Hybrid Hexagonal/Classical Tiling for GPUs

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

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

Justin Holewinski
NVIDIA Corporation
jholewinski@nvidia.com

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

Sven Verdoolaege
INRIA, École Normale Supérieure and KU Leuven
sven.verdoolaege@inria.fr

ABSTRACT

Time-tiling is necessary for the efficient execution of iterative stencil computations. Classical hyper-rectangular tiles cannot be used due to the combination of backward and forward dependences along space dimensions. Existing techniques trade temporal data reuse for inefficiencies in other areas, such as load imbalance, redundant computations, or increased control flow overhead, therefore making it challenging for use with GPUs.

We propose a time-tiling method for iterative stencil computations on GPUs. Our method does not involve redundant computations. It favors coalesced global-memory accesses, data reuse in local/shared-memory or cache, avoidance of thread divergence, and concurrency, combining hexagonal tile shapes along the time and one spatial dimension with classical tiling along the other spatial dimensions. Hexagonal tiles expose multi-level parallelism as well as data reuse. Experimental results demonstrate significant performance improvements over existing stencil compilers.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance

Keywords

Polyhedral compilation, GPGPU, CUDA, code generation, loop transformations, time tiling, stencil

1. INTRODUCTION

Tiling is one of the most important loop transformations, allowing to exploit parallelism while enhancing data locality. Its importance grows with the widening gap between the combined computational throughput of chip multi-processors and the aggregate bandwidth to off-chip memory: grouping operations into tiles that exhibit temporal reuse is an essential technique to reduce off-chip memory transfer. There has been a long history of efforts to develop tiling algorithms for compilers [12, 26, 3]. Several publicly available research compilers implement advanced forms of tiling for affine loop nests [23, 1, 3, 14, 16], and some of them [23, 1] also perform automatic parallelization of sequential code to parallel code on GPUs.

Despite significant compiler advances in tiling, it still is a challenging problem to perform effective tiling of stencil computations for GPUs. Stencil computations involve the repeated updating of values associated with points on a multi-dimensional grid, using only the values at a set of neighboring points. Stencils represent an important computational pattern used in scientific applications in many domains including computational electromagnetism [18], solution of PDEs using finite difference or finite volume discretization [17], and image processing. While stencil computations expose significant amounts of parallelism across spatial dimensions, these dimensions are generally much larger than on-chip memories. Time-tiling "blocks" the computations over multiple time steps, making the intra-tile spatial domains small enough to fit into caches. But simple rectangular tiling is not acceptable when the dependence components along spatial dimensions have negative components. For stencils, the most common solution is to resort to parallel wavefronts, effectively skewing spatial dimensions w.r.t. time, so that all dependence components become positive. Unfortunately this inhibits inter-tile parallelism [13]. Other approaches such as split tiling [7, 13], overlapped tiling [11, 13], and diamond tiling [2] have been proposed to address the loss of concurrency when time-tiling stencil computations. However, as explained in the next section, all of the previously proposed approaches have some limitations and drawbacks in the context of GPU computing.

The paper makes the following contributions:

- A novel hybrid hexagonal/classical tiling approach that enables reuse along the time dimension while ensuring adequate thread-level parallelism and fully avoiding thread divergence.

- Unlike other specialized stencil compilers, our method builds on a general-purpose polyhedral optimizer (PPCG), complementing it with domain-specific tiling and code generation strategies for stencil computations on GPUs.
- Experimental results provided on a number of stencil benchmarks demonstrating consistent superiority over existing general-purpose (Par4All, PPCG) and domain-specific stencil compilers (Overtile, Patus) for GPUs.

The paper is organized as follows. Sec. 2 provides a high-level overview of our approach. Sec. 3 presents the computation of a hybrid hexagonal/classical tiled schedule and proves its correctness. Sec. 4 explains the integration of our work with the CUDA code generator of PPCG and how we use and modify it to translate the tiled schedule into efficient GPU code. Sec. 5 discusses related work on tiling and optimization of stencils on GPUs. Sec. 6 presents experimental results, and we conclude in Sec. 7.

2. OVERVIEW OF APPROACH

An effective tiling scheme for GPUs must address a number of constraints. Unrolled inner loops must be carefully specialized to avoid divergent control flow among threads, minimize cumbersome address computations, effectively exploit register reuse, access shared memory instead of global memory as often as possible while avoiding bank conflicts, and achieve coalesced transfers for essential global memory accesses.

Figure 1 shows a 2D Jacobi stencil in source form, and Figure 2 shows the core of the PTX code, as generated by our tool and extracted from the CUDA compiler. This highly tuned block is free of control flow, performs only 3 shared memory loads and 1 store for 5 compute instructions, no global memory access, and 2 out of the 5 values in flight are being reused in registers across sequential time steps.

```
for (t=0; t < T; t++)
 for (i=1; i < N-1; i++)
  #pragma ivdep
  for (j=1; j < N-1; j++)
   A[(t+1)%2][i][j] = 0.2f * (A[t%2][i][j] +
        A[t%2][i+1][j] + A[t%2][i-1][j] +
        A[t%2][i][j+1] + A[t%2][i][j-1]);
```

Figure 1: Jacobi 2D stencil

```
ld.shared.f32   %f361, [%rd10+8200];
add.f32         %f362, %f353, %f361;
add.f32         %f363, %f362, %f345;
ld.shared.f32   %f364, [%rd10+7656];
add.f32         %f365, %f363, %f364;
ld.shared.f32   %f366, [%rd10+7648];
add.f32         %f367, %f365, %f366;
mul.f32         %f368, %f367, 0f3E4CCCCD;
st.shared.f32   [%rd10+1624], %f368;
```

Figure 2: Generated PTX (CUDA bytecode)

Generating such optimized core loops and thread code is a significant challenge, especially for higher-dimensional stencils. We address this challenge by developing a sophisticated tiling scheme, paired with an advanced code generation strategy.

We choose a hybrid tiling scheme that combines hexagonal tiling on the outer dimension with classical tiling on all remaining ones. Like most tiling schemes, our approach enables *reuse along the time dimension* while ensuring *balanced parallelism*, but hybrid tiling also addresses issues that make other approaches difficult to use on GPUs. In contrast to overlapped tiling [11], we perform *no redundant computations* and more importantly we avoid reserving shared memory space for data used only in redundant computations. This is important to ensure a *high compute-to-memory ratio* for each tile. Our hexagonal tiling approach is closely related to diamond tiling [2], but has two important differences. First, diamond tiles always have a narrow peak, whereas the peak of hexagonal tiles is adjustable in width. For stencil codes, adjusting the width translates into an *adjustable set of iterations available for fine-grained parallelism.* The second difference is that for diamond tiling, even though all tiles may have identical shapes, the actual number of integer points may vary between different tiles (see [9] for details). This difference may induce control flow divergence, when the diamond peaks sometimes fall on an integer point and sometimes do not. Our hexagonal tiling ensures an *identical number of computations within each full tile.*

Since hexagonal tiling along all spatial dimensions is not required to achieve an adequate degree of coarse-grained parallelism across thread blocks, we combine hexagonal tiling on an outer spatial dimension with classical tiling along the other dimensions, thereby bounding the data footprint of tiles to enable *all temporary values to be kept in shared memory.* Also due to the use of classical tiling we can ensure that the width along the classical tiled dimension remains constant. By setting the tile width to a multiple of the warp size we can always ensure *full warp execution*, *stride one accesses* and *avoidance of bank conflicts.* Also, as tiles are now always offset by a multiple of the warp size, we can position them to always ensure *cache-line aligned loads.*

With our advanced code generation strategy we also exploit the fact that along the classical tiled dimension, tiles are executed in sequence. This enables them to be executed in the same kernel thread and thereby *exploit reuse between successive tiles.* This is by itself already beneficial, but the real benefit is that the set of values that need to be loaded per tile is now a multiple of the tile width, which when chosen to be a multiple of the warp size will ensure that we *always load full cache lines.* Finally, we want *no conditional execution* and *no thread divergence* in the core computation. To ensure this we parameterize our code generation to create specialized code for full and for boundary tiles separately and we extensively unroll the innermost loops.

The stencil-specific tiling scheme is incorporated into a general-purpose polyhedral compiler framework currently translating C input to CUDA or OpenCL output. Our tiling scheme is largely independent of the input and output language such that applying it to a low-level compiler IR is possible e.g., by using the polyhedral optimizer Polly [8] which transforms the LLVM intermediate representation. Similarly, the integration in a DSL compiler e.g., halide [15] is possible. We claim that expressing domain specific optimization on a high-level representation, portable between compilers, is essential to address the upcoming compilation challenges posed by the increased need for optimized compilation of domain specific languages, the wide variety of specialized hardware and the need to generate optimized code for a wide variety of programming languages (including legacy code). Even though our compiler makes it easy to optimize existing stencil codes written in C, we see the more important contribution in it paving the way for a smoother integration of general purpose and domain specific optimizations. We believe that a close integration of domain and target specific optimizations in general purpose compilers will become unavoidable due to the increasing diversity in hardware and software. Such a close integration is not only desirable for mixed codes, but it also allows different code generation schemes to benefit from common infrastructure and optimizations. We have already seen such benefits while developing this optimizer and we expect them to become even more relevant down the road.

3. THE HYBRID SCHEDULE

To calculate a hybrid hexagonal/classical schedule that can be mapped nicely to the CUDA execution model we take several steps. First, the input program is analyzed statically and translated into a polyhedral representation. This representation is then canonicalized for stencil computations. Next, from this abstract information we derive a hexagonal schedule as well as a set of classically tiled schedules. Finally, the individual schedules are then combined into a hybrid hexagonal/classical execution schedule that materializes the ordering of iterations in a hybrid hexagonal/classical tiling. In addition, we explain how the calculated description of our tile shapes can be used to select good tile sizes.

3.1 Polyhedral model

In the polyhedral model [6], control and data flow are abstracted using sets and relations defined by affine constraints. The main constituents of a polyhedral representation are the iteration domain, the access relations, the dependence relation and the schedule. The iteration domain contains all the statement instances, where a statement instance $L[t, i, j]$ is represented by the name of the statement L and the values of the surrounding loop iterators t, i, j (see Figure 1). The access relations map the statement instances to the array elements read or written by the instance. The schedule defines the relative execution order of the statement instances by mapping them to a single schedule space where the execution order is determined by the lexicographical order in the schedule space. The dependence relation consists of those pairs of statement instances such that the second statement instance depends on the first statement instance. This dependence relation can be computed from the iteration domain, the access relation and a description of the original execution order [5]. A dependence distance vector is the difference in the schedule space between a statement instance and a statement instance on which it depends.

3.2 Preprocessing

As a first step, we extract a polyhedral description from our input C program using `pet` [24], compute dependences using `isl` [22] and transform the polyhedral description into some canonical form that later simplifies the construction of the schedule. The C input can contain modulos, non-unit stride loops and piecewise affine expressions, the latter are useful for example to model boundary conditions. There is also no limit on the number of arrays in the kernel. Focusing on the algorithmic domain of stencil computations, we assume that the input program consists of an outer loop containing $k \geq 1$ perfect nests of loops such that none of the loops in these nests carry any dependences. That is, all dependences are either carried by the outer loop or connect instances from different loop nests. If these conditions are met, then we construct a schedule of the form $L_i[t, s_0, \ldots, s_n] \to [k \cdot t + i, s_0, \ldots, s_n]$, where i satisfying $0 \leq i < k$ reflects the order in which the loop nests appear inside the outer loop. If the loop nests have different nesting depths, then they are currently manually aligned. In the constructed schedule, all dependences are carried by the outer dimension $k \cdot t + i$, meaning that the remaining dimensions s_i are fully parallel.

More generally, we could use a general purpose optimizer such as Pluto [3] to construct such an initial schedule (i.e., one with a single outer sequential dimension followed by only parallel dimension). This would allow us to consider more general inputs, but is left for future work.

The hybrid tiling of Section 3.6 is applied on top of the initial schedule. This tiling consists of a hexagonal tiling along the time and the first space dimension as well as classical tilings along the inner dimensions. We first describe the hexagonal and the classical tiling individually and then show how they are combined into a hybrid tiling.

3.3 Hexagonal tiling

We build hexagonal tiles starting from a two dimensional schedule space $P = [t, s_0]$ and a set of dependences $D \subseteq P \times P$. We first describe the restrictions on the input problem, then we construct the hexagonal tile shape and derive from it a hybrid tiling schedule. Finally, we show that the algorithm computes a correct tiled iteration space and that it allows parallel execution of the inner tile dimension.

3.3.1 Constraints on input

We require that the lexicographic order of the iterations in P is a valid schedule and that all dependences in D are such that t, the outer dimension of the index space, carries all dependences. As a result, the inner dimension s_0 is fully parallel. Finally, we assume that the dependence distances in the s_0-direction are bounded by a fixed constant times the dependence distance in the t-direction, both from above and below. Essentially, this assumption corresponds to the fact that we are dealing with a stencil computation.

3.3.2 Hexagonal tile shapes

To derive the tile shape of our hexagonal tiling we calculate two valid tiling hyperplanes from our dependences and use those hyperplanes to construct a tile shape for a given height h and width w_0. We illustrate the process on a slightly contrived example that computes

```
A[t][i] = f(A[t-2][i-2], A[t-1][i+2]);
```

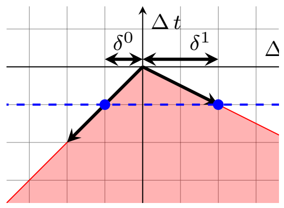

Figure 3: Opposite dependence cone

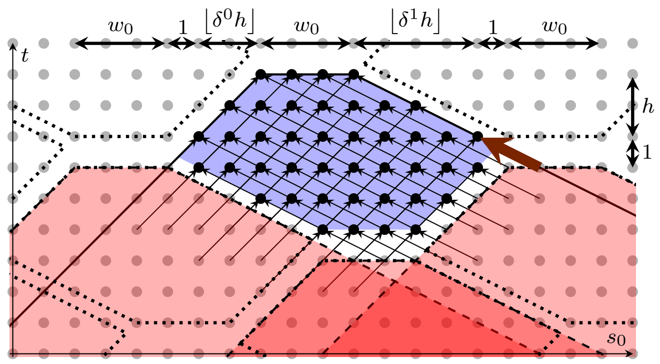

Figure 4: A hexagonal tile

We derive the tiling hyperplanes from the given dependences. We first compute the set of dependence distance vectors. In the example, we have $\{(1,-2);(2,2)\}$, meaning that the statement instances that directly depend on a given statement instance are executed in the original schedule at an offset $(\Delta t, \Delta s_0) = (1,-2)$ or $(2,2)$. Conversely, the opposites of these distance vectors are the offsets of statement instances on which the current statement instance directly depends. The cone generated by these opposite distance vectors is an over-approximation of the set of offsets of statement instances on which the current statement instance depends directly or indirectly. This cone (for the example) is shown as the red area in Figure 3. As we required the input to have strictly positive dependence distances in the first dimension, the cone lies entirely in the negative Δt half-space. Furthermore, because of our requirement of bounded distances in the s_0-direction, we can compute constants δ^0 and δ^1 such that $\Delta s_0 \le \delta^0 \Delta t$ (or, equivalently, $-\Delta s_0 \ge \delta^0(-\Delta t)$) and $\Delta s_0 \ge -\delta^1 \Delta t$. These constants can be computed through the solution of an LP-problem. Figure 3 shows the points $(-1,-\delta^0)$ and $(-1,\delta^1)$ in blue and the cone generated by these two points in red.

The basic idea is now that a tile will compute one or more s_0-instances at a given time step t together with all the instances on which it depends, except those that have already been computed by previous tiles. We therefore take w_0+1 instances at a given time step and construct a truncated cone that contains all the instances on which these selected instances depend by taking the union of the opposite dependence cones (the red cone from Figure 3) shifted to each of these instances. Figure 4 shows three such truncated cones in red, bounded by dashed lines. The blue tile shape is the result of subtracting these three truncated cones from the truncated cone bounded by solid lines. The offsets of the truncated cone have been carefully selected such that the entire space can be tiled using a single shape. In particular, the truncated cone on the left has offset $(-h-1, -w_0-1-\lfloor\delta^0 h\rfloor)$, the cone on the right has offset $(-h-1, w_0+1+\lfloor\delta^1 h\rfloor)$ and the cone on the bottom has offset $(-2h-2, \lfloor\delta^1 h\rfloor - \lfloor\delta^0 h\rfloor)$. The tiling is shown in dotted lines. In the figure, $w_0 = 3$ and $h = 2$. If there are multiple statements in the kernel, then choosing h such that $h+1$ is a multiple of the number of statements ensures that each tile starts with the same statement. To ensure that the result of the subtraction is a convex shape, the width w_0 has to be large enough. This is illustrated by the large brown dependence vector in Figure 4. If w_0 were equal to 1, then the result of the subtraction would contain an extra component to the right of the right truncated cone. Such extra components can be avoided by imposing

$$w_0 \ge \max\left(\delta^0 + \left\{\delta^0 h\right\}, \delta^1 + \left\{\delta^1 h\right\}\right) - 1, \tag{1}$$

with $\{x\}$ the fractional part of x, i.e., $\{x\} = x - \lfloor x\rfloor$. In the example, we have $w_0 \ge 1$. The correctness of (1) will be shown in Section 3.3.3.

3.3.3 Scheduling hexagonal tiles

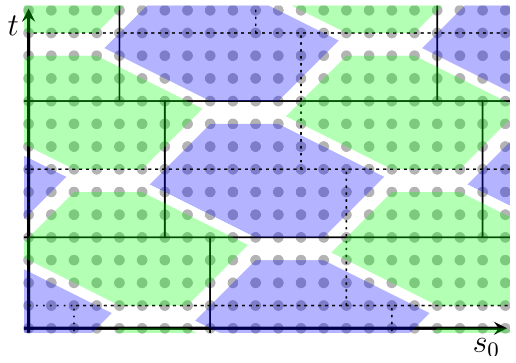

Figure 5: Hexagonal tiling pattern

The schedule of our hexagonal tiling maps the two iteration space dimensions $[t, s_0]$ into a three dimensional tile space $[T, p, S_0]$. The schedule alternates between two phases, 0 and 1. In particular, within each time tile T, the schedule first executes the blue tiles of Figure 5 (phase 0) and then the green tiles (phase 1). The tiles that belong to the same time tile and the same phase are indexed by S_0 and can be executed in parallel. In Figure 5 such tiles form a horizontal wavefront of identically colored tiles. For phase 0, we have

$$T = \lfloor (t+h+1)/(2h+2) \rfloor \tag{2}$$

$$S_0 = \left\lfloor \frac{s_0 + \lfloor\delta^1 h\rfloor + w_0 + 1 + T\left(\lfloor\delta^1 h\rfloor - \lfloor\delta^0 h\rfloor\right)}{2w_0 + 2 + \lfloor\delta^0 h\rfloor + \lfloor\delta^1 h\rfloor} \right\rfloor \tag{3}$$

while for phase 1, we have

$$T = \lfloor t/(2h+2) \rfloor \tag{4}$$

$$S_0 = \left\lfloor \frac{s_0 + T\left(\lfloor\delta^1 h\rfloor - \lfloor\delta^0 h\rfloor\right)}{2w_0 + 2 + \lfloor\delta^0 h\rfloor + \lfloor\delta^1 h\rfloor} \right\rfloor. \tag{5}$$

The difference in the numerators of the expressions for T ensures that the blue tiles belong to the same T-tile as the green tiles that have the same and greater t coordinates. Within this T-tile, the blue tiles are then executed before the green tiles. The other offsets are required to make all the tiles line up.

The (T, S_0)-coordinates refer to the boxes in Figure 5, the solid boxes for phase 0 and the dotted boxes for phase 1. To ensure that each (t, s_0) is only executed once, we only execute parts of these overlapping boxes. In particular, we execute the blue tile in each solid box and the green tile in each dotted box. To describe the hexagons, we use local coordinates (a, b) within each box. For example, for the green tiles, we have

$$a = t \bmod (2h+2)$$
$$b = s_0 + T\left(\lfloor \delta^1 h \rfloor - \lfloor \delta^0 h \rfloor\right) \bmod \left(2w_0 + 2 + \lfloor \delta^0 h \rfloor + \lfloor \delta^1 h \rfloor\right).$$

Using these local coordinates, the constraint of the top of the hexagons can be derived directly from the constraints of the opposite dependence cone. In particular, we have

$$\delta^0 a - b \le (2h+1)\delta^0 - \lfloor \delta^0 h \rfloor \tag{6}$$
$$a \le 2h+1 \tag{7}$$
$$\delta^1 a + b \le (2h+1)\delta^1 + \lfloor \delta^0 h \rfloor + w_0. \tag{8}$$

The remaining constraints are obtained from subtracting the earlier truncated cones. Let (a', b') be the local coordinates in the box at offset $(-h-1, -w_0 - 1 - \lfloor \delta^0 h \rfloor)$, i.e., $a' = a + h + 1$ and $b' = b + w_0 + 1 + \lfloor \delta^0 h \rfloor$. When subtracting the truncated cone associated to this box, we need to add the negation of the constraint

$$\delta^1 a' + b' \le (2h+1)\delta^1 + \lfloor \delta^0 h \rfloor + w_0, \tag{9}$$

i.e., $\delta^1 a + b \le h\delta^1 - 1$. Let d^1 be the denominator of δ^1. The negation of this constraint can then be written as

$$\delta^1 a + b \ge h\delta^1 - \frac{d^1 - 1}{d^1}. \tag{10}$$

In principle, we now also need to consider other pieces of the difference that satisfy (9), but that do not satisfy one of the other two constraints. Because of the vertical position of truncated cone we are subtracting it is impossible for there to be any integer points that lie in the original truncated cone, satisfy (9) and do not satisfy $a' \le 2h+1$. To verify that there can be no points in the current truncated cone that do not satisfy the constraint

$$\delta^0 a' - b' \le (2h+1)\delta^0 - \lfloor \delta^0 h \rfloor, \tag{11}$$

we again rewrite the constraint in terms of the current local coordinates and obtain

$$\delta^0 a - b \le (2h+1)\delta^0 - \lfloor \delta^0 h \rfloor + w_0 + 1 + \lfloor \delta^0 h \rfloor - \delta^0(h+1).$$

Due to our choice of w_0 in (1), we have $w_0 - \delta^0 - \{\delta^0 h\} + 1 \ge 0$, meaning that (11) is implied by the corresponding constraint on the original truncated cone.

The truncated cone at offset $(-h-1, w_0 + 1 + \lfloor \delta^1 h \rfloor)$ similarly yields the constraint

$$\delta^0 a - b \ge \delta^0 h - \lfloor \delta^0 h \rfloor - w_0 - \lfloor \delta^1 h \rfloor - \frac{d^0 - 1}{d^0}, \tag{12}$$

with d^0 the denominator of δ^0. Finally, the box at offset $(-2h-2, \lfloor \delta^1 h \rfloor - \lfloor \delta^0 h \rfloor)$ yields the constraint

$$a \ge 0. \tag{13}$$

3.4 The classical tile schedule

In the remaining spatial dimensions, we apply a more traditional form of tiling. This means that we lose parallelism along these dimensions, but it allows to reduce the working set within each tile. Each spatial dimension s_i with $i \in [1, n]$ is stripmined separately. Just like hexagonal tiling (see Figure 3), one computes the projection of the dependence cone onto the time dimension and the given spatial dimension s_i. Yet in this case, we only need to consider dependences on statement instances with higher values for the spatial dimension. This means that we only need to compute δ_i^1 and that therefore the dependence distance in the spatial dimension only needs to be bounded in terms of the distance in the time dimension from below. The resulting tile shape is a parallelogram with sides that are parallel to the corresponding side of the opposite dependence cone. Since this tiling needs to be combined with the hexagonal tiling, the height of these tiles is equal to $2h+2$. The width can be independently chosen as w_i. In sum, the corresponding tile dimension is given by

$$S_i = \lfloor (s_i + \delta_i^1 u)/w_i \rfloor, \tag{14}$$

where u is a normalized version of t that ensures that the starting positions of the tiles in the spatial direction are the same for all time tiles and for both phases. That is, we set

$$u = (t + h + 1) \bmod (2h+2) \quad \text{for phase 0 and} \tag{15}$$
$$u = t \bmod (2h+2) \quad \text{for phase 1.} \tag{16}$$

The above normalization is beneficial in two ways. First, the generated code is simpler because the offset is a constant instead of an expression that needs to be (re)calculated at each time tile step. Secondly, constant offsets make it easier to align the load instructions that fetch data from global to local memory. This is because the location and alignment of the load instructions directly depends on the position of the individual tiles.

3.5 Intra-tile schedules

We also specify non-trivial intra-tile schedules $t', s'_0, \ldots, s'_n$. It is desirable to minimize the intra-tile coordinates of the schedule, ideally starting from zero, to ensure an efficient thread to iteration mapping. To achieve this, we derive the intra-tile schedules from the tile schedule by replacing the outermost integer division by the corresponding remainder. For the classically tiled dimension this yields

$$s'_i = (s_i + \delta_i^1 u) \bmod w_i, \tag{17}$$

3.6 Hybrid tiling

The final hybrid tiling is a combination of the hexagonal tiling of Section 3.3 and the classical tiling of Section 3.4 as well as the intra-tile schedules of Section 3.5. This tiling is of the form

$$[t, s_0, \ldots, s_n] \to [T, p, S_0, \ldots, S_n, t', s'_0, \ldots, s'_n],$$

with tile dimensions defined by (2), $p = 0$, (3) (for S_0), (14) (for S_i with $i \ge 1$) and (15) for phase 0 and by (4), $p = 1$, (5), (14) and (16) for phase 1. Each phase is only applied to the subset of the domain that satisfies the conditions (6), (8), (10) and (12) in the local coordinates of the rectangular tile defined by (T, p, S_0). The constraints (7) and (13) are automatically satisfied for all points in the rectangular tile. As an example, Figure 6 shows the phase-0 part of a hybrid tiling where are δs are equal to 1.

The schedule is parameterized with the values h, w_0, …, w_n. The parameter h allows to adjust the distance between two subsequent tiles on the time dimension, and the different values w_i define the distance between subsequent tiles along the space dimensions s_i. For dimensions s_i with $i \ge 1$ the parameter w_i gives the exact width along this dimension, whereas for the dimension s_0 the value of parameter w_0 only gives the minimal width. The maximal tile width along this dimension may increase depending on the current time step.

$$
\begin{aligned}
&[t, s_0, s_1, \ldots, s_n] \to [T, 0, S_0, S_1, \ldots S_n, t', s_0', s_1', \ldots, s_n'] : \\
&\quad \exists a, b : a = (t + h + 1) \bmod (2h + 2) \land \\
&\qquad b = (s_0 + h + 1 + w_0) \bmod (2h + 2 + 2w_0) \land \\
&\qquad a - b \le h + 1 \land a + b \le 3h + 1 + w_0 \land \\
&\qquad a + b \ge h \land a - b \ge -w_0 - h \land \\
&\qquad T = \lfloor (t + h + 1)/(2h + 2) \rfloor \land \\
&\qquad S_0 = \lfloor (s_0 + h + 1 + w_0)/(2h + 2 + 2w_0) \rfloor \land \\
&\left(\bigwedge_{k:1 \le k \le n} S_k = \lfloor (s_k + ((t + h + 1) \bmod (2h + 2)))/w_k \rfloor \right) \land \\
&\qquad t' = (t + h + 1) \bmod (2h + 2) \land \\
&\qquad s_0' = (s_0 + h + 1 + w_0) \bmod (2h + 2 + 2w_0) \land \\
&\left(\bigwedge_{k:1 \le k \le n} s_k' = (s_k + ((t + h + 1) \bmod (2h + 2))) \bmod w_k \right)
\end{aligned}
$$

Figure 6: n-dimensional tile schedule (± 1 distances)

It should be noted that there is no need to map the spatial dimensions in the order to $s_0, \ldots, s_n$ in which the spatial loops are nested in the input code. Instead, any spatial dimension can be chosen as the one that is hexagonally tiled. However, to ensure our assumptions about aligned and coalesced memory accesses hold, it is necessary that the innermost dimension is the dimension that yields stride one access. This is a property that inputs normally already have and that we currently rely on.

3.7 Tile size selection

In order to determine appropriate values for the tile size parameters h and w_i, we use a simple model based on the load-to-compute ratio. In particular, we take a generic tile (not at the border) and compute the number of iterations in the tile and the number of loads performed by the tile. Since the set of iterations and the set of loads can be described using quasi-affine constraints, these numbers can be computed exactly as a function of the tile size parameters. For the experiments in this work, we use manually derived functions, but tools to count points in integer polyhedra [25] can automate this. For a 3D stencil with $\delta^0 = \delta^1 = 1$, the number of iterations in a tile is $2(1+2h+h^2+w_0(h+1))w_1w_2$, while the number of loads depends on the type of stencil and on various optimization choices described in Section 4. We then evaluate these formulas for all values of the tile size parameters that yield a memory tile size within a specified bound and select those parameters that yield the smallest load-to-compute ratio.

4. CUDA CODE GENERATION

To generate GPU code, we use the generic CUDA code generator of PPCG. The use of a generic infrastructure opens the opportunity to later integrate our stencil specific optimizer closely with a more generic GPU code generator. Even today it allows us to take advantage of the CUDA specific optimizations in PPCG. On top of the existing optimizations, we added additional optimizations that highly increase the performance of our generated CUDA code (see Section 6.2). These additional optimizations have been selected to make the execution of our hybrid-hexagonal schedule highly efficient, but we aimed to developed them in a way that enables the generic part of PPCG or other domain specific optimizations to benefit from them.

4.1 Generating CUDA code

Our tool uses the previously generated hybrid schedule to create CUDA code by mapping the schedule's output dimensions $[T, p, S_0, S_1, \ldots, t, s_0, s_1]$ to nested loops in the generated code. The T dimension is mapped to the host code, where it takes the form of a `for` loop repeatedly iterating over two CUDA kernels — one kernel for $p = 0$ and the other one for $p = 1$. For each kernel call, the dimension S_0 is mapped to a one dimensional grid of thread blocks that are executed in parallel. In case dimension S_0 has more elements than there are thread blocks supported by CUDA, the individual thread blocks execute multiple elements of S_0.

The remaining dimensions $[S_1, \ldots, S_n, t, s_0, \ldots, s_n]$ are code generated within each kernel. The dimensions $[S_1, \ldots, S_n, t]$ are code generated as sequential loops. As the dimensions $[s_0, \ldots, s_n]$ are fully parallel they can be mapped to different CUDA thread dimensions. In case there are more parallel dimensions than there are CUDA thread dimensions, the outer dimensions will be enumerated sequentially. To ensure all iterations of a dimension are executed even though there may be more iterations than threads in a thread block, additional iterations are assigned to threads in a cyclic way: iteration i is mapped to thread $i \bmod T_i$ with T_i being the number of threads used for dimension i. The sequential execution of subsequent time steps is ensured by generating a synchronization call at the end of each iteration of the sequential loops.

4.2 Shared memory

For hybrid-hexagonal tiled code the use of explicitly managed shared memory can be more efficient than a hardware managed cache. PPCG provides the following cache management strategy. Instead of performing all computations on global memory, PPCG allocates shared memory of the size of the smallest rectangular box that is large enough to accommodate the data accessed within a single tile. Now instead of just performing the computation of each tile, PPCG generates code that loads all data from global to shared memory, executes the computation on shared memory, and finally writes the modified elements back to global memory. To avoid thread divergence in the load phase, PPCG can over approximate the shape of the values to load with the rectangular box used to define the shared memory allocation.

4.2.1 Interleaving computations and copy-out

When developing our hybrid-hexagonal tiling we have seen that the separate copy-out phase makes the shared memory usage inefficient due to a possibly complex to describe set of values that needs to be copied out, but also due to the absence of overlap between the compute and the copy phase. We consequently extended the generic code generator to optionally write out values right at the time at which they are calculated. The unnecessary stores that may possibly be introduced are not overly costly, as for stencils the number of stores is low compared to the number of reads. Also, because our hybrid schedule ensures no thread divergence in the compute phase, executing the copy out next to the computation avoids all thread divergence.

4.2.2 Inter-tile reuse

Reducing the number of loads by taking advantage of reuse between subsequently executed tiles is another highly beneficial optimization It is possible due to the sequential execution of tiles enforced by the classical schedule at the inner dimension. Specifically, values that have already been loaded by the preceding tile, either because they are used there or because of the over approximation, do not need to be loaded from global memory. Instead, they can directly be moved from the shared memory location assigned in the preceding tile to the shared memory location where the current tile would store the element.

Another option would be to enforce a static mapping, where a single global location is always mapped to the same shared memory location. While this would eliminate the internal shared memory copy, accesses to statically mapped shared memory may induce more complex access patterns.

4.2.3 Aligned loads

It is important to ensure that loads from global memory are aligned to cache line boundaries. The location of the data that is loaded from global memory directly depends on the position of the tiles in space, specifically, the offsets of the tiles along the different space dimensions. When calculating the schedule we ensured that all these offsets are independent of the time dimension T. Assuming the size of the innermost data space dimension is a multiple of the minimal alignment, we select a tile width along the innermost dimension that is also a multiple of the minimal alignment. This ensures that as soon as the first load from an array is perfectly aligned, the subsequent loads are also perfectly aligned. We allow the tiles in the schedule to be translated by manually specifying the translation offset. By specifying the right offset it is possible to fully align the initial (and therefore all) global memory loads from a specific array. In case of multiple arrays, it may not always be possible to align the loads from all arrays.

4.3 Stencil specific code generation heuristics

During the final translation from the polyhedral program representation back to an abstract syntax tree (AST), domain specific knowledge can be used to adapt the code generation heuristics. The same schedule can be written out as an AST in many different ways, resulting in code that is functionality equivalent but that may have different performance behavior. The `isl` AST generator offers a flexible mechanism for allowing the user to choose between different ways of generating code across different parts of the schedule. We exploit this flexibility to implement specialized code generation heuristics for hybrid tiling.

4.3.1 Specialized code for the core computation

To generate optimal code for the core part of the computation we parameterize the code generation strategy such that specialized code is generated for full tiles and generic code for the remaining partial tiles.

When generating our schedule we have been especially careful to ensure that the number of integer points contained in a tile is the same for all tiles in the program and that the offsets used to derive the iterations that belong to a tile are constant within a single phase of our tiling scheme. We also made sure that within a core tile, there is no need for conditional execution that would cause thread divergence. To ensure that the simplicity of the core tiles is maintained and not lost by the need to handle rarely executed boundary cases we pass a description of the full tiles to `isl`'s AST generator, instructing it to generate code for these full tiles and the remaining partial tiles separately.

4.3.2 Unrolling for hybrid tiled stencils

Unrolling is often beneficial, but it is especially profitable in conjunction with our hybrid approach. As stated in the previous section, we construct a hybrid schedule such that the core computation is free of any thread divergence. In fact it does not require conditional control flow. However, due to the limited amount of shared memory and the large number of parallel threads, the number of iterations that need to be executed within a single thread is relatively low. Hence, we can unroll the point loops within the tile to create straightline code. This also contributes to exposing instruction level parallelism. Furthermore, depending on the tiling parameters chosen, we unroll neighboring points next to each other such that they can use a single load to get values that are within the neighborhood of both points.

Note that unrolling is not performed at the AST level, but on the constraint representation of the kernel. Constraint-based unrolling ensures that all conditions can be specialized or eliminated in the unrolled code, simplifying them according to the context in which an instruction is unrolled [20].

5. RELATED WORK

There has been much recent progress in automatic generation of high-performance code for stencil computations. Holewinski's Overtile [11] and Grosser's split tiling [7] compilers represent the state-of-the-art for the automatic generation of efficient GPU code relying on overlapped and split tiling, respectively. Patus is a domain-specific framework for stencils, driving multiple compilation strategies with autotuning, and targeting both CPUs and GPUs [4].

The PPCG [23] system is a state-of-the-art parallelizer for CPUs and GPUs, performing classical (time) tiling with parallel boundaries; PPCG relies on affine transformations to extract parallelism and improve locality, using a variant of the Pluto algorithm [3]. Reservoir Labs' R-Stream is also a reference polyhedral compiler targeting GPUs [14, 21]. Par4All [1] is an open source parallelizing compiler developed by Silkan targeting multiple architectures. The compiler is not based on the polyhedral model, but uses abstract interpretation for array regions, performing powerful inter-procedural analysis on the input code.

CPU-only frameworks are also available. Pochoir [19] is a domain-specific framework embedded into C++. Henretty et al. also proposed a DSL-based approach [10] for generating high-performance code for multi-core vector-SIMD architectures, using hybrid prismatic tile shapes. Our approach differs in the specific GPU constraints considered, including multi-level concurrency, local/shared memory and register transfers, and specialized code generation for full and partial tiles. Also, as a domain specific optimization embedded in a general purpose polyhedral compiler, our approach seems to be more general. The diamond tiling algorithm by Bandishti et al. [2] is closely related to our approach. In contrast to our approach, it combines tiling with transformations such as loop fusion and shifting, which is important when combining multiple stencils or non-stencil computations. We address the more constrained problem of generating code for

GPUs, and we overcome performance caveats of diamond tiling through our hybrid hexagonal/classical scheme with unique benefits on higher dimensional stencils.

The reader may ask, if a diamond tiling scheme could yield code of similar quality to the code we produce with our hybrid hexagonal/classical scheme. To our knowledge, no implementation of diamond tiling for GPUs is available. Consequently a direct performance comparison is not possible. However, as mentioned in Section 2, there are various qualitative advantages to our tiling scheme that are essential for high-performance GPU code. In contrast to diamond tiling, the adjustable size of the tile's peak ensures enough thread-level parallelism at all times; the identical integer point placement prevents thread divergence and the independence of time-tile height and tile-width allows more flexible tile-size choices, which consequently means better usage of available shared memory resources. We do not see how high-performance GPU code with all these features can be constructed using diamond-tiling. Further, we are not aware of any previously proposed tiling scheme that addresses such a comprehensive set of GPU performance issues.

6. EXPERIMENTAL RESULTS

To assess the effectiveness of our approach, in Section 6.1 we compare hybrid hexagonal tiling with state-of-the-art tools, and in Section 6.2 we analyze the impact of the various optimization strategies on performance.

6.1 Comparison with state-of-the-art tools

We evaluate our approach by comparing hybrid hexagonal tiling against Patus-0.1.3 [4], Overtile [11], Par4All-1.4.1 [1] as well as the current development version of the unmodified PPCG compiler [23]. We were not able to obtain a license for comparative evaluation with R-Stream [14].

	Loads	FLOPs/Stencil	Data-size	Steps
laplacian 2D	5	6	3072^2	512
heat 2D	9	9	3072^2	512
gradient 2D	5	15	3072^2	512
fdtd 2D	3	3	3072^2	512
	3	3	3072^2	512
	5	5	3072^2	512
laplacian 3D	7	8	384^3	128
heat 3D	27	27	384^3	128
gradient 3D	7	20	384^3	128

Table 3: Characteristics of Stencils

For benchmarks we use a Laplace kernel with two space dimensions, a 2D heat and a 2D gradient stencil as well as a two-dimensional, multi-stencil fdtd kernel. We also evaluate Laplace, heat and gradient kernels each having three space dimensions. Table 3 provides detailed characteristics of the stencils used. We did not evaluate our approach on one dimensional examples, because the hybrid method boils down to existing hexagonal or split tiling in this case [7]. All calculations were performed as single precision floating point computations and all timings include the data transfer overhead to and from the GPU. The experiments were conducted on NVIDIA GPUs: the NVS 5200M for mobile devices and a more powerful GeForce GTX 470.

For each tool, we sought to tune for the optimal tile sizes for the implemented tiling scheme and a specific benchmark. For PPCG, we used empirically optimized tile sizes used by the developers of the tool [23]. For Patus and overtile we used the provided autotuner. The Patus autotuner was run until completion, while we explored 800 tile sizes for each benchmark with overtile. For hybrid tiling we selected tile sizes aiming for a low load-to-compute ratio. Par4All was run with its dynamic tile sizing heuristic, using the options `-cuda -com-optimization` to enable GPU code generation. The flags defined in [23] were used for PPCG, and the hybrid tiling approach was combined with the optimizations discussed in Section 6.2. All other tools where used in the default configuration.

Tables 1 and 2 show the results for the GTX 470 and NVS 5200, respectively. As a baseline, the general purpose compiler PPCG is able to create code for all benchmarks, but does not reach optimal speed. We do not include performance numbers for Patus, because due to its experimental CUDA support, only laplacian and heat 3D code could be generated. However, it should be noted that Patus reaches 3.5 GStencils/second for laplacian 3D on the GTX 470 and 0.50 GSTencils/second on the NVS5200, a 75% (56%) of speedup over PPCG. Except for some slowness on the heat-2D kernel, Par4All produces reasonably well performing code with good performance on the gradient 2D and 3D kernels. Par4All uses an internal heuristic to derive tile sizes. Overtile shows consistently good performance, attaining speedups over PPCG code of up to 96% for 2D kernels, very high speedups of up to 818% for fdtd 2D and up to 106% on 3D kernels. These results demonstrate the performance a stencil DSL compiler combined with autotuning can reach. Looking at the auto-tuned tile sizes we see that Overtile is not able to effectively exploit time tiling for 3D kernels. Instead, it falls back to a space-tiled version. This is also in line with Patus, Par4All and PPCG, which do not support time-tiling in general.

The last row presents results from our hexagonal-hybrid tiling compiler. For all 2D kernels, on both the GTX470 and the NVS 5200, we observe better performance than all previous techniques. Compared to base PPCG, we observe speedups ranging from 71% and 211%, with an exceptional 920% speedup for fdtd-2d. The consistently superior performance for 2D and 3D kernels across the board demonstrates the effectiveness of our approach. The 2D and 3D heat kernels showcase our hybrid-hexagonal tiling with performance results that are in three cases more than two times faster than the second best implementation.

One of the main reasons for the good performance is that we have been able to effectively exploit time-tiling for all benchmarks. Each 2D kernel executes eight time steps per tile and each 3D kernel executes four time steps per tile. Exploiting time tiling has only become beneficial due to the careful management of shared memory, as well as the reduction of overhead due to full-partial tile separation, code specialization and unrolling. Combined together, this enabled excellent performance.

6.2 Hybrid tiling and shared memory

Even though hybrid tiling can be beneficial by itself, its full benefits only manifest when combined with explicitly managed shared memory. In this section, we analyze how shared memory usage as well as different shared memory optimizations impact the performance of a hybrid tiled kernel. As explicit cache management has proven to be especially

	laplacian 2D		heat 2D		gradient 2D		ftdt 2D		laplacian 3D		heat 3D		gradient 3D	
PPCG	5.4		5.1		3.9		0.76		2.0		1.8		2.1	
Par4All	7.0	+30%	5.4	+2%	5.5	+41%	invalid CUDA		2.0	±0%	1.9	+6%	3.1	+ 48%
Overtile	10.6	+96%	6.9	+35%	6.7	+72%	5.3	+597%	3.1	+55%	2.6	+44%	**3.6**	**+71%**
hybrid	**15.0**	**+177%**	**15.0**	**+194%**	**7.3**	**+87%**	**7.3**	**+860%**	**4.3**	**+115%**	**3.9**	**+116%**	**3.6**	**+71%**

Table 1: Performance on NVIDIA GTX 470: GStencils/second & Speedup

	laplacian 2D		heat 2D		gradient 2D		fdtd 2D		laplacian 3D		heat 3D		gradient 3D	
PPCG	1.0		0.97		0.61		0.098		0.32		0.29		0.32	
Par4All	1.1	+10%	0.79	-18%	0.9	+55%	invalid CUDA		0.34	+6%	0.35	+20%	0.69	+116%
Overtile	2.1	+90%	1.5	+54%	1.1	+80%	0.9	+818%	0.66	+106%	0.37	+30%	0.61	+90%
hybrid	**3.2**	**+211%**	**2.9**	**+198%**	**1.4**	**+130%**	**1.0**	**+920%**	**0.91**	**+184%**	**0.73**	**+150%**	**0.73**	**+128%**

Table 2: Performance on NVS 5200: GStencils/second & Speedup

challenging for 3D kernels, we choose to analyze the three dimensional heat kernel.

	NVS 5200		GTX 470	
(a) no shared memory	8		39	
(b) shared memory	8	±0%	44	+12%
(c) (b) + interleave copy-out	11	+37%	65	+47%
(d) (c) + align loads	12	+9%	70	+7%
(e) (d) + value reuse (static)	11	-8%	73	+5%
(f) (d) + value reuse (dynamic)	19	+58%	105	+50%

Table 4: Optimization steps: GFLOPS & Speedup

Table 4 gives an overview of the different configurations we analyzed and their performance on an NVS 5200 as well as a GTX 470 GPU. All configurations where run with 1x10x32 threads and hybrid tiles of size $h = 2, w_0 = 7, w_1 = 10, w_2 = 32$. As described in Section 3.7, tile sizes have been selected to minimize the load-to-compute ratio and to ensure that the inner dimension is a multiple of the warp size.

Configuration (a) only uses global memory, but no shared memory. (b) uses shared memory. For each tile we first copy all required values into shared memory, we then perform the computation within shared memory and finally we copy the results back to global memory. (c) eliminates the explicit copy out phase. Instead, results are copied out as soon as they have been calculated. In (d) we adjust the position of the tiles in the data space such that all loads from global memory are aligned. Finally, (e) and (f) show two different approaches that both enable the reuse of values used and loaded in one tile and used in a subsequently executed tile. In (e) we eliminate the need to reload values by statically assigning each global value to a shared memory location. In (f) we allow a single global value to be dynamically placed for different tiles at different shared memory locations. To still enable reuse we add an explicit copy phase scheduled between two subsequent tiles. This phase moves values from their old shared memory location to the location where the next tile expects them to be.

To understand the performance results shown in Table 4 we analyze the different configurations together with relevant performance counters. The results are shown in Table 5, in units of 10^9. The first one, configuration (a) gives a solid performance baseline. Introducing explicit shared memory in (b) does not change performance on the NVS 5200 and gives a 12% performance increase on the GTX470. The small performance difference is not surprising. Even though the number of global load instructions is reduced by a factor of 20, the actual reads from DRAM are mostly unaffected. This shows that our shared memory management is as effective in avoiding DRAM loads as the automatic caches are. Looking at the L2 transactions we see large benefits due to our explicit shared memory management. Unfortunately, the almost unchanged performance suggests that other effects such as a reduced global load efficiency and the explicit cache management overhead itself hide the benefits. One cache management problem is the missing overlap of computation and data-transfers. (c) shows that that by overlapping copy-out and the actual computation, we can increase performance by 37-47% without changing the amount of data transferred. Another inefficiency we see is the global load efficiency of only 30%. (d) partially addresses this by ensuring that all loads from global memory are fully aligned. However, only after removing partial global loads in (e) and (f) we are able to fully achieve 100% global load efficiency. Interestingly, at this point our kernel has been moved from being bound by global loads to being bound by shared memory loads. (f) has as efficient global loads as (e), but due to the way memory is accessed, it is very likely to cause bank conflicts in shared memory. This is reflected by the number of shared memory load transactions, which is twice that of all other kernels. The overhead caused by these bank conflicts unfortunately hides the gains from the reduction in global loads. On the other hand, (f) shows that we are able to create a highly performing kernel that achieves 100% global load efficiency, 100% shared load efficiency and that significantly reduces the requests that reach the L2 cache and global memory.

The overall speedup of 250% for this kernel was only possible due to the combination of hybrid-hexagonal tiling with careful shared memory management. Our optimization reaches a point where the kernel is mostly bound by shared memory. Further reducing the number of shared memory loads through register tiling would be an interesting angle to increase performance even further.

7. CONCLUSION

We presented hexagonal tiling and its combination with classical tiling, a hybrid algorithm for the automatic parallelization of iterative stencil computations on GPUs. Hexagonal tile shapes simultaneously enable parallel tile execution and reuse along the time dimension. The hybrid extension offers unprecedented performance on higher dimensional stencils, thanks to coalesced global-memory accesses, data reuse in shared-memory/cache and registers, avoiding thread divergence and maximizing the exploitation of concurrency at all levels. Experimental results demonstrate

	gld_inst_32bit	dram_read_transactions	l2_read_transactions	shared loads per request	gld_efficiency
(a)	171.0	1.7	12.0	n/a	54%
(b)	8.7	1.8	1.4	1.0	30%
(c)	8.7	1.8	1.4	1.0	30%
(d)	8.8	1.0	0.95	1.0	56%
(e)	7.6	0.97	0.49	1.8	100.00%
(f)	7.6	0.95	0.48	1.0	100.00%

Table 5: Performance counters (units of 10^9 events)

significant performance improvements over existing stencil compilers. We are combining this domain-specific approach with loop transformations for general, non-stencil codes, integrating the technique into a polyhedral research compiler.

Acknowledgments. This work is partly funded by a Google European Fellowship in Efficient Computing, by the European FP7 project CARP id. 287767, by the COPCAMS ARTEMIS project, and award 0926688 from the U.S. NSF.

8. REFERENCES

[1] M. Amini, B. Creusillet, S. Even, R. Keryell, O. Goubier, S. Guelton, J. O. McMahon, F.-X. Pasquier, G. Péan, P. Villalon, et al. Par4All: From convex array regions to heterogeneous computing. In *IMPACT*, 2012.

[2] V. Bandishti, I. Pananilath, and U. Bondhugula. Tiling stencil computations to maximize parallelism. In *Supercomputing*, page 40. IEEE Computer Society Press, 2012.

[3] U. Bondhugula, J. Ramanujam, and et al. PLuTo: A practical and fully automatic polyhedral program optimization system. In *PLDI*, 2008.

[4] M. Christen, O. Schenk, and H. Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *IPDPS*, 2011.

[5] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[6] P. Feautrier. *The Data Parallel Programming Model*, volume 1132 of *LNCS*, chapter Automatic Parallelization in the Polytope Model, pages 79–100. Springer, 1996.

[7] T. Grosser, A. Cohen, P. H. Kelly, J. Ramanujam, P. Sadayappan, and S. Verdoolaege. Split tiling for GPUs: automatic parallelization using trapezoidal tiles. In *GPGPU-6*, pages 24–31. ACM, 2013.

[8] T. Grosser, A. Grösslinger, and C. Lengauer. Polly – performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[9] T. Grosser, S. Verdoolaege, A. Cohen, and P. Sadayappan. The relation between diamond tiling and hexagonal tiling. In *1st Int. Workshop on High-Performance Stencil Computations (HiStencils 2014)*, Vienna, Austria, Jan. 2014.

[10] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector simd architectures. In *ICS*. ACM, 2013.

[11] J. Holewinski, L.-N. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on GPU architectures. In *ICS*, 2012.

[12] F. Irigoin and R. Triolet. Supernode partitioning. In *POPL*, pages 319–328, San Diego, CA, Jan. 1988.

[13] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *PLDI*, pages 235–244, 2007.

[14] A. Leung, N. Vasilache, B. Meister, M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin. A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In *GPGPU-3*, 2010.

[15] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Seattle, WA, June 2013.

[16] M. Ravishankar, J. Eisenlohr, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. Code generation for parallel execution of a class of irregular loops on distributed memory systems. In *Supercomputing*, pages 1–11, 2012.

[17] G. Smith. *Numerical Solution of Partial Differential Equations: Finite Difference Methods.* Oxford University Press, 2004.

[18] A. Taflove. *Computational electrodynamics: The Finite-difference time-domain method.* Artech House, 1995.

[19] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The Pochoir stencil compiler. In *SPAA*, pages 117–128. ACM, 2011.

[20] N. Vasilache, C. Bastoul, and A. Cohen. Polyhedral code generation in the real world. In *Proceedings of the International Conf. on Compiler Construction (ETAPS CC)*, Vienna, Austria, Mar. 2006. Springer.

[21] N. Vasilache, B. Meister, M. Baskaran, and R. Lethin. Joint scheduling and layout optimization to enable multi-level vectorization. In *IMPACT*, Paris, France, Jan. 2012.

[22] S. Verdoolaege. isl: An integer set library for the polyhedral model. In *Mathematical Software–ICMS 2010*, pages 299–302. Springer, 2010.

[23] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for CUDA. *ACM TACO*, 9(4):54, 2013.

[24] S. Verdoolaege and T. Grosser. Polyhedral extraction tool. In *IMPACT*, 2012.

[25] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica*, 48(1):37–66, June 2007.

[26] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison Wesley, 1996.