Simple, Accurate, Analytical Time Modeling and Optimal Tile Size Selection for GPGPU Stencils

Nirmal Prajapati
Colorado State University
Fort Collins, CO, USA
nirmal171@gmail.com

Waruna Ranasinghe
Colorado State University
Fort Collins, CO, USA
warunapww@gmail.com

Sanjay Rajopadhye
Colorado State University
Fort Collins, CO, USA
svr@cs.colostate.edu

Rumen Andonov
IRISA/INRIA
Rennes, France
Rumen.Andonov@irisa.fr

Hristo Djidjev
Los Alamos National Lab.
Los Alamos, NM, USA
djidjev@lanl.gov

Tobias Grosser
Dep. of Computer Science
ETH Zurich, Switzerland
tobias.grosser@inf.ethz.ch

ABSTRACT

Stencil computations are an important class of compute and data intensive programs that occur widely in scientific and engineering applications. A number of tools use sophisticated tiling, parallelization, and memory mapping strategies, and generate code that relies on vendor-supplied compilers. This code has a number of parameters, such as tile sizes, that are then tuned via empirical exploration.

We develop a model that guides such a choice. Our model is a simple set of analytical functions that predict the execution time of the generated code. It is deliberately optimistic, since we are targeting modeling and parameter selections yielding highly tuned codes.

We experimentally validate the model on a number of 2D and 3D stencil codes, and show that the root mean square error in the execution time is less than 10% for the subset of the codes that achieve performance within 20% of the best. Furthermore, based on using our model, we are able to predict tile sizes that achieve a further improvement of 9% on average.

1. INTRODUCTION

As we move to address the challenges of exascale computing, one approach that has shown promise is *domain specificity:* the adaptation of application, compilation, parallelization, and optimization strategies to narrower classes of domains. An important representative of such a domain is called *Stencil Computations*, and includes a class of typically compute bound parts of many applications such as partial differential equation (PDE) solvers, numerical simulations in domains like oceanography, aerospace, climate and weather modeling, computational physics, materials modeling, simulations of fluids, and signal and image-processing algorithms. One of the thirteen Berkeley dwarfs/motifs [2], is "structured mesh computations," which are nothing but stencils. Many *dynamic programming* algorithms also exhibit a similar dependence pattern. The importance of stencils has been noted by a number of researchers, indicated by the recent surge of research projects and publications on this topic, ranging from optimization methods for implementing such computations on a range of target architectures, to Domain Specific Languages (DSLs) and compilation systems for stencils [11, 12, 13, 31, 30, 32, 34, 38, 42, 51, 50, 53, 48]. Workshops and conferences devoted exclusively to stencil acceleration have recently emerged. Stencils belong to a class of programs called *uniform dependence computations*, which are themselves a proper subset of "affine loop programs." Such programs can be analyzed and parallelized using a powerful methodology called the polyhedral model [46, 44, 35, 15, 16, 17, 10, 6].

A second aspect of domain specificity is reflected in the emergence of specialized architectures, called *accelerators*, for executing compute intensive parts of many computations. They include GPGPU, general purpose computing on graphics processing units (GPUs), and other co-processors (Intel Xeon Phi, Knight's Landing, etc.). Initially they were "special purpose," limited to highly optimized image rendering libraries (aka. graphics processing). Later, users realized that they could be used for other computations. Eventually, the emergence of tools like CUDA and OpenCL enabled general purpose parallel programming on GPUs.

Exploiting the specificity of the applications and the specificity of target architectures leads to domain-specific tools to map high level programs to highly tuned and optimized implementations on the target architecture. Many such tools, both academic research prototypes and productions systems, are widely available. One example is PPCG, developed by the group at ENS, Paris [54]. PPCG includes a module that targets GPUs and incorporates a sophisticated, compiler developed by Grosser et al. [22]. We call it the HHC compiler because it employs a state-of-the-art tiling strategy called *hybrid hexagonal classic tiling.*

A number of parameters can be specified as inputs/flags to the HHC compiler, e.g., the tile sizes, and the number of threads in each dimension. These parameters have a tremendous influence on the performance of the code. An important element of such tools is a step called *auto tuning*: empirical evaluation of the actual performance of a, hopefully small, set of code instances for a range of mapping param-

PPoPP '17, February 04-08, 2017, Austin, TX, USA

 ISBN 978-1-4503-4493-7/17/02...$15.00

DOI: http://dx.doi.org/10.1145/3018743.3018744

eters. This enables the system to choose these parameters optimally for actual "production runs" on real data/inputs.

Modern architectures are extremely complicated, with sophisticated hardware features that interact in unpredictable ways, especially since the latency of operations is stochastic due to the deep memory hierarchy. It is widely believed that because of this, auto tuning is unavoidable in order to obtain good performance. In this paper, we make the case that domain specificity enables us to develop *good* analytical models to predict the performance of specific codes on specific target architectures. This allows a significant reduction in the autotuning search space. Our contributions are:

- **Contribution 1.** We develop a simple analytical model to predict the execution time of a tiled stencil program and apply it to codes generated by the HHC compiler. It is deliberately *optimistic*, ignores the effect of some parameters, and is an analytic function of
  - program, machine, and compiler parameters that are easily available statically, and
  - one stencil-specific parameter, obtained by running a *micro-benchmark* derived from the loop body.
- **Contribution 2.** Although our model may not accurately predict the performance for all tile size combinations, it is very accurate for the ones that matter, i.e., those that give top performance. To show this, we generated more than 60,000 programs for
  - two modern **target platforms** (NVIDIA GTX 980 and Titan X),
  - four 2D and two 3D **stencil codes** (Jacobi, Heat, Laplacian, and Gradient)
  - over a range of ten input data **sizes**, and
  - for each such *platform-stencil-size* combination, a wide range of tile sizes and thread counts (the HHC compiler inputs).

  As we expected, the RMS error over the entire data set was—"seemingly disappointingly"—over 100%. However, our model is very accurate where it matters. When we restricted ourselves to codes whose performance is within 20% of the best, the RMSE is less than 10%.[1]
- **Contribution 3.** To test the predictive abilities of the model, we evaluated the model over the entire feasible space (for each platform-stencil-size combination) and obtained the tile sizes that were within 10% of the best predicted execution time. There were less than 200 such points. We called the HHC compiler with these tile sizes and were able to observe among this set a performance improvement of 9% on average (max improvement was 17%). We also observed that the "conventional wisdom" to choosing tile sizes so as to maximize shared-memory footprint is not always optimal.

[1]The restriction to the better performing subset was exactly our motivation. We designed the model to help predict/explore data points that would give *good* performance. It is also why we made optimistic assumptions in developing the model.

The remainder of this paper is organized as follows. After a discussion of related work and backend (Sections 2 and 3) we describe the domain specific parallelization used for stencils and, in particular, the strategies used by the HHC compiler. Then, Section 4 develops our analytical model. Section 5 describes our experimental results on validating the model on a baseline set of tile sizes. Section 6 illustrates the predictive power of the model. Finally, we discuss our results, describe ongoing and future work, and conclude in Sections 7 and 8.

2. RELATED WORK

At the algorithmic level, most stencil applications are *compute bound* in the sense that the ratio of the total number of *operations* to the total number of *memory locations* touched can always be made "sufficiently large" because it is an asymptotically increasing value. We may expect that such codes can be optimized to achieve very high performance relative to machine peak. However, naive implementations turn out to be memory-bound. Therefore, many authors seek to exploit data locality for these programs [29, 43, 5]. One successful technique is called *time tiling* [56, 6, 58, 59, 19, 20, 52, 5], an advanced form of loop tiling [57, 56, 60]. Time tiling first partitions the whole computation space into tiles extending in all dimensions, and then optionally executes these tiles in a so called "45 degree wavefront" fashion. We assume, like most of the work in the literature, that *dense* stencil programs are *compute bound* after time tiling. However, due to the intricate structure of time tiled code, writing it by hand is challenging. Automatic code generation, is an attractive solution, and has been an active research topic.

There has been much work on time modeling and performance optimization. For stencil graphs, which are directed acyclic graphs (DAGs) of non-iterated stencil kernels, various DSLs compilers have been proposed. Halide [45] and Stella [24] are two DSLs from the context of image processing and weather modeling that separate the specification of the stencil computation from the execution schedule, which allows for the specification of platform specific execution strategies derived either by platform experts or automatic tuning. Both DSLs support various hardware targets, including CPUs and GPUs. Polymage [39] also provides a stencil graph DSL—this time for CPUs only—but pairs it with an analytical performance model for the automatic computation of optimal tile size and fusion choices. MODESTO [23] proposes an analytical performance model in the context of Stella, for multiple cache levels and fusion strategies, for both GPUs and CPUs.

For iterative stencils a large set of optimizing code generation strategies have been proposed. Ahmed et. al [1] describe time tiling as part of their work on synthesizing transformations for imperfectly nested loops. Li and Song [33] consider fusion and skewing in a unified framework and derive minimal skewing factors for exploiting data reuse along the time dimension of an iterative stencil. Both works do not consider GPU performance. Patus [9] provides an autotuning environment for stencil computations which can target CPU and GPU hardware. It does not use software managed memories and also does not consider any time tiling strategies. Pochoir [53] is a CPU-only code generator for stencil computations that exploits reuse along the time dimension by recursively dividing the computation in trapezoids. Diamond tiling [3], Hybrid-hexagonal tiling [22], and Overtile [26] are all tiling strategies that allow to exploit reuse along the time dimension, while ensuring a balanced amount of coarse-grained parallelism throughout the computation. While the former has only been evaluated on CPU systems, the last two tiling schemes have been implemented to target GPUs. Overtile uses redundant computation whereas hybrid-hexagonal tiling uses hexagonal tiles to avoid redundant computation and the increased shared memory that would otherwise be required to store temporary values. Another time tiling strategy has been proposed with 3.5D blocking by Nguyen et. al [41], who manually implemented kernels that use two dimensional space tiling plus streaming along one space dimension with tiling along the time dimension to target both CPUs and GPUs. A slightly orthogonal stencil optimization has been proposed by Henretty et. al [25], who use data-layout transformations to avoid redundant non-aligned vector loads on CPU platforms. All of the previously discussed frameworks either come with their own auto-tuning framework or require auto tuning to derive optimal tile sizes.

For stencil GPU code generation strategies that use redundant computations in combination with ghost zones an analytical performance model has been proposed [37] that allows to automatically derive "optimal" code generation parameters. Yotov et. al [61] showed already more than ten years ago that an analytical performance model for matrix multiplication kernels allows to generate code that is performance-wise competitive to empirically tuned code generated by ATLAS [55], but at this point no stencil computations have been considered. Shirako et al. [49] use cache models to derive lower and upper bounds on cache traffic, which they use to bound the search space of empirical tile-size tuning. Their work does not consider any GPU specific properties, such as shared memory sizes and their impact on the available parallelism.

In contrast to tools for tuning, Hong and Kim [27] present a precise GPU performance model which shares many of the GPU parameters we use. It is highly accurate, but low level, and requires analyzing the PTX assembly code. It is therefore unsuitable for use in a compiler.

3. STENCILS AND THEIR PARALLELIZATION

We now describe the class of computations we tackle, the overall parallelization strategy, and how the HHC compiler implements it.

In codes that implement *dense iterative stencils*, values of array elements are updated iteratively at every time step using the values of some of their neighbors from previous time steps[2] according to a fixed pattern. We consider stencil codes of the following kind. Let $\mathcal{S} = \{(i_1, \ldots, i_k) \mid 1 \leq i_j \leq S_j, \text{ for } j = 1, \ldots, k\}$ be a k-dimensional *space index set* and $\mathcal{T} = \{1, \ldots, T\}$ be a *time index set*. Then, given a set $\mathcal{N}$ defining the "neighborhood" of any point in terms of a pattern of relative coordinates and a coefficient w_a associated with each element $a \in \mathcal{N}$, a (convolutional) stencil code defines an iterative evaluation of the following weighted sum

$$A^t(s) = \Big(\sum_{a \in \mathcal{N}} w_a * A^{t-1}(s+a)\Big) + c\,, \tag{1}$$

$s \in \mathcal{S}$, $t \in \mathcal{T}$, where we assume that appropriate values are given for the "initial value" (when $t = 0$) and the "boundary values" (when the points $s + a$ fall outside $\mathcal{S}$). Stencils are usually implemented as nested loops with the loop body evaluating the rhs of (1) and storing it in a data array.

Efficient parallelization of stencils on GPUs requires careful consideration of at least two factors at two separate levels, and there is significant interplay between them: parallelism and data locality/reuse, at the fine grain (threads, synchronization, shared and/or scratchpad memory) and at the coarse grain (thread blocks, and global memory). Tiling is a widely used technique that has been developed to manage this, and it is applied at multiple levels and to both data and iterations of the loop. In a typical implementation, data, initially stored on the CPU, is first transferred to the GPU, subsequently a sequence of *kernel calls* is issued on the CPU to perform the computation on the GPU-resident, and finally the result is moved back to the CPU. For ease of explanation, it is convenient to view the entire stencil computation as defined by its *iteration space*: the set of legal values of the space and time coordinates.

3.1 Hybrid Hexagonal/Classical Tiling

The HHC compiler [22], which we are using, implements a hybrid of two strategies: *hexagonal* tiling of the outer two loops/dimensions, and the *classic* time skewing of the remaining (inner/space) dimensions. A 1D stencil is thus, a special case (the iteration space is 2D, and only hexagonal tiling is applicable). For more than two nested loops, it tiles the inner loops using the classic time skewing approach.

Therefore, we first explain hexagonal tiling. The iteration space, a $S \times T$ rectangle, is partitioned into a set of "staggered" hexagons, as shown in Figure 1.a, and we view a "row of hexagons" as those whose (leftmost) corners have the same value of t. Each row is independent, accesses distinct data, and can, hence, be executed in a single kernel call. The main program is just a sequence of such kernel calls.

Now consider a 3-D iteration space (see Figure 2) where each hexagon on the outer two (t-k) dimensions now becomes a "prism" extending along the i dimension. Stencil dependences precludes directly blocking this prism. Rather, *time-skewing* has to be applied (illustrated by the oblique hexagonal faces above). After this, the HHC compiler generates code that executes the tiles in the prism via a sequential loop—executed within a single kernel call—whose body is the execution of a tile, the outer loop structure remains the same. The idea is extended to higher dimensions, where the prisms become "slabs" and "hyper-slabs" and the sequential loop iterating over tiles becomes a nested loop.

3.2 Details of the HHC Compiler

HHC generates highly tuned code for specific stencils, problem sizes, and tile size parameters, taking advantage of properties of the "hybrid-hexagonal schedule". The HHC compiler is one module within a complete polyhedral tool suite, PPCG, developed by the group at ENS, Paris [54]. Independently of the tiling scheme, PPCG automatically simplifies generated loop bounds and index expressions, taking into account problem sizes and tile-size parameters.

[2]Stencils where some updates may use values from the current time step (called Gauss-Seidel stencils) are not included in the definition, as the HHC compiler does not consider these.

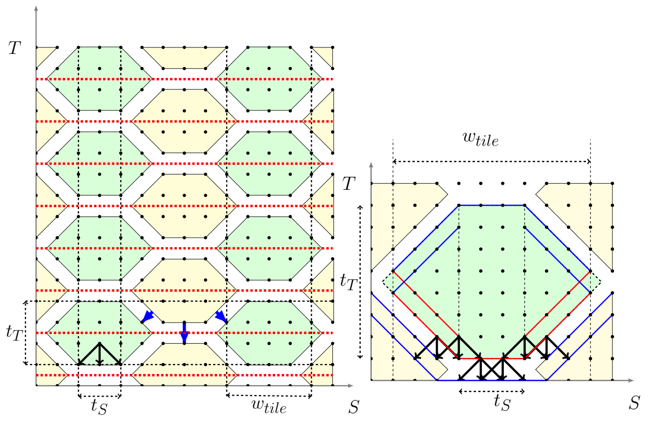


Figure 1: Hexagonal tiling for 1D stencils: [a] [b] the $S \times T$ iteration space is partitioned into hexagons (left). The Jacobi 1D inter and intra-tile dependencies are illustrated as blue and black arrows, respectively. There are two kinds of "tile rows" colored green and yellow. The tiles in each row are independent, and can executed in a single GPU kernel call. A single tile (right) and its I/O (red: iterations reading data from global memory; blue: iterations writing to global memory).

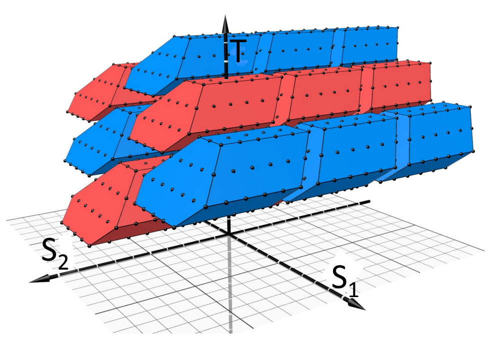


Figure 2: Hybrid-hexagonal tiling for 2D stencils.

When generating HHC tiled code, the effectiveness of PPCG's specialization can be largely improved by unrolling the global-to-shared memory copy code as well as the per-tile compute code. When unrolling both, all control flow within a tile is eliminated such that only a sequence of (possibly predicated) instructions remains to be executed by each thread. When using HHC tiling, it is also possible to take advantage of data-reuse between two tiles that are run in sequence by the same threadblock. In this situation, a subset of the data that is loaded by each tile is already in shared memory and does not need to be loaded again. However, as PPCG derives for each tile an optimal data-mapping strategy, the reusable data is not always placed such that it can be immediately reused between tiles. To still exploit this property, PPCG provides two different options. Option 1) enforces a shared (commonly less optimal) memory placement strategy. Option 2) moves reusable data within shared memory to account for different data placement between tiles before loading the remaining data from global memory.

4. EXECUTION TIME MODEL

Name	Type	Description
$\mathtt{S_i}$	EP	i-th space dimension
$\mathtt{T}$	EP	time dimension
t_{S_i}	ES	tile size along the i-th space dimension
t_T	ES	tile size along time dimension
$n_{\mathrm{thr,i}}$	ES	number of threads per threadblock in the i-th dimension/loop
n_{SM}	EH	number of SMs in the device
n_{V}	EH	number of vector units per SM
R_{SM}	EH	number of registers per SM
M_{SM}	EH	size of shared memory per SM
MTB_{SM}	EH	max threadblocks per SM
L	EH	time per word of global memory access
τ_{sync}	EH	time for a single synchronization
T_{sync}	EH	time for a host-GPU synchronization
N_w	CS	number of wavefronts
m_i	CS	input memory footprint of a tile (amount of data read from global memory)
m_o	CS	output memory footprint of a tile (amount of data written to global memory)
m'	CS	time for global↔shared data transfer for a tile
c	CS	time to perform the computation in a tile (collectively by all the threads)
k	CS	"hyper-threading" factor (threadblocks per SM)
$T_{\mathrm{tile}}(k)$	CS	time to compute a tile (accounting for k-way "hyper-threading")
$w(i)$	CS	width of the i-th wavefront (number of threadblocks in the i-th kernel call)
w_{tile}	CS	width of (number of iterations in) a tile
R_{tile}	CS	number of registers needed per tile
M_{tile}	CS	shared memory needed per tile
M_{io}	CS	I/O volume per tile (global↔shared)
C_{iter}	CSH	(optimized) execution time of one iteration
T_{alg}	C	total execution time of stencil
T_{exec}		observed execution time (not a model parameter)

Table 1: The execution time model parameters. E/C denote Elementary/Composite, and S/H/P denote Software/Hardware/Problem; M_{io} is measured in 4-byte words.

We now develop a model for the execution time of a stencil computation on the GPU as a function of software, hardware, and problem parameters. These parameters are shown in Table 1. Some of these have to be measured or are chosen by the compiler, we call them *elementary*, while others, called *composite*, are functions of elementary and other composite parameters. In addition to this distinction, we also divide them into three classes, hardware, software, and problem, depending on their origin. Hardware parameters are specific to the machine. Software parameters such as tile size, number of threads per tile, etc., are determined by the user or the compiler. And the problem parameters are determined by the type of stencil, the computation in the loop body, number of variables, nature of memory accesses, etc.

4.1 Model for Hexagonal Tiling

We first derive the execution time for an 1D stencil, Jacobi 1D (for which we drop the subscript on S_1, using just S). Later we extend the model to higher dimensions.

The $T \times S$ rectangular iteration space is tiled into hexagons with a base of size t_S and a height of size t_T (see Figure 1.a). In the following, we suppose that t_T is even, since the HHT compiler only supports this case.

The total execution time for the tiled code can be evaluated by adding the time spent by the GPU in each kernel call and the total synchronization time spent between kernel calls

$$T_{\text{alg}} = \sum_{i=1}^{N_w} \left(\left\lceil \frac{1}{n_{SM}} \left\lceil \frac{w(i)}{k} \right\rceil \right\rceil T_{\text{tile}}(k) + T_{\text{sync}} \right). \qquad (2)$$

The i^{th} kernel computes the tiles from the i^{th} *wavefront*. The i^{th} wavefront consists of the tiles that intersect the i^{th} red dashed horizontal line. Note that they contain either yellow tiles only (for odd wavefronts indexed from one) or green tiles only (for even wavefront indices). Let us estimate the number n_0 of even-indexed wavefronts (colored in green). Since the height of each tile is t_T and the height of the iteration space is T, then $n_0 = \lceil T/t_T \rceil$. For estimating the total number of wavefronts, we note that for any even-indexed wavefront (say j) we can associate exactly one, possibly partial, odd-indexed wavefront—the one whose index is $j-1$. Moreover, depending on the relative values of T and t_T, the last wavefront may be even (green) and numbered $2n_0$, or odd (yellow) and numbered $2n_0+1$. Which case holds depends on the relationship between $T - \lfloor T/t_T \rfloor t_T$ and $t_T/2$. More precisely, the total number of wavefronts is equal to

$$N_w = 2\left\lceil \frac{T}{t_T} \right\rceil + \epsilon \approx 2\left\lceil \frac{T}{t_T} \right\rceil, \qquad (3)$$

where $\epsilon = 0$ if $0 < T - \lfloor T/t_T \rfloor t_T \leq t_T/2$, otherwise $\epsilon = 1$.

To estimate the tile width, w_{tile}, we decompose each hexagon into a rectangle of size $t_S \times t_T$ and two right isosceles triangles with a hypotenuse t_T, each of which spans $t_T/2$ columns (see Figure 1.b). Adding these, we get

$$w_{\text{tile}} = t_S + t_T - 2. \qquad (4)$$

Furthermore, the distance between two consecutive tiles in a wavefront, called *pitch*, can be derived as $w_{\text{tile}} + t_S + 2 = 2t_S + t_T$. Now, there may be one more or less tile in alternate wavefronts, so the number of tiles in a wavefront (i.e., the width of a wavefront) is

$$w(i) = \left\lceil \frac{S}{2t_S + t_T} \right\rceil + \epsilon' \approx \left\lceil \frac{S}{2t_S + t_T} \right\rceil, \qquad (5)$$

where ϵ' is 1 or 0, and is ignored. Since $w(i)$ is actually independent of i, the summation in (2) can be simplified to yield

$$T_{alg} = N_w T_{\text{tile}}(k) \left\lceil \frac{1}{n_{SM}} \left\lceil \frac{w}{k} \right\rceil \right\rceil + N_w\, T_{\text{sync}}. \qquad (6)$$

4.1.1 Execution Time of a Tile

In the case of hexagonal tiling, the amount of data read from global memory is the sum of the bottom base (t_S) plus the data needed to compute its two adjacent oblique sides. Each of these oblique sides has $t_T/2$ points, and there are *two* such lines of points that need data from *two* oblique lines (shown in red in Figure 1.b). Collectively, this data comes from *two* other (blue) line segments in a neighboring south-east or south-west tile. The blue points thus depict the input footprint of the tile, m_i, which can be shown to be $t_S + 4t_T/2 = t_S + 2t_T$. In the case of Jacobi 1D, this amount also equals the output tile memory footprint m_o. The later is depicted in blue (north-oriented tile's facets) in Figure 1.b. Therefore, for the total input/output tile memory footprint we obtain

$$m_{io} = m_i + m_o = 2(t_S + 2t_T). \qquad (7)$$

To obtain m' we multiply m_{io} by L and add twice the synchronization time. Hence

$$m' = m_{io}L + 2\tau_{sync} = 2(t_S + 2t_T)L + 2\tau_{sync}. \qquad (8)$$

Finally, for these hexagonal tiles, $M_{\text{tile}} = 2(w_{\text{tile}} + 2) = 2(t_S + t_T)$. To determine T_{tile} we consider two cases: a single tile per SM (no hyperthreading) and multiple tiles per SM (hyperthreading).

4.1.2 No Hyperthreading

In this case $k = 1$ and only one tile is executed on each SM at a time. To compute a tile, a read operation, a compute operation, and a write operation are performed in sequence with synchronizations in between them. We assume that both read and write operations take an equal amount of time.

The iteration space dependences indicate that the computations in a tile can be done in parallel in each row, and in a sequential manner between rows from bottom to top. Since C_{iter} denotes the computation time per iteration and, considering the shape of each hexagon, we find that the computation time of a tile is given by

$$\begin{aligned} c &= 2 \sum_{x=t_S, step=2}^{w_{\text{tile}}} \left(\left\lceil \frac{x}{n_V} \right\rceil C_{\text{iter}} + \tau_{sync} \right) \\ &= 2C_{\text{iter}} \sum_{x=t_S, step=2}^{w_{\text{tile}}} \left\lceil \frac{x}{n_V} \right\rceil + t_T \tau_{sync}. \end{aligned} \qquad (9)$$

Combining (8) and (9), the total time to process a tile is

$$T_{\text{tile}} = m' + c. \qquad (10)$$

When $n_V \geq w_{\text{tile}}$, each tile row can be computed in C_{iter} time and the computation time c of a tile is just $c = t_T(C_{\text{iter}} + \tau_{sync})$. However, note that this is a very inefficient use of the fine grain resources of the SMs, and we expect that for the optimal solution, $n_V \ll w_{\text{tile}}$.

4.1.3 Hyperthreading

Consider the case k > 1, where more than one tile is executed on each SM at a time. The value of k, the number of tiles per SM, depends on the available resources, shared memory and registers in a SM as well as the resources consumed by a tile (thread block), and is bounded as follows:

$$1 < k \leq \min\left(\left\lfloor \frac{R_{SM}}{R_{\text{tile}}} \right\rfloor, \left\lfloor \frac{M_{SM}}{M_{\text{tile}}} \right\rfloor \right). \qquad (11)$$

Read/write operations can now overlap with computations. Therefore, reading of the second tile input data can be synchronized and overlapped with the first tile computation. However, the very first read and the very last write cannot overlap with anything. Thus the execution time is the sum of this and the dominant one between $(k-1)$ read-writes and computes. The time to compute k tiles is then

$$T_{\text{tile}}(k) = m' + c + (k-1)\max(m', c). \qquad (12)$$

4.2 Hybrid Hexagonal/Classic Tiling for 2D Stencils

Here, the outer two loops are tiled with hexagons, and the inner dimension(s) are tiled using classic time skewing techniques. We illustrate this for the Jacobi2D stencil.

4.2.1 Total Execution Time of Jacobi 2D

As illustrated in Figure 2, each hexagon from Figure 1 now becomes a "prism" with a hexagonal cross section, whose length is S_2 along the S_2-axis. Its data footprint may be too large for the entire prism to be executed as a single tile, so it needs to be tiled. To respect the dependences, time skewing is applied to each prism (notice how the front face is oblique in the S_1-T plane), and then each prism is partitioned using vertical cuts (other than the front face, all the inter-tile faces are vertical). A threadblock executes the entire *sequence of tiles* in a single kernel call. Let T_{prism} denote the time that this takes, and postpone its derivation for now.

The formulæ for the number of wavefronts (N_w), the tile width (w_{tile}), the width of a wavefront in respect to the S_1-axis (w), as well as the total execution time (T_{alg}), are identical to the Jacobi 1D case, and are given by equations 3, 4, 5 and 6 respectively, where the parameters S, t_S are replaced by S_1 and t_{S_1} while the term T_{tile} is substituted by T_{prism}.

4.2.2 Execution Time of a Tile

Since tiles chosen as above could be very large and inconvenient for the shared memory size, we need to further partition them into smaller chunks. In the Jacobi 2D hybrid approach these are hexagonal (non-orthogonal) sub-prisms with a length t_{S_2} and bases defined by the normal vector $(1, 0, 1)$ where time is the first dimension (vertical axis in Figure 2). The number of these sub-prisms in an entire prism is $\left\lceil \frac{S_2+t_T}{t_{S_2}} \right\rceil$ and the dependencies allow to compute them sequentially from bottom to top (right to left in Figure 2). We therefore assume from now on that one tile is computed by a single SM which iterates $\left\lceil \frac{S_2+t_T}{t_{S_2}} \right\rceil$ times for a given prism, and at each iteration computes one of the above sub-prism.

Since the data belonging to the oblique hexagonal faces are allocated in the local SM memory, the amount of data to be transferred from global to shared memory is simply the amount of data as for Jacobi 1D case (7) multiplied by the tile's length t_{S_2}. Hence

$$m_i = m_o = t_{S_2}(t_{S_1} + 2t_T). \tag{13}$$

For the corresponding time we obtain respectively

$$m' = (m_i + m_o)L + 2\tau_{\text{sync}}. \tag{14}$$

The iteration space dependences indicate that the computations in a tile can be done in parallel in each row, and in a sequential manner between rows from bottom to top. We therefore find that the computation time for a **non-boundary/steady state** tile is given by

$$\begin{aligned} c &= 2 \sum_{x=t_{S_1}, step=2}^{w_{\text{tile}}} \left(\left\lceil \frac{x t_{S_2}}{n_v} \right\rceil C_{iter} + \tau_{\text{sync}} \right) \\ &= 2C_{iter} \sum_{x=t_{S_1}, step=2}^{w_{\text{tile}}} \left\lceil \frac{x t_{S_2}}{n_v} \right\rceil + t_T \tau_{\text{sync}}. \end{aligned} \tag{15}$$

Now, the execution time of an entire prism depends on whether or not hyper-threading is performed. If we have a single tile on each SM, $T_{\text{tile}}(k) = m' + c$. On the other hand, with hyper-threading enabled, $T_{\text{tile}}(k)$ is dominated by $\zeta = \max\{m', c\}$, and so,

$$T_{prism}(k) = \begin{cases} k = 1 \quad : & \left(m' + c\right) \left\lceil \frac{S_2+t_T}{t_{S_2}} \right\rceil \\ k > 1 \quad : & m' + k\zeta \left\lceil \frac{S_2+t_T}{t_{S_2}} \right\rceil. \end{cases} \tag{16}$$

Plugging this into (2) and simplifying we get

$$T_{\text{alg}} = N_w T_{\text{sync}} + N_w T_{\text{prism}} \left\lceil \frac{1}{n_{SM}} \left\lceil \frac{w}{k} \right\rceil \right\rceil. \tag{17}$$

Finally, we extend the analysis for the 1D case to determine $m_i = m_o$ and M_{tile} as

$$m_i = t_{S_2}(t_{S_1} + 2t_T) \tag{18}$$

$$M_{\text{tile}} = 2(t_{S_1} + t_T + 1)(t_{S_2} + t_T + 1). \tag{19}$$

4.3 Hybrid Hexagonal/Classic Tiling for 3D stencils

Here, the outer two loops are tiled with hexagons, and the inner dimension(s) are tiled using classic time skewing techniques. We illustrate this for the Jacobi 3D stencil.

4.3.1 Total Execution Time of Jacobi 3D

Each hexagon from Figure 1 now becomes an $S_2 \times S_3$ "slab" with a hexagonal cross section. Its data footprint is surely too large for the entire slab to be executed as a single tile, so it needs to be further tiled in the two inner dimensions. Indeed, even a single dimensional slice out of this slab, i.e., a 3-dimensional prism, will most likely have too large a data footprint. To respect the dependences, time skewing is applied to each slab (notice how the front face is oblique in the S_1-T plane (see Figure 2 that illustrates the 2D case) and then each slice is partitioned using vertical cuts (other than the front face, all the inter-tile faces are vertical). A threadblock executes the entire *sequence of tiles* in a single kernel call. Let T_{slab} denote the time that this takes, and postpone its derivation for now.

The formulæ for the number of wave-fronts (N_w), the tile width (w_{tile}), the width of a wave-front in the t_{S_1}-axis (w), are identical to the Jacobi 1D case, and respectively, are

$$N_w \approx 2 \left\lceil \frac{T}{t_T} \right\rceil \tag{20}$$

$$w_{\text{tile}} = t_{S_1} + t_T - 2, \tag{21}$$

$$w \approx \left\lceil \frac{S_{s_1}}{t_{S_1} + t_T} \right\rceil. \tag{22}$$

The total execution time is hence similar to the one of Jacobi 1D 6 with the unique difference that the term T_{tile} is substitute now by T_{slab}.

4.3.2 Execution Time of a Slab

Since slabs chosen as above could be very large and inconvenient for the shared memory size, we need to further partition them into smaller chunks. In the Jacobi 2D hybrid approach (see Figure 2) these are hexagonal (non-orthogonal) sub-slabs with a length t_{S_2} and bases defined by the normal vector $(1, 0, 1)$, where time is the first dimension (vertical axis). In case of Jacobi 3D the number of these sub-slabs in an entire slab is

$$N_{sslabs} = \left\lceil \left(\frac{S_2 + t_T}{t_{S_2}}\right)\left(\frac{S_3 + t_T}{t_{S_3}}\right)\right\rceil \tag{23}$$

and we assume from now on that one slab is computed by a single SM, which iterates N_{sslabs} times for a given slab, and at each iteration computes one of the above sub-slab.

Since data belonging to the oblique hexagonal faces are allocated in the local SM memory, the amount of data to be transferred from global to shared memory is simply the amount of data as for Jacobi 1D case (7) multiplied by the tile's length in the S_2 and S_3 axes. Hence

$$m_i = m_o = t_{S_2} t_{S_3} (t_{S_1} + 2t_T). \tag{24}$$

For the corresponding time we obtain respectively

$$m' = (m_i + m_o)L + 2\tau_{\text{sync}}. \tag{25}$$

To obtain the volume of a sub-slab, we multiply the hexagon area by its length in the S_2 and S_3 axes and we obtain

$$V_{\text{tile}} = t_{S_2} t_{S_3} \frac{t_T(w_{\text{tile}} + t_{S_1})}{2}. \tag{26}$$

The iteration space dependences indicate that the computations in a tile can be done in parallel in each row, and in a sequential manner between rows from bottom to top. We therefore find that the computation time for a **non-boundary/steady state** tile is given by

$$\begin{aligned} c &= 2 \sum_{x=t_{S_1}, step=2}^{w_{\text{tile}}} \left(\left\lceil \frac{x t_{S_2} t_{S_3}}{n_V} \right\rceil C_{iter} + \tau_{\text{sync}} \right) \\ &= 2C_{iter} \sum_{x=t_{S_1}, step=2}^{w_{\text{tile}}} \left\lceil \frac{x t_{S_2} t_{S_3}}{n_V} \right\rceil + t_T \tau_{\text{sync}}. \end{aligned} \tag{27}$$

Now, the execution time of an entire slab depends on whether or not hyper-threading is performed. If we have a single tile on each SM (i.e. $k = 1$) we obtain

$$T_{\text{slab}}(1) = (m' + c)N_{sslabs}. \tag{28}$$

On the other hand, with hyper-threading enabled (i.e. $k > 1$), $T_{\text{tile}}(k)$ is dominated by $\max(m', c)$, and so,

$$T_{slab}(k) = m' + k \max(m', c) N_{sslabs}. \tag{29}$$

Plugging this into (2) and simplifying yields

$$T_{\text{alg}} = N_w T_{\text{sync}} + N_w T_{\text{slab}}(k) \left\lceil \frac{1}{n_{SM}} \left\lceil \frac{w}{k} \right\rceil \right\rceil. \tag{30}$$

All the equations developed here hold true for all stencil codes generated by HHC compiler. However, the parameter C_{iter}, which corresponds to computation time of the loop body, varies with number and type of computations. Again, the model is not restricted to HHC style codes. It can be applied to other parallelization strategies. Consider, wavefront parallel Jacobi1D stencil. The total execution time is the sum of the times for each wavefront. The time for each kernel call (or wavefront) is the sum of the time needed for the SM with maximum number of tiles assigned to it to finish. Hence, equation 6 holds for wavefront parallel codes.

5. EXPERIMENTAL VALIDATION

To validate the model, we perform a number of experiments on two NVIDIA platforms: GTX 980 and Titan X. Our benchmarks include four 2D stencils: Jacobi, Heat, Laplacian and Gradient, all first order stencils. The benchmarks also include two 3D stencils: Heat and Laplacian. All 2D stencils have two space dimensions and one time dimension. The two space dimension sizes we explore are 4096^2 and 8192^2. For each such size, we explore five problem sizes in time dimension (T): 1024, 2048, 4096, 8192 and 16384. In total, we explore 10 different combinations of problem size parameters. With 4 benchmarks, 10 size combinations, and 2 machines, we have a total of 80 combinations, which we refer to as *2D stencil experiments*. Similarly, all 3D stencils have three space dimensions and one time dimension. The three space dimension sizes are 384^3, 512^3 and 640^3. For each such size, we explore five problem sizes in time dimension (T): 128, 256, 384, 512 and 640 where $T \leq S$. In total, we explore 12 different combinations of the problem size parameters. With 2 benchmarks, 12 size combinations, and 2 machines, we have a total of 48 combinations, which we refer to as *3D stencil experiments*.

5.1 Baseline Experiments

We maximize the memory footprint of the tile subject to capacity constraints. Hence, we obtain tile sizes, which are as large as shared memory capacity M_{SM}. This means we execute only one tile per SM at a time. However, both GPUs allow $48K$ shared memory per thread block. This constraint is enforced such that we experience the benefit of hyperthreading factor of two. In HHT paper [22], the authors suggests tile sizes that maximize the compute to IO ratio. We use similar strategies to construct what we call the *baseline experiments* that enable a good exploration of the feasible space.

The shared memory requirement of a tile is given by M_{tile}, which is a function of tile size parameters. Shared memory constraints limit the feasible number of tile sizes. We take data points that maximize M_{tile} over M_{SM} per thread block. To explore hyperthreading, we add data points that allow multiple thread blocks to execute concurrently on one SM. Using this approach, for each experiment we select a set of tile sizes t_{T}, t_{S_1}, and t_{S_2} for 2D stencils. In addition to these tile sizes, we select t_{S_3} for 3D stencils. We generate 85 unique tile size combinations per experiment and, for each of them, we explore 10 different values of $n_{\text{thr,i}}$. Each unique combination of an experiment with the parameters t_{T}, t_{S_1}, t_{S_2}, and $n_{\text{thr,i}}$ is called a *data point*. Hence, our baseline-experiment set contains 850 data points for each experiment.

The HHC compiler generates a separate program (code) for every data point (it cannot produce codes with parametric tile sizes), a total of $850 \times (80+48) = 108,800$ data points. We measure execution time of each data point over five runs, and select the *smallest* of the five measurements. We made this choice of the minimum (rather than the average) as this is a common strategy in performance tuning/optimization and, also, since our model makes optimistic assumptions regarding run time behavior, choosing the smallest time is consistent with our modeling objective.

In order to complete the time model, additional hardware parameters need to be specified. Some of the needed values can be taken from vendor-provided hardware specifications. Table 2 shows such parameters for our two platforms.

Table 2: GPU configuration

Architecture Parameters	Type	GTX 980	Titan X
n_{SM}	EH	16	24
n_{v}	EH	128	128
M_{SM} [KB]	EH	96	96
R_{SM}	EH	65536	65536
shared memory banks	EH	32	32
max threadblocks per SM	EH	32	32

Table 3: Parameter values for the micro-benchmarks

Parameter [unit]	GTX 980	Titan X
L [s/GB]	7.36×10^{-3}	5.42×10^{-3}
τ_{sync} [s]	7.96×10^{-10}	6.74×10^{-10}
T_{sync} [s]	9.24×10^{-7}	9.00×10^{-7}

We also need values of the remaining parameters L, τ_{sync}, T_{sync} and C_{iter}, which could not be obtained from hardware specifications. We conduct the following microbenchmark experiments to gather these values.

5.2 Microbenchmarks

For L, τ_{sync} and T_{sync}, the micro-benchmarks are implemented such that the execution time is dominated by the operation of interest. The experimental parameter values that we empirically determined are listed in Table 3.

Another crucial component in our model is C_{iter}. The parameter C_{iter} denotes the execution time of one iteration of the loop body per vector unit provided that all the necessary data is available in shared memory. Its value depends on the types and number of operations in the loop body and on the platform. Since we have 6 benchmarks and 2 platforms, we need to determine 12 values for C_{iter}, one per combination. This is because C_{iter} is independent of the problem size parameters.

Notice that C_{iter} is not a simple function of the number of arithmetic operations of each type, but is quite complex, depending also on the instruction fetch/issue/execution latency, instruction issue pipeline, control flow, shared memory bank conflicts, data dependency, and many other factors. Analytically determining the execution time of a single iteration considering all these factors is a difficult problem. Thus, we estimate the value of C_{iter} empirically. For this purpose, we remove all global⇔shared memory data transfers, while making sure that the computations we want to measure do not get optimized away. The execution time is measured for 70 randomly picked problem and tile sizes and is determined by dividing the execution time per vector unit by the number of iterations of the particular instance. Finally, we take the average over all 70 runs to compute the value of C_{iter} for a benchmark-machine combination. The resulting values are given in Table 4.

Now that we have the values of all parameters in our model, we can use them to validate the model.

5.3 Validation Results

Using the parameter values from the previous subsection, we compute the predicted execution time T_{alg} for each data point. For each benchmark-machine combination we have 10 problem sizes and 850 data points, which overall are

Table 4: Values of C_{iter} in seconds

Benchmark	GTX 980	Titan X
Jacobi2D	3.39×10^{-8}	3.83×10^{-8}
Heat2D	3.68×10^{-8}	4.23×10^{-8}
Laplacian2D	3.11×10^{-8}	3.81×10^{-8}
Gradient2D	6.09×10^{-8}	7.60×10^{-8}
Heat3D	1.55×10^{-7}	1.64×10^{-7}
Laplacian3D	1.36×10^{-7}	1.44×10^{-7}

8500 data points. We compute the root mean square error (RMSE) and observe that the RMSE is in the range of 45%–200% when considering the whole set of data points. However, as our model is designed to optimistically predict the execution time, these inaccuracies are expected. In fact, they are inevitable as our model is not designed to precisely predict the performance of data-points that result in inefficient implementations.

However, when restricting the analysis to the top performing (in terms of GFLOPS per second) data points, our model turns out to be very accurate. Out of the 850 points for each benchmark-platform combination, we observe that for all the data points that are within 20% of the top performing one, the RMSE is for all stencils on both GPUs below 10%. Figure 3 shows the correlation between the predicted and the measured time over the top performing points.

In the following section, we show that the time model can be used for tile size optimization.

6. TILE SIZE OPTIMIZATION

The model we developed in Section 4 can be used to predict the efficiency of a code for given values of size parameters, such as S and T, but more importantly, it can be used to select the compiler parameters, in our case, the tile sizes, that lead to the best performance. We first formulate the optimization problem mathematically, describe the limitations that prevent a standard non-linear solver from finding an optimal solution, and then describe how we solved the problem via a simple exhaustive enumeration. Finally, we present our experimental results.

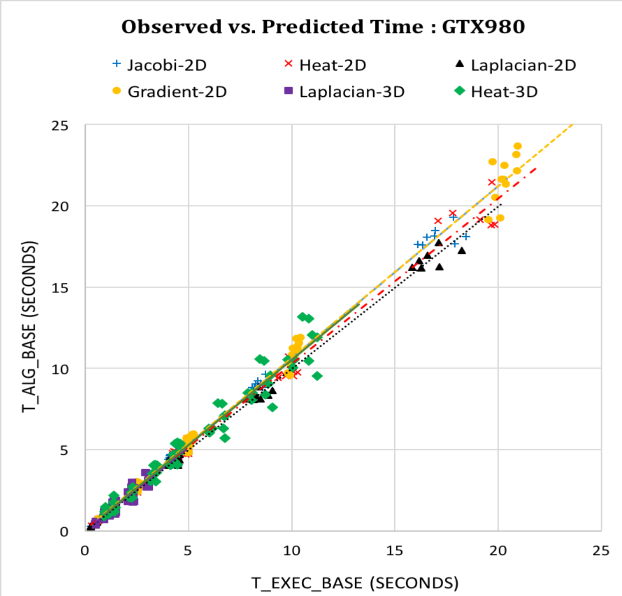
6.1 The Optimization Problem

We formulate the problem of finding optimal tile sizes as a mathematical optimization problem of the following type:

$$\begin{aligned} \underset{t_{\mathrm{S}_1}, t_{\mathrm{S}_2}, t_{\mathrm{T}}}{\text{minimize}} \quad & T_{\mathrm{alg}}(t_{\mathrm{S}_1}, t_{\mathrm{S}_2}, t_{\mathrm{T}}) \\ \text{subject to} \quad & M_{\mathrm{tile}} \le M_{\mathrm{SM/threadblock}} \\ & k \le MTB_{\mathrm{SM}} \\ & kM_{\mathrm{tile}} \le M_{\mathrm{SM}} \\ & t_{\mathrm{S}_1}\text{–integer}, t_{\mathrm{S}_2}\text{–multiple of 32}, t_{\mathrm{T}}\text{–even} \end{aligned} \tag{31}$$

where M_{tile} and k are functions of the tile sizes. We require t_{T} to be even, as necessary for hybrid-hexagonal tiling [22], and t_{S_2} to be a multiple of 32 to ensure that neighboring threads in S_2 fill complete warps (groups of 32 threads).

The optimization problem at hand is of a type that does not allow very efficient solution methods as it is non-linear and non-convex and has integer variables. On the other hand, it has a small number of variables (only three). Also, despite the problem being non-continuous due to the ceiling



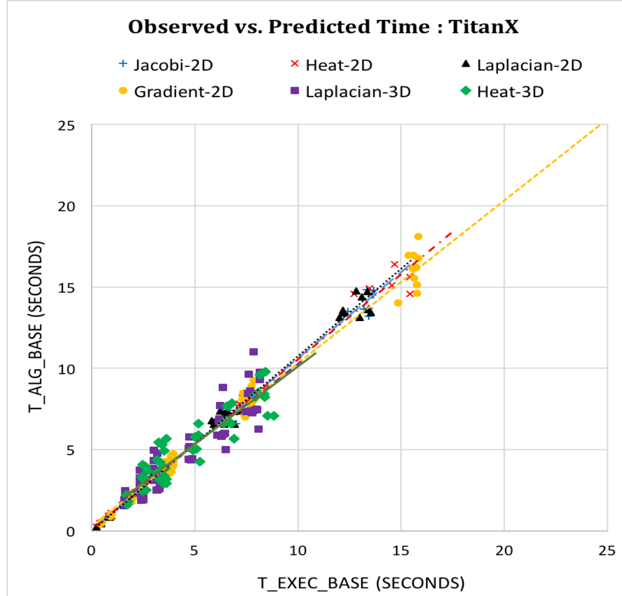


Figure 3: Observed execution time vs. model predicted time on GTX 980 and on Titan X, where $T_{\text{alg\_base}}$ denotes the model predicted time and $T_{\text{exec\_base}}$ denotes the measured execution time for the baseline experiments.

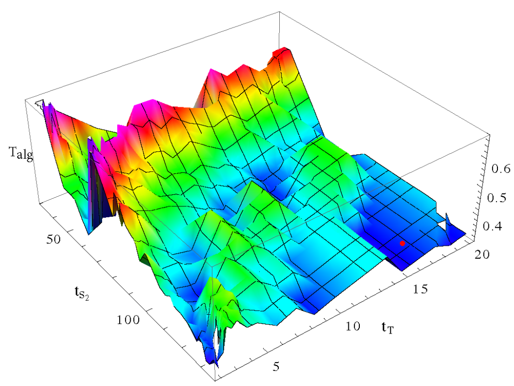


Figure 4: T_{alg} for Heat2D and GTX 980 as a function of t_{T} and t_{S_2} and with t_{S_1} fixed at 8. The red dot shows $T_{\text{alg\_min}}$, the point of minimum over all T_{alg}.

and floor functions, it can be made continuous by replacing these functions with new variables and inequality constraints, e.g., the ceiling in $\lceil x \rceil$ can be eliminated by introducing a new integer variable xc to replace $\lceil x \rceil$ and adding the inequality $x \leq xc$, assuming T_{alg} is a non-decreasing function with respect to $\lceil x \rceil$. Figure 4 illustrates the shape of T_{alg} as a 2D function (the 3D plot is sliced at $t_{\text{S}_1} = 8$) and shows that T_{alg} varies significantly with the tile sizes, so careful tile size selection is indeed important for getting good performance.

We encoded the optimization problem in the modeling language AMPL [18] and solved it using several non-linear solvers, including commercial ones. The best results were obtained using the open-source solver Bonmin [7]. All those solvers use heuristics that allow relatively good (but sub-optimal) solution to be found for large problems, but for small problems like ours they do not offer an option to do an exhaustive search that would find the optimal solution.

One of the main reasons for the somewhat disappointing performance is that the feasible space of the optimization problem 31 does not capture an important pragmatic aspect of the GPU code, as we are unable to model it, namely the number of physical registers that the generated code uses: this information is only available *after the generated code is compiled*. It is well known that if the number of registers exceeds the number of physical registers in the SM, namely the hardware parameter R_{SM}, the additional registers are implemented as "virtual registers" and get spilled and restored from global memory. This is known to be extremely inefficient and slows down the generated code.

In our model, we do not have a function for R_{tile}, since this is very difficult to model analytically. Because of this, we used the following approach.

- We evaluate our objective function within the entire feasible space of of Eqn 31.
- We keep all points that yield execution times within 10% of model predicted minimum value of T_{alg}. Many of these were not in our set of 850 baseline experiments.
- We generated codes for the new tile sizes in this set and evaluated their performance.

6.2 Experimental Results

We observe that the new tile sizes perform better than those obtained in baseline. Figure 5 shows the execution times and model-predicted times for the Gradient-2D stencil for a problem size of $S_1 = 8192$, $S_2 = 8192$ and $T = 8192$. Clearly, the optimal tile sizes predicted by the model outperform the best baseline tile sizes. As we search within the 10% vicinity of $T_{\text{alg\_min}}$, we find multiple near-optimal points. Baseline observed best is at *19.8 seconds*, whereas our model predicted optimal gives us a tile size that takes 16.5 *seconds*, which is a 17% improvement in performance. This model predicted tile size was not explored in our set of

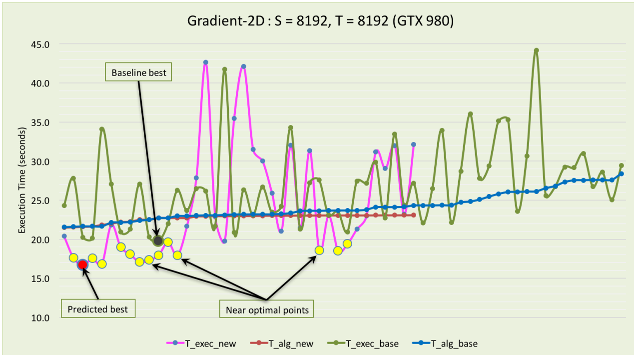


Figure 5: Predicted tile size performance of Gradient-2D for $S_1 = 8192$, $S_2 = 8192$ and $T = 8192$ on GTX 980.

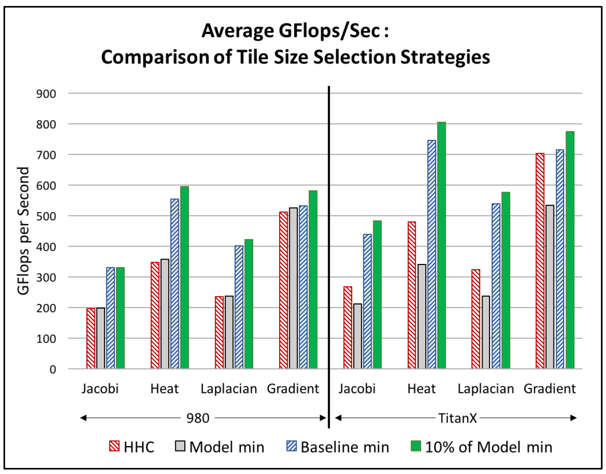


Figure 6: Average (over 10 problem sizes) GFlops/Sec achieved by different tile size selection strategies for 2D stencils.

baseline tile sizes. Moreover, we observe multiple near optimal points in the range of *16.5-19.8 seconds*. We get similar performance improvements for all 2D stencils on both platforms over all different problem sizes.

We compare the performance of different tile sizes obtained from HHC, $T_{\text{alg\_min}}$, Exhaustive search and best within 10% of $T_{\text{alg\_min}}$. Figure 6 shows the average GFlops per second achieved by different tile size selection strategies for 2D stencils over ten different problem sizes. It is clear that tile sizes corresponding to $T_{\text{alg\_min}}$ have poor performance in all cases. Another important conclusion is that the conventional wisdom of using large tile sizes does not yield best performance. The tile sizes that are within 10% of $T_{\text{alg\_min}}$ give the best performance with improvement of 60% over HHC and 9% over Baseline.

Finally, note that exhaustive searching over the entire feasible space is not practical, and no autotuner does this. HHC does not have an established autotuner. Comparing against a generic autotuner (e.g., opentuner) could be interesting, but tuning without good domain knowledge will be difficult, as the search space isn't easy to navigate. The feasible space is at least 200 times larger than the number of experiments we ran, and these took many weeks of dedicated machine time.

7. DISCUSSION

There are many "rules of thumb" used to optimize GPU programs. Expert programmers strive to ensure "high occupancy," avoid control-divergence, "coalesce" their memory accesses, and tune the number of threads. They also suggest maximizing the shared-memory footprint, while ensuring latency-hiding of memory accesses (usually by virtualizing multiple threadblocks per physical core). While all these parameters are very difficult to tune, many of these conditions are already ensured by a highly tuned domain specific compiler like HHC. Moreover, our model deliberately ignores some of these. We now discuss the limitations, and also the generality of our approach.

Execution of full warps without thread divergence (except on data-space boundaries) is guaranteed by the HHC compiler, if tile sizes in the innermost dimension are multiples of 32. So we use this to ensure divergence-freedom for all configurations that we predict/model/generate. Similarly, the HHC compiler generates code that guarantees coalesced accesses. This justifies some of our optimistic assumptions.

Limitations.

The two key limitations of our model are in register usage and the number of threads-per-block (in possibly multiple dimensions). Both are very difficult to model analytically. Although the register usage does not feature directly in our objective function, it appears in a constraint of the feasible space: our optimistic model holds only if register spills do not slow down the execution. However, this can only be known after the back-end `nvcc` compiler is called. Because of this, the results using an off-the-shelf solver like Bonmin were disappointing, and we used the script-driven exhaustive analytical evaluation described in Section 6.1.

The threads-per-block parameter(s) have a significant impact on performance, and this is also hard to model. Largely because of this, our tile selection could not be accomplished using only model-based optimization, but needed additional exploration of the search space in the vicinity of the model-predicted optima. However, we did take it into account during experimentation. Among the high-performing instances, we found that the values of this parameter that yielded the locally best performance was easily predictable—empirically, rather than analytically. The threads-per-block parameter used in our final, optimization experiments use this empirically predicted value.

Generality.

Our model can be extended to other stencil types for e.g., higher order stencils (provided they can be handled by HHC-compiler). When dependences change, the slopes of the hexagons change by constant factors, the memory footprints change similarly, etc. These are exactly the terms that a polyhedral compiler like HHC manipulates when allocating memory buffers, and constructing loop bounds. Our ongoing work is in incorporating the model into the compiler itself.

We reiterate that although the model is specific to HHC, the approach itself, which involves the choice of parameter hierarchy and the methods of their estimation/approximation, is extensible. For rectangular tiles, for instance, the formulae for N_w, m_i, m_o, etc., will be different, but the elementary software and hardware parameters will be the same, and the methods for deriving the formulae will be similar.

Revisiting conventional wisdom.

A commonly used "rule of thumb" suggests that the optimal tiling strategy is to choose the "largest possible tile size that fits" i.e., its memory footprint matches the available capacity. Our results suggest that we should question this. First of all, this strategy precludes overlapping of computation and communication (the "hyperthreading effect"). But this can be avoided by explicitly accounting for hyperthreading. Indeed, many modern GPU platforms preclude such large size by limiting the maximum data footprint of a thread block to only *half* the shared memory capacity. So the "hyperthreading-adjusted conventional wisdom" would seek to maximize tile volume subject to the half-capacity constraint—the best strategy is the largest tile volume for the given footprint.

Our experimental data suggests otherwise—an even higher hyperthreading factor turns out to yield best performance in a wide range of our experiments. We still don't know why, and this is the subject of our ongoing investigation.

8. CONCLUSION

We developed a model for the execution time of stencil codes on the GPU platform and used it for tile size selection for stencil codes generated by the HHC polyhedral compiler. Our model is very accurate for predicting the times of problem instances whose performance is within 20% of the optimal and, hence, it can be used to find values for tunable parameters that will give near optimal performance. We applied our model for optimizing the tile sizes and experimentally observed a noticeable improvement in performance when compared with manually determined best tile sizes found after significant numbers of experiments.

To investigate the predictive capability of the model, we explored all points with predicted performance within 10% of $T_{\text{alg\_min}}$. This is because our model does not explicitly model many architectural and code features: thread divergence, imbalance among threads in the same warp, branch divergence, memory bank conflicts, etc. We also do not model the effect of the number of registers per thread block, a factor that can only be obtained "post mortem" after the `nvcc` compiler. This is why it is still necessary to have an empirical tuning phase, but we have shown that the number of points that need to be explored is relatively small.

Finally, we would like to note that a large part of the time and effort of conducting our experiments was the code generation effort. The HHC compiler generated codes where the tile size and many other parameters are fixed at compile time, necessitating a separate call to the compiler for each data point in our experiments. For some of the points this ran into several tens of seconds, and was a significant fraction of the total time that the experimentation took. We are therefore also exploring the use of parametric tiled code generation, where a single parametric code is generated once for a given input program, and can be reused for different tile sizes. Here, tile sizes may be set a launch time or even dynamically during execution. The trade-off this brings between code efficiency and compilation time is the subject of our ongoing research.

9. REFERENCES

[1] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. *International Journal of Parallel Programming*, 29(5):493–544, 2001.

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, P. Gebis, J. J. abd Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. EECS Tech Report EECE-2006-183, UC Berkeley, Decembeer 2006.

[3] V. Bandishti, I. Pananilath, and U. Bondhugula. Tiling stencil computations to maximize parallelism. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 40:1–40:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[4] C. Bleck, R. Rooth, D. Hu, and L. T. Smith. Salinity-driven Thermocline Transients in a Wind- and Thermohaline-forced Isopycnic Coordinate Model of the North Atlantic. *Journal of Physical Oceanography*, 22(12):1486–1505, 1992.

[5] U. Bondhugula, V. Bandishti, A. Cohen, G. Potron, and N. Vasilache. Tiling and optimizing time-iterated computations on periodic domains. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 39–50, New York, NY, USA, 2014. ACM.

[6] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2008.

[7] *Bonmin Project Page*. https://projects.coin-or.org/Bonmin, 2015 (accessed March 11, 2016).

[8] R. A. Chowdhury, H.-S. Le, and V. Ramachandran. Cache-oblivious dynamic programming for bioinformatics. *TCBB*, 7(3):495–510, July-September 2010.

[9] M. Christen, O. Schenk, and H. Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE Int.*, pages 676–687, May 2011.

[10] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Birkhaüser, 2000.

[11] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 4:1–4:12, Austin, TX, November 2008. http://portal.acm.org/citation.cfm?id=1413370.1413375.

[12] H. Dursun, K. Nomura, L. Peng, R. Seymour, W. Wang, R. K. Kalia, A. Nakano, and P. Vashishta. A multilevel parallelization framework for high-order stencil computations. In *Euro-Par 09*, pages 642–653, Delft, The Netherlands, August 2009.

[13] H. Dursun, K. Nomura, W. Wang, M. Kunaseth,

L. Peng, R. Seymour, R. K. Kalia, A. Nakano, and P. Vashishta. In-core optimization of high-order stencil computations. In *PDPTA*, pages 533–538, Las Vegas, NV, July 2009.

[14] J. F. Epperson. *An Introduction to Numerical Methods and Analysis*. Wiley-Interscience, 2007.

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

[16] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part I. one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–347, 1992.

[17] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part II. multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.

[18] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modelling Language for Mathematical Programming*. Duxbury Press, Brooks/Cole Publishing Company, 2nd edition, 2002.

[19] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS: IEEE Symposium on Foundations of Computer Science*, pages 285–297, New York, NY, October 1999.

[20] M. Frigo and V. Strumpen. Cache oblivious stencil computations. In *Proc. of the 19th Annual Int. Conf. on Supercomputing*, ICS '05, pages 361–366, New York, NY, USA, 2005. ACM.

[21] S. M. Griffies, C. Böning, F. O. Bryan, E. P. Chassignet, R. Gerdes, H. Hasumi, A. Hirst, A.-M. Treguier, and D. Webb. Developments in Ocean Climate Modelling. *Ocean Modelling*, 2:123–192, 2000.

[22] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege. Hybrid hexagonal/classical tiling for GPUs. In *CGO*, page 66, Orlando, FL, Feb 2014.

[23] T. Gysi, T. Grosser, and T. Hoefler. Modesto: Data-centric analytic optimization of complex stencil programs on heterogeneous architectures. In *Proc. of the 29th ACM on Int. Conf. on Supercomputing*, ICS '15, pages 177–186, New York, NY, USA, 2015. ACM.

[24] T. Gysi, C. Osuna, O. Fuhrer, M. Bianco, and T. C. Schulthess. Stella: A domain-specific tool for structured grid methods in weather and climate models. In *Proc. of the Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 41:1–41:12, New York, NY, USA, 2015. ACM.

[25] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector simd architectures. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 13–24, New York, NY, USA, 2013. ACM.

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

[27] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 152–163, New York, NY, USA, 2009. ACM.

[28] C. John. *Options, Futures, and Other Derivatives*. Prentice Hall, 2006.

[29] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, April 2010.

[30] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *MSPC 2005: Workshop on Memory Systems Performance*, pages 36–43, Chicago, IL, June 2005. ACM Sigplan.

[31] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *MSPC 2006: Workshop on Memory Systems Performance and Correctness*, pages 51–60, San Jose, CA, October 2006. ACM Sigplan.

[32] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *PLDI 2007: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 235–244, San Diego, CA, June 2007. ACM.

[33] Z. Li and Y. Song. Automatic tiling of iterative stencil loops. *ACM Trans. Program. Lang. Syst.*, 26(6):975–1028, Nov. 2004.

[34] P. Liu, R. Seymour, K. Nomura, R. K. Kalia, A. Nakano, P. Vashishta, A. Loddoch, M. Netzband, W. R. Volz, and C. C. Wong. High-order stencil computations on multicore clusters. In *IPDPS 2009: IEEE International Parallel abd Distributed Processing Symposium*, pages 1–11, Rome, Italy, May 2009.

[35] C. Mauras, P. Quinton, S. Rajopadhye, and Y. Saouter. Scheduling affine parameterized recurrences by means of variable dependent timing functions. In S. Y. Kung and E. Swartzlander, editors, *International Conference on Application Specific Array Processing*, pages 100–110, Princeton, New Jersey, Sept 1990. IEEE Computer Society.

[36] W. Mei, W. Shyy, D. Yu, and L. S. Luo. Lattice Boltzmann Method for 3-D Flows with Curved Boundary. *Journal of Computational Physics*, 161(2):680–699, 2000.

[37] J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 256–265, New York, NY, USA, 2009. ACM.

[38] P. Micikevicius. 3D finite difference computation on GPUs using CUDA. In *GPPGPU*, pages 79–84, Washington, DC, March 2009.

[39] R. T. Mullapudi, V. Vasista, and U. Bondhugula. Polymage: Automatic optimization for image processing pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 429–443, New York, NY, USA,

2015. ACM.

[40] A. Nakano, R. K. Kalia, and P. Vashishta. Multiresolution Molecular Dynamics Algorithm for Realistic Materials Modeling on Parallel Computers. *Computer Physics Communications*, 83(2-3):197–214, 1994.

[41] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5D blocking optimization for stencil computations on modern CPUs and GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–13, Washington, DC, USA, 2010. IEEE Computer Society.

[42] A. Nitsure. Implementation and optimization of a cache oblivious lattice boltzmann algorithm. Master's thesis, Institut für Informatic, Friedrich-Alexander-Universität Erlangen-Nürnberg, July 2006.

[43] L. Peng, R. Seymour, K. ichi Nomura, R. K. Kalia, A. Nakano, P. Vashishta, A. Loddoch, M. Netzband, W. R. Volz, and C. C. Wong. High-order stencil computations on multicore clusters. In *IPPS*, 2009.

[44] P. Quinton and V. Van Dongen. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI Signal Processing*, 1(2):95–113, 1989.

[45] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[46] S. V. Rajopadhye, S. Purushothaman, and R. M. Fujimoto. On synthesizing systolic arrays from recurrence equations with linear dependencies. In *Proceedings, Sixth Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 488–503, New Delhi, India, December 1986. Springer Verlag, LNCS 241.

[47] G. Rizk, D. Lavenier, and S. Rajopadhye. *GPU accelerated RNA folding algorithm*, chapter 14. Morgan Kauffman, 2010. in GPU Computing Gems 4, editor: W-M. Hwu.

[48] M. Shaheen and R. Strzodka. Numa aware iterative stencil computations on many-core system. In *26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Shanghai, China, 2012.

[49] J. Shirako, K. Sharma, N. Fauzia, L.-N. Pouchet, J. Ramanujam, P. Sadayappan, and V. Sarkar. *Analytical Bounds for Optimal Tile Size Selection*, pages 101–121. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[50] R. Strzodka, M. Shaheen, and D. Pajak. Time skewing made simple (poster). In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 295–296, New York, NY, USA, 2011. ACM.

[51] R. Strzodka, M. Shaheen, D. Pajak, and H.-P. Seidel. Cache oblivious parallelograms in iterative stencil computations. In *24th ACM/SIGARCH International Conference on Supercomputing (ICS)*, pages 49–59, Tsukuba, Japan, June 2010.

[52] R. Strzodka, M. Shaheen, D. Pajak, and H. P. Seidel. Cache accurate time skewing in iterative stencil computations. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 571–581, Sept 2011.

[53] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM.

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

[55] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1):3–35, 2001.

[56] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *ACM Sigplan Not.*, volume 26, pages 30–44. ACM, 1991.

[57] M. J. Wolfe. Iteration space tiling for memory hierarchies. *Parallel Processing for Scientific Computing (SIAM)*, pages 357–361, 1987.

[58] D. Wonnacott. Time skewing for parallel computers. In *Languages and Compilers for Parallel Computing, 12th International Workshop, LCPC'99, La Jolla/San Diego, CA, USA, August 4-6, 1999, Proceedings*, pages 477–480, 1999.

[59] D. Wonnacott. Achieving scalable locality with time skewing. *International Journal of Parallel Programming*, 30(3):1–221, 2002.

[60] J. Xue. *Loop Tiling for Parallelism*, volume 575 of *Kluwer International Series in Engineering and Computer Science*. Kluwer, 2000.

[61] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 63–76, New York, NY, USA, 2003. ACM.