codes on CPUs and GPUs using a proposed extension to OpenMP. Micikevicius et al. [14] hand-tune a 3-D finite difference computation stencil. Datta et al. [7, 8] highlight the importance of auto-tuning and develop an optimization and auto-tuning framework for stencil computations, targeting multi-core systems, NVIDIA GPUs, and Cell SPUs. But Han et al., Micikevicius et al. and Datta et al. do not consider time-tiled implementations on GPUs. Nguyen et al. [15] develop a data blocking scheme that optimizes memory bandwidth and computation resources on GPU devices. Di and Xue [9] address tile size selection for GPU kernels related to stencils. However, these works do not address automatic code generation for GPUs.

Tang et al. [20] propose the Pochoir stencil compiler which uses a DSL embedded in C++ to produce high-performance code for stencil computations using cache-oblivious parallelograms for parallelism on x86 targets; they do not address issues specific to GPU code generation for stencil computations. Meng and Skadron [13] develop a performance model for the evaluation of ghost zones for stencil computations on GPU architectures, relying on user-provided annotations for GPU code generation; they do not consider fully automated code generation.

The standard approach to time-tiling of stencil computations requires skewing of the iteration space along the spatial dimensions relative to the time dimension. Skewing makes subsequent rectangular tiling along spatial as well as the time dimension legal, thereby enabling time-tiling and enhanced data reuse. While this alleviates the impact of the memory bandwidth constraint for the time-tiled code, inter-tile dependencies are created along the spatial dimensions, restricting inter-tile parallelism to be only along diagonal wavefronts in the iteration space. In this paper, we develop a novel static code generation approach for regular stencils targetting GPUs using *split* tiling. The idea of alternating between different phases was inspired by the approach of Strout et al. [17], which dynamically generates sparse tiles for irregular stencils. Both *split* and *overlapped* tiling have been proposed as alternative tiling techniques to avoid the loss of inter-tile parallelism along the spatial dimensions [12]. Overlapped tiling achieves this by introducing redundant computations. In contrast, split tiling does not require redundant computations, but introduces instead differently shaped tiles and the need for additional synchronization. Overlapped tiling is used to enhance tile-level concurrency on multicore systems by Krishnamoorthy et al. [12]. Holewinski et al. [11] use overlapped tiling to generate code for GPUs but do not consider split tiling. Hierarchical overlapped tiling is described by Zhou et al. [24], and evaluated in the context of OpenCL code generation but only for CPUs. To date, no compiler algorithms have been developed for the generation of effective split-tiled code for (multi-statement) stencil codes.

In this paper, we present a novel polyhedral approach for generating split-tiled code for GPUs. We propose a generic algorithm to calculate a schedule for index set splitting that enables us to perform tiling while simultaneously maintaining parallelism, without the need for skewing or redundant computations. Our algorithm performs split tiling for an arbitrary number of dimensions and without the need to construct any large integer linear programming problem instances. The method and its implementation are evaluated on several stencil kernels and compared with a state-of-the-art domain-specific GPU-stencil compiler. The newly developed algorithm is being incorporated into the publicly available PPCG tool flow [23].

2. ILLUSTRATIVE EXAMPLE

In this section, we use a simple example to elaborate on the problem addressed in this paper. The following simple computation kernel contains fine-grained parallelism at the inner loop dimension.

```
   for (t = 1; t < n; t++)
     for (i = 1; i < m - 1; i++)
R:     A[t][i] = A[t-1][i-1] + A[t-1][i+1];
```

The polyhedral representation (visualized in Figure 2) is:

$$D_R = \{A[t,i] : 1 \leq t < n \wedge 1 \leq i < m-1\}$$
$$S_R = \{A[t,i] \rightarrow [t,i]\}$$
$$P_R = \{A[t,i] \rightarrow A[t-1,i-1]; A[t,i] \rightarrow A[t-1,i+1]\}.$$

D_R describes the set of statement instances executed in this kernel, S_R maps each instance to an execution time and P_R describes the dependences that must be respected when changing the execution order of different statement instances. Details on the polyhedral model and its representation can be found in [21].

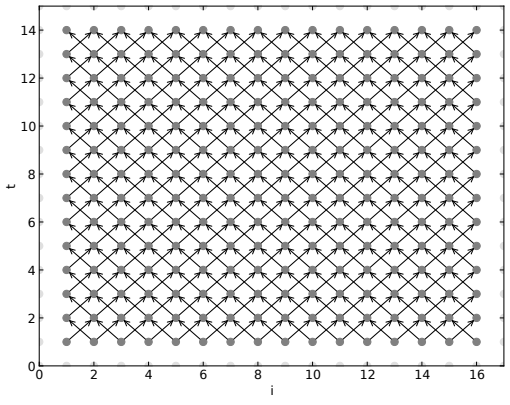

Figure 1: The iteration space of the simple kernel.

Listing 1: Wavefront parallel code

```
for (t1 = 0; t1 <= floord(2*n+m-4,32); t1++) {
  lb1 = max(ceild(t1,2), ceild(32*t1-n+1,32));
  ub1 = min(min(floord(n+m-3,32), floord(32*t1+m+29,64)), t1);

  #pragma omp parallel for shared(t1,lb1,ub1) private(t2,t3,t4)
  for (t2 = lb1; t2 <= ub1; t2++)
    for (t3 = max(max(1,32*t1-32*t2),32*t2-m+2);
          t3 <= min(min(n-1,32*t2+30),32*t1-32*t2+31); t3++)
      for (t4 = max(32*t2,t3+1); t4 <= min(32*t2+31,t3+m-2); t4++)
        A[t3][-t3+t4] = A[t3-1][-t3+t4-1]+A[t3-1][-t3+t4+1];;
}
```

In the given form, tiling with rectangular tiles in the 2D iteration space is not feasible because we would have mutual inter-dependences between spatially adjacent tiles. The standard approach to tiling such a code is to first skew the iteration space so that no dependences are oriented along the negative direction along any dimension. The Pluto algorithm developed by Bondhugula et al. [5] can generate such tiled code for any affine, imperfectly nested loop.

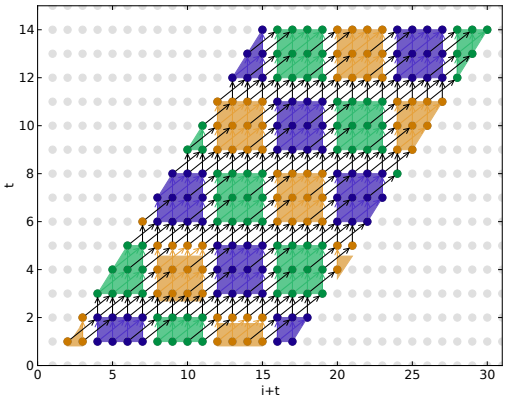

Figure 2: Pluto generated code for the simple kernel (tile size 3×4).

For our example this yields the code in Listing 1,[1] whose execution is depicted in Figure 2.

We can see that while iterations at a fixed time value are parallel for different values of i within a rectangular tile, there are inter-tile dependences between horizontally adjacent tiles. This inter-tile parallelism is not available along the i dimension. Inter-tile parallelism is only feasible among tiles in a common diagonal wavefront.

Such an automatic parallelization approach may be used effectively for multicore CPUs; however, it is not effective for GPUs, which require a very high degree of thread-level parallelism within a thread block and multiple concurrent thread blocks to utilize parallelism across the streaming multiprocessors in a GPU. We claim that wavefront parallelism is not optimal for GPUs, due to the low number of parallel tiles at the beginning and at the end of the computation. An even more effective approach using "diamond" tiles has very recently been developed for multi-core CPUs [3], but it remains to be evaluated on GPUs. As illustrated in the next section, the split tiling approach does not skew the iteration space but instead creates trapezoid shaped tiles of suitable shape so that they can be executed in a small number of phases. In each phase, we have a set of concurrently executable tiles with intra-tile parallelism that can be mapped to threads in a thread block.

[1] Generated with 'polycc –tile –parallel'.

3. SPLIT TILING OVERVIEW

Figure 3 illustrates split tiling for the previously discussed example. Two execution phases are needed for each time-tile. The green upright trapezoidal tiles in a time band can be executed in parallel. Each of these tiles is atomically executable since there are no incoming dependence edges from either neighboring purple tile. The only incoming dependences are from iteration points from a previous time band that have already completed. In mapping this to a GPU, different green tiles map to different thread blocks, and the points along the spatial dimension within each tile map to threads within a thread block. After all the green tiles in a time band have executed, all purple tiles in the same time band can execute concurrently, since all incoming dependences are either from points in the previous time band or points within a green tile in the same time band that have already executed.

Let us now provide an overview of the code generation approach for this simple example. Split tiling divides the iteration space into subspaces and tiles each subspace with a different schedule. For our example we define the subspaces T_G, the green tile space, and T_P, the purple tile space. The schedule for T_G is S_G and the schedule for T_P is S_P. The overall schedule S is a map where S_G is used for all elements in T_G and S_P is used for all elements in T_P. For each subspace we can now define a schedule that executes the convex subsets (our tiles) in parallel. S_G and S_P define such schedules. For our example we define T_G, T_P, S_P and S_P as follows (tile size 64×32):

$$
\begin{aligned}
T_G &= \{A[t,i] : (\exists i',t' : t' = t \bmod 32 \wedge i' = i \bmod 64 \\
&\quad \wedge i' - t' \leq 0 \wedge i' + t' \leq 64 - 2)\} \\
T_P &= \{A[t,i] : (\exists i',t' : t' = t \bmod 32 \wedge i' = i \bmod 64 \\
&\quad \wedge i' - t' < 64 \wedge i' + t' > 64 - 2)\} \\
S_G &= \{A[t,i] \to [t',0,i',t,i] : t' = \lfloor t/32 \rfloor * 32 \\
&\quad \wedge i' = \lfloor i/64 \rfloor * 64\} \\
S_P &= \{A[t,i] \to [t',1,i',t,i] : t' = \lfloor t/32 \rfloor * 32 \\
&\quad \wedge i' = \lfloor (i-31)/64 \rfloor * 64\}
\end{aligned}
$$

Applying the schedule S with a code generator like CLooG yields the code in Listing 2.

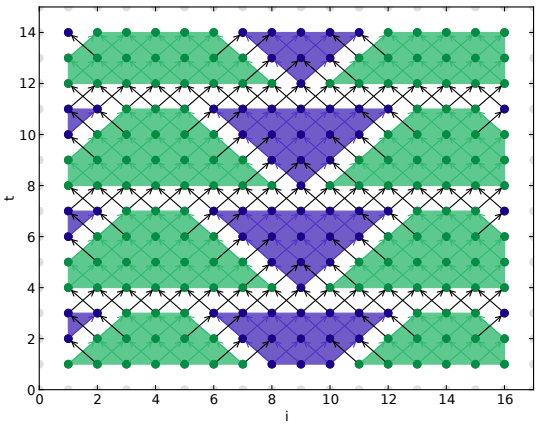

Figure 3: Split tiling for the simple example (tile size 10×4).

4. SPLIT TILING ALGORITHM

We present our new algorithm in steps, starting with the main idea applied to a single statement stencil, then generalizing it to multi-statement kernels, and refining the method with necessary optimizations. The algorithm is implemented in PPCG.[2]

4.1 Core algorithm

Given an iteration space and a set of dependences, we want to construct a schedule that implements split tiling. We start from a kernel where there is a single outermost loop that carries all dependences such that the remaining inner dimensions are fully parallel. Such a kernel is typical for a single statement stencil. As a first step, we partition the iteration space by placing equally distanced hyperplanes orthogonal to the axis of the time dimension. The different partitions form bands of fixed height (the height of the tiles). As the time dimension increases from band to band and as all dependences are carried by the time dimension, the bands can and must be executed sequentially. To obtain parallelism, we split the iterations within a single band into tiles of different colors, such that dependences may enforce an execution order between the different colors, but that within a single color all tiles can be executed in parallel.

To partition the band into different colors, we derive a tile shape for each color such that the full band can be covered with these shapes. The tile shape of the first color C_0 is constructed by choosing an arbitrary point X. X will be the apex of a pyramid that contains all iterations within the band that are needed to satisfy the (transitive) dependences of X. To construct this pyramid, we calculate the dependence distance vectors of the program and attach all of them to X. Together they form the upper part of a pyramid. We now extend the dependence vectors backward until their length along the time dimension matches the tile height we are aiming for. The convex hull of the extended dependence vectors forms a pyramid. This pyramid is the minimal set of points that we consider as the shape of the first color. In some cases it is preferable to have a shape that is wider along certain space dimensions. We can form such wider shapes by "stretching" the initial pyramid along these space dimensions. Stretching along a dimension means to position two copies of the original shape, such that the positions of the copies only differ in the dimension along which we stretch. The stretched shape is now the convex hull of the two shapes.

In addition to the first color, we derive one color for each space dimension in the input. The shape of a color C_x (where x corresponds to some space dimension) is derived by stretching the pyramid of C_0 along the x-dimension and by subsequently subtracting the shapes of all previously calculated colors.

In the case of more than one space dimension, additional colors are needed. Besides the initial color C_0 and the colors for individual dimensions, we introduce a color for each combination of dimensions. This means, for a 3D input, the colors C_{xy}, C_{xz}, C_{yz} as well as C_{xyz} are introduced. Their tile shapes are derived by stretching the initial pyramid along the set of dimensions they are named after. This can be compared to calculating the different faces of a cube, where the pyramid itself forms the shape of a vertex, the pyramids stretched along a single dimension form the differently oriented edges, the pyramids stretched along two dimensions form the facets and the pyramid stretched along all three dimensions forms the cube itself. Stretching the pyramid along more than one dimension (e.g., along x-y-z) is done recursively. We select one dimension (e.g., y) and calculate the union of the tile shapes that correspond to the colors of the remaining dimensions (here C_{xy}, C_x, C_z, C_0). This union is then replicated along the selected dimension, the convex hull of the entire construct is calculated, and finally the previous colors as well as their replicated copies are subtracted.

The split tiling schedule is constructed by tiling the original iteration space with the previously calculated tile shapes, such that the sequential execution of the different bands as well as of the different colors is ensured. Tiles of the same color and within the same band are mapped to parallel dimensions. The iterations within the tiles are executed according to the original schedule. As the index set splitting is calculated without considering the bounds of the iteration space, there is no constraint on the shape of the iteration space. Only as the very last step, we constrain the schedule to the actual iteration space.

The left part of Figure 4 shows a split tiling of the Jacobi 2D kernel. The pyramid that forms the first color was placed in the center of the iteration space. At time step one (the upper left illustration) the number of elements in the first color is still large. When going down to time step two and three we are moving up the pyramid such that the number of elements executed becomes smaller. At time step three, color one consists only of a single point, the summit of the pyramid. The shape of color two forms the connection between two vertical neighbors of color one. The shape is non-convex and resembles a simple butterfly. Color three now forms the horizontal connection between two neighboring shapes of color one. Color four, the last color constructed, fills the space enclosed by the previously calculated colors.

4.2 Tile shape simplification

The previously introduced split-tiling algorithm starts from a single pyramid that exactly covers the dependence vectors. Depending on the dependence vectors, such a minimal pyramid may not always be parallel to the axes of the iteration space. In case it is not, such as in Figure 4, subsequent tile shapes may have a non-convex form. Such non-convex tile shapes are undesirable, not only because they increase the required amount of communication between the different tiles, but they also introduce more complex control flow structures. To avoid such complex control flow structures, we normally widen the original pyramid to create a rectan-

[2]http://repo.or.cz/w/ppcg.git

Listing 2: Split tiled code

```
for (c1=0;c1<=M-1;c1+=32) {
  lb = 0;
  ub = min(N-2,c1+N-3);
#pragma omp parallel for shared(c1,ub,lb) private(c3,t,i)
  for (c3 = lb; c3 <= ub; c3+=64)
    for (t = max(1,c1); t <= min(min(M-1,c1+31),c1-c3+N-2); t++)
      for (i = max(1,-c1+c3+t); i <= min(N-2,c1+c3-t+62); i++)
        A[t][i] = A[t-1][i-1] + A[t-1][i+1];

 lb = max(-64,-64*floord(-c1+M-3,64)-64);
 ub = min(N-34,-c1+M+N-66);
#pragma omp parallel for shared(c1,ub,lb) private(c3,t,i)
  for (c3 = lb; c3 <= ub; c3+=64)
    for (t = max(max(max(1,c1),c1-c3-62),c1+c3-N+65); t <= min(M-1,c1+31); t++)
      for (i = max(1,c1+c3-i+63); i <= min(N-2,-c1+c3+t+63); j++)
        A[t][i] = A[t-1][i-1] + A[t-1][i+1];
}
```

gular base. This can avoid the construction of non-convex tiles. Figure 4 illustrates the tile shapes resulting from the widening of the original pyramid to a rectangular base. As can been seen, this leads to greatly simplified (and convex) shapes.

```
  for (t = 0; t < T; t++)
    for (i = 0; i < N; i++)
      A[t][i] += A[t-1][i+1] + A[t-2][i-3];
```

Figure 5: Example

4.3 Multi-statement loop nests

Up to this point, our split-tiling algorithm is only defined for kernels typical of single-statement stencils. In this section, we extend it to stencil computations that apply more than one statement in each iteration of the time loop. Such kernels have an additional sequential dimension with a known non-parametric number of iterations that models the lexicographic order of the different statements.

Figure 6 shows a simple two-statement kernel. Its iteration domain is $\{S[t,i] : 0 \leq t < T \wedge 0 \leq i < N; P[t,i] : 0 \leq t < T \wedge 0 \leq i < N\}$ and its execution order is defined by the following schedule $\{S[t,i] \rightarrow [t,0,i]; P[t,i] \rightarrow [t,1,i]\}$. The following dependences exist: $\{S[t,i-2] \rightarrow P[t,i]; P[t-1][i-1] \rightarrow S[t,i]; S[t-1][i] \rightarrow S[t][i]; P[t-1][i] \rightarrow S[t-1][i]\}$. Mapped into the scheduling space, this yields $\{[t,0,i-2] \rightarrow [t,1,i]; [t-1,1,i-1] \rightarrow [t,0,i]; [t-1,0,i] \rightarrow [t,0,i]; [t-1,1,i] \rightarrow [t,1,i]\}$. By analyzing the dependences, we see that the two outermost dimensions both carry loop dependences. This means our split-tiling algorithm is not directly applicable.

By applying a simple pre-transformation we can canonicalize the code such that it is again possible to use the previously presented split-tiling algorithm. We detect that only the outermost time dimension can have parametric size related to the number of time steps executed. The size of the second sequential dimension is independent of the number of executed time steps. As the second dimension represents the lexical order of the statements in the source code, its size is bounded by the number of statements in the source code. As the integer value of this bound is available at compile time, we can fold the two time dimensions into a single one. For a two statement kernel this transformation can be described by the following mapping $\{[t,1,i] \rightarrow [2t+1,i] : 0 \leq l \leq 1\}$. Applying this mapping on the original schedule gives us $\{S[t,i] \rightarrow [2t,i]; P[t,i] \rightarrow [2t+1,i]\}$ as well as the dependences $\{[2t,i-2] \rightarrow [2t+1,i]; [2t-1,i-1] \rightarrow [2t,i]; [t-2,i] \rightarrow [t,i]\}$. After this transformation, all dependences are again carried by the outermost dimensions and the inner parallel dimensions remain unchanged. Now, the previously presented split-tiling algorithm can be applied.

```
for (t = 0; t < T; t++) {
  for (i = 0; i < N; i++)
S:      A[1][i] += A[0][i+1]

  for (i = 0; i < N; i++)
P:      A[0][i] += A[1][i+2]
}
```

Figure 6: Two statement kernel

5. CUDA CODE GENERATION

To generate CUDA code, we extended the polyhedral GPU code generator PPCG [23]. PPCG is a state-of-the-art GPU code generator that translates static control programs in C to CUDA enabled programs. For certain classes of computations (e.g., linear algebra kernels), this produces very efficient code that reaches the performance of highly-tuned libraries. For stencil computations, PPCG performs a basic mapping where for each time step, a new kernel instance is spawned and each kernel applies a single stencil to just a couple of data points. This mapping exposes a high level of parallelism, but at each time step all data is read from and written to global memory. This means the global memory bandwidth becomes the limiting factor of the stencil computation.

By adding sipport for split-tiling, we enabled PPCG to produce time-tiled CUDA code for stencil like computations. Such code executes several iterations of the time loop within each kernel and keeps intermediate results in shared memory. This significantly lowers the pressure on the global memory bandwidth and consequently allows a higher computational throughput.

The split-tiling support for PPCG was developed by enhancing

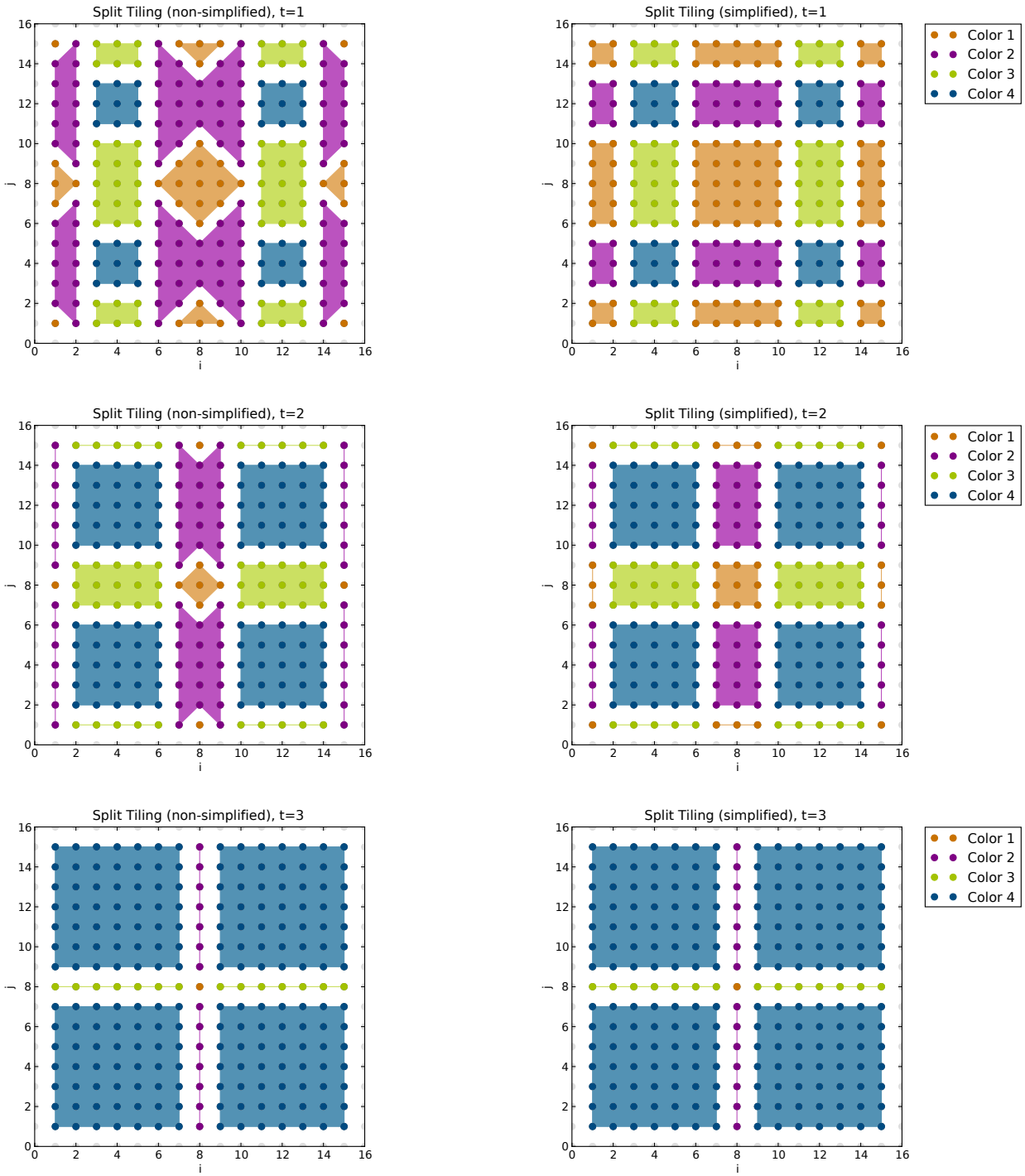

Figure 4: Split tiled jacobi-2d kernel

and parameterizing the polyhedral optimization infrastructure already available in PPCG. We specifically avoided the development of a new domain-specific code generator, but aimed instead at enhancing an existing GPU optimization framework. From a user's point of view, this provides a smoother experience as the same framework can be used for a wide range of kernels. The only difference is that it is now possible to obtain improved code for stencil computations. From the developer's point of view, the use of a uniform optimization framework has several benefits. Developing on top of an existing infrastructure speeds up the development of the CUDA code generator. It also enabled us to develop generic features and optimizations that can be beneficial for PPCG itself, but that show immediate benefits for split tiling if parameterized accordingly. The uniform framework makes it again very easy to

specify and communicate the necessary parameters to the relevant transformations.

When generating split-tiled CUDA code we start from the C code of the program. This code is read by the polyhedral extraction tool *pet* [22] which is available from within PPCG. Based on the extracted polyhedral description, we check if the program is suitable for split tiling. If this is the case, we derive a split-tiled schedule according to the generic algorithm described above. This new schedule is now provided to the generic PPCG transformation infrastructure where it replaces the PPCG internal schedule optimizer as well as the PPCG internal tiling. Instead, we parameterize PPCG with information about the schedule we provide. This information includes the number of dimensions of the entire schedule, the number of outer loops that should be executed on the host side, the first parallel loop that should be mapped to the GPU, the dimensionality of the tiles, the number of parallel dimensions in the tiles as well as information about the number of dimensions that should be considered when keeping values in shared memory.

PPCG uses this information to map the split tiles to the GPU. The mapping itself is rather straightforward. The tile loop of the time dimension is generally kept in the host code where it loops over a sequence of kernel calls. Each kernel call executes a set of thread blocks which in turn execute the parallel tiles as available at a certain time point using a one-to-one mapping from tiles to thread blocks. Within a tile, the parallel loops that enumerate the space dimensions are mapped to individual threads in a way that ensures coalesced memory accesses. The non-parallel loop for the time dimension is executed sequentially in each kernel. `__synchthreads` calls are introduced to ensure that each time step is finished before the next one is started.

5.1 Shared memory usage

The most important optimization for split tiling is the use of shared memory. The standard code that PPCG generates for stencils only uses shared memory to take advantage of special reuse within a single calculation. Such special reuse rarely happens for stencils and the additional synchronization overhead often outweighs the benefits of shared memory usage. However, with split-tiling, we can now take advantage of reuse along the time dimension. This means, all calculations within a single tile can be performed in shared memory. Only at the beginning and at the end of the kernel execution, the used values need to be loaded from and stored back into global memory.

5.2 Instruction level parallelism

On CUDA architectures several kinds of parallelism are available. Parallelism due to the execution of parallel threads is the most obvious one. However, even when generating code without split tiling, PPCG maps by default several data points to a single thread. Mapping several data points to a single thread ensures that there is a certain number of instructions between two subsequent `__syncthreads` calls. Exposing this instruction level parallelism is beneficial as it helps to hide memory access latency. Our split tiling implementation uses loop unrolling to increase the amount of available instruction level parallelism.

5.3 Full/partial tile separation

One way to avoid overhead due to the evaluation of boundary conditions is to use full/partial tile separation. The idea here is to generate specialized code for full tiles as well as for tiles that intersect with the iteration space boundary (i.e., partial tiles). Due to the absence of checks for the iteration space boundaries, the code of the full tiles evaluates a lot less conditionals. This is beneficial not only due to the reduced number of evaluated conditions, but also as it opens up new possibilities for loop-invariant code motion. Our split-tiling compiler automatically performs full/partial tile separation.

6. EVALUATION

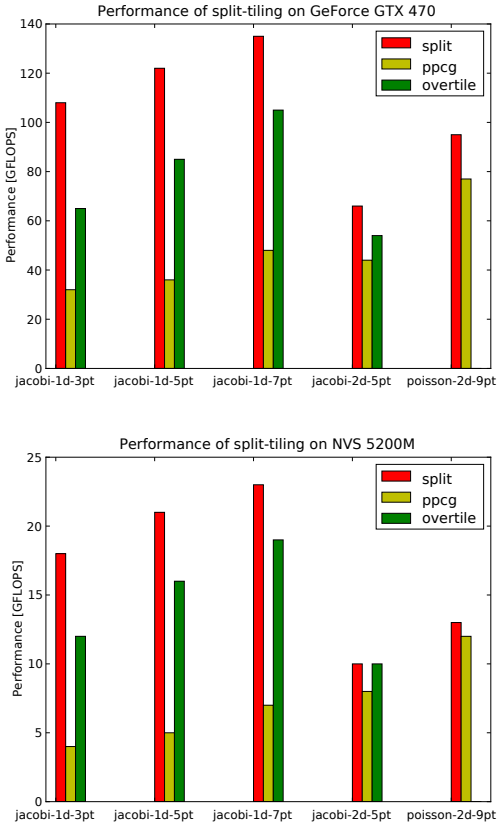

Figure 7: Performance results on different stencil compilers

To evaluate the performance of our split-tiling implementation we compare it against two compilers: (i) PPCG [23], a state-of-the-art GPU code generator; (and) (ii) Holewinski et al.'s [11] recent GPU stencil compiler, that we refer to as **`overtile`** (since it uses overlapped tiling).

We evaluate on a couple of different kernels consisting of jacobi-1d (3pt, 5pt, 7pt), jacobi-2d (5pt) as well as a 9pt poisson solver. The experiments are run on a NVIDIA GeForce GTX 470 desktop GPU as well as on a NVIDIA NVS 5200M mobile GPU. The performance is shown in GFLOPS and includes both the time of the computation itself as well as the data transfer to the GPU.

The results in Figure 7 show that the split-tiled code is consistently faster than both PPCG and **`overtile`**. Especially for the one dimensional test cases, we can see an almost 4x improvement over PPCG and between 30% and 40% over **`overtile`**. For the 2D cases on the mobile GPU, the performance of the different tools is again closer. On the desktop GPU, on the other hand, the difference between split tiling as well as PPCG and **`overtile`** is again considerable.[3]

[3]Unfortunately, we cannot report **`overtile`** numbers for the poisson solver, as nvcc 4 and 5 both crashed on the CUDA file that **`overtile`** produced.

7. CONCLUSIONS

We have presented a novel polyhedral method for generating split-tiled code for GPUs. It is based on an index-set splitting which is directly derived from dependence vectors, without the need of any additional algorithm to calculate valid tiling hyperplanes. Our algorithm has been implemented in a prototype extension of the PPCG tool flow. It is capable of tiling multi-statement stencils over both time and parallel loops, achieving excellent multi-level exploitation of the parallelism and local memory resources of a modern GPU. Future work includes the investigation of hybrid schemes for imperfectly nested loops, combining affine transformations with index-set splitting [3], evaluating the impact on a wider range of CPU and GPU targets, and providing a formal definition of the algorithms.

Acknowledgments. This work is partly funded by a Google European Fellowship in Efficient Computing, by the European FP7 project CARP id. 287767 as well as EPSRC (Project ref EP/I00677X/1).

8. REFERENCES

[1] M. Amini, F. Coelho, F. Irigoin, and R. Keryell. Static compilation analysis for host-accelerator communication optimization. In *Workshop on Languages and Compilers for Parallel Computing (LCPC'11)*, LNCS. Springer-Verlag, Oct. 2011.

[2] M. Amini, B. Creusillet, S. Even, R. Keryell, O. Goubier, S. Guelton, J. O. McMahon, F. X. Pasquier, G. Péan, and P. Villalon. Par4all: From convex array regions to heterogeneous computing. In *IMPACT'12*, Paris, France, Jan. 2012.

[3] V. Bandishti, I. Pananilath, and U. Bondhugula. Tiling stencil computations to maximize parallelism. In *Proceedings of SC '12*, pages 40:1–40:11, Los Alamitos, CA, USA, 2012. IEEE.

[4] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In *Proceedings of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction*, CC'10/ETAPS'10, pages 244–263, Berlin, Heidelberg, 2010. Springer-Verlag.

[5] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI'08, pages 101–113, New York, NY, USA, 2008. ACM.

[6] M. Christen, O. Schenk, and H. Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 676–687, Washington, DC, USA, 2011. IEEE.

[7] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. A. Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review*, 51(1):129–159, 2009.

[8] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of SC '08*, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[9] P. Di and J. Xue. Model-driven tile size selection for DOACROSS loops on GPUs. In *Proceedings of the 17th international conference on Parallel processing - Volume Part II*, Euro-Par'11, pages 401–412, Berlin, Heidelberg, 2011. Springer-Verlag.

[10] D. Han, S. Xu, L. Chen, and L. Huang. Pads: A pattern-driven stencil compiler-based tool for reuse of optimizations on gpgpus. In *ICPADS*, pages 308–315, 2011.

[11] J. Holewinski, L.-N. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 311–320, New York, NY, USA, 2012. ACM.

[12] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI'07, pages 235–244, New York, NY, USA, 2007. ACM.

[13] J. Meng and K. Skadron. A performance study for iterative stencil loops on gpus with ghost zone optimizations. *International Journal of Parallel Programming*, 39(1):115–142, 2011.

[14] P. Micikevicius. 3d finite difference computation on gpus using cuda. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 79–84, New York, NY, USA, 2009. ACM.

[15] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-d blocking optimization for stencil computations on modern cpus and gpus. In *Proceedings of SC '10*, pages 1–13, Washington, DC, USA, 2010. IEEE Computer Society.

[16] The OpenACC standard, 2011.

[17] M. M. Strout, L. Carter, J. Ferrante, J. Freeman, and B. Kreaseck. Combining performance aspects of irregular gauss-seidel via sparse tiling. In *LCPC*, pages 90–110, 2002.

[18] R. Strzodka, M. Shaheen, D. Pajak, and H.-P. Seidel. Cache oblivious parallelograms in iterative stencil computations. In *ICS*, pages 49–59, 2010.

[19] R. Strzodka, M. Shaheen, D. Pajak, and H.-P. Seidel. Cache accurate time skewing in iterative stencil computations. *2012 41st International Conference on Parallel Processing*, 0:571–581, 2011.

[20] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM.

[21] S. Verdoolaege. Counting affine calculator and applications. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Charmonix, France, Apr. 2011.

[22] S. Verdoolaege and T. Grosser. Polyhedral extraction tool. In *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12)*, Paris, France, Jan. 2012.

[23] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization(TACO)*, Dec. 2012. Selected for presentation at the HiPEAC 2013 Conf.

[24] X. Zhou, J.-P. Giacalone, M. J. Garzarán, R. H. Kuhn, Y. Ni, and D. Padua. Hierarchical overlapped tiling. In *Proceedings of the 10th Intl. Symp. Code Gen. and Opt.*, CGO '12, pages 207–218, New York, NY, USA, 2012. ACM.