High-Performance Code Generation for Stencil Computations on GPU Architectures

Justin Holewinski Louis-Noël Pouchet P. Sadayappan

Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{holewins, pouchet, saday}@cse.ohio-state.edu

ABSTRACT

Stencil computations arise in many scientific computing domains, and often represent time-critical portions of applications. There is significant interest in offloading these computations to high-performance devices such as GPU accelerators, but these architectures offer challenges for developers and compilers alike. Stencil computations in particular require careful attention to off-chip memory access and the balancing of work among compute units in GPU devices.

In this paper, we present a code generation scheme for stencil computations on GPU accelerators, which optimizes the code by trading an increase in the computational workload for a decrease in the required global memory bandwidth. We develop compiler algorithms for automatic generation of efficient, time-tiled stencil code for GPU accelerators from a high-level description of the stencil operation. We show that the code generation scheme can achieve high performance on a range of GPU architectures, including both nVidia and AMD devices.

Categories and Subject Descriptors

D.1.3 [**Programming techniques**]: Concurrent programming—*Parallel Programming*; D.3.4 [**Programming Languages**]: Processors—*Code Generation, Compilers*

Keywords

GPU, OpenCL, Overlapped Tiling, Stencils, DSL

1. INTRODUCTION

Stencils represent an important computational pattern used in scientific applications in a variety of domains including computational electromagnetics [18], solution of PDEs using finite difference or finite volume discretizations [16], and image processing for CT and MRI imaging [3, 4]. A number of recent studies have focused on optimizing stencil computations on multicore CPUs [2, 6, 8, 17, 19] as well as GPUs [11–13].

ICS'12, June 25–29, 2012, San Servolo Island, Venice, Italy.

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. For multi-core processors, stencil computations are often memory-bandwidth bound when the collective data for all grid points exceeds cache size, since each grid point is accessed at each time step. Time-tiling, i.e., tiling along the time dimension, is useful in enhancing data locality. The standard approach to time-tiling of stencil computations requires loop skewing to make tiling legal and this results in loss of inter-tile concurrency [10], since inter-tile dependences are introduced in the spatial directions due to the skewing. The approach of "overlapped tiling" [10], also called "ghost zone" optimization [3,11], has been used for preserving concurrency in parallel time-tiled execution of stencil computations. However, we are unaware of any fully automated compiler approach for the generation of overlapped-tiling code for execution on GPUs. In this paper, we develop compiler algorithms for automated GPU code generation for stencil computations and demonstrate effectiveness through experimental evaluation using a number of stencils on four GPU platforms.

The paper is organized as follows. In Sec. 2, we provide some background on GPUs and the key issues in achieving high performance with them. In Sec. 3, we formalize the class of stencil computations we consider. The compiler algorithms for generation of overlapped tiled code for GPUs are presented in Sec. 4. Sec. 5 presents experimental results. Related work is covered in Sec. 6, and we conclude in Sec. 7.

2. GRAPHICS PROCESSING UNITS

Graphics Processing Units (GPUs) are massively-threaded, many-core architectures with peak floating-point throughput of over 1 TFLOP/s. NVIDIA GPUs contain hundreds of cores (streaming processors) arranged in tightly coupled groups of 8–32 scalar processors per streaming multi-processor. Threads are grouped into thread blocks that are scheduled on a streaming multi-processor and cannot migrate. A single multi-processor can concurrently handle several blocks of threads using zero-overhead hardware multi-threading to interleave their execution on its cores. Parallelism is exposed both across thread blocks and within thread blocks. Threads within a block are cooperative and can synchronize with each other, but threads in different blocks cannot synchronize, even if they are scheduled on the same streaming multi-processor.

Each thread has access to global, off-chip memory and a shared scratch-pad memory that is shared among all threads within a block. Threads within a block can communicate

and exchange data through shared memory. Thread synchronization can be achieved by the use of barrier instructions that cause all threads within a block to stop at the barrier until all threads have reached the barrier. Thread synchronization is, in general, not feasible across thread blocks.

Architectural Model: Low-level programming models are commonly used to write GPU programs. The two most common models are CUDA [14] and OpenCL [9, 15]. In both models, the programmer writes an imperative program (called a *kernel*) that is executed by each thread on the device. Threads are spawned in 1-, 2-, or 3-dimensional rectangular groups of cooperative threads, called blocks (CUDA) or work-groups (OpenCL). A 1- or 2-dimensional grid of blocks is used to schedule the thread blocks. Both the size and number of thread blocks are fixed when launching a GPU kernel and cannot be changed after the threads have launched.

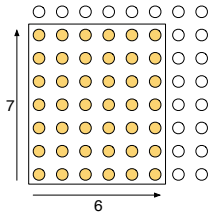
Efficient GPU programs typically involve the scheduling of hundreds of threads per streaming multi-processor to hide memory latency. The streaming multi-processors schedule threads at the granularity of warps, which comprise 32 threads on previous and current generation architectures. The thread scheduler time-shares the streaming multi-processors between all currently active warps, and thread context switches incur no overhead.

Challenges: Several sources of inefficiency can arise when developing GPU applications. GPU devices provide a very high off-chip memory bandwidth (up to 192 GB/sec for the GTX 580), but *this bandwidth is only achievable with coalesced access*. Data from the off-chip memory is transferred to the GPU device in contiguous blocks and therefore high bandwidth can be achieved only when requests by concurrent threads in a warp fall within such contiguous blocks. When non-contiguous memory locations are accessed by threads, the achieved bandwidth can be much lower than the peak, leading to stalling and wasted compute cycles. Branch divergence is another source of inefficiency. Threads within a warp that follow different control paths are serialized, again leading to wasted compute cycles. Traditional approaches to time tiling of stencil computations to enhance data reuse for CPUs do not translate well to GPUs because they lead to uncoalesced memory access and divergent branching of threads.

Another challenge comes from shared scratch-pad memory implemented as a banked memory system. If concurrently executing threads in a block make requests to shared memory locations in the same bank, a bank conflict occurs and the requests are serialized. Therefore, to achieve optimal usage of shared memory, *concurrently executing threads should access data from different banks*. We present in this paper an automated code generation approach to overcome these challenges, for the class of stencil computations as described in Section 3.

3. STENCIL COMPUTATIONS

Recent work has shown promise for high performance by use of overlapped tiling on GPUs [11] for stencil computations. In this paper, we present an automated approach to generate efficient overlapped tiling code for stencil computations on GPUs. We first describe the features of a Domain-Specific Language (DSL) to describe stencil computations, such that any program written in this language can be processed automatically by our compiler. The code generation algorithms for overlapped tiling are detailed later in Section 4.



$$G = [0,5] \times [0,6]$$

Figure 1: Example of a 2-D grid

3.1 Stencil DSL

Our stencil DSL models iterative methods operating on (dense) *fields* atop a Cartesian *grid*. In addition to the the data space, we describe the *stencil function* which is applied iteratively on each point of the grid.

We enforce some constraints on the DSL to facilitate transformation to efficient code. First, to enable compile-time generation of overlapped tiled code, we need to model the halo of the stencil, and it needs to be exactly computable at compile time. This implies that the neighboring relationship remains constant during the computation. Second, in order to perform tiling along the time dimension, the computation must iterate a constant number of time steps before any field-spanning operation is performed (e.g., a convergence check). Finally, we remark that to ease code generation, we enforce the use of a second temporary field to store the result of the application of the point function on the input field. Thus in this work, we address Jacobi-like methods, and not Seidel-like methods.

We now define the three key concepts of computation grids, fields, and stencil functions before putting it together to describe a full program using this DSL.

Computation Grid: Every stencil computation that we consider is defined on a *computation grid*, which is a bounded, rectangular region in $\mathbb{Z}^n$. The supporting grid can be seen as a Cartesian coordinate system on a contiguous subset of $\mathbb{Z}^n$. A grid can be defined as a union of "sub-grids". This is particularly relevant to define regions in the space that may have different physical properties, such as boundaries. So we have:

$$G \subset \mathbb{Z}^n$$

Sub-grids G^i of the grid G are non-intersecting subsets of integer points, defining a partition of G.
A sub-grid is defined by an origin point α_i and and end point β_i in each of the n dimensions, with $\alpha_i \leq \beta_i$. That is:

$$G^i = \{[\alpha_1..\beta_1] \times [\alpha_2..\beta_2] \times \ldots \times [\alpha_n..\beta_n], \alpha_i, \beta_i \in \mathbb{Z}, \alpha_i \leq \beta_i\}$$

Fields: Once we have defined our computation grid, we can define the data attached to this grid for a particular computation. This takes the form of a (series of) fields attached to a particular (sub-)grid. Multiple fields are associated with a single grid typically when a stencil uses data from multiple sources during the computation. A field is implemented as a data structure that maps a value to every point in the grid. The type of these values can be simple scalar values or

complex recursive types, defined by the following grammar:

$$\begin{aligned} ElemType &\longrightarrow \text{real} \mid \text{integer} \mid \text{vector } n \text{ of } ElemType \\ &\quad \mid StructType \\ StructType &\longrightarrow \{field_1 : ElemType_1, \\ &\quad field_2 : ElemType_2, ...\} \end{aligned}$$

We denote $\mathcal{F}_G$ a field $\mathcal{F}$ associated to a (sub-)grid G.

Stencil Functions: The last component of a stencil computation is the sequence of *stencil functions* that are applied iteratively to the fields defined on the computation grid. A stencil function defines a computation that is applied to each point of a (sub-)grid. Different functions can be used for different sub-grids, as for instance to handle boundary conditions. The stencil function uses neighboring field points in the same field or other fields in its computation of the new value for the point.

Given a collection of p fields $\mathcal{F}^k_{G^k}$, a stencil function f is a function

$$f : \mathcal{F}^1_{G^1} \times \ldots \times \mathcal{F}^p_{G^p} \rightarrow T_{\mathcal{F}^p}$$

where $T_{\mathcal{F}^p}$ is the type of elements in field $\mathcal{F}^p$. The domain of definition G_f of this function is a sub-grid of G^p, and we have $\forall i \in [1..p], G_f \in G^i$. This function is implicitly invoked for each point of G_f.

As an example, consider a simple 2-D stencil function that is defined on the computation grid introduced in Figure 1. We define the stencil as:

$$\begin{aligned} f(A_G)[1..N-2, 1..M-2] = &A[-1,0] + A[0,-1] + A[0,0] + \\ &A[+1,0] + A[0,+1] \end{aligned}$$

where the notation $A[i,j]$ shows an access to the field A at grid coordinates (i,j) from the current grid point, and $[1..N-2, 1..M-2]$ specifies the range of the grid over which the stencil function should be applied[1]. Since we are using Cartesian grids in $\mathbb{Z}^n$, the grid offsets must be integer values. Furthermore, they must be constant. This notation naturally extends to grids of any dimensionality. This stencil function effectively computes a new value for a grid point based on the immediate neighbors in both the horizontal and vertical directions.

In this example, we assume that the addition operator $(+)$ is well-defined for the value type of elements in the field A. For scalar, vector, and matrix types, the operator is clearly well-defined. However, if our value type is of structure type, the addition operator may not be well-defined in general.

Program Specification: Using the previous definitions of *computation grid*, *field*, and *stencil function*, we can now define a complete *stencil program*. A stencil program defines the underlying computation grid, a set of one or more fields that are associated to the grid, a sequence of stencil functions that are applied on the (sub-)grids, and the number of steps the iterative process is repeated. For each field, we must define the value type for the points. We must also define the region over which each stencil function operates. Finally, we must define the number of iterations over which the stencil computation will occur, which is attached as an attribute of the grid. In each iteration, every stencil function is evaluated in the proper region. Stencil programs in our framework are formally defined as:

$$\begin{aligned} Program &\longrightarrow GridDef \;\; FieldDefs \;\; Funcs \\ GridDef &\longrightarrow \texttt{Bounds}, \;\; \texttt{TimeSteps} \\ FieldDefs &\longrightarrow \texttt{Name} \; : \; ElemType \;\; FieldDefs \\ &\mid \epsilon \\ Funcs &\longrightarrow \texttt{Name} \; (\; \texttt{Arrays} \;) \; [\; \texttt{Bounds} \;] \\ &\qquad = \; Expr \; Funcs \\ &\mid \epsilon \\ Expr &\longrightarrow \texttt{FieldAccess} \mid Expr \; \texttt{Op} \; Expr \end{aligned}$$

The bounds assigned to stencil functions define the sub-grids in the computation. A stencil function always operates on a sub-grid of the entire computation grid, and it may span the entire computation grid.

In the presence of multiple stencil functions, the semantics of a stencil program asserts that they are executed in sequential order. That is, the first stencil function is applied to each point in its region and the results are written back to the data grid before any evaluation of the second stencil function occurs. There is no ordering constraint between grid points within a stencil function evaluation. That is, each evaluation of a stencil function within a single outer iteration occurs concurrently.

```
1  real A[M];
2  real A_tmp[M];
3
4  copy(A_tmp /* dest */, A /* src */);
5
6  for (n = 0; n < T; ++n) {
7    A[0] = A_tmp[0];
8    for (i = 1; i < M-1; ++i) {
9      A[i] = 0.333 * (A_tmp[i-1] + A_tmp[i] + A_tmp[i+1]);
10   }
11   A[M-1] = A_tmp[M-1];
12   swap(A_tmp, A);
13 }
```

Figure 2: Pseudocode for a simple 3-point stencil.

3.2 Example

As a concrete example, we consider the following stencil program, where M is a parameter:

$$\begin{aligned} G &= [0, M-1], 64 \\ A_G &: \text{real} \\ f_1(A_G)[0] &= A[0] \\ f_2(A_G)[1..M-2] &= 0.333 * (A[-1] + A[0] + A[1]) \\ f_3(A_G)[M-1] &= A[0] \end{aligned}$$

In this program, we define the computation grid as the region $[0, M-1]$ in $\mathbb{Z}$, where the stencil should be applied iteratively over 64 time steps. We define one field A_G which associates a field of real numbers onto the computation grid. Finally, we define a three stencil functions f_1, f_2 and f_3. f_2 computes an average of grid point values, defined in the range $[1, M-2]$ on the computation grid. The boundary points are updated with specific equations defined by the stencil functions f_1 and f_3.

The semantics of this program is that of the C-like pseudocode as shown in Figure 2. Note the use of the `A_tmp` array to help implement the semantics of the stencil program. In

[1]Here, it is assumed that N and M are the bounds of the computation grid.

each iteration (`t`-loop), the new values for A are computed using the *old* values of A from the previous iteration. Therefore, the `A_tmp` array is used to cache the old values of A. Also, the variable `T` is used as a parameter for the number iterations over which to perform the stencil computation.

4. OVERLAPPED TILING

4.1 Overview

Tiling for stencil computations is complicated by data sharing between neighboring tiles. Cells along the boundary of a tile are often needed by computations in surrounding tiles, requiring communication between tiles when neighboring tiles are computed by different processors. To compute a stencil on a cell of a grid, data from neighboring cells is required. These cells are often referred to as the *halo* region. In general, to compute an $N \times M$ block of cells on a grid, we need an $(N+n) \times (M+m)$ block of data to account for the cells we are computing as well as the surrounding halo region, where n and m are constants derived from the shape of the stencil. For GPU devices, the halo region needs to be re-read from global memory for every time step as surrounding tiles may update the values in these cells. This limits the amount of re-use we can achieve in scratchpad memory before having to go back to global memory for new data. The new cell values produced in each time step must also be re-written to global memory as other tiles may require the data in their halo regions. In addition to the cost of going to global memory for each time step, global synchronization is also required to ensure all surrounding tiles have completed their computation and written their results to global memory before new halo data is read for each tile.

To get around these issues, overlapped tiling has been proposed [10] as a technique to reduce the data sharing requirements for stencil computations by introducing redundant computations. Instead of forcing tile synchronization after each time step to update the halo region for each tile, each tile instead redundantly computes the needed values for the halo region. This allows us to efficiently perform time tiling and achieve high performance on GPU targets.

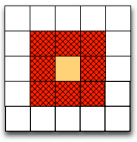
Consider a simple 2-D Jacobi 9-point stencil like the one shown in Figure 3. In order to compute one time step of an $n \times n$ tile, we need to read $(n+2) \times (n+2)$ cells into scratchpad memory, compute the stencil operation on each of the $n \times n$ points, then write back $n \times n$ points to global memory. Now, let us consider the computation of two time steps of the stencils on a single block, without having to go back to global memory between the two time steps. If we read $(n+4) \times (n+4)$ cells into memory, we can compute a $(n+2) \times (n+2)$ tile of cells in the first time step, which includes the results for our original $n \times n$ tile as well as the halo region needed for the next time step. If we again apply the stencil operation to the $(n+2) \times (n+2)$ tile, we correctly compute the inner $n \times n$ tile for the second time step but the results computed in the halo region for the second time step are not correct. However, this is not a problem since we only care about the inner $n \times n$ region. An illustration of this computation is presented in Figure 4.

4.2 Code Generation

To generate efficient GPU code, we need to tile the data space of the stencil grids into units that can be computed by a single thread block on a GPU device without any needed block synchronization. For each stencil operation, we need

```
1  for(t = 0; t < T; ++t) {
2    for(i = 1; i < N-1; ++i) {
3      for(j = 1; j < N-1; ++j) {
4        A[i][j] = CNST * (B[i  ][j  ] +
5          B[i  ][j-1] + B[i  ][j+1] +
6          B[i-1][j  ] + B[i-1][j-1] +
7          B[i-1][j+1] + B[i+1][j  ] +
8          B[i+1][j-1] + B[i+1][j+1]);
9      }
10   }
11   for(i = 1; i < N-1; ++i) {
12     for(j = 1; j < N-1; ++j) {
13       B[i][j] = A[i][j];
14     }
15   }
16 }
```

(a)



(b)

Figure 3: Jacobi 9-Point Stencil. In (a), C code is shown for the stencil, and in (b), the accessed data space is shown for one grid point. The tan cell is the (i, j) point, and each red hashed cell is read during the computation of the (i, j) cell.

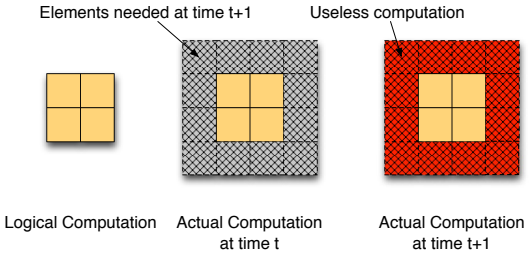


Figure 4: Overlapped Jacobi 9-Point Stencil for $T = 2$.

Algorithm 1: Overlapped-tiling code generation algorithm

Input: P : Program, B : Block Size, T : Time Tile Size, E : Elements Per Thread
Output: P_{gpu} : GPU Kernel, P_{host} : CPU function

1 R ← DetermineTileSize(P, B, T)
`//Generate GPU Kernel`
2 P_{gpu} ← `GenerateGPUKernel` (P, B, R, T) (§ 4.3)
`//Generate CPU Function`
3 P_{host} ← `GenerateHostCode` (P, B, R, T) (§ 4.4)

to determine the region of grid points that will be computed by each thread block, including any needed redundant computation for intermediate time steps. The input to our code generation algorithm is a sequence of stencil operations and the output is overlapped-tiled GPU code and a host driver function.

Code generation for overlapped tiling on GPU architectures requires us to first identify the footprints of the stencil(s), determine proper block sizes for the target architecture, and then generate the GPU kernel code and host wrapper code. The overall algorithm is presented in Algorithm 1. The following sections describe all of the steps for the code generation algorithm.

Determining Abstract Tiles: To implement overlapped tiling, we need to create tiles such that a single thread block can compute all grid points for all stencil operations for T time steps, where T is the time tile size, without requiring any inter-block synchronization. For single-stencil computations, we only need to worry about the halo regions that are needed in intermediate time steps. However, for multi-stencil computations, we need to account for grid cells that may be needed for later stencil computations.

To determine the required computation for each sub-grid, we work backwards across all stencils and all time steps within our time tile. We start by assuming the last stencil computation in the last time step operates on a rectangular tile that is defined by an origin $\vec{x}_0$ and length $\vec{l}_0$. For each sub-grid used to compute this stencil, we can build a rectangular region that forms a tighest-possible bound on the sub-grid that includes all grid points needed for that stencil computation.

Once rectangular regions have been built for all involved sub-grids, we move onto the previous stencil in execution order and perform the same computation. Once we finish with the first stencil in execution order, we wrap around to the last stencil and continue the process again until we have processed all stencils for all time steps in our time tile. When we generate a new rectangular tile for a sub-grid for which we have already computed a tile, we form the new tile by forming the union of the old and new result and then taking the tightest-possible rectangular bound.

This tile size selection procedure is formalized in Algorithm 2. The `GetUpdatedGrid`(s) function returns the sub-grid that is updated for the given stencil, and the function `GetSourceGrids`(s) returns all sub-grids that are used on the right-hand side of the given stencil. The function `GetRequiredBounds`($g, s, \vec{x}, \vec{l}$) returns a rectangular region that represents the grid cells that are needed in grid g to compute stencil s in the region defined by $\vec{x}$ and $\vec{l}$. For a given stencil program P, the `ExtractSubGrids`(P) function returns all sub-grids that are touched by any point function. Finally, the `RectangularHull`($\cup_i(\vec{x}_i, \vec{l}_i)$) function returns the region that forms the tightest-possible rectangular bound on the given (possibly non-rectangular) region.

The results of this algorithm are a set of rectangular regions defined by $\vec{x}_g$ and $\vec{l}_g$, one for each sub-grid g. These regions define the grid cells that must be processed in each time step for each sub-grid in order to properly compute the final stencil in the final time step of the time tile. These regions will later be used to derive the per-block computation code. For the case of multiple grid points processed per thread, the $\vec{l}_g$ quantities are just multiplied by the number of elements per thread, element-wise in each dimension.

Note that to compute the final result for our $(\vec{x}_0, \vec{l}_0)$ region for each sub-grid, we do not need to compute values for the entire $(\vec{x}_g, \vec{l}_g)$ grid in each time step. However, on GPU architectures, it is more efficient to perform this redundant computation and let each thread perform the same amount of work in each time step. Otherwise, threads would contain control-flow instructions that would cause branch divergence and lower overall performance.

1-D Multi-Stencil Example: Let us now consider a multi-stencil computation. In the following example, we use two

Algorithm 2: Finding the tile size.

```
Name: DetermineTileSize
Input: P : Stencil Program, T : Time Tile Size
Output: (x_g, l_g) ∀ Sub-Grids
S ← ExtractStencilFunctions (P)
DG ← ExtractSubGrids (P)
foreach SubGrid g ∈ DG do
    (x_g, l_g) ← (x_0, l_0)
end
foreach TimeStep t in (T, ..., 1) do
    foreach StencilFunction s in Reverse (S) do
        g ← GetUpdatedGrid (s)
        src ← GetSourceGrids (s)
        foreach SubGrid dg ∈ src do
            (x'_dg, l'_dg) ← GetRequiredBounds (dg, s, x_g, l_g)
            (x_dg, l_dg) ← RectangularHull ((x_dg, l_dg) ∪ (x'_dg, l'_dg))
        end
    end
end
```

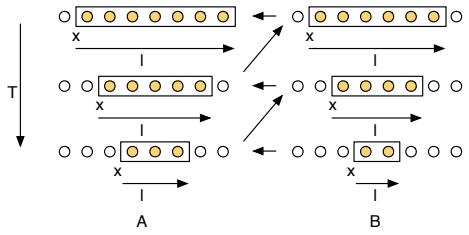


Figure 5: Tile region computation for 1-D multi-stencil.

stencil computations operating over two data grids, defined as:

$$f_A(B_G)[1..N-2] = B[-1] + B[0]$$
$$f_B(A_G)[1..N-2] = A[0] + A[1]$$

Again, we use a time tile size of three. We start by assigning an initial tile of $(\vec{x}_B, \vec{l}_B) = (x_0, l_0)$ to grid B. We see that the computation of B only depends on the grid A, so we use the index expressions to compute the needed region of A, which is $(\vec{x}_A, \vec{l}_A) = (x_0, l_0 + 1)$. We then backtrack to the computation for grid A, and determine that the needed region in grid B to compute A is $(\vec{x}_B, \vec{l}_B) = (x_0 - 1, l_0 + 2)$. This completes time step three, so we reverse to time step two and perform the same computation. Backtracking all of the way to the beginning of time step one, we have:

$$(\vec{x}_A, \vec{l}_A) = (x_0 - 2, l_0 + 5)$$
$$(\vec{x}_B, \vec{l}_B) = (x_0 - 2, l_0 + 4)$$

A pictorial representation of this process is shown in Figure 5

4.3 Generation of GPU Kernel Code

Now that we know the amount of redundant computation that is needed for each sub-grid, we can generate the GPU kernel code for the stencil time tile. For this, we need to

Algorithm 3: Generating shared-memory definitions

Name: GenerateSharedMemory
Input: P : GPU Program, $\vec{B}$: Block Size, $\vec{E}$: Elements Per Thread, D : Sub-Grids
Output: P : GPU Program
1 **foreach** SubGrid g *in* D **do**
2 P ← `DeclareSharedRegion` (P, `NumberOfPoints` ($\vec{B}$, $\vec{E}$), g)
3 **end**

define a set of parameters that are used as input to the code generation algorithm:

Parameter	Description
E	The number of cells to process per thread
$\vec{B}$	The desired GPU block size (x, y, z)
T	The time tile size

Using the tile regions determined by Algorithm 2, we know that for each sub-grid g, the region of computation for a given thread block is given by $(\vec{x}_g, \vec{l}_g)$. Let us define the region with the maximum size as $(\vec{x}_{max}, \vec{l}_{max})$, where $\vec{x}_{max}$ and $\vec{l}_{max}$ are affine expressions in $\vec{x}_0$ and $\vec{l}_0$, our abstract tile size. Here, our definition of maximum size is the largest number of grid points contained in the region defined by $(\vec{x}_g, \vec{l}_g)$. Then, given that $\vec{B}$ is our desired thread block size, we can solve for $\vec{l}_0$ with:

$$\vec{l}_{max} = \vec{B} \tag{1}$$

The value $\vec{x}_0$ is still a symbolic entity, but we now have a concrete value for $\vec{l}_0$ which we can use to generate GPU kernel code. An important observation at this point is that the region $(\vec{x}_g, \vec{l}_g)$ defines the region of real computation for the stencil operation writing sub-grid g and the halo region (e.g. the region that is computed redundantly) is defined as:

$$Halo(g) = (\vec{x}_{max}, \vec{l}_{max}) - (\vec{x}_g, \vec{l}_g) \tag{2}$$

Note that the halo region is, in general, not a rectangular region. It is instead a rectangular bound around the non-halo region.

Shared/Local Memory Size: To take advantage of the GPU memory hierarchy, it is necessary to use shared/local memory to cache results whenever possible. To generate high-performance GPU code using overlapped tiling, we need to determine how much shared memory is needed to cache the redundant computations that are performed. This computation is straight-forward, since we only need to account for the results produced by each thread. Therefore, the amount of shared memory we need for the computation (in number of data elements) is simply:

$$SharedSize = \sum_g (\text{Area}(\vec{B})) \tag{3}$$

This allows each thread to cache the result it produces for each stencil computation. The Area() function simply returns the number of grid points within the region defined by $(\vec{x}_g, \vec{l}_g)$.

This process is formalized in Algorithm 3. Given a block size and the number of grid points to process per device thread, a shared memory region is declared for each sub-grid. This shared memory region is declared for the GPU program P. The `NumberOfPoints` function simply returns the number of integer points contained within the block defined by taking the component-wise multiplication of $\vec{B}$ and $\vec{E}$.

Thread Synchronization: Thread barriers are used between stencil operations to ensure that all threads have finished the computation for a particular stencil operation before any thread starts computing a value for the next stencil operation. The thread blocks compute completely independently, but the threads within a thread block must be synchronized as one thread may produce a result that is needed by another thread in the next stencil operation.

Block Synchronization: Our computation model dictates that for each stencil operation, a snapshot of the data is used as input for every evaluation of every grid point, and writes back to the grid are done after all evaluations have finished. In other words, all reads from global memory must see the same data. Unfortunately, this is hard to guarantee on GPU architectures which lack block synchronization and GPU-wide memory fences. To get around this issue, a buffering approach is used. For each sub-grid used as an output for a stencil computation, two grids are actually maintained in GPU memory. For each time tile, one version is used as input and another is used as output. Between time tiles, the host will swap the buffer pointers before the next kernel invocation. This ensures that all reads within a kernel invocation see the same data and no issues can arise from one thread block completely finishing a computation before another block starts.

Thread Code: We can now generate the per-thread code that will implement the stencil computation. This code implements the computation of all sub-stencils for $\left|\vec{E}\right|$ data elements over T time steps. In the first time step, each thread reads its needed data from global memory, performs the computation of the first stencil operation, and stores the result to shared/local memory. This process is repeated for each stencil operation in the stencil computation. For each subsequent time step, each thread reads its needed data from shared/local memory, performs the computation of each sub-stencil, in order, and stores the result to shared/local memory. In the last time step, *if the thread is not part of the halo*, the final result is written back to global memory.

This process is formalized in Algorithm 4. Here, code is generated for each stencil function evaluation for every time step of our time tile. A new GPU program object is created that represents our generated kernel code, and we generate code to evaluate multiple grid points per thread (if needed), and use shared memory to cache results whenever possible. Different implementations of the generate functions can be used to target different languages, including CUDA, OpenCL, LLVM IR, etc.

Example: As an example, let us consider a Jacobi 5-point stencil over 2 time steps. The stencil function can be defined as:

$$f(A_G)[1..N-1, 1..M-1] = 0.2 * (A[-1,0] + A[0,0] + A[1,0] + A[0,-1] + A[0,1])$$

The generated code for an OpenCL target will apply the stencil function twice, once at time t and again at time $t+1$. Global memory will only be read at the beginning of time

Algorithm 4: Thread code generation

Name: GenerateGPUKernel
Input: P : Stencil Program, $\vec{B}$: Block Size, $\vec{E}$: Elements Per Thread, T : Time Tile Size
Output: P_{gpu} : GPU Program

```
S ← ExtractStencilFunctions (P)
D ← ExtractSubGrids (P)
P_gpu ← NewGPUProgram ()
P_gpu ← GenerateSharedMemory (P_gpu, B, E, D) (Algorithm 3)
InShared ← ∅
foreach TimeStep t in T do
    foreach Function f in S do
        foreach Element e in Iterate (E) do
            foreach SubGrid g in GetSources (f) do
                if g ∈ InShared then
                    P_gpu ← GenerateSharedMemoryReads (P_gpu, g, e, f)
                else
                    P_gpu ← GenerateGlobalMemoryReads (P_gpu, g, e, f)
                end
            end
            P_gpu ← GenerateFunctionEvaluation (P_gpu, f, e)
        end
        P_gpu ← GenerateThreadSync (P_gpu)
        P_gpu ← GenerateSharedMemoryWrite (P_gpu, g, e)
        InShared ← InShared ∪ GetDestination (f)
        P_gpu ← GenerateThreadSync (P_gpu)
    end
end
foreach SubGrid g in D do
    foreach Element e in Iterate (E) do
        if ¬ Halo (g, e) then
            P_gpu ← GenerateGlobalMemoryWrite (P_gpu, g, e)
        end
    end
end
```

Algorithm 5: Host code generation

Name: GenerateHostCode
Input: P : Stencil Program, P_{gpu} : GPU Program, $\vec{B}$: Block Size, $\vec{E}$: Elements Per Thread, T : Time Tile Size, $\vec{N}$: Problem Size, T_{total} : Total Number of Time Steps, $(\vec{x}_0, \vec{l}_0)$
Output: P_{host} : Host Program
Output: P_{host} : Host Program

```
P_host ← NewHostProgram ()
DG ← ExtractSubGrids (P)
NumBlocks ← (N_x/l_{0,x}, N_y/l_{0,y}, ...)
foreach SubGrid g in DG do
    g_gpu,0 ← AllocateGPUBuffer (g)
    g_gpu,1 ← AllocateGPUBuffer (g)
    P_host ← CopyHostToDevice (P_host, g_gpu,0, g)
    P_host ← CopyHostToDevice (P_host, g_gpu,1, g)
end
Body ← { InvokeKernel (P_gpu, NumBlocks); Swap(g_gpu,0, g_gpu,1) ∀ SubGrid g in DG }
P_host ← GenerateTimeLoop (P_host, T_total / T, Body)
foreach SubGrid g in DG do
    P_host ← CopyDeviceToHost (P_host, g_gpu,†, g)
end
```

step t, and written at the end of time step $t+1$. The results computed in time step t will be cached in shared memory and read in time step $t+1$. A barrier will be placed between the time steps to ensure all shared memory writes complete before any shared memory reads occur in time step $t+1$.

4.4 Generation of Host Code

To be a complete code generation framework, host code is also needed to properly configure and execute the generated GPU kernels. For our purposes, this involves creating a C/C++ function that is responsible for copying input data to the GPU device before kernel invocation, setting up the proper grid and block sizes, invoking the kernel, and copying output data back to the host after kernel invocation. The general procedure is outlined in Algorithm 5, and details are provided in the following subsections.

Copy In/Out: Copy-in/copy-out code is generated by determining the amount of global halo that is needed, allocating buffers of the appropriate size, and copying data to/from the device at the appropriate time. A global halo is used to eliminate conditional behavior around the boundary in GPU kernel code. When time tiling is used, a thread that would ordinarily access the sub-grid boundary may now read several points beyond the edge of the grid. To make sure these memory accesses stay within bounds and that the intermediate results are correct, the boundary grid cells are replicated into a halo region of width equal to the maximum radius of all stencils that read from the sub-grid.

Thread Blocks: The GPU block size is pre-determined as an input to the code generation algorithm, but we also need to determine the number of thread blocks that will be executed. Remember that each thread block computes a region of $(\vec{x}_{max}, \vec{l}_{max})$ for grid g, where only $(\vec{x}_0, \vec{l}_0)$ is useful work. If $\vec{N}$ is our total problem size, then we need $|\vec{N}/\vec{l}_0|$ total blocks, where the division is performed element-wise and the length is a measure of the total number of blocks in the volume.

Time Loop: The host code is also responsible for implementing the outer time loop. Each invocation of the GPU kernel will compute T time steps of the stencil. If S is the total number of time steps, then the host code will invoke the kernel $\frac{S}{T}$ times. After each time tile, the input and output buffers are swapped to make the output from the previous time tile the input to the next time tile.

5. EVALUATION

In this section, we report on an experimental evaluation of the code generation scheme. using GPU devices from both AMD and nVidia – AMD A8-3850 APU (Radeon HD 6550D GPU), nVidia GTX 280, GTX 580, and Tesla C2050 devices. For the nVidia devices, we used the publicly available CUDA SDK 4.1 RC2 to execute OpenCL programs. For the AMD devices, we used the AMD Accelerated Parallel Processing SDK 2.6. Across all devices, we tested on Red Hat Enterprise Linux 6.2 AMD64 with the kernels and device driver versions recommended by the respective device vendors.

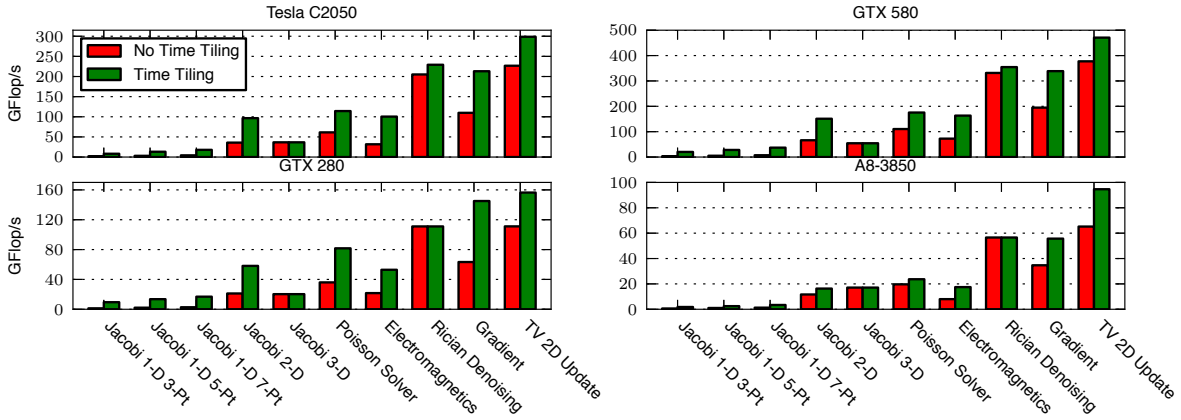


Figure 6: Performance of stencils across architectures

The characteristics of the tested GPU devices are as follows:

	GTX 280	GTX 580	Tesla C2050	AMD A8-3850
Peak SP	933 GF/s	1.58 TF/s	1.03 TF/s	480 GF/s
Peak B/W	141.7 GB/s	192 GB/s	144 GB/s	Variable[2]

The characteristics of the tested stencil programs are as follows:

Program	Elements Per Pt	Number of Arrays	Ops Per Pt	Dim.
Jacobi 1-D 3-Pt	3	1	3	1
Jacobi 1-D 5-Pt	5	1	5	1
Jacobi 1-D 7-Pt	7	1	7	1
Jacobi 2-D	5	1	5	2
Jacobi 3-D	7	1	7	3
Poisson Solver	9	1	9	2
Electromagnetics	2/2/3	3	11	2
Rician Denoising	5	2	44	2
Gradient	5	1	18	2
TV Update 2D	7	1	59	2

The *Jacobi* stencils are synthetic Jacobi-style stencil programs. The *Poisson Solver* stencil is an application of the Poisson PDE in two dimensions. The *Electromagnetics* stencil is an application of the 2-D Finite-Difference Time-Domain method [18]. The *Rician Denoising* and *TV Update 2D* stencils are components of the CDSC CT/MRI imaging pipeline [1]. The *Gradient* stencil is an application of a gradient operator on a two-dimensional grid.

5.1 Performance Analysis

We evaluated the performance improvement of overlapped tiled stencil code as generated by our algorithm by examining the performance over many stencil programs. We performed an exhaustive search of the parameter space (block size, time tile size, and spatial tile sizes) to determine the best performing version. For comparison purposes, we also show the results obtained by only searching in the spatial tile size and not performing time tiling. This allows us to evaluate the effectiveness of time tiling using overlapped tiling for these stencils.

[2]The AMD Fusion chips are integrated CPU/GPU devices with the GPU cores using system memory, making the peak bandwidth variable.

Figure 6 shows the results of this experiment. Each graph shows a different GPU devices, and each bar shows performance results for a particular stencil program. Overlapped tiling is particularly effective for 1-D and 2-D stencils, as shown in the Jacobi 1-D, Jacobi 2-D, Poisson, Electromagnetics, TV Update 2D, and Gradient stencils. For 3-D stencils, the computational overhead of overlapped tiling often offsets the savings in global memory access. The same is also true in the 2-D Rician Denoising stencil, where the ratio of computation to memory access is already high.

Peak floating-point performance on a GPU device can only be achieved by issuing a multiply-and-add instruction in each clock cycle. Due to the ratio of floating-point addition to multiplication being quite high in all of the tested stencil programs (often on the order of 5–10), it is expected that the actual performance is significantly lower than device peak. Further, most stencil operations involve multiplying a number by the result of a chain of additions, which does not map well to multiply-and-add instructions (instead, we would need an add-and-multiply instruction). As an example, consider the Gradient stencil program. For each point, there is one reciprocal square-root operation and 17 additions. If we consider only the additions, the achievable peak single-precision floating-point performance of the GTX 580 drops to 790 GFlop/s. Taking this into account, we achieve about 44.3% of the achievable peak on the GTX 580 for the Gradient stencil.

To evaluate our performance results, we compare against the results published in other stencil literature. Datta [5] reports 15.8 GFlop/s and Tang et al. [19] report 19.9 GFlop/s for a 7-point Jacobi 3D stencil on an Intel Nehalem using double-precision. On the GTX 580, we achieve 50 GFlop/s for a 7-point Jacobi 3D stencil in single-precision mode, and 28.7 GFlop/s in double-precision mode. Tang et al. also report a maximum of 5.3 GPoints/s for a 2-D heat equation stencil on an Intel Nehalem, which translates into 31.8 GFlop/s. Our equivalent Jacobi 2-D stencil achieves 49.5 GFlop/s in double-precision mode on the GTX 580. Meng et al. [11] report approximately 2×10^6 cycles per iteration on a GTX 280 for a Poisson stencil that has been manually tiled using overlapped tiling with a time tile size of 3. With a clock speed of 1.3 GHz, this gives approximately 70.2

GFlop/s. In comparison, the Poisson stencil generated by our framework achieves 81.7 GFlop/s on the GTX 280.

In a different work, Datta et al. [6] show 36 double-precision GFlop/s for a 7-point Jacobi 3D stencil on a GTX 280 after aggressive auto- tuning. On the GTX 280, we achieve 20 GFlop/s. We note that the results presented here are shown for automatically generated code that implements overlapped tiling. We do not yet perform some of the manual optimizations performed by Datta et al. in their auto-tuning work. Additionally, 3D stencils are not handled well by overlapped tiling on GPU architectures due to the amount of redundancy computation that is needed for even small time tile sizes. In such cases, the optimizations proposed by Datta el al. are more beneficial than overlapped tiling on GPUs.

5.2 Impact of Tile Size Selection

The proper selection of block size, time tile size, and elements per thread is essential for the performance of the generated code. Figure 7 shows the performance of the Jacobi 2-D stencil for a fixed block size and varying time tile size and elements per thread. The performance on four different architectures is shown: an nVidia GTX 580 (Fermi), Tesla C2050 (Fermi), GTX 280 (GT200), and an AMD A8-3850 APU (Radeon HD 6550D). The block sizes were chosen such that the stencil achieves optimal performance on the architecture for some time tile size and elements per thread.

The trend in each case is that time tiling, through overlapped tiling, improves the performance of the stencil program for increasing time tile sizes up to a point. After this point, the overhead of overlapped tiling offsets the benefits of the time tiling. Therefore, there is an optimal time tile size. The choice for the number of elements per thread to process has an effect on this optimal time tile size, as shown in the graphs. For the GTX 580, this point is at $T = 6$ for 10 and 12 elements per thread, but only $T = 5$ for 6 and 8 elements per thread. For the A8-3850 APU, the optimal time tile size is around 5 for each case.

Insights into this optimal time tile size can be gained by looking at the actual computation being performed on the device for each time tile size. Figure 8 shows normalized GPU performance counter data for a time tile size of 1 to 10 for a fixed elements per thread value of 10 (GTX 280) and 12 (GTX 580). We see that as we increase the time tile size, the total number of global loads generally decreases up to a point and then saturates. For larger time tile sizes, more work is being cached in shared memory resulting in the decrease in global loads. There is a corresponding increase in the number of shared memory loads/stores and total instructions executed as we increase the time tile size. Again, as we increase the time tile size, we are doing more work on data in shared memory, but the percentage of redundant computation also increases, leading to the increase in total instructions executed.

From the GPU time counter, we see that minimum total time spent on the computation is at a time tile size of 6. As we increase this size, we see that the total number of instructions executed continues to increase, but the total number of global loads remains relatively constant. Hence, we begin to increase the overall execution time. It is important to note, however, that the floating-point throughput of the device does continue to increase as we continue to increase the time tile size and hence perform a larger portion of work in shared memory. However, larger time tile sizes also mean that smaller percentages of the total

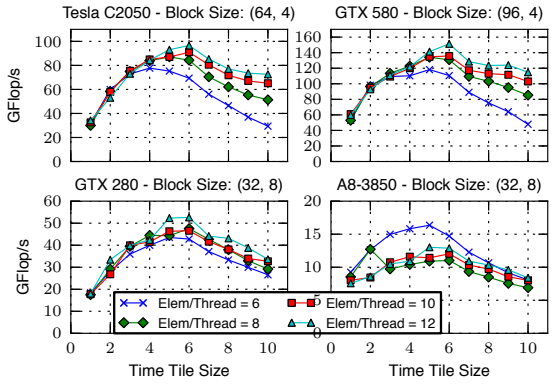


Figure 7: Effects of time tile size selection on performance for the Jacobi 2-D stencil.

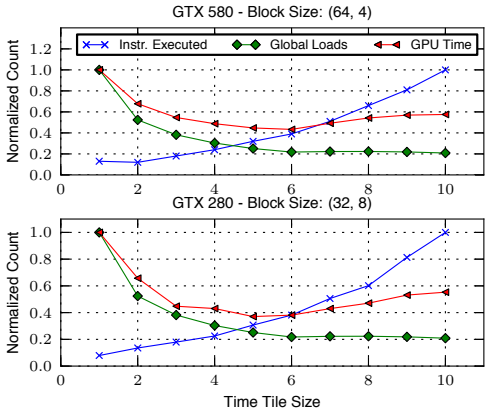


Figure 8: GPU performance counter data for varying time tile sizes

floating-point throughput is actually useful and not just redundant computation. Therefore, after a time tile size of 6, the cost of the redundant computation starts to exceed the savings in global memory transfer. The limiting factor for performance for smaller time tile sizes is thus global memory bandwidth. This changes to floating-point instruction throughput for larger time tile sizes.

Note that even when time tiling through overlapped tiling is not profitable, our code generation algorithm still produces high-performance code. Our framework can still utilize spatial tiling of the stencil computation in order to achieve levels of performance that surpass that of multi-core CPUs.

6. RELATED WORK

A number of recent studies have focused on optimizing stencil computations for multicore CPUs and GPUs [2, 6, 8, 12, 17, 19, 20]. Strzodka et al. [17] use time skewing and cache-size oblivious parallelograms to improve the memory system pressure and parallelism in stencils on CPUs. PATUS [2] is a stencil compiler proposed by Christen et al. that uses both a stencil description and a machine mapping de-

scription to generate efficient CPU and GPU code for stencil programs. Han et al. [8] propose an extension to OpenMP to allow for pattern-based optimization of stencil programs on CPUs and GPUs. Micikevicius et al. [12] hand-tuned a 3-D finite difference computation stencil and achieved an order of magnitude performance increase over existing CPU implementations on GT200-based Tesla GPUs. Datta et al. [6] developed an optimization and auto-tuning framework for stencil computations, targeting multi-core systems, NVidia GPUs, and Cell SPUs. They proposed autotuning as essential in order to achieve performance levels on GPUs where the benefits outweigh the cost of sending data across the PCIe bus. But neither of these studies considered time-tiled implementations on GPUs.

The work of Tang et al. [19] is perhaps the most closely-related to ours. They propose the Pochoir stencil compiler which uses a DSL embedded in C++ to produce high-performance code for stencil computations using cache-oblivious parallelograms for parallelism. They target x86 using the Intel C++ Compiler and the Intel Cilk Plus library. They show good performance on x86 targets, but do not address issues specific to GPU code generation for stencil computations.

Closely related to our work is that of Meng et al. [11]. They propose a performance model for the evaluation of ghost zones for stencil computations on GPU architectures. They consider the effects of tile size selection on the performance of the final code, and propose an approach based on user provided annotations for GPU code generation, but do not consider fully automated code generation.

Overlapped tiling, the technique used in our automatic code generation framework, was used by Krishnamoorthy et al. [10] for enhancing tile-level concurrency for multi-core systems. Nguyen et al. [13] proposed a data blocking scheme that optimizes both the memory bandwidth and computation resources on GPU devices. Peng et al. [7] investigate the selection of tile sizes for GPU kernels, with an emphasis on stencil computations. However, none of these works consider fully automatic, high-performance code generation for stencil computations on GPUs.

7. CONCLUSION

In this paper, we have introduced an automatic code generation scheme for stencil computations on GPU architectures. This scheme uses overlapped tiling to provide efficient time tiling on GPU architectures, which are massively threaded but are susceptible to performance degradation due to branch divergence and a lack of memory coalescing. We have shown that our scheme produces high performance code on a variety of GPU devices for many stencil programs. We further performed an analysis of the resulting code for various time tile sizes to identify the limiting factors, showing that global memory access is the limiting factor for smaller time tile sizes, and computational overhead is the limiting factor for larger time tile sizes.

Acknowledgments

We thank the reviewers for their valuable comments. This work was supported in part by the U.S. National Science Foundation through award 0926688, and by the Center for Domain-Specific Computing (CDSC) funded by NSF "Expeditions in Computing" award 0926127.

8. REFERENCES

[1] CDSC, the Center for Domain-Specific Computing. http://www.cdsc.ucla.edu.

[2] M. Christen, O. Schenk, and H. Burkhart. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. IPDPS '11, pages 676–687, 2011.

[3] J. Cong, M. Huang, and Y. Zou. Accelerating fluid registration algorithm on multi-FPGA platforms. FPL, 2011.

[4] J. Cong and Y. Zou. Lithographic aerial image simulation with FPGA-based hardwareacceleration. FPGA, pages 67–76, 2008.

[5] K. Datta. *Auto-tuning Stencil Codes for Cache-Based Multicore Platforms*. PhD thesis, University of California, Berkeley, 2009.

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

[7] P. Di and J. Xue. Model-driven tile size selection for doacross loops on gpus. In *Euro-Par*, volume 6853, pages 401–412. 2011.

[8] D. Han, S. Xu, L. Chen, and L. Huang. PADS: A pattern-driven stencil compiler-based tool for reuse of optimizations on GPGPUs. In *ICPADS '11*, pages 308 –315, dec. 2011.

[9] Khronos OpenCL Working Group. The OpenCL specification - version 1.2. November 2011.

[10] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. PLDI '07, pages 235–244, 2007.

[11] J. Meng and K. Skadron. A performance study for iterative stencil loops on GPUs with ghost zone optimizations. *IJPP*, 39:115–142, 2011.

[12] P. Micikevicius. 3d finite difference computation on GPUs using CUDA. GPGPU-2, pages 79–84, 2009.

[13] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-d blocking optimization for stencil computations on modern CPUs and GPUs. SC '10, pages 1–13, 2010.

[14] NVIDIA Corporation. CUDA C programming guide - version 4.0. May 2011.

[15] NVIDIA Corporation. OpenCL programming guide for the CUDA architecture. February 2011.

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

[17] R. Strzodka, M. Shaheen, D. Pajak, and H.-P. Seidel. Cache oblivious parallelograms in iterative stencil computations. ICS '10, pages 49–59, 2010.

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

[19] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The Pochoir stencil compiler. SPAA '11, pages 117–128, 2011.

[20] J. Treibig, G. Wellein, and G. Hager. Efficient multicore-aware parallelization strategies for iterative stencil computations. *ArXiv e-prints*, Apr. 2010.