The Promises of Hybrid Hexagonal/Classical Tiling for GPU

Tobias Grosser, Sven Verdoolaege, Albert Cohen, P. Sadayappan

To cite this version:

Tobias Grosser, Sven Verdoolaege, Albert Cohen, P. Sadayappan. The Promises of Hybrid Hexagonal/Classical Tiling for GPU. [Research Report] RR-8339, INRIA. 2013. hal-00848691

HAL Id: hal-00848691

https://hal.inria.fr/hal-00848691

Submitted on 27 Jul 2013

The Promises of Hybrid Hexagonal/Classical Tiling for GPU

Tobias Grosser, Sven Verdoolaege,
Albert Cohen, P. Sadayappan

RESEARCH REPORT

N° 8339

July 2013

Project-Teams PARKAS

ISSN 0249-6399 ISRN INRIA/RR--8339--FR+ENG

The Promises of Hybrid Hexagonal/Classical Tiling for GPU

Tobias Grosser[*], Sven Verdoolaege[*],
Albert Cohen[*], P. Sadayappan[†]

Project-Teams PARKAS

Research Report n° 8339 — July 2013 — 22 pages

Abstract: Time-tiling is necessary for efficient execution of iterative stencil computations. But the usual hyper-rectangular tiles cannot be used because of positive/negative dependence distances along the stencil's spatial dimensions. Several prior efforts have addressed this issue. However, known techniques trade enhanced data reuse for other causes of inefficiency, such as unbalanced parallelism, redundant computations, or increased control flow overhead incompatible with efficient GPU execution.
We explore a new path to maximize the effectivness of time-tiling on iterative stencil computations. Our approach is particularly well suited for GPUs. It does not require any redundant computations, it favors coalesced global-memory access and data reuse in shared-memory/cache, avoids thread divergence, and extracts a high degree of parallelism. We introduce hybrid hexagonal tiling, combining hexagonal tile shapes along the time (sequential) dimension and one spatial dimension, with classical tiling for other spatial dimensions. An hexagonal tile shape simultaneously enable parallel tile execution and reuse along the time dimension. Experimental results demonstrate significant performance improvements over existing stencil compilers.

Key-words: polyhedral model, GPGPU, CUDA, code generation, compilers, loop transformations, time tiling, stencil

[*] INRIA and École Normale Supérieure
[†] Ohio State University

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Les promesses du partitionnement hybride hexagonal/classique pour GPU

Résumé : Le partitionnement temporel est indispensable pour l'exécution efficace de stencils itératifs. En revanche les tuiles hyper-parallélépipédiques usuelles ne sont pas applicables en raison du mélange de dépendances en avant et en arrière suivant les dimensions spatiales du stencil. Plusieurs études ont été consacrées à ce problème. Pourtant, les techniques connues tendent à échanger une meilleure réutilisation des données contre d'autres sources d'inefficacité, telles que le déséquilibre du parallélisme, des calculs redondants, ou un surcoût induit par la complexité du flot de contrôle incompatible avec l'exécution sur GPU.

Nous explorons une autre voie pour maximiser l'efficacité du partitionnement temporel sur des stencils itératifs. Notre approche est particulièrement bien adaptée aux GPUs. Elle n'induit pas de calculs redondants, favorise l'agglomération des accès à la mémoire globale et la réutilisation de données dans les mémoires locales ou caches, tout en évitant la divergence de threads et en exposant un degré élevé de parallélisme. Nous proposons le partitionnement hybride hexagonal, qui repose sur des tuiles hexagonales selon la dimension temporelle (séquentielle) et une dimension spatiale, combinées avec un partitionnement classique selon les autres dimensions spatiales. La forme de tuile hexagonale autorise l'expression de parallélisme entre tuiles et la réutilisation selon la dimension temporelle. Nos résultats expérimentaux mettent en évidence des améliorations sensibles de performance par rapport aux compilateurs spécialisés dans l'optimisation de stencils.

Mots-clés : modèle polyédrique, GPGPU, CUDA, génération de code, compilateurs, transformations de boucles, partitionnement temporel, stencil

Contents

1 Introduction **4**

2 Analysis **5**

3 Algorithm **7**

3.1 Polyhedral Model 7

3.2 Preprocessing 8

3.3 Hexagonal tiling 8

3.3.1 Constraints on input 8

3.3.2 The hexagonal tile shape 9

3.3.3 The schedule for hexagonal tiling 10

3.4 Classical tiling 12

3.5 The hybrid tiling 13

4 Code Generation **13**

4.1 Generating CUDA code 13

4.2 Use of shared memory 15

4.3 Refinement of the schedule 15

4.3.1 Alignment 16

4.3.2 Avoiding modulo operations 16

4.4 Stencil specific code generation heuristics 16

4.4.1 Specialized code for the core computation 17

4.4.2 Unrolling for hybrid tiled stencils 17

5 Related Work **17**

6 Experimental Results **18**

7 Conclusion **19**

1 Introduction

Tiling is a critical loop transformation to enable the effective exploitation of data locality in computations. With the growing divergence between combined computational rates of cores on a chip and the aggregate bandwidth to off-chip memory, the grouping of operations in the iteration space into tiles of contiguous operations that exhibit reuse is a fundamentally important technique to reduce the amount of off-chip memory transfer. There has been a long history of efforts to develop compiler algorithms to perform tiling [9, 20, 3]. Several publicly available research compilers exist that implement advanced tiling transformations for affine loops [18, 1, 3, 10, 11], and some of them [18, 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 [13], solution of PDEs using finite difference or finite volume discretization [12], and image processing. While stencil computations manifest significant amounts of parallelism at each time step across the spatial domain, the challenge is to avoid being memory-bandwidth bound because the spatial domains are often much larger than cache capacity. Time-tiling, or blocking of the computations over multiple time steps for a sufficiently small spatial domain that can fit within cache is essential to achieving high performance. But simple rectangular tiling over the iteration space is not feasible since the dependence components along spatial dimensions have negative components. The "classical" solution to time-tiling of stencil computations is to skew all spatial dimensions with respect to time, so that all dependence components are positive along all dimensions. But this inhibits inter tile parallelism [21].

Other approaches to tiling, such as split tiling [7], overlapped tiling [8], and diamond tiling [2] have been proposed to address the loss of concurrency with standard time-tiling of stencil computations. However, as elaborated further in the next section, all previously proposed approaches have drawbacks for GPUs. Achieving high performance on GPUs requires that several considerations all be satisfied by the generated tiled stencil code: 1) coalesced data access from global memory, 2) two-level parallel structure, where efficient synchronization is feasible only at the inner (within a thread-block) and high degree of parallelism is desired to effectively mask access latency to global memory, but only moderate level of parallelism is needed at the outer (across thread blocks, 3) avoidance of thread divergence within thread blocks.

Previously proposed approaches to stencil computations on GPUs, whether manual or assisted by compiler transformations, suffer from one or more drawbacks, as explained in Sec. 2. In contrast, in this paper we develop a compiler transformation and code generation strategy for stencil execution on GPUs, which addresses all the above requirements, without resorting to execution of any redundant or unnecessary computations. The paper makes the following contributions:

- It develops a novel hybrid hexagonal/standard tiling approach that achieves adequate thread-level parallelism, while fully avoiding thread divergence.
- Unlike other specialized stencil compilers, the developed compiler algorithms for stencil optimization can be integrated into general polyhedral frameworks, and are being incorporated into the publicly distributed ppcg polyhedral tool.
- Experimental performance data are provided on a number of stencil benchmarks demonstrating consistent superiority over all existing GPU stencil compilers on 2D stencils, and

comparable performance with the best on 3D stencils. Our stencil compiler is also the only one generating correct GPU code for important multi-statement stencils like FDTD.

The paper is organized as follows. In Sec. 2, we elaborate on the challenges of tiling stencils for GPUs, pointing out the limitations of previously proposed approaches. Sec. 3 presents the compilation algorithm and proof of correctness. Sec. 4 explores important code generation issues pertinent to optimizing for GPU execution. 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 Analysis

A cleverly chosen tiling scheme must navigate through a host of constraints. At the core, unrolled inner loops must be carefully specialized to avoid control flow and cumbersome address computations, exploiting register reuse and relying on shared memory as efficiently as possible. Only then, such highly optimized building blocks can be tiled at higher levels of the memory hierarchy and distributed over the whole parallel architecture. Figure 1 shows a 2D Jacobi stencil in source form, and Figure 2 show 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 gets even more complicated for higher-dimensional stencils. Let us review some of the pitfalls and limitations of state-of-the art methods, before we introduce our new approach to the conciliation of all important constraints.

Split the wrong dimensions → divergence Tiling on a GPU is exposes coarse-grained parallelism exploited as thread blocks, and it is also essential to control shared memory usage. Previous automated techniques did not use specific heuristics to address these two different goals: we claim that both should be addressed separately and with dedicated tiling strategies. Specifically, *we use hexagonal tiling to expose parallelism, but classical tiling to bound the memory size of the kernels.* In addition, we *do not* apply hexagonal tiling to the iteration space dimension carrying contiguous (stride-one) array accesses: this allows for coalesced memory loads. The tile size for the contiguous access dimension is thus constant over all time steps (parallel tile boundaries), and choosing it to be a multiple of the warp size avoids any thread divergence in the core computations.

Overlapped tiling → wastes shared memory Overlapped tiling [8] seems to be a promising strategy, as trading in redundant computations to reduce memory accesses is likely to be beneficial on devices such as GPUs where individual computations are cheap and memory bandwidth is commonly a limiting factor. However, it needs to be stressed that overlapped tiling degrades shared memory usage on GPU kernels, as temporary values must be stored in shared memory before being atomically committed. Having to allocate additional memory for temporary values consequently means less iterations can be executed before synchronizing and communicating with global memory. As the IO/compute ratio improves with larger tile sizes, relying on temporary values negatively impacts the computation throughput.

Diamond tiling → memory access bottleneck Diamond tiling [2] is a very successful strategy to enable time-tiling while exploiting parallelism along the main dimensions of the iteration space. This has load balancing and code quality benefits. Our proposal is directly related to diamond tiling, but introduces two major advances. The first one is that diamond tiles always have a narrow peak. This is not good for GPUs, because the amount of parallelism at this level is too low to exploit fine-grain parallelism in kernels, and because accesses to global memory are too narrow to make efficient use of the available bandwidth and to hide latency. This can lead to the weird situation where a computation is memory bound but the available memory bandwidth is not fully used. Lastly, even though diamond tiling uses identical tile shapes, the actual integer points may vary in the different tiles. This may induce additional control flow and code bloat, when the peaks of the diamonds sometimes fall on an integer point and sometimes do not.

Data-transfer to compute ratio Finally, the intrinsic IO/Compute ratio of the stencil algorithm must be considered, measuring its impact on the tile shape and strategy. When applying different time-tiling strategies to Jacobi-style stencils, differences in their IO/Compute behavior can be observed. Understanding these differences and their performance impact on GPU computations is important. Figure 3 illustrates the behavior of overlapped tiling, split tiling and diamond tiling on 1 space dimension, for a three point stencil. Setting a bound on shared memory usage to 3 elements per array (for the sake of the illustration), we maximize tile sizes while keeping intra-tile computations in shared memory. The light blue and green line illustrate in the figure the amount of data that is read in and written out.

For overlapped tiling, as a first step all data from one array needs to be communicated before the computation can begin. The computation phase needs two arrays of 3 elements each. Overlapped tiling executes 1 unit of actual computation and another unit of redundant computation. After the whole computation has finished, 1 unit of data can be written out. Overall, 4 data elements must be communicated, and 1 redundant computation is required for each useful computation. This results in an IO/Compute ratio of 4.0.

For split tiling, there are two symmetric cases. In the first case one loads 3 data elements before any computation can begin. But it is then possible to perform 2.25 units of computation. After the tile has finished 6 results need to be written back. Overall 9 data elements are communicated to perform 2.25 computations. This gives an IO/Compute ratio of 4.0. Unlike overlapped tiling, it is possible to stream data in and out on the fly or to recycle registers and shared memory rather than having to wait for the completion of the full tile.

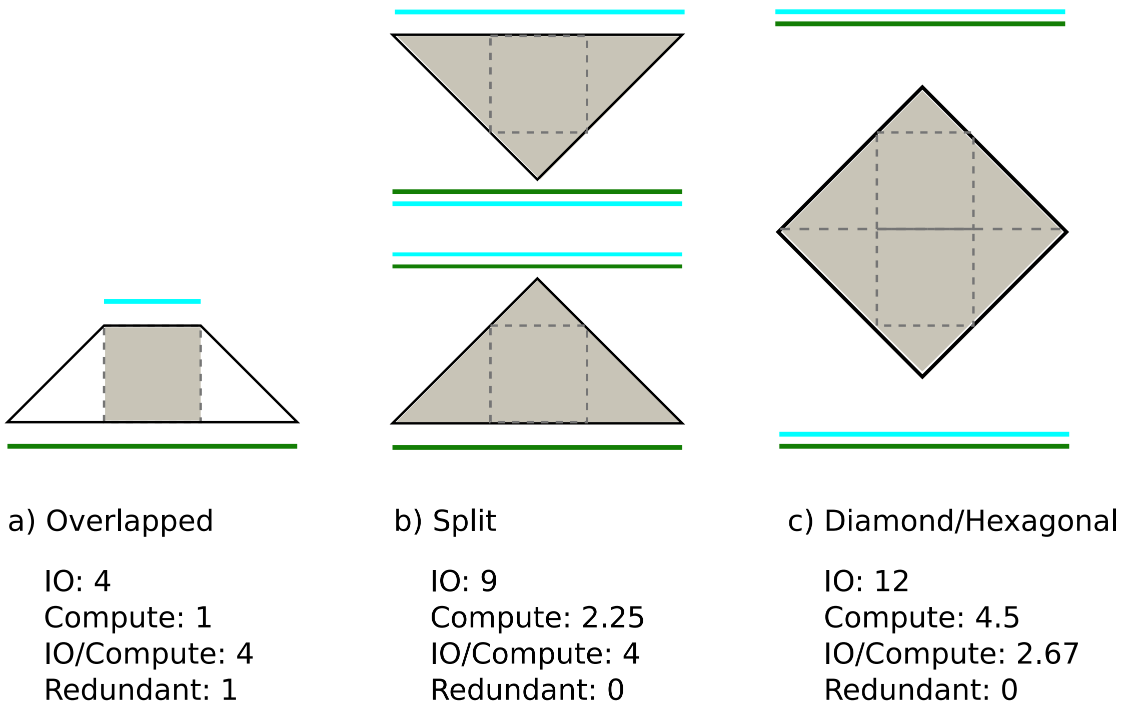

Figure 3: IO/Compute ratio w.r.t. tiling strategy

For diamond tiling, 3 loads are again necessary from each of the two arrays, but these allow to perform 4.5 computations, before writing back 3 values to each of the two arrays. Overall, 12 elements are communicated to perform 4.5 computations. This gives an IO/Compute ratio of 2.67. Again, data can be streamed in and out on the fly, like split tiling. Hexagonal tiling behaves the same way, but improves on parallelism and memory transfers.

3 Algorithm

Our approach can be divided into several steps. First, the input program is analyzed statically and translated into a polyhedral representation. This representation is then canonicalized for stencil computations. Next, this abstract information allows to derive an execution schedule that materializes the ordering of iterations in a hybrid hexagonal/classical tiling. Finally, we use this schedule to parameterize our generic CUDA code generator. After presenting the general structure we propose non-essential amendments addressing specific performance issues. As a last critical point, we show how to use a parameterized code generation algorithm to achieve very high code quality.

3.1 Polyhedral Model

The polyhedral model [6] allows for an instance-based analysis of a program by representing its main features using sets and relations bounded 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` [19], compute dependences using `isl` [17] and transform the polyhedral description into some canonical form that later simplifies the construction of the schedule. We currently assume that the input program consists of an outer loop containing $k \geq 1$ perfectly nested loop nests such that none of the loops in these loop 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 $L_i[t, s_0, \ldots, s_n] \rightarrow [k \cdot t + i, s_0, \ldots, s_n]$, 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 3.5 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 tiling 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 calculate a hexagonal tiling from a two dimensional initial schedule space $P = [t, s_0]$ and a set of dependences $D \subseteq P \times P$. We first describe the constraints we impose on the input data. Having conforming input data, we then derive the tiling by constructing the hexagonal tile shape and consequently calculating a schedule that implements the tiling. Finally, we show that the constructed tiling is correct and that it allows the 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 The hexagonal tile shape

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]);
```

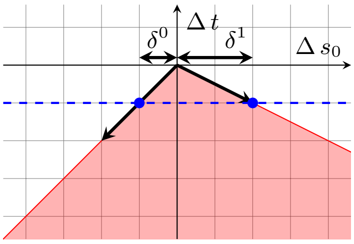

Figure 4: Opposite dependence cone

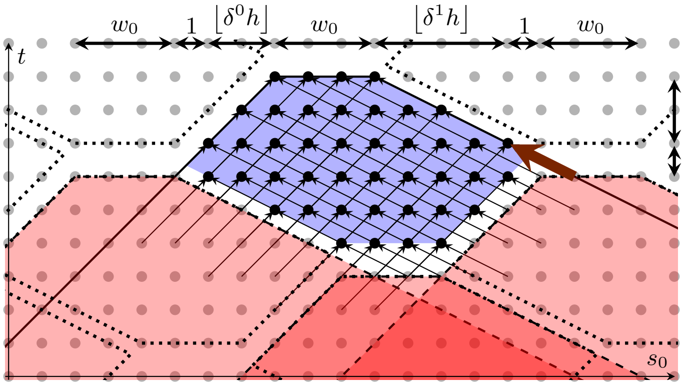

Figure 5: A hexagonal tile

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 4. 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 $\Delta s_0 \leq \delta^0 \Delta t$ (or, equivalently, $-\Delta s_0 \geq \delta^0(-\Delta t)$) and $\Delta s_0 \geq -\delta^1 \Delta t$. These constants can be computed through the solution of an LP-problem. Figure 4 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 4) shifted to each of these instances. Figure 5 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 5. 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

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

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

3.3.3 The schedule for hexagonal tiling

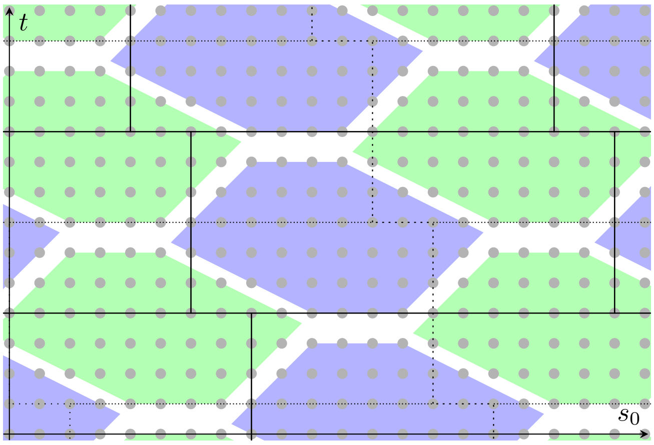

Figure 6: Hexagonal tiling pattern

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 6 (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. 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 numerator of expression 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 6, the solid boxes for phase 0 and the dotted boxed 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

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

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 $a' \leq 2h + 1$. To verify that there can be no points in the current truncated cone that do not satisfy the constraint

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

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

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

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

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

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

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

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

3.4 Classical tiling

In the remaining spatial dimensions, we apply a more traditional tiling. This means that we lose parallelism along these dimensions, but it allows us to reduce the working set within each tile. These additional tilings are performed on each spatial dimension s_i with $i \in [1, n]$ separately. As in the case of hexagonal tiling (see Figure 4), we first need to consider the projection of the dependence cone onto the time dimension and the given spatial dimension. However, 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 tile shape for these tilings 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

$$S_i = \left\lfloor (s_i + \delta_i^1 t')/w_i \right\rfloor, \tag{14}$$

where t' 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

$$t' = (t + h + 1) \bmod (2h+2) \qquad \text{for phase 0 and} \tag{15}$$

$$t' = t \bmod (2h+2) \qquad \text{for phase 1.} \tag{16}$$

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 The 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. This tiling is of the form

$$[t, s_0, s_1, \ldots] \to [T, p, S_0, S_1, \ldots, t, s_0, s_1, \ldots],$$

with tile dimensions defined by (2), $p = 0$, (3) (for S_0), (14) (for S_i with $i \geq 1$) 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 7 shows the phase-0 part of a hybrid tiling where are δs are equal to 1.

$$\begin{aligned}
&[t, s_0, s_1, \ldots, s_n] \to [T, 0, S_0, S_1, \ldots S_n, t, s_0, s_1, \ldots, s_n] : \\
&\quad \exists a, b : a = (t + h + 1) \bmod (2h + 2) \wedge \\
&\qquad\quad b = (s_0 + h + 1 + w_0) \bmod (2h + 2 + 2w_0) \wedge \\
&\qquad\quad a - b \leq h + 1 \wedge a + b \leq 3h + 1 + w_0 \wedge \\
&\qquad\quad a + b \geq h \wedge a - b \geq -w_0 - h \wedge \\
&\qquad\quad T = \lfloor (t + h + 1)/(2h + 2) \rfloor \wedge \\
&\qquad\quad S_0 = \lfloor (s_0 + h + 1 + w_0)/(2h + 2 + 2w_0) \rfloor \wedge \\
&\qquad\quad S_1 = \lfloor (s_1 + ((t + h + 1) \bmod (2h + 2)))/w_1 \rfloor \wedge \\
&\qquad\qquad \vdots \\
&\qquad\quad S_n = \lfloor (s_n + ((t + h + 1) \bmod (2h + 2)))/w_n \rfloor
\end{aligned}$$

Figure 7: n-dimensional hybrid tile schedule for ±1 dependence distances

The schedule is parameterized with the values h, w_0, $\ldots w_n$. 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 $i \geq 1$ 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.

4 Code Generation

To generate GPU code, we use the generic CUDA code generator from ppcg, feeding it with additional information to exploit the structure of the hybrid tiled iteration space. We start by providing ppcg the hybrid schedule as well as some essential information to generate tiled GPU code. As a next step we drive ppcg into moving most of the core computation into shared memory.

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 for p equals one, the other for p equals two. 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. Figure 8 presents the host code that is generated for a simple 2D kernel.

```
for (T = 0; T <= 132; T += 1) {
  {
    dim3 dBlock(64, 2);
    dim3 dGrid(93);
    kernel0<<<dGrid, dBlock>>>(dev_A, T);
  }
  {
    dim3 dBlock(64, 2);
    dim3 dGrid(94);
    kernel1<<<dGrid, dBlock>>>(dev_A, T);
  }
}
```

Figure 8: The CUDA host code

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. Figure 9 shows the CUDA code generated for the jacobi 2D kernel shown in Figure 1 using two threads along the s_0 dimension and 64 threads along the s_1 dimension.

```
b0 = blockIdx.x;
t0 = threadIdx.y, t1 = threadIdx.x;

for (S1 = 1; S1 <= 14; S1 += 1)
  for (t = 6 * T; t <= 6 * T + 5; t += 1) {
    for (s0 = max(-((t0 + t) % 2) + 22 * b0 -
                   6 * T + t - 4,
                 -((t0 + t + 1) % 2) +
                   22 * b0 + 6 * T - t + 1);
         s0 <= ms0n(22 * b0 - 6 * T + t + 8,
                  22 * b0 + 6 * T - t + 13);
         s0 += 2)
      for (s1 = ((t1 - 6 * T + t + 64) % 64) +
                 6 * T + 128 * S1 - t;
           s1 <= 6 * T + 128 * S1 - t + 127;
           s1 += 64)
S:      A[-(t % 2) + 1][s0][s1] = ...
    __syncthreads();
  }
```

Figure 9: CUDA kernel code for a 2D problem (without border tile conditions)

4.2 Use of shared memory

We want the generated CUDA code to take advantage of shared memory. One of the core goals of time tiling is to reduce the number of memory accesses. For CUDA this means we want to reduce the amount of global memory accesses and instead keep values that are only used locally in shared memory. This could either be achieved by relying on the automated caches or by explicitly managing shared memory. In our work we decided to explicitly manage shared memory.

Our memory management strategy is as follows. We keep all values that are used within a single tile in shared memory. ppcg allows us to define the loop level up to which data that is used multiple times should be kept in shared memory. We set this loop level to the last schedule dimension that enumerates the tiles. ppcg then automatically derives the subset of global memory on which reuse occurs and which can be placed in shared memory. If there is a sufficient amount of shared memory ppcg introduces code to copy values from global to shared memory, ppcg rewrites the memory accesses to point to shared memory and ppcg introduces code to copy the values back from shared memory to global memory. Figure 10 gives an example that shows how before each tile all values are copied into shared memory and how after each tile all values are copied back to global memory. Within a tile, all calculations are performed in shared memory.

```
b0 = blockIdx.x;
t0 = threadIdx.y, t1 = threadIdx.x;
__shared__ float shared_A[2][15][135];

for(S1=1;S1<=14;S1+=1){
  for(c0=0;c0<=1;c0+=1)
    for(c1=t0;c1<=14;c1+=2)
      for(c2=t1;c2<=134;c2+=64)
        shared_A[c0][c1][c2]
                =A[c0][22*b0+c1-3][128*S1+c2-6];
  for(t=6*h0;t<=6*h0+5;t+=1){
    __syncthreads();
    for(s0=max(-((t0+t)%2)+22*b0-6*T+t-4,
              -((t0+t+1)%2)+22*b0+6*T-t+1);
           s0<=ms0n(22*b0-6*T+t+8, 22*b0+6*T-t+13);
             s0 += 2)
      for(s1=((t1-6*T+t+64)%64)+6*T+128*S1-t;
          s1<=6*T+128*S1-t+127; s1 += 64)
        shared_A[-(t%2)+1][-22*b0+s0+3][-128*S1+s1+6] = ...
    __syncthreads();
  }
  for(c0=0;c0<=1;c0+=1)
    for(c1=-t0+2;c1<=13;c1+=2)
      for(c2=max(((t1-c1+74)%64)+c1-10,
                ((t1+63)%64)+1),
                ((t1+c1+60)%64)-c1+4);
          c2<=min(c1+130,-c1+144,133);
          c2+=64)
        A[c0][22*b0+c1-3][128*S1+c2-6]
                =shared_A[c0][c1][c2];
  __syncthreads();
}
```

Figure 10: CUDA kernel code using shared memory (without border tile conditions)

4.3 Refinement of the schedule

We would like to present a set of adjustments to the previously presented hybrid schedule which take advantage of additional knowledge about how our generic CUDA code generator to further improve the generated code. The three main changes introduced are: 1) making the offset of the tiles invariant to the time dimension, 2) ensure aligned memory loads and 3) avoid modulo operations in the core computations.

4.3.1 Alignment

We want to ensure that load instructions 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. In the previous section we ensured that all these offsets are independent of the time dimension T. For the case where the size of the innermost data space dimension is a multiple of the minimal alignment, and the tile width along the innermost dimension is also a multiple of the minimal alignment, all loads are either perfectly aligned or none of them is aligned. To ensure perfect alignment, the schedule is translated such that the tiles are distributed along individual dimensions, the global memory loads being fully aligned.

To implement this translation, we replace the occurrences of s_i in the original schedule by variables s'_i, defining them as the translation by offset o_i of the original variable s_i.

4.3.2 Avoiding modulo operations

Another point that needs to be addressed is the occurrence of modulo operations in the array subscripts of the computational statement. When mapping the different iterations of one of the inner space dimensions s_i with $i \geq 1$ to the CUDA threads the mapping is cyclic. In the optimal case where there are exactly as many threads as iterations along a space dimension, all statement instances are executed in a single line of code without any conditions and with no thread divergence. However, with the current schedule the access functions of this optimal statement still require a modulo operation, because it may still happen that the upper part of the thread group is calculating the lower part of the iteration space. This is because the subset of the iteration space that is executed along a dimension changes due to the sloped tiles, and the first thread does not always point to the first iteration that needs to be executed. By ensuring that the first thread always executes the first iteration on a certain dimension, we can avoid the introduction of modulo operations.

4.4 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 performance heuristics used during code generation time.

For a given input program and tiled schedule, there may be a wide spectrum of code generation choices. All of these respect the ordering of statement iterations dictated by the schedule, but they may result in widely different code quality and performance [15]. Polyhedral code generators regularly need to choose between different ASTs that implement the same behavior. In general such decisions are taken automatically by some internal heuristics. For the code generator provided by isl [17] we made it possible to influence such decisions on a per statement and per loop iteration level, such that it is now possible to adapt decisions e.g. to unroll iterations, to balance between code size and control overhead and even to apply different code generation strategies for different parts of the program. We used this flexibility to specify code generation heuristics that are optimal for stencil computations and our specific tiling scheme.

4.4.1 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 schema. 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 generate specialized code for the core tiles.

To parameterize the code generation accordingly, we first need to compute the set of iterations that are part of the core computation. For this we use the previously calculated schedule, but project out the point loops of the tiling. This yields a map that maps elements of the iteration space to the tile coordinate they are executed in ($[t, s_0, s_1, \ldots, s_n] \to [T, 0, S_0, S_1, \ldots, S_n]$). If we now apply this mapping once on the iteration space itself and once on the complement of the iteration space, we get a set of tiles that are fully or partially executed as well as a set of tiles tiles that are partially or not at all executed. Subtracting the latter from the former yields the set of full tiles. We now provide this set to the polyhedral code generator, which uses it to check at each loop level if it is legal to introduce specialized code for the core computation. If this is the case, such specialized code is introduced and protected with the necessary run-time conditions.

4.4.2 Unrolling for hybrid tiled stencils

Unrolling is often a beneficial transformation, but it is especially useful for stencil operations tiled with our hybrid tiling approach. As stated in the previous section, we construct our 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 needs to be executed within a single thread is relatively low. Hence, we can unroll the point loops within the tile to create straight code without any conditions. This does not only reduce control, but it also exposes 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 is necessary to ensure that all conditions can be specialized or eliminated in the unrolled code, simplifying them according to the context in which an instruction is unrolled [15].

5 Related Work

Automatic generation of high-performance code for stencil computations has made rapid progress in the past 2 years. Holewinski's overtile [8] and Grosser's split tiling [7] have set the reference performance 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 auto-tuning, and targeting both CPUs and GPUs [4].

On the other hand, ppcg [18], 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, a variant of the Pluto algorithm [3]. Reservoir Labs' R-Stream is also a reference polyhedral compiler targeting GPUs [10, 16]. 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 interprocedural analysis on the input code.

CPU-only frameworks are also available. Pochoir [14] is a domain-specific framework embedded into C++. The diamond tiling algorithm by Bandishti et al. [2] is closely related our approach while supporting combinations of affine 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.

6 Experimental Results

	laplacian 2D	**heat 2D**	**gradient 2D**	**ftdt 2D**	**laplacian 3D**	**heat 3D**	**gradient 3D**
ppcg	30.6	46.8	37.3	15.5	11.5	38.0	28.7
Par4All	39.3	49.0	103.6	invalid CUDA	18.2	**52.3**	75.0
Patus	crash	crash	n/a	n/a	20.1	19.6	n/a
Overtile	63.5	60.5	128.0		21.8	41.1	**89.0**
hybrid	**82.5**	**125.0**	**140.0**	**21.6**	**26.5**	43.8	66.7

Table 1: NVIDIA GTX 470

	laplacian 2D	**heat 2D**	**gradient 2D**	**fdtd 2D**	**laplacian 3D**	**heat 3D**	**gradient 3D**
ppcg	5.8	8.2	5.6	2.4	1.9	6.7	4.9
Par4All	7.0	7.1	18.9	invalid CUDA	3.1	9.1	14.5
Patus	crash	crash	n/a	n/a	3.9	3.2	n/a
Overtile	15.8	13.5	21.2		4.7	8.7	**16.9**
hybrid	**18.6**	**26.0**	**25.7**	**5.3**	**6.4**	**9.9**	14.5

Table 2: NVS 5200

We evaluate our approach by comparing hybrid hexagonal tiling against Patus [4], overtile [8], Par4All [1] as well as the unmodified ppcg compiler [18]. We were not able to obtain a license for comparative evaluation with R-Stream [10].

As benchmarks we use a laplace kernel with two space dimensions, a 2D heat and 2D gradient stencil as well as a two-dimensional, multi-statement fdtd kernel. We also run a laplace, heat and gradient kernel each having three space dimensions. For the two-dimensional stencils we run a data size of 2048^2 with 512 time steps whereas the three dimensional stencils where run on a data set with 256^3 elements using 128 time steps. We do not use 1D stencils because the hybrid method boils down to existing hexagonal or split tiling in this case [7]. All calculations have been performed as `float` and all timings include the data transfer overhead to and from the GPU. The experiments were performed on a NVIDIA NVS 5200M mobile GPU as well as a NVIDIA GeForce GTX 470.

For each tool, we sought to auto tune for the optimal tile sizes for the implemented tiling schema and a specific benchmark. For ppcg, we used the tile sizes found empirically and used in a prior publication [18]. For Patus and overtile we used the provided autotuner. For Patus the auto-tuner was run until it finished, for overtile we explored around 800 tile sizes for each benchmark. For our hybrid tiling we manually selected reasonable tile sizes. For Par4All we used the provided tile size heuristic. All tools where run with default arguments, except for Par4All we used the flags `-cuda -com-optimization` to enable GPU code generation.

Figure 1 gives the results for the GTX 470 and Figure 2 gives the result for the mobile GPU NVS 5200. As a general-purpose compiler, ppcg is able to create code for all benchmarks, but does not reach optimal speed. Patus, on the contrary, does always succeed. For the gradient kernels it does not support the `sqrt` instruction, whereas it simply throws an exception for the other 2D kernels.[1] Even though Patus supports multiple statements executed at one time point as well as different stencils executed at different regions of the iteration space, we were not able to model a multi-statement stencil like fdtd. For the remaining 3D benchmarks, Patus performs reasonably well on the laplacian 3D benchmark, but not as good on the heat 3D kernel. Except for one benchmark, Par4All produces reasonably well performing code, even reaching the top performance for the 3D heat kernel with 52 GFLOPS on the GTX 470. This is quite impressive for a general-purpose compiler. overtile performs well on 2D kernels. And with gradient 3D, overtile even produces the fastest kernels of all GPU compilers.

The last row describes our own hybrid tiling compiler. For all 2D kernels, on both the GTX470 and the NVS 5200, we are able to report notably better performance than all previous techniques. For the 3D kernels the performance of hybrid tiling is also excellent, but generally not the best of all approaches. This can be explained when observing that the top performers, Par4All and overtile, did not use time tiling for 3D stencils. Par4All does not support time tiling in general, whereas the overtile auto tuner found that time-tiling does not give the best performance. When the dimension increases, time tiling brings diminishing returns in terms of data reuse, because much reuse can already been exploited on space dimensions. Similarly, the extra dimensions expose sufficient amounts of concurrency to saturate the processor, without the need to leverage one additional (time) dimension to free some space iterations for an increased parallelism degree. To validate this analysis, we disabled time tiling for our hybrid hexagonal tiling, and repeated the experiment on 3D stencils. Table 3 shows the results of this experiment for two data sizes, the original 256^3 as well as a larger data size of 384^3. In this configuration, our tool generates the best code for laplacian 3D, and about the same as the others for heat and gradient 3D.

GTX 470	256^3	384^3
laplacian-3d	22.40	33.9
heat-3d	32.3	48.5
gradient-3d	45.0	75.0

NVS5200	256^3	384^3
laplacian-3d	7.1	7.0
heat-3d	11.7	11.4
gradient-3d	16.6	17.0

Table 3: Space tiling for 3D stencils

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

[1] The crash was reported to the author, but is not yet fixed

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

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: Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012*, 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)*, pages 676–687. IEEE, 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] J. Holewinski, L.-N. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *Supercomputing*, pages 311–320. ACM, 2012.

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

[10] 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 *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 51–61, New York, NY, USA, 2010. ACM.

[11] 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 *International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*, pages 72:1–72:11. IEEE Computer Society Press, 2012.

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

[13] A. Taflove. Computational electrodynamics: The finite-difference time-domain method. 1995.

[14] 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.

[15] 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)*, LNCS, pages 185–201, Vienna, Austria, Mar. 2006. Springer.

[16] N. Vasilache, B. Meister, M. Baskaran, and R. Lethin. Joint scheduling and layout optimization to enable multi-level vectorization. In *IMPACT'12*, Paris, France, Jan. 2012.

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

[18] 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.

[19] S. Verdoolaege and T. Grosser. Polyhedral extraction tool. In *PACT*, 2012.

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

[21] D. G. Wonnacott and M. M. Strout. On the scalability of loop tiling techniques. In *Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques (IMPACT)*, January 2013.

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399