DISSERTATION

ANALYTICAL COST METRICS : DAYS OF FUTURE PAST

Submitted by

Nirmal Prajapati

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Summer 2019

Doctoral Committee:

Advisor: Sanjay Rajopadhye

Wim Böhm
Edwin Chong
Louis-Noël Pouchet

ABSTRACT

ANALYTICAL COST METRICS : DAYS OF FUTURE PAST

Future exascale high-performance computing (HPC) systems are expected to be increasingly heterogeneous, consisting of several multi-core CPUs and a large number of accelerators, special-purpose hardware that will increase the computing power of the system in a very energy-efficient way. Specialized, energy-efficient accelerators are also an important component in many diverse systems beyond HPC: gaming machines, general purpose workstations, tablets, phones and other media devices. With Moore's law driving the evolution of hardware platforms towards exascale, the dominant performance metric (time efficiency) has now expanded to also incorporate power/energy efficiency.

This work builds analytical cost models for cost metrics such as time, energy, memory access, and silicon area. These models are used to predict the performance of applications, for performance tuning, and chip design. The idea is to work with domain specific accelerators where analytical cost models can be accurately used for performance optimization. The performance optimization problems are formulated as mathematical optimization problems. This work explores the analytical cost modeling and mathematical optimization approach in a few ways. For stencil applications and GPU architectures, the analytical cost models are developed for execution time as well as energy. The models are used for performance tuning over existing architectures, and are coupled with silicon area models of GPU architectures to generate highly efficient architecture configurations. For matrix chain products, analytical closed form solutions for off-chip data movement are built and used to minimize the total data movement cost of a minimum op count tree.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Sanjay Rajopadhye, for his guidance and support. He has been a continuous source of motivation. His brilliant ideas and farsightedness have awed me throughout my Ph.D. I learnt a lot from him and enjoyed being his student. I not only gained technical knowledge in the field but also learnt values of a good human being like patience, kindness, selflessness, compassion, and dedication. I feel honored to have the opportunity to work with him.

I am thankful to my committee members Wim Böhm, Louis-Noël Pouchet, and Edwin Chong for their valuable time and inputs. Their expertise and insights have helped me improve the quality of my work. In addition to the committee members, my co-authors Hristo Djidjev, Rumen Andonov, Tobias Grosser, and Nandkishore Santhi have mentored me at various stages of my Ph.D. and have been a great source of knowledge. I would like to thank all the members of the mélange group at Colorado State University for the pleasant and joyful company.

My family has supported me throughout my education. I would like to thank my parents Amrutbhai Prajapati and Nayanaben Prajapati for believing me. They recognized my love for science early in my childhood and encouraged me to pursue my passion. My parents marched against the social norms of educating a girl and invested their blood, sweat and tears to help me achieve my dreams. I would like to thank my lovely sister Anjali Prajapati and my amazing brother Vishal Prajapati for being the never ending source of energy and happiness. It is because of my siblings that I saw hopes in my darkest times.

I would like to thank Ritesh Banka, Amrapalli Garanaik, Anantkumar Shah, Aditi Prasad, Ankitaben Patel, and Athith Rao for their selfless love and support. All of these awesome, crazy people have been a huge source of motivation during the highs and lows of my Ph.D. life. Thanks to them, I survived this roller coaster!

Finally, I would like to thank everyone who contributed to my education. I would like to thank Cindy Norris, Jay Fenwick, and Alice McRae for giving me an amazing flavor of research during my Masters at Appalachian State University. I would also like to thank my middle school teacher

Late Mrs. Annama Thomas for recognizing the potential in me and molding my brain since my early childhood. She cultivated the problem solving skills in me and constantly motivated me to push my boundaries. She is no more but her lessons and values are alive in me. I would like to extend my thanks to all my relatives who have been looking forward to my graduation. I am proud to be the first generation Ph.D. in the family!

DEDICATION

I would like to dedicate this thesis to the memory of my loving grandparents,

Amthiben Shankarlal Prajapati, and Kalidas Maganlal Prajapati.

You are gone but your belief in me has made this journey possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

"When you change the way you look at things, the things you look at change." – Max Planck

As we move towards the exascale era, the new architectures must be capable of running the massive computational problems efficiently. Scientists and researchers are continuously investing in tuning the performance of extreme-scale computational problems. These problems arise in almost all areas of computing, ranging from big data analytics, artificial intelligence, search, machine learning, virtual/augmented reality, computer vision, image/signal processing to computational science and bioinformatics.

With Moore's law driving the evolution of hardware platforms towards exascale, the dominant performance metric (time efficiency) has now expanded to also incorporate power/energy efficiency. Therefore the major challenge [2, 3, 4, 5, 6] that we face in computing systems research is: "*how to solve massive-scale computational problems in the most time/power/energy efficient manner*?"

The architectures are constantly evolving making the current performance optimizing strategies less applicable and new strategies need to be invented. The solution is for the new architectures, new programming models, and applications to go forward together. Doing this is, however, extremely hard. There are too many design choices in too many dimensions. The algorithms/applications may have a wide range of parameters, making them either intensely bandwidth bound or heavily compute bound, limited only by the hardware's operational throughput. This significantly affects the algorithms used to obtain the best solution. At the hardware end, the target platform affects not only the parallelization methods but also the tool-chains used. To overcome this complications, cost models and simulators are used.

We propose the following strategy to solve the problem, i.e., *"how to solve massive-scale computational problems in the most time/power/energy efficient manner*?":

1. **Models** Develop accurate analytical models (e.g. *execution time, energy, silicon area*) to predict the cost of executing a given program.

2. **Complete System Design** Simultaneously optimize all the cost models for the programs (computational problems) to obtain the most *time/area/power/energy efficient* solution. Such an optimization problem evokes the notion of codesign.

Codesign—the simultaneous design of hardware and software—has two common interpretations [7, 2]. *System codesign* [8, 9] is the problem of simultaneously designing hardware, run-time system, compilers, and programming environments of entire computing systems, typically in the context of large-scale, high-performance computing (HPC) systems and supercomputers. *Application codesign*, also called *hardware-software codesign* [10, 11, 12], is the problem of systematically and simultaneously designing a dedicated hardware platform and the software to execute a single application (program). The proposed approach is applicable to both contexts.

Solving the problem of *System codesign* in its full generality is very difficult [8, 5, 2, 3] because of the wide range of (i) computational problems, (ii) the programming languages and abstractions used to express them, and (iii) varying target architectures, from data centers and cloud-based platforms, to distributed heterogeneous mobile platforms such as phones, and even "things." Therefore, we suggest the following breakdown:

1. **Employ Domain-Specificity** Choose (i) a (small set/class) of programs, (ii) highly optimized hardware accelerators, and (iii) the optimal compiler transformations.

2. **Develop Cost Models** Develop accurate analytical cost (Time/Data/Area/Energy) models for performance prediction and optimization.

3. **Mini Sub Prob - Specialize** Solve the codesign problem for the specific domain in (1) using the models in (2) i.e. optimize the program, compiler and hardware parameters simultaneously.

4. **Large Prob - Generalize** Repeat the above process to cover various important classes of programs, assign a weight to each class of programs and formulate an optimization problem

for this weighted set of program parameters. Simultaneously, solve for all the parameters and for all classes of programs.

1.1 Specialization

Application codesign [13] is particularly important for embedded systems. In many uses of these systems (e.g., self-driving cars, computational fluid dynamics, neural networks, medical imaging, smart cameras, and cyber-physical systems) general purpose platforms based on standard CPUs deliver inadequate "performance" on a combination of many cost metrics: speed/throughput, power/energy, weight, size, and manufacturing/fabrication cost, especially in volumes that the market can sustain. As a result, specialized hardware is essential.

Hardware platforms for embedded systems are usually heterogeneous, with (instruction-set) programmable processors (CPUs and micro-controllers), accelerators that are either instruction-set programmable (e.g., GPUs), or "hardware programmable" ones like FPGAs and reconfigurable logic, as well as Application Specific Integrated Circuits (ASICs). For HPC systems, ASICs are usually not a designer option. In either case, the platforms have specialized, highly parallel, often fine-grain, components like FPGAs, GPUs, or DSPs, called *accelerators*.

The challenge is exacerbated when we consider the fact that accelerators are not one single architecture, and moreover, are constantly evolving. For example,

- Google developed the ***TPU*** (Tensor-flow Processing Unit), an ASIC to accelerate machine learning computations. It is completely invisible to end users, who access it via the TensorFlow tool, and whose back end presumably makes direct library calls to TPUs.
- Microsoft released ***Catapult***, an FPGA based fabric for accelerating large-scale data-center services. It is a custom design, written in Verilog, and accessed via library calls.
- At the other end of the spectrum, Facebook is developing its large scale machine learning applications using off-the-shelf GPUs and conventional tool chains: CUDA/OpenCL.

Mapping an application to an accelerator platform, *even when the hardware is fixed*, is extremely difficult. Codesign seeks to *simultaneously design* the hardware itself, and is, therefore, an even harder problem. Multiple cost metrics must be optimized, while still providing flexibility to the programmer and end user, and the design space is huge.

The key element of our approach is to exploit multiple forms of *domain-specificity* [2]. First, we tackle a specific (family of) computations that are nevertheless very important in many embedded systems. This class of computations, called *dense stencils*, includes the compute-intensive parts of many applications such as computational fluid dynamics, neural networks, medical imaging, smart cameras, image processing kernels, simulation of physical systems relevant to realistic visualization, as well as the solution of partial differential equations (PDEs) that arise in many cyber-physical systems such as automobile control and avionics.

Second, we target NVIDIA GPUs, which are *vector-parallel programmable accelerators*. Such components are now becoming de-facto standard in most embedded platforms and MPSoCs since they provide lightweight parallelism and energy/power efficiency. We further argue that they will become ubiquitous for the following reasons. Any device on the market today that has a screen (essentially, *any device*, period) has to render images. GPUs are natural platforms for this processing (for speed and efficiency). So all systems will have an accelerator, by default. If the system now needs any additional dense stencil computations, the natural target for performing it in the most speed/power/energy efficient manner is on the accelerator.

The third element of domain specificity is that we exploit a formalism called the *polyhedral model* as the tool to map dense stencil computations to GPU accelerators. Developed over the past thirty years [14, 15, 16, 17, 18], it has matured into a powerful technology, now incorporated into `gcc`, `llvm` and in commercial compilers like Rstream. Tools targeting GPUs are also available [19, 20].

Thus, we formulate the domain-specific optimization problem: *simultaneously optimize compilation and hardware/architectural parameters to compile stencil computations to GPUs*.

Previously, we presented [21] an approach to solve the above problem as follows:

1. **Develop Models**

   (a) **Time Model [22]** We show that the elements of the domain specificity can be combined to develop simple, analytical (as well as accurate) models for the execution time of tiled stencil codes on GPUs and that these model can be used to solve for optimal tile size selection. Our model was able to predict tile sizes that achieve 30% of theoretical machine peak on NVIDIA Maxwell GTX 980 and Titan X.

   (b) **Area Model [21]** We develop a simple, analytical model for the silicon area of programmable accelerator architectures, and calibrate it using the NVIDIA Maxwell class GPUs. Our model proved to be accurate to within 2% when validated.

   (c) **Energy Model [23]** We also developed energy models, as an explicit analytic function of a set of compiler and hardware parameters, that predict the energy consumption by analyzing the source code. We used these energy models to obtain optimal solutions to tile size selection problem.

2. **Codesign [21]** We combine the proposed execution time model [22] and the area model [21] with a workload characterization of stencil codes to formulate a mathematical optimization problem that minimizes a common objective function of all the hardware and compiler parameters. We propose a set of Pareto optimal designs that represent optimal combination of the parameters that would allow up to 126% improvement in performance (GFlops/sec).

Despite domain specificity, the problem remains difficult. Even when done by hand for single target architecture and an application kernel, it is more art than science. Although smart designers and implementers have worked for many decades on such problems for the "application/architecture du jour," each one was usually a point-solution [24, 25, 26]. Designers invested blood, sweat and tears to find the best implementation, used it to solve their problem of interest, usually published a paper explaining the design, and moved on. Their invested effort, particularly the trade-offs they made, and lessons they learned, are lost: future designers are left to reinvent wheels.

The high-level objective is to optimize stencil codes while tuning the hardware accelerator (GPUs) developing a complete ecosystem. The goal is to ***automatically and provably optimally, using time and/or energy as the objective function, map stencils to the hardware accelerators***. The idea is to obtain *provably optimal* mappings through rigorous mathematical optimizations. **The proposed approach can have the following benefits.**

- ***Automation with Optimality****:* the most time/power/energy efficient implementations can be derived, reducing programmers' effort. *Compilation tools* can be used to guide the optimal choice of transformations which will, in turn, *optimize the performance* of the workloads such as deep learning, image rendering, cyber-physical systems, autonomous vehicle systems, etc.
- ***Future proofing:*** Porting applications to new GPU architectures will require less effort. Instead of a redesign of each program, our methods can be used to develop new parallelization strategies and transformations, refine/redefine objective functions and constraints, and re-target the compiler. This one-time effort can then be amortized over many application kernels.
- ***Codesign:*** By casting hardware/architectural parameters as *variables* in the mathematical optimization framework, we can *solve* for their optimal values. This will enable us to systematically explore alternate GPU architectures and simultaneously tune compilation parameters. Such a codesign approach will help speed up the research work and the chip design process. The cost models can be used to quickly recognize the performance sinks and help *identify the design flaws in its early stages saving billions of dollars*.

Generalization Future exascale high-performance computing (HPC) systems are expected to be increasingly heterogeneous, consisting of several multi-core CPUs and a large number of accelerators, special-purpose hardware that will increase the computing power of the system in a very energy-efficient way [6]. Consequently, highly specialized coprocessors will become much more common in exascale than in the current HPC systems. Such specialized, energy-efficient acceler-

ators are also an important component in many diverse systems beyond HPC: gaming machines, general purpose workstations, tablets, phones and other media devices. In order to attain exascale level performance, accelerators have to become even more energy-efficient, and experts anticipate that a large part of this must come through increased specialization [2].

Our approach can be used to solve the problem of *System codesign* by applying proposed accelerator codesign techniques to all the classes of programs that optimize for all the parameters simultaneously. We provide a proof of concept of our approach, which is a stepping stone towards solving the larger problem of *transforming the GPUs into accelerators for HPC Systems.*

1.2 Breaking Abstractions

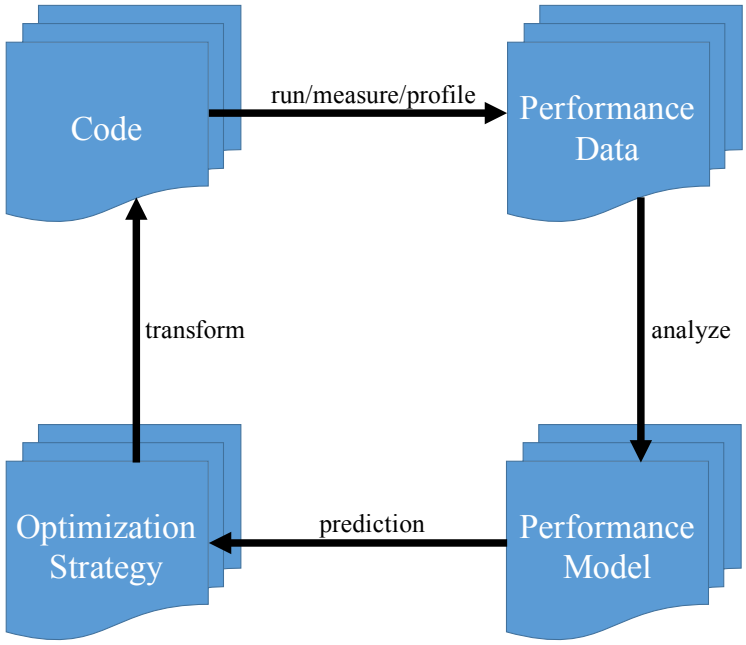


Figure 1.1: HPC application performance improvement cycle. The layers represent different architectures.

Since a long time the HPC developers and tool builders are using certain abstractions to improve the performance of applications. Figure 1.1 shows the performance improvement life cycle of an HPC application. The layers represent a separate life cycle for every architecture (e.g. GPUs, CPUs, FPGAs). A different Programming Language and System, each with its own Programming

Environment & Tools is used based on the underlying hardware architecture. Therefore, there are different codes for the same application on different hardware. Usually, the scientists who develop the algorithms/applications are completely isolated from the HPC performance tuning specialists. For every application, a profiler is used to get performance data which is analyzed to derive a performance model. The performance models are used to predict the performance and select an optimization strategy. Optimal transformation strategy is applied to get code. This cycle repeats until the satisfactory performance is obtained.

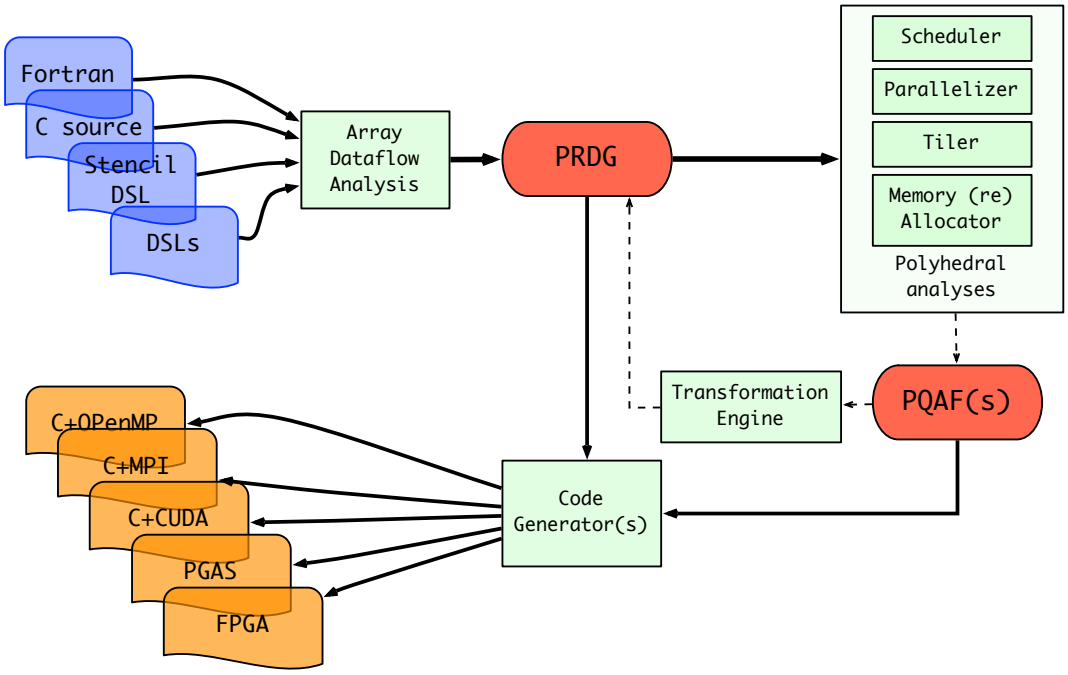


Figure 1.2: Polyhedral Compilation

Polyhedral compilers (see Figure 1.2) have a Polyhedral Reduced Dependence Graph (PRDG) as the intermediate representation of loops. Polyhedral analysis (such as scheduling, parallelizing, memory allocation, etc.) is performed on this graph. Piecewise Quasi-Affine Functions (PQAFs) are mathematical functions that transform the PRDG using cost functions. A transformed PRDG is used to generate codes. Notice the similarities and differences between Figures 1.1 and 1.2 and their performance cycles. The optimization strategies in Figure 1.1 are represent the PQAFs.

The performance models are subsumed in the polyhedral analysis phase. Our work focuses on using cost models for polyhedral analysis, in turn, breaking the cycle and reducing the time to find optimal solutions.

Design Space

1. The main idea of our work comes out as a consequence of some of the exascale challenges [2]. For exascale system design, various architectures, programs and transformation strategies are to be explored simultaneously in order to find the optimal. We add cost models to this design space and provide ***a unified view of the optimization space***. Figure 1.3 shows this view (more details in Section 2.2).

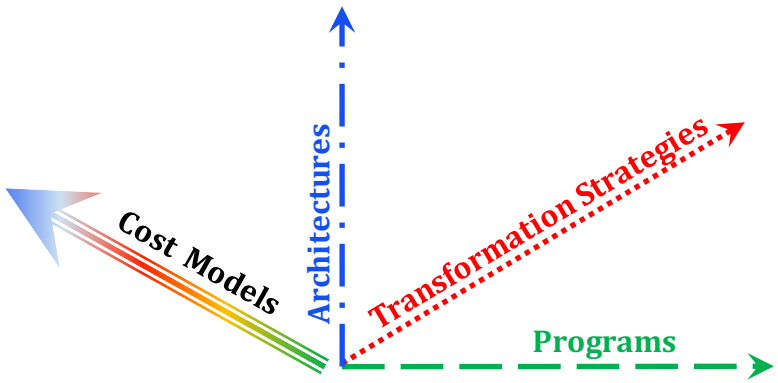


Figure 1.3: Unified View of the Design Space.

2. The above design space, however, is very large, has too many parameters and is too complicated to develop precise models. Therefore, we explore ***domain specificity and identify regions*** where optimization across multiple axes become possible.

We show how the analytical cost models can be used to optimize the performance of domain specific programs using transformation strategies for a given architecture in Chapter 2.

The rest of this document is organized as follows: Chapter 2 explains our proposed approach of navigating the design landscape. Chapter 3 discusses the relevant literature. Chapters 4, 5, 6, and 7 discuss the contributions. Finally, Chapter 8 concludes the document.

Chapter 2

The Landscape and Navigation

2.1 Domain Specificity

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 [27], 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 [28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40]. Workshops and conferences devoted exclusively to stencil acceleration have recently emerged.

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 occurring in graphics processing. Later, researchers realized that these processors could be used for more general computations, and, eventually, the emergence of tools like CUDA and OpenCL enabled general purpose parallel programming on these platforms.

Exploiting the specificity of the applications and the specificity of target architectures leads to domain-specific tools to map very high level program specifications to highly tuned and optimized implementations on the target architecture. Many such tools exist, both academic research prototypes and productions systems.

As indicated earlier, our domain specificity comes in multiple flavors. First, we investigate only stencil computations. They 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 [14, 15, 49, 16, 17, 18, 50, 51], and many tools are widely available, e.g., PPCG, developed by the group at ENS, Paris [52]. Second, we tackle a specific target platform, namely a single GPU accelerator, and PPCG includes a module that targets GPUs and incorporates a sophisticated code generator developed by Grosser et al. [19] that employs a state-of-the-art tiling strategy called *hybrid hexagonal classic tiling*. An open source compiler, implementing this strategy is also available, henceforth called the HHC compiler.

2.1.1 Comparison with Polyhedral Methods

The landscape described (in Figure 1.3) allows us to place our work in context. Although our methods are for domain specific purposes, an extreme situation with CPUs as the architecture, and the set of polyhedral programs allows us to compare with conventional compilation.

The optimization problem a compiler "solves" is: *pick transformation parameters so as to optimize the program property of interest, typically execution time.* Since it has a single (or a small handful of) predetermined strategies, it is a limited kind of mathematical optimization problem. The objective function is a surrogate for execution time.

Now consider PLuTO [51], a state-of-the art polyhedral compiler based on a mathematical representation of both programs and transformations. By considering only polyhedral programs and transformations, the optimization problems are rigorous. By default, PLuTO targets multi-core CPUs, and uses a transformation strategy that combines one level of tiling, loop fusion and

(loop/wavefront) parallelization of tiles. It solves a mathematical optimization problem where the schedule parameters (coefficients of tiling and schedule hyperplanes) are the unknown variables, and the cost function is the number of linearly independent tiling hyperplanes, combined with a quantitative measure of the length of the inter-tile dependences. This is again, a surrogate for the total execution time, and leads to solutions that while reasonable, are not *provably* optimal. Moreover, parameters like tile sizes, vectorization and inter-tile schedule are chosen using simple heuristics, and are not part of the optimization.

2.1.2 Limitations of current domain-specific compilation

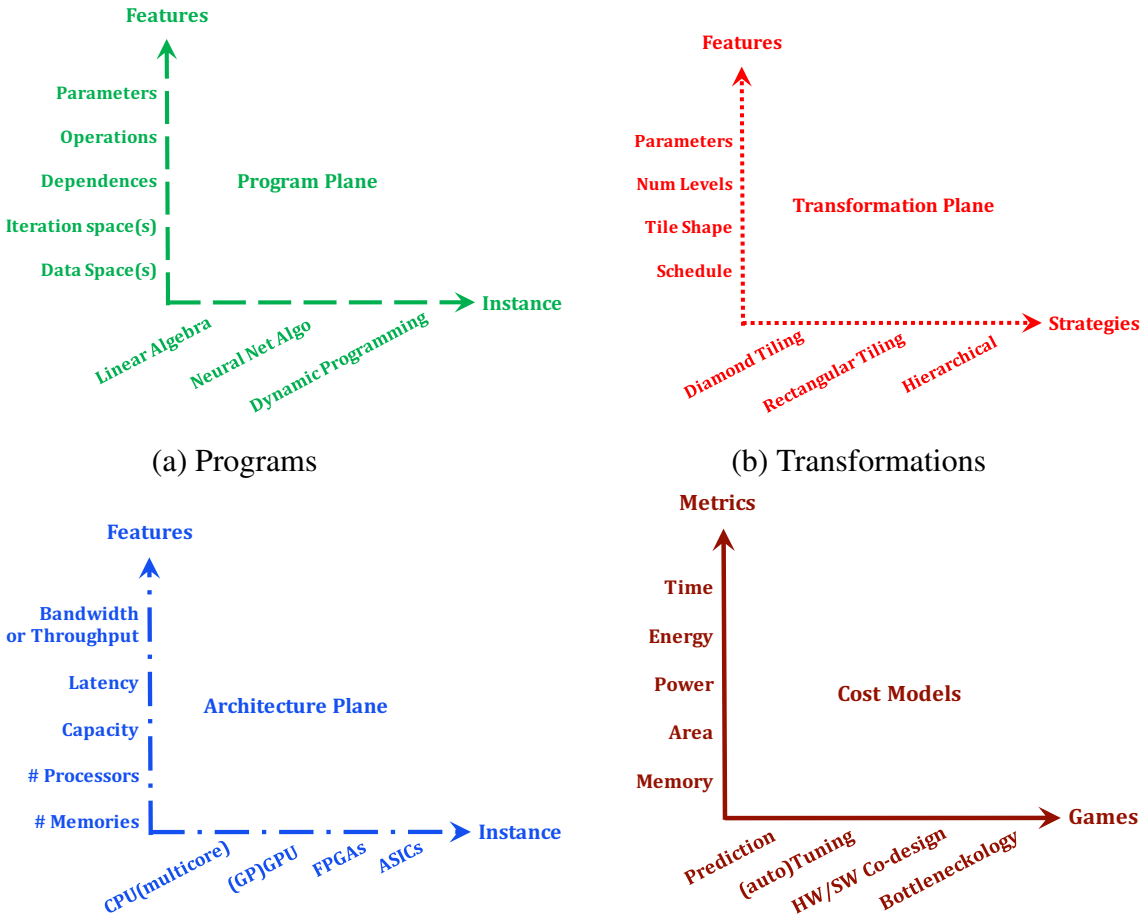
Consider how polyhedral compilation has recently evolved. Bondhugula et al. [53] proposed an extension of PLuTO for periodic stencil computations, and Bandishti et al. [54] developed another extension to allow concurrent starts. Since the objective functions in these strategies are all surrogates for the execution time, there is no way to compare across the strategies. Authors leave the choice of strategy to the user, via compiler flags. Recently it was shown (in [55], both quantitatively and empirically) that while concurrent start may be faster for iteration spaces with a certain aspect ratio of the program size parameters, the best performance for the same program with different aspect ratio is provided by the basic PLuTO algorithm.

As another example, Grosser et al. [19] proposed a novel combination of hexagonal and classic tiling for stencil programs on GPUs. They demonstrated—only empirically—performance gains compared to previous strategies, but did not quantitatively explore cases where HHC was better.

As a consequence, polyhedral compilation remains difficult. Every time a new strategy is developed, the authors publish a paper, and empirically show that their results are better than previous ones. They usually do not provide a quantitative, analytical comparison, thereby preventing a better, collective understanding of how to solve the bigger, global problem. Our intention is not to criticize the field: the problems today are difficult enough that significant effort is needed for even developing such "point solutions." Our approach is a step towards addressing these limitations.

2.2 Design Landscape

To place our work into context, to precisely formulate the problems we address, and to describe the approach we take to solve them, we show the design landscape of domain-specific optimization problems. It has six dimensions, organized into three planes (Fig. 2.1 (a), (b) and (c)). Each plane has two axes: *instances*, and *features*. The feature axis may be hierarchical, and parameterized.



(a) Programs

(b) Transformations

(c) Architectures

(d) Cost Models

Figure 2.1: The Domain-Specific Design Landscape. We tackle a narrow domain of programs (a) each one described by a small number of features. Similarly the mappings/transformations (b) we can apply are drawn form a small set, each one parameterized by a set of features. Architectures (c) are also drawn from a similarly small set, and are parameterized by their features. In each plane, the features may be hierarchical. The cost metrics (d) and the games that we can play.

The ***program plane*** (Fig. 2.1(a)) consists of instances of ***dense stencil computations***, such as ***Convolutional Neural Net*** (a machine learning kernel), ***Heat-3D*** (a stencil computation from computational science), ***Clustal-3D*** (a dynamic programming kernel from bioinformatics). Because of domain specificity, each program is *compactly* described with a small set of *features*, such as: (i) a set of *iteration spaces* (ii) a set of *data spaces*, (iii) a set of *dependences*, (iv) a set of *computational operators* (e.g., loop bodies), and (iv) one or more *size parameters*.

The ***transformation plane*** (Fig. 2.1(b)) defines the space of compiler transformations that can be applied to the program. Domain specificity again allows us to consider only a few instances, e.g., time skewing [56, 57], diamond tiling [54], diamond prisms [37, 58], or hybrid hexagonal-classic (HHC) tiling [59, 60, 19]. Transformation strategies are (potentially) hierarchical, and each level of the hierarchy represents a ***partitioning***.[1] They are also specified by a set of *features*, each of which is a mathematical function: (i) *tile shape*, specified by the so called "tiling hyperplanes," (ii) *tile schedule*, (iii) *processor mapping* specifying which (virtual/physical) processor in the hardware hierarchy will execute a tile, and (iv) *memory allocation* specifying where its inputs and outputs are stored. Note that the schedule usually also has components to specify when tile inputs are read and when tile outputs are written. The transformation plane features are also parameterized: mapping function coefficients, tile sizes, etc., are viewed as parameters.

The ***architecture plane*** (Fig. 2.1(c)) captures the accelerator hardware. Examples of architecture instances are ASICs, FPGAs/Reconfigurable Logic, and Instruction Set Architectures (such as GPGPUs).[2] Hardware features include (i) the number of processors at that granularity, (ii) the memories, and (iii) the interconnects. Parameters of each such feature specify the performance of that feature: e.g., speed in terms of throughput and latency, capacity, etc. Also included are specific access/scheduling constraints such as the need for gang scheduling in warps, the constraints on coalescing and bank conflicts, etc.

---

[1] *Partitioning* denotes a generalization of a crucial transformation called ***tiling*** to also include ***multiple passes***.

[2] In our taxonomy, an accelerator that is programmable in the conventional, von Neumann sense is an ISA, and accelerators where the "programming" is at the circuit level are FPGAs or ASICs.

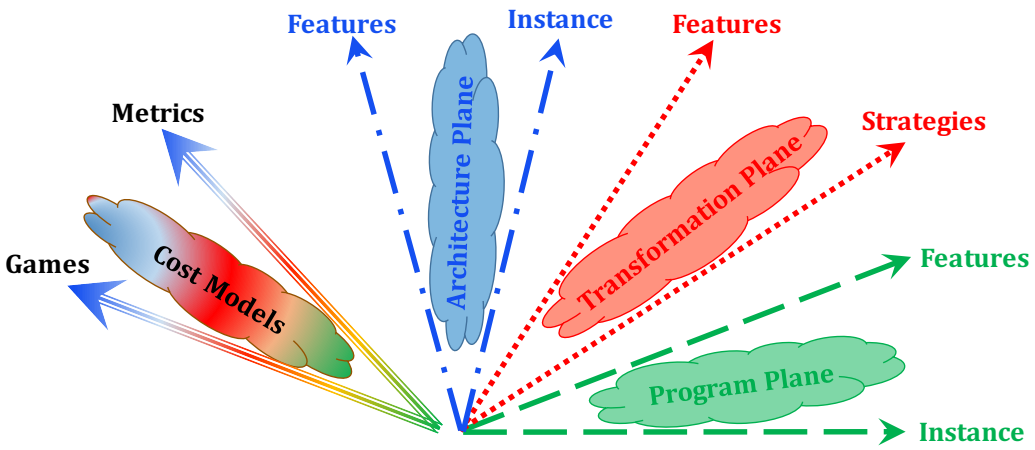


Figure 2.2: The main novelty of our work! The Domain-Specific Design Landscape that we must navigate is thus the cross-product of the four planes in Fig. 2.1.

We combine this six-dimensional design space with the ***cost modeling plane*** (Fig. 2.1(d)) which gives us a high level picture of various cost metrics and the games that we can play with these cost models. ***This unified view of the landscape is the main novelty of our work***. Fig. 2.2 shows the eight dimensional navigation space to be explored. A model can be used for performance ***prediction*** as well as performance improvement like ***(auto)tuning***, ***bottleneckology*** (bottleneck analysis), etc. Multiple models can contribute to ***multi-criteria optimization*** (eg. simultaneously optimize for both, time and energy) as well as ***HW/SW Co-design***.

2.3 Approach

We now describe our overall approach and how it can lead to the benefits mentioned earlier (i.e., *automatic optimal mappings*, *future proofing* and *codesign*). First, we develop analytical models for execution time and energy for a given program and a transformation strategy on a fixed architecture. We also develop silicon area models for GPU architectures and show its use in chip area prediction. Second, we show how these models can be used for performance optimization. And finally, we show how to formulate mathematical optimization problems using such cost models to solve the problem of software-hardware codesign. We show our initial results to justify our claims, and identify remaining challenges in later chapters.

2.3.1 Models and Validation

Execution Time Model and Prediction of GPGPU Stencils (Appendix A)

We develop an execution time model that predicts execution time of transformed stencil codes. 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 optimized 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.

Energy Model and Prediction of GPGPU Stencils (Appendix B)

Like the analytical execution time model, we develop a methodology for modeling the energy efficiency of tiled nested-loop codes running on a graphics processing unit (GPU) and use it for prediction of energy consumption. We assume that a highly optimized and parameterized version of a tiled nested-loop code—either written by an expert programmer or automatically produced by a polyhedral compilation tool—is given to us as an input. We then model the energy consumption as an analytical function of a set of parameters characterizing the software and the GPU hardware. Most previous attempts at GPU energy modeling were based on low-level machine models that were then used to model whole programs through simulations, or were analytical models that required low level details. In contrast, our approach develops analytical models based on (i) machine and architecture parameters, (ii) program size parameters as found in the polyhedral model, and (iii) tiling parameters. Our model therefore allows prediction of the energy consumption with respect to a set of parameters of interest. We illustrate the framework on three nested-loop codes: Smith-Waterman, and one-dimensional and two-dimensional Jacobi stencils, and analyze the accuracy of the resulting models. With optimal choice of model parameters the RMS error is less than 4%. Two factors allow us to attain this high accuracy. The first is domain-specificity: we focus only on tilable nested-loop codes. The second is that we decouple the energy model from a model of the execution time, a known hard problem.

Memory Access Models (Appendix A, B, and D)

We model memory accesses(between global and shared memories of a GPU) for GPGPU stencil computations and tiled codes. We use these memory access models to quantify the contribution of data transfers in execution time as well as energy consumption. For energy modeling we go a step further and develop cost models to reflect the data transfers between global memory and registers. These analytical models are based on (i) architecture parameters, (ii) program size parameters, and (iii) tile sizes. The tile sizes are constrained by the memory capacities.

We also develop data movement models for tiled matrix multiplication codes on GPUs. These cost models predict data transfers for different cases such as the one where an input matrix is read only once from global memory, or the one where output matrix is written only once to the global memory, etc. These models can be found in the literature of communication avoiding algorithms or lower bound algorithms. We develop data movement models for tiled and fused two matrix multiplications codes on GPUs. These data movement models allow us to minimize off-chip data movement for matrix chain products.

Area Model and Chip Area Prediction of GPUs (Appendix C)

We also develop an analytic model for the total silicon area of a GPU accelerator. We faced some difficulties in deriving an acceptable analytical model, as silicon data had to be reverse engineered from extremely limited public domain resources. As a general observation, within each GPU family, there is little diversity in the parameter configurations. For the Maxwell family of GPUs, the GTX980 and Titan X chips were chosen as two sufficiently distinct points to calibrate our analytical models. The calibration itself was performed by evaluating die photomicrographs, publicly available information about the nVidia GTX 980 (Maxwell series) GPU, and other generally accepted memory architecture models. The model validation was done by comparing the predictions with known data on the Maxwell series Titan X GPU. We found the model prediction to be accurate to within, 2%, though this number is not significant[3].

[3]Although a many configurations of any family of GPUs are spaced out, they come from binning only a small number of distinct dies. We ended up calibrating our model on one die and validating it on only another one.

Next, we develop mathematical objective functions to illustrate the use of these models in performance optimization and later we will show the same for software-hardware codesign.

2.3.2 Compilation and its optimization subspaces

To address the limitations of current domain-specific compilation noted in Section 2.1.2, we now describe our approach to systematically exploring well defined regions of the design landscape using exact (not surrogate) objective functions.

1. **(Auto) Tuning:** Let us consider a three-dimensional subspace in our landscape, of *instances* of programs $\times$ transformations/strategies $\times$ *execution time*. At each point in this three-dimensional space, we may define a mathematical optimization problem which has an *objective function*, and a *feasible space*. Both involve analytical functions of parameters from all three feature dimensions, defined by vectors: $\vec{P}$ for a program, $\vec{S}$ for a strategy, and $\vec{A}$ for an architecture. The objective function for execution time is $M_T(\vec{P}, \vec{S}, \vec{A})$. Similarly, we have constraints defining the feasible space of the optimization problem, $\mathcal{F}_T(\vec{P}, \vec{S}, \vec{A})$. The mathematical optimization problems can be formulated for various performance metrics as follows:

   (a) For execution time, minimize $M_T(\vec{P}, \vec{S}, \vec{A})$ subject to $\mathcal{F}_T(\vec{P}, \vec{S}, \vec{A})$,

   (b) For energy, minimize $M_E(\vec{P}, \vec{S}, \vec{A})$ subject to $\mathcal{F}_E(\vec{P}, \vec{S}, \vec{A})$,

   (c) For power, minimize $M_P(\vec{P}, \vec{S}, \vec{A})$ subject to $\mathcal{F}_P(\vec{P}, \vec{S}, \vec{A})$, etc.

   Each problem instance has an objective function that represents (is not just a surrogate for) the metric which we seek to optimize: execution time(M_T), power(M_P), energy(M_E), etc. It is a function of all the parameters of this three-dimensional point. Other parameters, e.g., the number of processors, the memory capacity, etc., may define a feasible space where this function is valid.

   Our approach is based on the hypothesis that domain-specificity of both the programs and the architecture allows us to develop such functions. Note that the objective function cannot

be a surrogate, it must be the actual cost metric of interest. Under this hypothesis, our entire strategy can be summarized as *collective solution of multiple optimization problems with common objective function(s)*. For GPGPU stencil codes, we explore two such common objective functions, $\mathcal{M}_T$ for execution time, and $\mathcal{M}_E$ for energy (expressed as GOPs/sec or GOPs/joule). For matrix multiplication and matrix product chains, we explore $\mathcal{M}_T$ for execution time, and $\mathcal{M}_D$ for data movement.

2. **(Auto) Super-Tuning:** The next step will be to extend the optimization across multiple strategies, say S_1 and S_2. Given two separate optimizations formulated as follows:

   (a) Minimize $M_{T_1}(\vec{P}, \vec{S_1}, \vec{A})$ subject to $\mathcal{F}_{T_1}(\vec{P}, \vec{S_1}, \vec{A})$, and

   (b) Minimize $M_{T_2}(\vec{P}, \vec{S_2}, \vec{A})$ subject to $\mathcal{F}_{T_2}(\vec{P}, \vec{S_2}, \vec{A})$

   We can formulate the problem of optimizing across strategies in two ways: (i) Take the minimum of the two optimizations min(minimize $M_{T_1}(\vec{P}, \vec{S_1}, \vec{A})$, minimize $M_{T_2}(\vec{P}, \vec{S_2}, \vec{A})$), or (ii) solve separate optimization problems, depending on the intersections and differences of the feasible spaces of each one.

   (a) $\min(M_{T_1}(\vec{P}, \vec{S_1}, \vec{A}), M_{T_2}(\vec{P}, \vec{S_2}, \vec{A}))$, subject to $\mathcal{F}_{T_1}(\vec{P}, \vec{S_1}, \vec{A}) \cap \mathcal{F}_{T_2}(\vec{P}, \vec{S_2}, \vec{A})$,

   (b) Minimize $M_{T_1}(\vec{P}, \vec{S_1}, \vec{A})$, subject to $\mathcal{F}_{T_1}(\vec{P}, \vec{S_1}, \vec{A}) \cap \sim \mathcal{F}_{T_2}(\vec{P}, \vec{S_2}, \vec{A})$, and

   (c) Minimize $M_{T_2}(\vec{P}, \vec{S_2}, \vec{A})$, subject to $\sim \mathcal{F}_{T_1}(\vec{P}, \vec{S_1}, \vec{A}) \cap \mathcal{F}_{T_2}(\vec{P}, \vec{S_2}, \vec{A})$

   This can be extended to a set of strategies, $\mathcal{S} = \{S_i\}$. Although the second option is not very scalable—the number of sub-problems grows exponentially with the number of strategies—it is reasonable for a small number of strategies, e.g., it would let us automatically choose between time skewing, diamond tiling, diamond prisms, and HHC.

3. **Multi-Metric (Auto) Tuning:** The above optimizations account for only one performance metric, which leads to a single objective function for the optimization. One might want to optimize for more than one metric. Let us consider a multi-metric optimization such as the

energy-delay product. The optimization problem can be formulated as

$$\text{Minimize } (M_T(\vec{P},\vec{S},\vec{A}) * M_E(\vec{P},\vec{S},\vec{A})), \text{ subject to } \mathcal{F}_T(\vec{P},\vec{S},\vec{A}) \cap \mathcal{F}_E(\vec{P},\vec{S},\vec{A})$$

Note, the feasible space consists of the intersection of the feasible space of time and energy. The program parameters (eg. problem sizes) and the features (eg. tile sizes) of the selected strategy (eg. Diamond tiling) are the parameters to the multi-metric objective function.

4. **Multi-Metric (Auto) Super-Tuning:** The above multi-metric objective function can be extended to multiple strategies, say S_1 and S_2. Consider, two optimization functions

   (a) Minimize $(M_{T_1}(\vec{P},\vec{S_1},\vec{A}) * M_{E_1}(\vec{P},\vec{S_1},\vec{A}))$, subject to $\mathcal{F}_{T_1}(\vec{P},\vec{S_1},\vec{A}) \cap \mathcal{F}_{E_1}(\vec{P},\vec{S_1},\vec{A})$, and

   (b) Minimize $(M_{T_2}(\vec{P},\vec{S_2},\vec{A}) * M_{E_2}(\vec{P},\vec{S_2},\vec{A}))$, subject to $\mathcal{F}_{T_2}(\vec{P},\vec{S_2},\vec{A}) \cap \mathcal{F}_{E_2}(\vec{P},\vec{S_2},\vec{A})$

   As in *(Auto) Super-Tuning*, we can formulate the problem of optimizing across strategies in two ways: (i) Take the minimum of the two optimizations, or (ii) solve separate optimization problems, depending on the intersections and differences of the feasible spaces of each one. The methods can be extended to a set of strategies, $\mathcal{S} = \{S_i\}$.

   For matrix chain products, we formulate a multi-metric optimization problem where we minimize the total op-count(i.e., execution time $\mathcal{M}_T$) as well as off-chip communication (i.e., data movement $\mathcal{M}_D$). First, we use two transformation strategies: single matrix multiplication and two matrix multiplications. Single matrix multiplication(*1MM*) comprises of finding an optimal tiling schedule(that minimizes data movement) for data transfers between global and shared memories. Two matrix multiplication(*2MM*) objective is to find an optimal loop fusion transformation that minimizes the data movement. Second, we use the classic matrix chain ordering algorithm [61] to find the minimum op-count tree($\mathcal{M}_T$). We formulate a combined optimization problem for *1MM*($\mathcal{M}_{D_1}$) and *2MM*($\mathcal{M}_{D_2}$) to minimize the total off-chip data transfers. Details in Appendix D.

Thus, our approach would allow us to deliver on the promise of *automatic* and *provably optimal* compilation, for any point in the program $\times$ transformations/strategies plane for a given performance metric.

Polyhedral compilation revisited: As noted before, current polyhedral compilers target a fairly broad class of programs, and make choices like tiling hyperplanes and shapes, and (inter and intra) tile schedules. They do this by this using classic scheduling algorithms [51, 17, 18] that use (integer) linear programming using surrogate objective functions. Tile sizes are chosen subsequently via auto-tuning.

2.3.3 Codesign and its optimization subspaces

Codesign—the simultaneous design of hardware and software—has two common interpretations. *System codesign* is the problem of simultaneously designing hardware, runtime system, compilers, and programming environments of entire computing systems, typically in the context of large-scale, high-performance computing (HPC) systems and supercomputers. *Application codesign*, also called *hardware-software codesign*, is the problem of systematically and simultaneously designing a dedicated hardware platform and the software to execute a single application (program). The proposed approach is applicable in both contexts.

Application Codesign and its optimization subspaces

1. **Application Codesign:** For *Application Codesign*, we fix a program, a strategy as well as a micro-architecture (technology node). In addition to the program parameters and features of the selected strategy, we now have $\vec{A}$, the architecture parameters, as unknowns. The architecture parameters can be number of processors, number of memories, size of memories, etc. Considering, execution time as the performance metric, the *four-dimensional* optimization problem can be formulated as

$$\text{Minimize } M_T(\vec{P}, \vec{S}, \vec{A}), \text{ subject to } \mathcal{F}_T(\vec{P}, \vec{S}, \vec{A})$$

The feasible space $\mathcal{F}_T$ is a combination of (i) all possible program parameters, (ii) all possible features of the selected strategy, (iii) all possible architecture parameters, and (iv) time as the performance optimizing criteria. Similar objective functions can be formulated for different performance metrics.

2. **Super Application Codesign:** Classic application codesign techniques consider optimization for one strategy. Domain specificity allows us to formulate the mathematical optimization across a set of strategies, say $\mathcal{S} = \{S_i\}$, as follows

$$\text{Minimize } M_T(\vec{P}, \vec{\mathcal{S}}, \vec{A}), \text{ subject to } \mathcal{F}_T(\vec{P}, \vec{\mathcal{S}}, \vec{A})$$

Note that the feasible space for each strategy will vary and not all architectures may be feasible for every strategy. Such optimization allows for exploration of a larger feasible space.

3. **Multi-Metric Application Codesign:** The architecture parameters may include configuration options that trade off energy for performance (as in execution time). One might, therefore, want to optimize across multiple performance metrics. A multi-metric SW-HW Codesign optimization can be formulated as

$$\text{Minimize } (M_T(\vec{P}, \vec{S}, \vec{A}) * M_E(\vec{P}, \vec{S}, \vec{A})), \text{ subject to } \mathcal{F}_T(\vec{P}, \vec{S}, \vec{A}) \cap \mathcal{F}_E(\vec{P}, \vec{S}, \vec{A})$$

considering *energy-delay product* as the performance metric. Note that this *four-dimensional* feasible space consists of program parameters, strategy features and architecture parameters as unknowns.

4. **Multi-Metric Super Application Codesign:** Again, the multi-metric application codesign objective function can be extended to multiple strategies, say $\mathcal{S} = \{S_i\}$. Considering energy-delay product as the optimizing criteria, the objective function will be

$$\text{Minimize } (M_T(\vec{P}, \vec{\mathcal{S}}, \vec{A}) * M_E(\vec{P}, \vec{\mathcal{S}}, \vec{A})), \text{ subject to } \mathcal{F}_T(\vec{P}, \vec{\mathcal{S}}, \vec{A}) \cap \mathcal{F}_E(\vec{P}, \vec{\mathcal{S}}, \vec{A})$$

Note that the feasible space here consists of multiple areas in the design space based on the strategy and performance metric.

System Codesign and its optimization subspaces

1. **System Codesign:** Let us now consider a set of program instances $\mathcal{P} = \{P_i\}$, and recall that we have *common* objective functions for all of them. The optimization problem

$$\text{Minimize } \mathcal{M}(\vec{\mathcal{P}}, \vec{S}, \vec{A}) \text{ subject to } \mathcal{F}(\vec{\mathcal{P}}, \vec{S}, \vec{A})$$

seeks to optimize a common performance metric (we drop the subscript for the common metric). The parameters to this function are a set of program instances, features of the strategy and the architecture features.

2. **Super System Codesign:** Let us now consider the following optimization problem, which we call the ***generalized optimization problem***.

$$\text{Minimize } \mathcal{M}(\vec{\mathcal{P}}, \vec{\mathcal{S}}, \vec{A}) \text{ subject to } \mathcal{F}(\vec{\mathcal{P}}, \vec{\mathcal{S}}, \vec{A})$$

This problem seeks to optimize the common performance metric for the set of programs on the given architecture. We treat $\vec{A}$, not as parameters, but rather as unknowns, in the generalized optimization problem, $\text{argmin}_{\vec{A}}\mathcal{M}(\vec{\mathcal{P}}, \vec{\mathcal{S}}, \vec{A})$ gives us the optimal architecture for the set of program instances. Thus we simultaneously solve for architecture and compilation, thereby resolving the ***codesign problem***.

3. **Multi-Metric System Codesign:** The next step would be to extend our ***generalized optimization problem*** to a multi-metric optimization.

$$\text{Minimize } (M_T(\vec{\mathcal{P}}, \vec{S}, \vec{A}) * M_E(\vec{\mathcal{P}}, \vec{S}, \vec{A})), \text{ subject to } \mathcal{F}_T(\vec{\mathcal{P}}, \vec{S}, \vec{A}) \cap \mathcal{F}_E(\vec{\mathcal{P}}, \vec{S}, \vec{A})$$

This is particularly useful in large system designs, where the transformation strategy is fixed and more than one performance metric is critical for system design. Note, that we show multi-metric optimization for two cost metrics which can be extended to more than two cost metrics as needed.

4. **Multi-Metric Super System Codesign:** The above multi-metric system codesign can be further extended to consider multiple strategies as shown below

$$\text{Minimize } (M_T(\vec{\mathcal{P}}, \vec{\mathcal{S}}, \vec{A}) * M_E(\vec{\mathcal{P}}, \vec{\mathcal{S}}, \vec{A})), \text{ subject to } \mathcal{F}_T(\vec{\mathcal{P}}, \vec{\mathcal{S}}, \vec{A}) \cap \mathcal{F}_E(\vec{\mathcal{P}}, \vec{\mathcal{S}}, \vec{A})$$

This would be the ***ultimate goal for system codesign*** where we can optimize across all the possible program, transformation and architecture planes for multiple performance metrics.

2.3.4 Bottleneckology

Our approach to system design seems deceptively simple, however, it is a very hard problem. Exploiting the resources to their full capacity is one of the objectives while optimizing for performance. The bottleneck analysis becomes helpful in studying the performance sinks and design flaws. There are many ways of utilizing the cost models to perform bottleneck analysis. The cost models can be used to identify the resources that have been saturated and the ones that have slack. We refer to this slack and saturation of the resources as ***Bottleneckology***. Chapter 8 discusses this in three ways: (i) investigate codesign-tradeoffs, (ii) perform overhead analysis, and (iii) explore the effect of hyperthreading.

2.4 Contributions

The following are the primary contributions of this dissertation:

1. An approach to automatically design an efficient system by navigating the design landscape
2. Analytical Cost Models
   (a) Stencil Computations

i. Execution Time (included in [22])

ii. Energy (included in [23])

iii. Memory Access (included in [22, 23])

(b) Matrix Multiplication

i. Data Movement (included in Appendix D)

(c) Matrix Chain Products

i. Data Movement (included in Appendix D)

(d) Chip Area for GPUs (included in [21])

3. Single Metric Tuning

(a) Tune for Speed (included in [22])

(b) Optimize for Energy (included in [23])

(c) Minimize Off-chip Memory Accesses (included in Appendix D)

i. Minimize Data Movement for a single Matrix Multiplication

ii. Minimize Data Movement for Two Matrix Multiplications

4. Multi-Metric Tuning (included in Appendix D)

(a) Minimize Op Count and off-chip memory accesses for Matrix Chain Products

5. Application and System Codesign (included in [21])

Chapter 3

Related Work

This work draws upon prior work in the following distinct areas.

3.1 Stencil Computations and Code Generation

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 [62, 63, 54]. One successful technique is called *time tiling* [64, 65, 66, 67, 68, 69, 58, 54], an advanced form of loop tiling [70, 64, 71]. 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.

For iterative stencils a large set of optimizing code generation strategies have been proposed. Pochoir [39] 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 [72], Hybrid-hexagonal tiling [19], and Overtile [73] 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 the hexagonal tiles to avoid the need for 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 [74], 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, who use data-layout transformations to avoid redundant non-aligned vector loads on CPU platforms.

3.2 Performance Modeling

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 graphs, which are directed acyclic graphs (DAGs) of non-iterated stencil kernels, various DSLs compilers have been proposed. Halide [75] and Stella [76] 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 [77] 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. With MODESTO [78] an analytical performance model has been proposed that allows to model multiple cache levels and fusion strategies for both GPUs and CPUs as they arise in the context of Stella.

For stencil GPU code generation strategies that use redundant computations in combination with ghost zones, an analytical performance model has been proposed [79] that allows to automatically derive "optimal" code generation parameters. Yotov et. al [80] 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 [81], but at this point no stencil computations have been considered. Shirako et al. [82] 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 [83] present a precise GPU performance model which shares many of the GPU parameters we use. It is highly accurate, low level, and requires analyzing the PTX assembly code. For stencil GPU code generation strategies that use redundant computations in combination with ghost zones an analytical performance model has been proposed [79] that allows to automatically derive "optimal" code generation parameters. Patus [84] provides an auto-tuning 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.

Renganarayana et al. [85] identifies positivity as a common property shared by the parameters used by tile size selection methods and show that the property can be used to derive efficient and scalable tile size selection frameworks.

3.3 Chip Reverse Engineering and Area Modeling

Chip area modeling can be formally considered a branch of semiconductor reverse engineering, which is a well researched subject area. Torrence et. al. [86] gives an overview of the various techniques used for chip reverse engineering. The packaged chips are usually decapped and the wafer die within is photographed layer by layer. The layers are exposed in the reverse order after physical or chemical exfoliation. *Degate* [87], for example, is a well known open source software that can help in analyzing die photographs layer by layer. The reverse engineering process can be coarse-grained to identify just the functional macro-blocks. Sometimes, the process can be very fine-grained, in order to identify standard-cell interconnections, and hence, actual logic-gate netlists. Degate is often used in association with catalogs of known standard cell gate layouts, such as those compiled by *Silicon Zoo* [88]. Courbon et. al. [89] provides a case study of how a modern flash memory chip can be reverse engineered using targeted scanning electron microscope imagery. For chip area modeling, one is only interested in the relatively easier task of demarcating the interesting functional blocks within the die.

3.4 Energy Modeling

GPU power/energy model is a very active area: a recent survey article on the topic [90] cites almost 150 references. We only discuss the relevant work here. The model we present complements Mittal and Vetter [90] by enabling us to find the optimal parameters (i.e., tile sizes) for the energy efficient execution of stencil like programs. Hong and Kim [91] present a GPU power model to predict the number of optimal GPU cores to achieve the peak memory bandwidth for a kernel. An analytical model is used to predict the execution time [92] which has enabled prediction of the power consumption statically. However, they have predicted the minimum number of cores required for a program to achieve the peak memory bandwidth of GPU. While this approach may work for memory bandwidth bound programs, it is unlikely to produce better results for compute-bound programs like tiled stencil computations. Our model is much simpler, because our model does not depend on warp and thread level parameters and number of PTX instructions.

Nagasaka et al [93] model GPU power of kernels using performance counters. Lim et al. [94], GPUWattch [95] and GPUSimPow [96] are simulation based power models. McPAT [97] is the basis for Lim et al [94] and GPUWattch [95] uses GPGPUSim [98] to simulate execution time. Simulation and performance counters-based models require execution (or simulation) of the program to predict the power consumption. Therefore, these models are not feasible solutions when it requires to take decisions at compile time to determine optimal software parameters. We do need to run some micro-benchmarks to find the energy parameters that our model use. In contrast, we run our micro-benchmarks only to determine parameters of a GPU architecture, while the power consumption can be predicted for a given program statically without running the program.

There are studies [99, 100] focused on reducing the energy for both CPU and GPU by balancing the load among CPU and GPU. Our study is only focused on modeling the energy consumption of GPUs. There are studies [99, 100] focused on reducing the energy for both CPU and GPU. The models are used to determine how to balance the load among CPU and GPU, so that it reduces the overall energy consumption. Our study is only focused on modeling the energy consumption of GPUs.

3.5 Codesign

Application codesign is a well established discipline and has seen active research for well over two decades [101, 102, 103, 104, 105]. The essential idea is to start with a program (or a program representation, say in the form of a CFDG—Control Data Flow Graph) and then map it to an abstract hardware description, often represented as a graph of operators and storage elements. The challenge that makes codesign significantly harder than compilation is that the hardware is not fixed, but is also to be synthesized. Most systems involve a search over a design space of feasible solutions, and various techniques are used to solve this optimization problem: tabu search and simulated annealing [106, 107], integer linear programming [108].

There is some recent work on accurately modeling the design space, especially for regular, or *affine control* programs [24, 25, 26]. However, all current approaches solve the optimization problem for a single program at a time. To the best of our knowledge, no one has previously considered the *generalized application codesign* problem, seeking a solution for a *suite of programs*.

There are multiple publications on codesign related to exascale computing, but they focus on different aspects. For instance, Dosanji et al. [109] focus on methodological aspects of exploring the design space, including architectural testbeds, choice of mini-applications to represent applications codes, and tools. The ExaSAT framework [110] was developed to automatically extract parameterized performance models from source code using compiler analysis techniques. Performance analysis techniques and tools targeting exascale and codesign are discussed in [7].

Kuck et al. [111, 112] analyze and model program hot-spots. They develop computational capacity models and propose an approach for the HW/SW codesign of computer systems. The hardware/software measurements of computational capacity (based on bandwidth usage) and power consumption (based on hardware counters) are used to find optimal solutions to various codesign problems and to evaluate codesign trade-offs. Their models are theoretical and are illustrated by numerical examples. They do not validate their models using real hardware.

Chapter 4

Models and Validation

Model predictions are used for estimating execution time, energy consumption, power consumption, etc. of a program. Cost metrics either appear in the objective function as a factor to be optimized or in the constraints. We will illustrate the use of a few of the many metrics - (i) execution time models, and (ii) energy models as cost in the optimizing functions; and (iii) memory access models, and (iv) silicon area models as the constraints to the objective function.

4.1 Execution Time Model for GPGPU Stencils

We develop an execution time models for GPGPU stencils [22] (Appendix A) that guides the optimal choice of compiler parameters(tile sizes). 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 optimized 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. We show the following.

- 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. The model is an analytic function of
  - program, machine, and compiler parameters that are easily available statically, and
  - one stencil-specific parameter that is obtained by running a handful of *micro-benchmarks*.

  It is deliberately *optimistic* and also ignores the effect of some parameters.
- 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 than 60,000 programs for

- two modern **target platforms** (NVIDIA GTX 980, and Titan X), and
- four 2D **stencil codes** (Jacobi2D, Heat2D, Laplacian2D, and Gradient2D) and two 3D stencils (Heat3D and Laplacian3D)
- over a range of ten input/problem **sizes**, and
- a wide range of tile sizes and thread counts (the HHC compiler inputs) for each *platform-stencil-size* combination.

As we expected, the root-mean-square error (RMSE) over the **entire** data set was "disappointingly" over 100%. However, when we restricted ourselves to the data points that have an execution time within 20% of the best value for that particular platform-stencil-size combination, the RMSE dropped to less than 10%[4], which we consider very good.

Our overall methodology is applicable, with simple extensions, to more general programs, e.g., those that fit the polyhedral model. But for achieving high GPU utilization, we need efficient GPU codes to start with, which are very hard and time consuming to produce manually, especially in higher dimensions. The highly optimized HHC-generated codes we are using for testing and validation have a few thousand lines of CUDA code each and we generated tens of thousands such codes in our experimental analysis. So our methodology is not limited to the HHC compiler (in fact we have applied it successfully to manually generated 1D stencil codes), but the use of HHC (or similar compiler) was necessary to produce for our experiments a high number of GPU codes that are also very efficient.

4.2 Energy Model for Tiled Nested-Loop Codes

Energy efficiency has been recognized as one of the biggest challenges in the roadmap to higher performance (exascale) systems for a number of reasons including cost, reliability, energy conservation, and environmental impact. The most powerful computers today consume megawatts

[4]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.

of power, enough to power small towns, and at cost of millions per year. And those estimations do not include the cost of cooling, which might be almost as high as the cost of computing itself [113]. In addition, the cost of building a power provisioning facility ranges at $10-22 per deployed IT watt [114] and every $10\,^{\circ}\mathrm{C}$ temperature-increase results in a doubling of the system failure rate, thereby reducing the reliability of HPC systems [115]. Designing accurate models for energy efficiency can help better predict the power and energy requirements of an application and aid developers optimize the parameters of their codes for better energy efficiency on HPC systems.

The goal of our work [23] (Appendix B) is to introduce a new approach for modeling the energy cost as an analytical function of tunable software parameters in a way that is both simple and accurate. Having such a model will allow the energy efficiency to be optimized with respect to (a subset of) the tunable parameters by solving the corresponding analytical optimization problem.

We target with our modeling approach tiled nested-loop code segments, which are the most compute-intensive portions of many application codes and which also allow a high degree of parallelism. In order to be more specific, we focus in our analysis on a subclass of the tiled nested-loop codes called *dense stencils*, which occur frequently in the numerical solution of PDEs and in many other contexts such as high-end graphics, signal and image processing, numerical simulation, scientific computing, and bioinformatics. We chose stencils for our case studies since that would allow us to model the entire class in a hierarchical way with a single generic model representing the whole class, while model parameters that are stencil-dependent have to be separately specified for each stencil of interest to complete its model. (However, the approach is applicable to any other class of nested-loop codes that allows tiling.) We completely develop and validate the detailed models (including the stencil-dependent parameters) of three specific stencils. Models for other stencils can be developed in a similar way with relatively small amount of extra work.

In order to efficiently optimize stencils on accelerators, we aim to represent the amount of energy consumed as an analytic function of the software parameters. We assume that the input codes have been analyzed and optimized with respect to parallelism and data-access efficiency by appropriate skewing and tiling transformations, say by a polyhedral code generator.

Our specific contributions are as follows.

- Our energy model predicts energy efficiency by analyzing source code only, unlike other approaches [93, 95] that rely on parameters computed by running benchmarks for each individual code. We do use micro-benchmarks, but they are used to characterize hardware, rather than codes.

- We are not aware of any previous work combining the polyhedral method with energy modeling. Our approach allows optimization of codes that are already very efficient having been significantly improved by applying the polyhedral method and by using advanced tiling strategies such as hexagonal and hybrid tilings [19].

- Our model is very accurate (one version with $\mathrm{RMS\,error} \leq 17.14\%$ and another with $\mathrm{RMS\,error} \leq 4\%$), with similar or higher precision than alternative existing models, e.g., GPUSimPow [116], which are simulation based.

4.3 Memory Access Model for GPGPU Stencils

We develop Memory Access models [22, 23] (Appendix A and B) for GPGPU stencils and use them for execution time models and energy models. The memory models appear in two specific contexts. Firstly, the total number of memory accesses made by a tile is used to model the data transfer time taken by a tile, similarly, the data movement requirement of a wavefront is modeled using the equations to calculate the data transfer time for wavefronts. Similarly, for the energy models we use memory access equations to determine the amount of transfer required and combine it with the energy consumption per data transfer to calculate the total energy consumption of a tile.

Second, the memory footprint of a tile are used as constraints to the formulated objective function for optimization. The memory requirements of a tile are not to exceed the shared memory capacity of a GPU. This in turn constrains the tile sizes and the feasible space.

These memory models are then used for codesign optimization [21]. Again for software-hardware codesign, the memory models appear both in the objective function as a part of time

model equations (i.e., to calculate the data transfer time) as well as the constraints where memory capacity defines the feasible space.

4.4 Silicon Area Model for GPUs

We develop an analytic model for the total silicon area of a GPU accelerator [21](Appendix C). We faced some difficulties in deriving an acceptable analytical model, as silicon data had to be reverse engineered from extremely limited public domain resources. As a general observation, within each GPU family, there is little diversity in the parameter configurations. For the Maxwell family of GPUs, the GTX980 and Titan X chips were chosen as two sufficiently distinct points to calibrate our analytical models. The calibration itself was performed by evaluating die photomicrographs, publicly available information about the nVidia GTX-980 (Maxwell series) GPU, and other generally accepted memory architecture models. The model validation was done by comparing the predictions with known data on the Maxwell series Titan X GPU. We found the model prediction to be accurate to within, 2%, though this number is not significant.[5]

4.5 Data Movement Model for Matrix Multiplication

We develop off-chip memory access models for hyper-rectangular tiled single matrix multiplication codes on GPUs (Appendix D). These models take into account the matrix sizes, tile sizes, and on-chip memory (shared memory) capacity of the architecture. We divide the computation space into hyper-rectangular tiles and map tiles to threadblocks such that partial results are not written to off-chip memory. For this mapping, we develop memory access models that compute the total number of off-chip memory accesses.

We also develop memory access models for hyper-rectangular tiled two matrix multiplications with 3 matrices to multiply. Conventionally, two matrices are multiplied to produce intermediate result matrix which is written to the off-chip memory. This intermediate result is then multiplied

[5]Although a many configurations of any family of GPUs are spaced out, they come from binning only a small number of distinct dies. We ended up calibrating our model on one die and validating it on only another one.

with the third matrix to produce the final output. We apply loop permutation and loop fusion techniques to avoid writing of the intermediate result to the off-chip memory. We develop memory access models for such fused and tiled loop codes. Section D.2 discusses this in detail.

In the next chapter, we show how to use these cost models for tile size selection that leads to improved performance.

Chapter 5

Single-Metric Tuning

An important element of compilation 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 parameters. This enables the compilation 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 manners, especially since the latency of operations is unpredictable because of the deep memory hierarchy. It is widely believed that because of this, autotuning is unavoidable in order to obtain good performance.

Our work challenges this. In particular, we make the case that domain specificity can have a third important benefit: it enables us to develop a good analytical model to predict the performance of specific types of codes on specific types of target architectures. We can then use the model to optimally choose the mapping parameters (notably tile sizes).

In order to address the challenges of exascale computing, many experts believe that a software-hardware co-design approach—where the software and the corresponding hardware are jointly co-developed and co-optimized—will be a "critical necessity" [117]. Since the architectures of exascale systems are in the flux, it is important to develop rigorous methods to map high level specifications of computations to diverse target architectures, ranging from multi-core CPUs, many-core GPUs, and accelerators over heterogeneous nodes of such CPU-GPU combinations to large distributed systems of many such nodes. In the overwhelming majority of cases, the mismatch between data communication patterns and hardware architecture prevents the efficient exploitation of all available computing resources and peak performance is almost impossible to achieve. Worse still, it is often not clear to the user when the point of diminishing returns is reached.

We address a key step of the optimization, namely mapping the software representation onto the hardware, and choosing the mapping parameters to optimize an objective function representing the performance, i.e., the execution time. In its full generality, the optimal mapping problem is

a discrete non-linear optimization problem, known to be NP-hard [118] and hence very difficult to solve efficiently. We therefore use a number of simplifying assumptions, as is common in the literature. A number of parameters can be specified as inputs to a compiler, e.g., the tile sizes. These parameters have a tremendous influence on the performance of the code. The problem we tackle here is how to select these parameters optimally.

5.1 Tune for Speed

To test the predictive abilities of our execution time models, 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 [22] (Appendix A). 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 with maximum of 17%. Prajapati et al. [22] illustrates the predictive power of our execution time models in detail.

We have two messages. The main one is that, contrary to widespread belief, it is possible to construct good analytical cost functions to drive performance tuning for GPGPUs. This can significantly reduce the space that autotuners need to explore. The second message, is that it may be necessary to revisit some of the "conventional wisdom," when choosing tile size parameters. 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 would like to note that our techniques can be easily extended to other type of stencils.

5.2 Optimize for Energy

Our proposed optimization methods can also be applied to optimize for energy. Our energy models represents the energy consumption as an explicit *analytic* function of a set of software and hardware parameters describing the specifics of the implementation and the hardware.

In our experiments, we observe that energy optimization has almost always had no loss of speed, as expected in the folklore. Thus a user could use both optimizations rather than having to choose one or the other. Finally, we use our energy model to select the optimal tile size for energy efficiency and report the number of non-optimal tile size selections and hence the error in energy due to the selection of non-optimal tile sizes. Prajapati et al. [23] (Appendix B) describes our energy models, the optimization methods, the results and experimental validation in details.

5.3 Minimize Data Movement

We build closed form solutions for tile sizes that minimize the total number of off-chip data transfers for hyper-rectangular tiled single matrix multiplication codes on a given architecture. We also develop closed form solutions for fused and tiled two matrix multiplication codes. The total number of off-chip data transfers for these tile sizes is provably minimum. Appendix D provides complete description.

Chapter 6

Multi-Metric Tuning

One of the challenges of exascale computing is energy consumption. We formulate an optimization problem that seeks to minimize the total energy consumption of matrix chain products. Energy consumption is related to both, the execution time and off-chip memory accesses. We decompose the problem into a two step optimization approach. We first minimize the total number of computations for matrix chain products and then minimize data movement. There is a well-known dynamic programming algorithm that minimizes the total number of computations required for matrix chain products [61]. This algorithm produces an optimal op-count tree. We use our closed form solutions for fused and tiled two matrix multiplication codes to minimize the total number of off-chip data transfers for a given optimal op-count tree. We develop a dynamic programming solution to the problem of minimizing the data movement globally over the minimum op-count tree. This is a clear example of multi-metric tuning where we first tune for performance (i.e., minimize op-count/execution time) which minimizes the static energy consumption, and then minimize data movement that will optimize dynamic energy consumption for matrix chain products. The dynamic programming solution to minimize op-count as well as off-chip data movement is provably optimal. This is discussed in more details in Appendix D.

Chapter 7

Accelerator Codesign

"Design is not just what it looks like and feels like. Design is how it works." – Steve Jobs

Software-hardware codesign is one of the proposed enabling technologies for exascale computing and beyond [7]. Currently, hardware and software design are done largely separately. Hardware manufacturers design and produce a high-performance computing (HPC) system with great computing potential and deliver it to customers, who then try to adapt their application codes to run on the new system. But because of a typically occurring mismatch between hardware and software structure and parameters, such codes are often only able to run at a small fraction of the total performance the new hardware can reach. Hence, optimizing both the hardware and software parameters *simultaneously* during hardware design is considered as a promising way to achieve better hardware usage efficiency and thereby enabling leadership-class HPC availability at a more manageable cost and energy efficiency.

The design of HPC systems and supercomputers is by no means the only scenario where such optimization problems occur. The execution platforms of typical consumer devices like smart phones and tablets consist of very heterogeneous Multi-Processor Systems-on-Chip (MPSoCs) and the design challenges for them are similar.

Despite the appeal of an approach to *simultaneously* optimize for software and hardware, its implementation represents a formidable challenge because of the huge search space. Previous approaches [119, 120, 121], pick a hardware model $\mathcal{H}$ from the hardware design space, a software model $\mathcal{S}$ from the software design space, map $\mathcal{S}$ onto $\mathcal{H}$, estimate the performance of the mapping, and iterate until a desirable quality is achieved. But not only each of the software and hardware design spaces can be huge, each iteration takes a long time since finding a good mapping of $\mathcal{S}$ onto $\mathcal{H}$ and estimating the performance of the resulting implementation are themselves challenging computational problems.

We propose a new approach(Appendix C) for the software-hardware codesign problem that avoids these pitfalls by considerably shrinking the design space and making its exploration possible by formulating the optimization problem in a way that allows the use of existing powerful optimization solvers. We apply the methodology to programmable accelerators: Graphics Processing Units (GPUs), and for stencil codes. The key elements of our approach are to exploit multiple forms of *domain-specificity*. Our main contributions are:

- We propose a new approach to software-hardware codesign that it is computationally feasible and provides interesting insights.
- We combine our area model with a workload characterization of stencil codes, and our previously proposed execution time model [22] to formulate a mathematical optimization problem that maximizes a common objective function of the hardware and software parameters.
- Our analysis provides interesting insights. We produce a set of Pareto optimal designs that represent the optimal combination of hardware and compiler parameters. They allow for up to 33% improvement in performance as measured in GFLOPs/sec.

We develop a framework for software-hardware codesign that allows the simultaneous optimization of software and hardware parameters. It assumes having analytical models for performance, for which we use execution time, and cost, for which we choose the chip area. We make use of the execution time model from Prajapati et al [22] that predicts the execution times of a set of stencil programs. For the chip area, we develop an analytical model that estimates the chip area of parameterized designs from the Maxwell GPU architecture. Our model is reasonably accurate for estimating the total die area based on individual components such as the number of SMs, the number of vector units, the size of memories, etc.

We formulate a codesign optimization problem using the time model and our area model for optimizing the compiler and architecture parameters simultaneously. We predict an improvement in the performance of 2D stencils by (104% and 69%) and 3D stencils by (123% and 126%) over existing Maxwell (GTX980 and Titan X) architectures.

The main focus is on the methodology; specifically, to develop a software-hardware codesign framework and to illustrate how models built using it can be used for efficient exploration of the design space for identifying Pareto-optimal configurations and analyzing for design tradeoffs. The same framework, possibly with some modifications, could be used for codesign on other type of hardware platforms (instead of GPU), other type of software kernels (instead of the set of stencils we chose, or even non-stencil kernels), and other kind of performance and cost criteria (e.g., energy as cost). Also, with work focused on the individual elements of the framework, the execution time and the chip area models we used could possibly be replaced by ones with better features in certain aspects or scenarios.

The analyses from our work indicate the following accelerator design recommendations, for the chosen performance, cost criteria, and application profile:

- Remove caches completely and
- Use the area (previously devoted to caches) to add more cores on the chip.
- The more precise the workload characterization and the specific area model parameters, the more useful the conclusions drawn from the study.

Hardware resources such as memories are often expensive and must be utilized wisely. In the next chapter, we discuss how to use cost models to identify the resources requirements for optimal performance via bottleneck analysis.

Chapter 8

Conclusions

Our work contributes to knowledge in following ways:

The unified view of the polyhedral design landscape We put together all the parameters to be considered for performance optimization in a single ***unified landscape*** (Chapter 2). The landscape (shown in Figure 2.1) considers the program, architecture, and compiler parameters and combines them with various cost metrics. This view lets us identify the pockets of domain specificity and allows us to study performance improvement across all cost metrics.

Analytical Models We develop ***analytical cost models*** (Chapter 4) for execution time, energy, and memory access models for stencil computations. We develop silicon area models of a GPU chip. Our models are reasonably accurate and help predict the associated cost. We also develop memory access models for a single matrix multiplication and for fused two matrix multiplications. We argue that these models can be used to break the HPC application performance improvement cycle.

Mathematical Optimization Approach We formulate mathematical optimization problems to address some of the challenges of exascale. We show how these optimizations can be used for ***single as well as multi-metric tuning*** (Chapter 5 and 6) and ***accelerator codesign*** (Chapter 7). For GPGPU stencil computations and polyhedral code generator, we illustrate a proof of concept [21] and present a novel optimization approach to accelerator codesign.

8.1 Future Work

We can explore ***bottleneckology*** in three ways: ***study codesign trade-offs, perform overhead analysis, and investigate the effect of hyperthreading***. Next three sections discuss these in more details.

Table 8.1: Workload sensitivity. The optimal architecture configuration for a single benchmark varies significantly.

Code	n_{SM}	n_{V}	M_{SM}	Area	GFLOPs/S
Jacobi 2D	32	128	24	438	2059
Heat 2D	22	256	12	447	3017
Gradient 2D	28	160	24	431	4963
Laplacian 2D	28	160	12	426	2549
Heat 3D	18	288	192	447	3600
Laplacian 3D	8	896	96	446	1427

8.1.1 Codesign Trade-offs

Workload Sensitivity Table C.2 illustrates the architectural parameters for the best performing designs for each of the six benchmarks(2D and 3D stencils), for an area budget between 425–450 mm^2. Observe how the parameters of the best architecture are significantly different. There are also differences in the achieved performance for each benchmark, but that is to be expected since the main computation in the stencil loop body has different number of operations across the benchmarks.

Shared Memory Requirements We can also observe that there are marked differences between the optimal architecture configurations for 2D and 3D stencils in Table C.2. 3D stencils seem to require larger shared memory ($\geq$ 96 kB / SM) compared to 2D stencils ($\leq$ 24 kB / SM). Indeed, for designs with lower than 48kB, the performance was nowhere near the optimal for 3D stencil programs. Comparing the optimal configurations for $Heat\ 2D$ stencil with that of $Heat\ 3D$ stencil (both have equal total die area of $447\ mm^2$), we observe that the amount of shared memory required for $Heat\ 3D$ stencil is 16 times more than that for $Heat\ 2D$ stencil. Also note, 3D stencils require higher number of vector units per SM for optimal performance. We do not have a clear explanation but the intuition lies in the dependence patterns. This is a subject of our ongoing investigation.

Resource Allocation Another interesting perspective is seen in Figure C.4 which plots the Pareto optimal design points in blue and all other non-Pareto configurations in orange. The axes show the relative percentages of the chip area devoted to memory, and to vector units. We notice that the

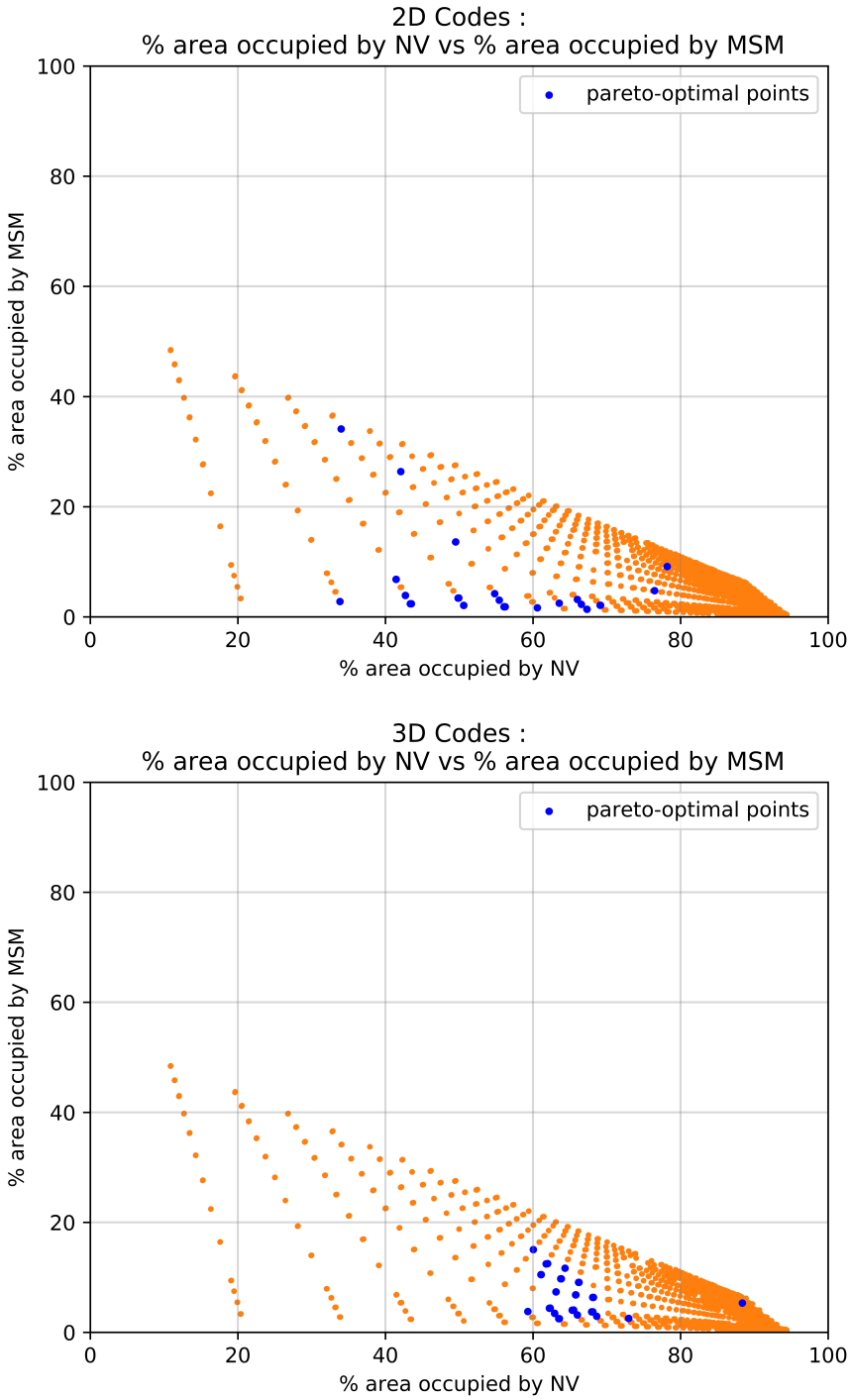


Figure 8.1: Resource Allocation.

optimal designs (blue points) lie in a relative cluster. This phenomenon is even more marked for 3D stencils. At present, we do not have a clear explanation for why the points are clustered in this manner, and we plan to mine this data to determine patterns, if any.

8.1.2 Overhead Analysis

As discussed before, a compiler generated code has a number of parameters, such as tile sizes, that are then tuned via empirical exploration. Our execution time model [22] guides such a choice. This execution time model is a simple set of analytical functions that predicts the execution time of the HHC [19] generated code.

The execution time model uses a machine dependent parameter, called C_{iter}, which is the execution time of one iteration of the loop body per vector unit provided that all the necessary data is available in shared memory. To measure C_{iter}, one needs to generate a random set of codes for a given stencil with different tile sizes, modify these codes to remove global memory accesses and then take the average of empirically measured execution times to obtain the value of C_{iter}. The process is very time consuming and requires expertise in Hybrid Hexagonal Tiled code generator [19]. Also, C_{iter} is dependent on both, the machine as well as the program. Therefore, its value changes as machine and program parameters vary. It is, therefore, very difficult to model/measure is C_{iter}. Also note that C_{iter} value is used to evaluate the objective function to find the optimal tile size. This imposes limitations on the use of execution time model for optimal tile size selection because the crucial model parameter C_{iter}, is to be empirically measured.

To address this problem, we propose a closed form solution that is completely independent of the machine and the program parameters. We are able to analytically predict the optimal tile sizes which are portable across platforms and is valid for all Jacobi-like stencils. We modify the objective function from Prajapati et al. [22] and develop a cost function which is independent of C_{iter}. For a 2D stencil, our closed form solution suggests that we maximize the size of the hexagonal face of a tile subject to some constraints. This allows us to significantly narrow down the tile size design space.

The mathematical optimization problem in Prajapati et al. [22] for a given 2D stencil is formulated as follows:

$$minimize \quad T_{alg}(t_{S_1}, t_{S_2}, t_T) \tag{8.1}$$

where t_{S_1}, t_{S_2}, and t_T are tile sizes and T_{alg} is the model predicted execution time of the code given by the following formula:

$$T_{\text{alg}} = N_w T_{\text{sync}} + N_w T_{\text{prism}} \left\lceil \frac{1}{n_{SM}} \left\lceil \frac{w}{k} \right\rceil \right\rceil . \tag{8.2}$$

where N_w is the number of wavefronts, T_{sync} is the time for synchronization for a wavefront, T_{prism} is the time to execute a tile, n_{SM} is the number of processors, w is the size of a wavefront and k is the number of tiles that execute simultaneously. For more details, please refer Prajapati et al. [22].

In addition to the time for computation, T_{alg} includes time taken by data transfers and the time for inter-tile and intra-tile synchronizations. We are interested in only those tile sizes that give optimal performance. Therefore, our tiles will be compute bound. Let us consider an ideal machine where the performance is given by the following equation:

$$T_{\text{ideal}} = \frac{S_1 S_2 T}{n_{SM} n_V} C_{iter} \tag{8.3}$$

where S_1, and S_2 are problem sizes in space dimensions, T is the problem size in time dimension, and n_V is the number of vector units. Such an ideal machine is free of all synchronization delays and takes no additional time to do data transfers. On a real machine, we would like to obtain the performance that is close to T_{ideal}. However, there is always an overhead price such that

$$T_{\text{overhead}} = T_{\text{alg}} - T_{\text{ideal}} \tag{8.4}$$

Substituting T_{alg} and T_{ideal} with their respective equations and solving gives us

$$T_{\text{overhead}} = C_{iter} S_1 T \frac{1}{t_{S_1} + \frac{t_T}{2}} \tag{8.5}$$

Instead of minimizing the execution time(as in the equation 8.1), we can now minimize the overhead. To minimize equation 8.5, we need to maximize $t_{S_1} + \frac{t_T}{2}$. This suggests that we should

increase the size of the hexagonal face of the tile as much as possible. Notice, t_{S_2} does not appear in the equation 8.5.

For the above formulation, we assumed that the tiles are compute bound. We need a mechanism to first prune the tile size design space and restrict it to only consider compute bound tiles and then use the above cost functions to further reduce the search space.

8.1.3 The effect of the hyper-threading

Our results in [22] suggest that we should revisit the "conventional wisdom" that says that an optimal strategy of a tiling is to choose the "largest possible tile size that fits" i.e., its memory footprint matches the available capacity. First of all, this falls into the trap that it precludes overlapping of computation and communication (the "hyperthreading effect"). But this can be avoided by explicitly accounting for hyperthreading. Indeed, our GPU platforms preclude such large size by disallowing the data footprint of a thread block to exceed *half* the shared memory capacity.

Thus, the hyperthreading-adjusted "conventional wisdom" would still seek to maximize tile volume subject to the half-capacity constraint—the best strategy is the largest tile volume for the given footprint. Our model and experimental data suggests otherwise—an even higher hyperthreading factor is turning out to yield the best performance. We still don't know why, and it is subject to investigation.

8.1.4 Evaluation of Minimum Data Movement Algorithm for Matrix Chain Products

The closed form solutions in Chapter D remain to be empirical validated in terms of execution time performance. The tile sizes that minimize data movement for fused two matrix multiplication should also have optimal execution time performance. Optimal execution time means reduction in the static energy consumption. Validation of fused codes requires a code generator that admits fused schedule and allocates shared memory to the tiles in the intermediate matrix. Such a code generator is not yet available and hence this empirical evaluation still remains. We will also need to

measure dynamic energy consumption of such codes and empirically validate that the total energy consumption is indeed minimized using our methods.

8.2 Limitations of our approach

Our approach is limited to the narrow area of domain specific applications, polyhedral model and GPU-like programmable hardware accelerators. The approach can, however, be extended to other set of *programs/architectures/transformations* by identifying other domain specific regions in the design landscape. More work is needed in order to extend our approach across different regions of domain specificity.

8.3 Open Questions

Among the many different uses of the analytical cost models, they can be further explored to answer important performance related questions. We list some of them below:

- Using analytical execution time and energy models we can find out (i) What happens when input parameters change? (ii) What happens when different number of processors is used? (iii) What is the largest possible problem size on a given architecture? (iv) When does the efficiency drop?

- Silicon area models can be used for the following: (i) Chip area prediction. (ii) Generate all possible configurations for a give die area. (iii) Calculate the cost of different configurations. (iv) Study the trade-offs. (v) System tuning.

- Performance tuning related questions such as (i) Sensitivity of problem size to tile size. (ii) Sensitivity of optimal tile size for different codes. (iii) Reconfirm the folklore : whether optimizing for time is equal to optimizing for energy? (iv) Reasons for the poor performance. can be answered using cost models.

The answer to these questions become helpful in two situations. One, for *performance portability* while moving from one architecture to another. Second, obtaining interesting insights to recognize promising areas for future research.

Bibliography

[1] Fritzchens Fritz's collection of nvidia die photographs. https://www.flickr.com/photos/130561288@N04, 2017. Accessed: 2017-April-07.

[2] National Research Council, Samuel H. Fuller, and Lynette I. Millett. *The Future of Computing Performance: Game Over or Next Level?* The National Academies Press, Washington, DC, 2011.

[3] David J. Kuck. *High Performance Computing: Challenges for Future Systems*. Oxford University Press, Oxford, UK, 1996.

[4] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, and many. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 15, 2008.

[5] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. In *International Conference on High Performance Computing for Computational Science*, pages 1–25. Springer, 2010.

[6] Bill Dally. Power, programmability, and granularity: The challenges of exascale computing. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 878–878. IEEE, 2011.

[7] Martin Schulz, Jim Belak, Abhinav Bhatele, Peer Timo Bremer, Greg Bronevetsky, Marc Casas, Todd Gamblin, Katherine E. Isaacs, Ignacio Laguna, Joshua Levine, Valerio Pascucci, David Richards, and Barry Rountree. *Performance analysis techniques for the exascale co-design process*, volume 25 of *Advances in Parallel Computing*, pages 19–32. Elsevier, 2014.

[8] Steve Ashby and Many. *The opportunities and challenges of exascale computing*. http://science.energy.gov/. U.S. Department of Energy, 2010.

[9] Paul Messina. The exascale computing project. *Computing in Science & Engineering*, 19(3):63–67, 2017.

[10] Donald E Thomas, Jay K Adams, and Herman Schmit. A model and methodology for hardware-software codesign. *IEEE Design & test of computers*, 10(3):6–15, 1993.

[11] Wayne H Wolf. Hardware-software co-design of embedded systems. *Proceedings of the IEEE*, 82(7):967–989, 1994.

[12] Wayne Wolf. A decade of hardware/software codesign. *Computer*, 36(4):38–43, 2003.

[13] Massimiliano Chiodo, Paolo Giusto, Attila Jurecska, Harry C Hsieh, Alberto Sangiovanni-Vincentelli, and Luciano Lavagno. Hardware-software codesign of embedded systems. *IEEE micro*, 14(4):26–36, 1994.

[14] 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.

[15] P. Quinton and V. Van Dongen. The mapping of linear recurrence equations on regular arrays. *J. of VLSI Signal Processing*, 1(2), 1989.

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

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

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

[19] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. Hybrid hexagonal/classical tiling for GPUs. In *CGO*, page 66, Orlando, FL, Feb 2014.

[20] C. Chen, J. Chame, and M. Hall. Chill: A framework for composing high-level loop transformations. Technical Report TR-08-897, USC Computer Science Department, June 2008.

[21] Nirmal Prajapati, Sanjay Rajopadhye, Hristo Djidjev, Nandkishore Santhi, Tobias Grosser, and Rumen Andonov. Optimization approach to accelerator codesign. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2019.

[22] Nirmal Prajapati, Waruna Ranasinghe, Sanjay Rajopadhye, Rumen Andonov, Hristo Djidjev, and Tobias Grosser. Simple, accurate, analytical time modeling and optimal tile size selection for gpgpu stencils. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2017*, pages 163–177. ACM, 2017.

[23] Nirmal Prajapati, Waruna Ranasinghe, Vamshi Tandrapati, Rumen Andonov, Hristo Djidjev, and Sanjay V. Rajopadhye. Energy modeling and optimization for tiled nested-loop codes. In *High Performance, Power-Aware Computing in conjunction with IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015*, pages 888–895, 2015.

[24] Louis-Noel Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. Polyhedral-based data reuse optimization for configurable computing. In *Proc. of the ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays*, FPGA '13, pages 29–38, New York, NY, USA, 2013. ACM.

[25] Wei Zuo, Warren Kemmerer, Jong Bin Lim, Louis-Noël Pouchet, Andrey Ayupov, Taemin Kim, Kyungtae Han, and Deming Chen. A polyhedral-based systemc modeling and generation framework for effective low-power design space exploration. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '15, pages 357–364, Piscataway, NJ, USA, 2015. IEEE Press.

[26] Wei Zuo, Peng Li, Deming Chen, Louis-Noël Pouchet, Shunan Zhong, and Jason Cong. Improving polyhedral code generation for high-level synthesis. In *Proc. of the Ninth IEEE/ACM/IFIP Int. Conf. on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '13, pages 15:1–15:10, Piscataway, NJ, USA, 2013. IEEE Press.

[27] 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. `www.eecs.berkeley.edu/Pubs/TechRpts/2006...` `/EECS-2006-183.pdf`.

[28] 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.

[29] 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.

[30] 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.

[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. 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.

[33] 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.

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

[36] 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.

[37] 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.

[38] Robert Strzodka, Mohammed Shaheen, and Dawid 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.

[39] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles 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.

[40] 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.

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

[42] 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.

[43] 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.

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

[45] 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.

[46] 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.

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

[48] 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.

[49] 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.

[50] Alain Darte, Yves Robert, and Frédéric Vivien. *Scheduling and Automatic Parallelization*. Birkhaüser, 2000.

[51] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. Pluto: A practical and fully automatic polyhedral program optimization system. In *ACM Conference on Programming Language Design and Implementation*, pages 101–113, Tuscon, AZ, June 2008. ACM SIGPLAN.

[52] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):54, 2013.

[53] U. Bondhugula, V. Bandishti, A. Cohen, G. Potron, and N Vasilache. Tiling and optimizing time-iterated computations over periodic domains. In *PACT*, 2014.

[54] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. Tiling stencil computations to maximize parallelism. In *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.

[55] Yun Zou. *Towards Automatic Compilation for Energy Efficient Iterative Stencil Computations*. PhD thesis, Colorado State University, Ft. Collins CO, July 2016.

[56] D. Wonnacott. Achieving scalable locality with time skewing. *IJPP: International Journal of Parallel Programming*, 30(3):181–221, Jun 2002.

[57] R. Andonov, S. Balev, S. Rajopadhye, and N. Yanev. Optimal semi-oblique tiling. *IEEE Transactions on Parallel and Distributed Systems*, 14(9):944–960, Sept 2003.

[58] Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and Hans-Peter Seidel. Cache accurate time skewing in iterative stencil computations. In *Proceedings of the International Conference on Parallel Processing (ICPP)*. IEEE Computer Society, September 2011.

[59] Tobias Grosser, Sven Verdoolaege, Albert Cohen, and P. Sadayappan. The relation between diamond tiling and hexagonal tiling. *Parallel Processing Letters*, 24(3), 2014.

[60] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. Polyhedral AST generation is more than scanning polyhedra. *ACM Trans. Program. Lang. Syst.*, 37(4):12:1–12:50, July 2015.

[61] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[62] 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.

[63] Liu Peng, Richard Seymour, Ken ichi Nomura, Rajiv K. Kalia, Aiichiro Nakano, Priya Vashishta, Alexander Loddoch, Michael Netzband, William R. Volz, and Chap C. Wong. High-order stencil computations on multicore clusters. In *IPPS*, 2009.

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

[65] Uday Bondhugula, Albert 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.

[66] David 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.

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

[68] 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.

[69] Matteo Frigo and Volker 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.

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

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

[72] Vinayaka Bandishti, Irshad Pananilath, and Uday 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.

[73] Justin Holewinski, Louis-Noël 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.

[74] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep 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.

[75] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman 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.

[76] Tobias Gysi, Carlos Osuna, Oliver Fuhrer, Mauro Bianco, and Thomas 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.

[77] Ravi Teja Mullapudi, Vinay Vasista, and Uday 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.

[78] Tobias Gysi, Tobias Grosser, and Torsten 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.

[79] Jiayuan Meng and Kevin 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.

[80] Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David Padua, Keshav Pingali, Paul Stodghill, and Peng Wu. A comparison of empirical and model-driven optimization. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 63–76, New York, NY, USA, 2003. ACM.

[81] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[82] Jun Shirako, Kamal Sharma, Naznin Fauzia, Louis-Noël Pouchet, J. Ramanujam, P. Sadayappan, and Vivek Sarkar. Analytical bounds for optimal tile size selection. In *Proc. of the 21st Int. Conf. on Compiler Construction*, pages 101–121, Berlin, Heidelberg, 2012. Springer.

[83] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *Proc. of the 36th Annual Int. Symposium on Computer Architecture*, ISCA '09, pages 152–163, New York, NY, USA, 2009. ACM.

[84] 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.

[85] Lakshminarayanan Renganarayana and Sanjay Rajopadhye. Positivity, posynomials and tile size selection. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 55:1–55:12, Piscataway, NJ, USA, 2008. IEEE Press.

[86] Randy Torrance and Dick James. The State-of-the-Art in IC Reverse Engineering. In *Proceedings of Cryptographic Hardware and Embedded Systems Conference (CHES), LNCS Volume 5747*, pages 363–381, 2009.

[87] *Degate*. http://www.degate.org/documentation/.

[88] Silicon Zoo open source standard cell catalog. http://www.siliconzoo.org, 2017. Accessed: 2017-April-07.

[89] Franck Courbon, Sergei Skorobogatov, and Christopher Woods. Reverse engineering flash eeprom memories using scanning electron microscopy. In *Proceedings of International Conference on Smart Card Research and Advanced Applications, CARDIS 2016: Smart Card Research and Advanced Applications, LNCS Volume 10146*, pages 57–72, 2016.

[90] Sparsh Mittal and Jeffrey S. Vetter. A survey of methods for analyzing and improving GPU energy efficiency. *ACM Comput. Surv.*, 47(2):19:1–19:23, August 2014.

[91] Sunpyo Hong and Hyesoon Kim. An integrated GPU power and performance model. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 280–289, New York, NY, USA, 2010. ACM.

[92] Sunpyo Hong and Hyesoon Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proc. of the 36th Annual Int. Symp. on Computer Architecture*, ISCA '09, pages 152–163, New York, NY, USA, 2009. ACM.

[93] H. Nagasaka, N. Maruyama, A Nukada, T. Endo, and S. Matsuoka. Statistical power modeling of GPU kernels using performance counters. In *Green Computing Conference, 2010 International*, pages 115–122, Aug 2010.

[94] Jieun Lim, Nagesh B. Lakshminarayana, Hyesoon Kim, William Song, Sudhakar Yalamanchili, and Wonyong Sung. Power modeling for GPU architectures using McPAT. *ACM Trans. Des. Autom. Electron. Syst.*, 19(3):26:1–26:24, June 2014.

[95] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. GPUWattch: Enabling energy optimizations in GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 487–498, New York, NY, USA, 2013. ACM.

[96] J. Lucas, S. Lal, M. Andersch, M. Alvarez-Mesa, and B. Juurlink. How a single chip causes massive power bills GPUSimPow: A GPGPU power simulator. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 97–106, Apr 2013.

[97] Sheng Li, Jung-Ho Ahn, R.D. Strong, J.B. Brockman, D.M. Tullsen, and N.P. Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and many-

core architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 469–480, Dec 2009.

[98] A Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174, April 2009.

[99] Da Qi Ren. Algorithm level power efficiency optimization for CPU-GPU processing element in data intensive SIMD/SPMD computing. *Journal of Parallel and Distributed Computing*, 71(2):245 – 253, 2011. Data Intensive Computing.

[100] Da-Qi Ren and Reiji Suda. Global optimization model on power efficiency of GPU and multicore processing element for SIMD computing with CUDA. *Computer Science - Research and Development*, 27(4):319–327, 2012.

[101] S. Prakash and A. C. Parker. Synthesis of application-specific multiprocessor systems including memory components. In *Proc. of the Int. Conf. on Application Specific Array Processors*, pages 118–132, Aug 1992.

[102] Wayne Wolf and Jorgen Staunstrup. *Hardware/Software CO-Design: Principles and Practice*. Kluwer, Norwell, MA, USA, 1997.

[103] Karam S. Chatha and Ranga Vemuri. MAGELLAN: Multiway hardware-software partitioning and scheduling for latency minimization of hierarchical control-dataflow task graphs. In *Proceedings of the Ninth International Symposium on Hardware/Software Codesign*, CODES '01, pages 42–47, New York, NY, USA, 2001. ACM.

[104] R. P. Dick and N. K. Jha. MOGAC: A multiobjective genetic algorithm for hardware-software cosynthesis of distributed embedded systems. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 17(10), November 2006.

[105] J. Teich. Hardware/software codesign: The past, the present, and predicting the future. *Proc. of the IEEE*, 100:1411–1430, May 2012.

[106] Petru Eles, Zebo Peng, Krzysztof Kuchcinski, and Alexa Doboli. System level hardware/-software partitioning based on simulated annealing and tabu search. *Trans. on Design Automation for Embedded Systems*, 2(1):5–32, January 1997.

[107] C. Erbas, S. Cerav-Erbas, and A. D. Pimentel. Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design. *IEEE Transactions on Evolutionary Computation*, 10(3):358–374, September 2006.

[108] Ralf Niemann and Peter Marwedel. An algorithm for hardware/software partitioning using mixed integer linear programming. *Design Automation for Embedded Systems*, 2(2):165–193, 1997.

[109] S. S. Dosanjh, R. F. Barrett, D. W. Doerfler, S. D. Hammond, K. S. Hemmert, M. A. Heroux, P. T. Lin, K. T. Pedretti, A. F. Rodrigues, T. G. Trucano, and J. P. Luitjens. Exascale design space exploration and co-design. *Future Gener. Comput. Syst.*, 30:46–58, January 2014.

[110] Didem Unat, Cy Chan, Weiqun Zhang, Samuel Williams, John Bachan, John Bell, and John Shalf. Exasat: An exascale co-design tool for performance modeling. *The International Journal of High Performance Computing Applications*, 29(2):209–232, 2015.

[111] David J. Kuck, Michael W. Berry, Kyle A. Gallivan, Efstratios Gallopoulos, Ananth Grama, Bernard Philippe, Yousef Saad, and Faisal Saied. *Computational Capacity-Based Codesign of Computer Systems*, pages 45–73. Springer London, London, 2012.

[112] David Kuck. A comprehensive approach to hw/sw codesign. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 1–2, Piscataway, NJ, USA, 2013. IEEE Press.

[113] K. Rajamani and C. Lefurgy. On evaluating request-distribution schemes for saving energy in server clusters. *2003 IEEE International Symposium on Performance Analysis of Systems and Software. ISPASS 2003.*, 2003.

[114] Luiz André Barroso, J Clidaras, and Urs Hölzle. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. In *Synthesis Lectures on Computer Architecture*, volume 4, pages 1–108. Morgan & Claypool Publishers, 2009.

[115] Ivan Rodero and Manish Parashar. *Energy Efficiency in HPC Systems*, pages 81–108. John Wiley & Sons, Inc., 2012.

[116] Jan Lucas, Sohan Lal, Michael Andersch, Mauricio Alvarez Mesa, and Ben H. H. Juurlink. How a single chip causes massive power bills GPUSimPow: A GPGPU power simulator. In *ISPASS*, pages 97–106. IEEE, 2013.

[117] Vivek Sarkar, William Harrod, and Allan E. Snavely. Software challenges in extreme scale systems. *Journal of Physics: Conference Series*, 180(1):012–045, 2009.

[118] J. D. Ullman. NP-complete scheduling problems. *J. Comput. Syst. Sci.*, 10(3):384–393, June 1975.

[119] Hristo Djidjev. Codesign mapping and optimization algorithms. Technical Report LA-UR-11-10169, Los Alamos National Laboratory, 2011.

[120] S. Eidenbenz, K. Davis, A. Voter, H. Djidjev, L. Gurvitz, C. Junghans, S. Mniszewski, D. Perez, N. Santhi, and S. Thulasidasan. Optimization principles for codesign applied to molecular dynamics: Design space exploration, performance prediction and optimization strategies. In *Proceedings of the U.S. Department of Energy Exascale Research Conference*, pages 64–65, 2012.

[121] Susan M. Mniszewski, Christoph Junghans, Arthur F. Voter, Danny Perez, and Stephan J. Eidenbenz. Tadsim: Discrete event-based performance prediction for temperature-accelerated dynamics. *ACM Trans. Model. Comput. Simul.*, 25(3):15:1–15:26, April 2015.

[122] R. M. Karp, R. E. Miller, and S. V. Winograd. The organization of computations for uniform recurrence equations. *JACM*, 14(3):563–590, July 1967.

[123] Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL: A Modelling Language for Mathematical Programming*. Duxbury Press, Brooks/Cole Publishing Company, 2nd edition, 2002.

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

[125] R. Andonov and S. Rajopadhye. Optimal orthogonal tiling of 2-D iterations. *Journal of Parallel and Distributed Computing*, 45(2):159–165, September 1997.

[126] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Totonto, CA, june 1991.

[127] J. McCalpin and D. Wonnacott. Time skewing: A value-based approach to optimizing for memory locality. Technical Report TR-379, Rutgers University, Department of Computer Science, March 1999.

[128] CUDA C programming guide v6.0, April 2014.

[129] NVIDIA Corporation. *NVIDIA Next Generation CUDA Compute Architecture: Kepler GK110*, 2012. Rev. 1.0.

[130] Jens Lang and Gudula RÃijnger. High-resolution power profiling of GPU functions using low-resolution measurement. In Felix Wolf, Bernd Mohr, and Dieter an Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 801–812. Springer Berlin Heidelberg, 2013.

[131] Cbc home page. https://projects.coin-or.org/Cbc.

[132] Richard H Byrd, Jorge Nocedal, and Richard A Waltz. Knitro : An Integrated Package for Nonlinear Optimization. *Energy*, 83:1–25, 2006.

[133] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *ISCA '11 Proceedings of the 38th*

Annual International Symposium on Computer Architecture, pages 365–376, San Jose, CA, June 2011. ACM.

[134] K. O'Brien, L. Di Tucci, G. Durelli, and M. Blott. Towards exascale computing with heterogeneous architectures. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 398–403, March 2017.

[135] K Amunts, A Lindner, and K Zilles. The human brain project: neuroscience perspectives and german contributions. *e-Neuroforum*, 5(2):43–50, 2014.

[136] P Chris Broekema, Rob V van Nieuwpoort, and Henri E Bal. The square kilometre array science data processor. preliminary compute platform design. *Journal of Instrumentation*, 10(07):C07004, 2015.

[137] Rik Jongerius, Stefan Wijnholds, Ronald Nijboer, and Henk Corporaal. An end-to-end computing model for the square kilometre array. *Computer*, 47(9):48–54, 2014.

[138] Gabe Rudy. *CUDA-CHiLL: A programming language interface for GPGPU optimizations and code generation*. The University of Utah, 2010.

[139] Robin Lee, Jung-Ping Yang, Chia-En Huang, Chih-Chieh Chiu, Wei-Shuo Kao, Hong-Chen Cheng, Hong-Jen Liao, and Jonathan Chang. A 28nm high-k metal-gate sram with asynchronous cross-couple read assist (ac2ra) circuitry achieving 3x reduction on speed variation for single ended arrays. In *Proc., IEEE VLSIC 2012*, pages 64–65, 2012.

[140] HP Labs Cacti 6.5 sram area and power estimation tool. https://github.com/HewlettPackard/cacti, 2014. Accessed: 2017-May-31.

[141] NVidia GeForce GTX-980 photomicrograph. http://international.download.nvidia.com/geforce-com/international/images/geforce-gtx-980-970-feature-article/Maxwell_Die_FlatShade_colorA.jpg, 2014. Accessed: 2016-April-08.

[142] Alexander Grebhahn, Sebastian Kuckuk, Christian Schmitt, Harald Köstler, Norbert Siegmund, Sven Apel, Frank Hannig, and Jürgen Teich. Experiments on optimizing the performance of stencil codes with spl conqueror. *Parallel Processing Letters*, 24(03):1441001, 2014.

[143] Wenlei Bao, Sriram Krishnamoorthy, Louis-Noel Pouchet, and P Sadayappan. Analytical modeling of cache behavior for affine programs. *Proceedings of the ACM on Programming Languages*, 2(POPL):32, 2017.

[144] Louis-Noel Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. Polyhedral-based data reuse optimization for configurable computing. In *Proc. of the ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays*, FPGA '13, pages 29–38, New York, NY, USA, 2013. ACM.

[145] Joo Hwan Lee, Jiayuan Meng, and Hyesoon Kim. Sesh framework: A space exploration framework for GPU application and hardware codesign. In Stephen A. Jarvis, Steven A. Wright, and Simon D. Hammond, editors, *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation*, pages 182–202, Cham, 2014. Springer International Publishing.

[146] Guanwen Zhong, Vanchinathan Venkataramani, Yun Liang, Tulika Mitra, and Smaïl Niar. Design space exploration of multiple loops on FPGAs using high level synthesis. *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pages 456–463, 2014.

[147] Guanwen Zhong, Alok Prakash, Siqi Wang, Yun Liang, Tulika Mitra, and Smaïl Niar. Design space exploration of FPGA-based accelerators with multi-level parallelism. *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1141–1146, 2017.

[148] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter perfor-

mance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12. ACM, 2017.

[149] Jason Cong, Vivek Sarkar, Glenn Reinman, and Alex Bui. Customizable domain-specific computing. *IEEE Design & Test of Computers*, 28(2):6–15, 2011.

[150] Samuel Steffl and Sherief Reda. Lacore: A supercomputing-like linear algebra accelerator for soc-based designs. In *2017 IEEE 35th International Conference on Computer Design (ICCD)*, pages 137–144. IEEE, 2017.

[151] *Polyhedral Benchmark suite*. http://web.cs.ucla.edu/~pouchet/software/polybench/, 2012 (accessed June 19, 2019).

[152] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu. Google workloads for consumer devices: Mitigating data movement bottlenecks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 316–331, New York, NY, USA, 2018. ACM.

[153] J. W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, STOC '81, pages 326–333, New York, NY, USA, 1981. ACM.

[154] Lynn Elliot Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm*. PhD thesis, Bozeman, MT, USA, 1969. AAI7010025.

[155] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 39(5):575–582, September 1995.

[156] Dror Irony, Sivan Toledo, and Alexander Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64(9):1017–1026, September 2004.

[157] Robert A. van de Geijn and Jerrell Watts. Summa: Scalable universal matrix multiplication algorithm. Technical report, Austin, TX, USA, 1995.

[158] S. Lennart Johnsson. Minimizing the communication time for matrix multiplication on multiprocessors. *Parallel Computing*, 19(11):1235–1257, November 1993.

[159] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 285–, Washington, DC, USA, 1999. IEEE Computer Society.

[160] Edgar Solomonik and James Demmel. Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, Euro-Par'11, pages 90–109, Berlin, Heidelberg, 2011. Springer-Verlag.

[161] William F McColl and Alexandre Tiskin. Memory-efficient matrix multiplication in the bsp model. *Algorithmica*, 24(3-4):287–297, 1999.

[162] James Demmel, David Eliahu, Armando Fox, Shoaib Kamil, Benjamin Lipshitz, Oded Schwartz, and Omer Spillinger. Communication-optimal parallel recursive rectangular matrix multiplication. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 261–272, Washington, DC, USA, 2013. IEEE Computer Society.

[163] G. Ballard, E. Carson, J. Demmel, M. Hoemmen, N. Knight, and O. Schwartz. Communication lower bounds and optimal algorithms for numerical linear algebra. *Acta Numerica*, 23:1–155, 2014.

[164] Jinsung Kim, Aravind Sukumaran-Rajam, Vineeth Thumma, Sriram Krishnamoorthy, Ajay Panyala, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. A code generator for high-performance tensor contractions on gpus. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, pages 85–95, Piscataway, NJ, USA, 2019. IEEE Press.

[165] Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *LCPC*, 1993.

[166] Sharad K Singhai and Kathryn S. McKinley. A parametrized loop fusion algorithm for improving parallelism and cache locality. *The Computer Journal*, 40(6):340–355, 1997.

[167] Apan Qasem and Ken Kennedy. Profitable loop fusion and tiling using model-driven empirical search. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 249–258. ACM, 2006.

[168] Uday Bondhugula, Sanjeeb Dash, Oktay Gunluk, and Lakshminarayanan Renganarayanan. A model for fusion and code motion in an automatic parallelizing compiler. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 343–352. IEEE, 2010.

[169] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[170] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. Loop transformations: Convexity, pruning and optimization. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on*

Principles of Programming Languages, POPL ?11, page 549?562, New York, NY, USA, 2011. Association for Computing Machinery.

[171] Samyam Rajbhandari, Jinsung Kim, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, Robert J. Harrison, and P. Sadayappan. On fusing recursive traversals of k-d trees. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 152–162, New York, NY, USA, 2016. ACM.

[172] Samyam Rajbhandari, Jinsung Kim, Sriram Krishnamoorthy, Louis-Noel Pouchet, Fabrice Rastello, Robert J. Harrison, and P. Sadayappan. A domain-specific compiler for a parallel multiresolution adaptive numerical simulation environment. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 40:1–40:12, Piscataway, NJ, USA, 2016. IEEE Press.

[173] Samyam Rajbhandari, Fabrice Rastello, Karol Kowalski, Sriram Krishnamoorthy, and P. Sadayappan. Optimizing the four-index integral transform using data movement lower bounds analysis. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 327–340, New York, NY, USA, 2017. ACM.

[174] Henrik Barthels, Marcin Copik, and Paolo Bientinesi. The generalized matrix chain algorithm. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, pages 138–148, New York, NY, USA, 2018. ACM.

Appendix A

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

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 (such as auto-tuning).

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 optimized 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.

A.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 [27], 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 [28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40]. Workshops and conferences devoted exclusively to stencil acceleration have recently emerged.

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 occurring in graphics processing. Later, researchers realized that these processors could be used for more general computations, and, eventually, the emergence of tools like CUDA and OpenCL enabled general purpose parallel programming on these platforms.

Exploiting the specificity of the applications and the specificity of target architectures leads to domain-specific tools to map very high level program specifications to highly tuned and optimized implementations on the target architecture. Many such tools exist, both academic research prototypes and productions systems.

As indicated earlier, our domain specificity comes in two flavors. First, we investigate only stencil computations. They 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 [14, 15, 49, 16, 17, 18, 50, 51], and many tools are widely available, e.g., PPCG, developed by the group at ENS, Paris [52].

Second, we tackle a specific target platform, namely a single GPU accelerator, and PPCG includes a module that targets GPUs and incorporates a sophisticated code generator developed by Grosser et al. [19] that employs a state-of-the-art tiling strategy called *hybrid hexagonal classic*

tiling. An open source compiler, implementing this strategy is also available, henceforth called the HHC compiler. We show the following.

- 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. The model is an analytic function of
  - program, machine, and compiler parameters that are easily available statically, and
  - one stencil-specific parameter that is obtained by running a handful of *micro-benchmarks*.

  It is deliberately *optimistic* and also ignores the effect of some parameters.

- Although our model is not accurate for all tile size combinations—the root mean square error(RMSE) is over 100%—it is very accurate for the ones that matter. For codes whose performance is within 20% of the best, the RMSE is less than 10%[6]

  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 than 60,000 programs for
  - two modern **target platforms** (NVIDIA GTX 980, and Titan X), and
  - four 2D **stencil codes** (Jacobi2D, Heat2D, Laplacian2D, and Gradient2D) and two 3D stencils (Heat3D and Laplacian3D)
  - over a range of ten input/problem **sizes**, and
  - a wide range of tile sizes and thread counts (the HHC compiler inputs) for each *platform-stencil-size* combination.

  As we expected, the root-mean-square error (RMSE) over the **entire** data set was "disappointingly" over 100%. However, when we restricted ourselves to the data points that have

[6]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.

an execution time within 20% of the best value for that particular platform-stencil-size combination, the RMSE dropped to less than 10% , which we consider very good.

Our overall methodology is applicable, with simple extensions, to more general programs, e.g., those that fit the polyhedral model. But for achieving high GPU utilization, we need efficient GPU codes to start with, which are very hard and time consuming to produce manually, especially in higher dimensions. The highly optimized HHC-generated codes we are using for testing and validation have a few thousand lines of CUDA code each and we generated tens of thousands such codes in our experimental analysis. So our methodology is not limited to the HHC compiler (in fact we have applied it successfully to manually generated 1D stencil codes), but the use of HHC (or similar compiler) was necessary to produce for our experiments a high number of GPU codes that are also very efficient.

The remainder of this section is organized as follows. We describe the domain specific parallelization used for stencils, and in particular, the strategies used by the HHC compiler. Then, Section A.4 develops our analytical execution time model, Section A.5 describes our experimental results on validating the model on a baseline set of tile sizes. After a discussion of related work (Section C.5), we discuss our results, describe ongoing and future work, and conclude in Section C.6.

A.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 [62, 63, 54]. One successful technique is called *time tiling* [64, 65, 66, 67, 68, 69, 58, 54], an advanced form of loop tiling [70, 64, 71]. 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 [75] and Stella [76] 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 [77] 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. With MODESTO [78] an analytical performance model has been proposed that allows to model multiple cache levels and fusion strategies for both GPUs and CPUs as they arise in the context of Stella.

A.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.

We assume, like most of the work in the literature, that dense iterative 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, and polyhedral compilation is one of the most successful technique.

One of the desired, but often loosely defined properties of a good mapping is "locality of data." Although it is often difficult to formulate mathematically, it is used as a surrogate measure of the quality of the mapping. The ultimate metric remains the speed and/or energy, but it is widely believed that the actual quantitative performance is difficult to model precisely. Modern

processors be they multicore CPUs or manycore GPUs or other accelerators, have multiple levels of memory/communication hierarchy that are poorly understood, and very difficult to analyze.

It is well known that they have limited amount of local memory and the time of data transfer between that memory and the global memory is large compared to the cost of computation or the cost of local access. Hence, it is important to have a method that maps the operations of a software code onto the hardware in a way that optimizes for locality and data reuse. This is achieved by a well known transformation called tiling (of both the computation space and the data sets). Tiling is usually applied at multiple levels (hierarchical tiling) and used to generate sequential or parallelized code with multiple levels of parallelism. These transformations must preserve the semantics of the original code thereby guaranteeing that the mapping will not affect the code correctness.

The architecture targeted in this study is a GPU architecture, illustrated in Figure A.1. There are N_{SM} parallel units (called streaming multiprocessors, or SMs) that communicate through slow global memory. Each SM contains a number of fine grain vector units called streaming processors (SPs) that have access to fast, on-chip memory (called shared memory), as well as to slow, off-chip main memory (called global memory). Thread blocks of 32 threads with consecutive IDs in an SM form a so-called warp, which is executed in a Single Instruction Multiple Data (SIMD) mode. We will consider a number of parameters of such an architecture as variables that can be optimized. Such parameters include number of SMs, shared and global memory size, and data transfer bandwidth between different components.

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[7] 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

[7]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.

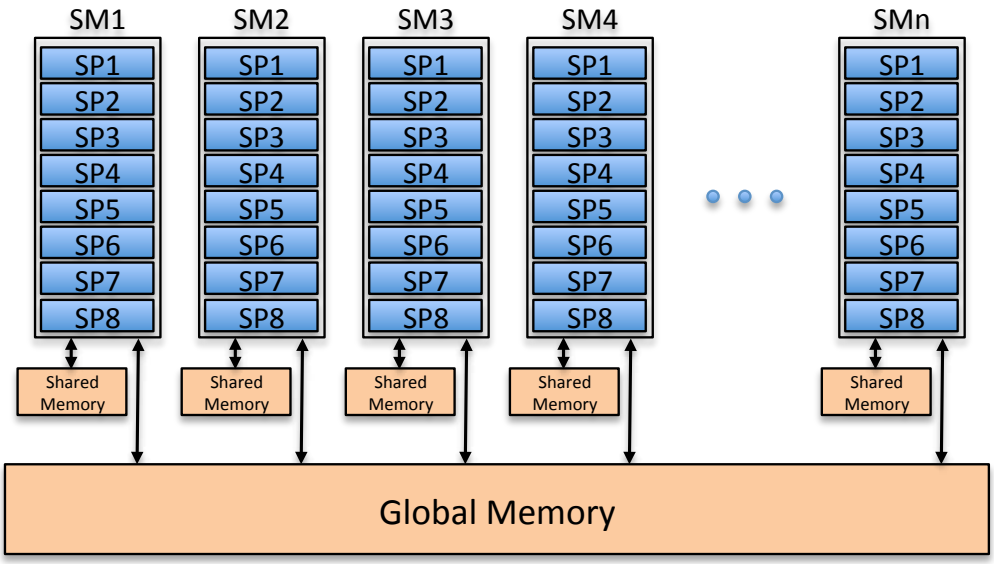


Figure A.1: The GPU architecture.

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.

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

$s \in \mathcal{S}$, $t \in \mathcal{T}$, 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 $\mathcal{S}$).

Furthermore, the elements of the set $\mathcal{N}$ must be such that the computation is "explicitly defined" (see [122]). The definition may be extended to (i) allow a computation to depend on values at more than the previous time step, and (ii) allow the coefficients to be computed values, rather than compile-time constants, provided the evaluation of these coefficients is also described by a stencil computation, and (iii) allow generalization where the computation of $A^t(s)$ involves more than just a weighted sum. Stencils are usually implemented as nested loops with the loop body evaluating the rhs of (C.3) and storing it in a data array.

Efficient parallelization of stencils on GPUs requires careful consideration of at least four factors: parallelism at the fine grain (thread level) and the coarse grain (block level) and data locality and reuse at the fine grain (shared memory and/or scratchpad) and the coarse grain (global mem-

ory), and there is significant interplay between them. Tiling is a widely used technique that has been developed to manage this and 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.

A.3.1 Hybrid hexagonal/classical tiling

The HHC compiler [19], 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 B.4.a, and we view a "row of hexagons" as those whose (leftmost) corners have the same value of t. Each row is completely 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 A.3) where each hexagon on the outer two (t-k) dimensions now becomes a "prism" extending along the i dimension. The nature of the stencil dependences precludes directly blocking this prism. Rather, a *classic time-skewing* has to be applied: in the figure this is illustrated by the oblique hexagonal faces. 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 structure of the code remains the same. The idea is extended to higher dimensions, where the prisms become "slabs" and "hyperslabs" and the sequential loop iterating over tiles becomes a nested loop.

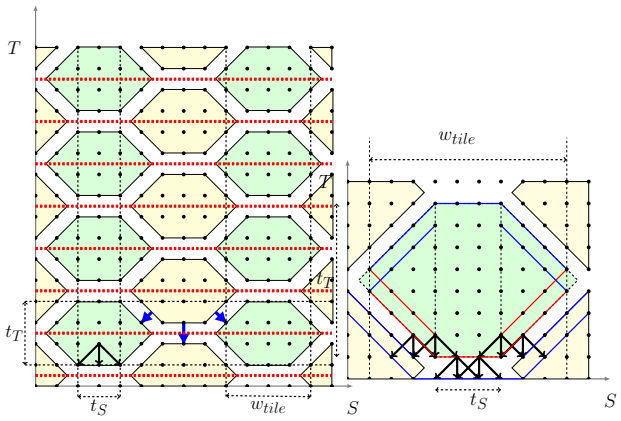


Figure A.2: Hexagonal tiling for 1D stencils: the $S \times T$ iteration space is partitioned into hexagons (left). The Jacobi 1D inter and intra-tile dependencies are illustrated as blue and black arrows, respectively. There are two kinds of "tile rows" colored green and yellow. The tiles in each row 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).

Using hybrid-hexagonal classical tiling [19] we partition such a stencil program in tasks that can be effectively mapped to a GPU. The specific details of the schedule are beyond the scope of this paper, but we give a simplified explanation when we develop our model (Section A.4). As typical for GPGPU, initial data is first stored at the CPU and then transferred to the GPU, subsequently a sequence of *kernel calls* is issued on the CPU to perform the actual computation on the GPU, and finally the result is moved back to the CPU. Further specifics of our execution model are as follows.

- Each kernel call invoked from the host is called a *wavefront* and consists of a set of threadblocks.
- The iteration space of each threadblock in a kernel is either a single *tile* or a sequence of tiles, each one being a bounded polytope with some parameterized size. These we call the tile size parameters.
- Each tile is executed in parallel by the threads within a threadblock, that *collectively*
  - fetch some data from global to shared memory
  - do some computation to update data in shared memory (and registers)

  - write results back from shared to global memory

- Within each tile, some of the dimensions may be parallel (i.e., carry no dependences) in which case, they may be used as the "thread dimension", possibly with additional synchronization. Otherwise, it may be necessary to apply a second level of transformation/skewing to expose such parallel dimensions.

- Independently of this parallelization, each tile may be divided into *subtiles,* possibly to improve the register access, or to reduce synchronization overheads.

- If subtiling is applied, then the iterations in the subtile are executed sequentially by a thread, and the parallelization is *across* subtiles, i.e., subtiles are executed in parallel by multiple threads. The innermost level always consists of iterations that are executed by a single thread within the threadblock between successive synchronizations.

A.3.2 Details of the HHC compiler

HHC generates highly tuned code for specific stencil pattern, problem sizes, and tile size parameters, taking advantage of specific 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 [52]. Independently of the tiling scheme, PPCG automatically simplifies generated loop bounds and index expressions, taking into account problem sizes and tile-size parameters.

When generating HHC tiled code, the effectiveness of PPCG's specialization can be largely improved by unrolling the global⇔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

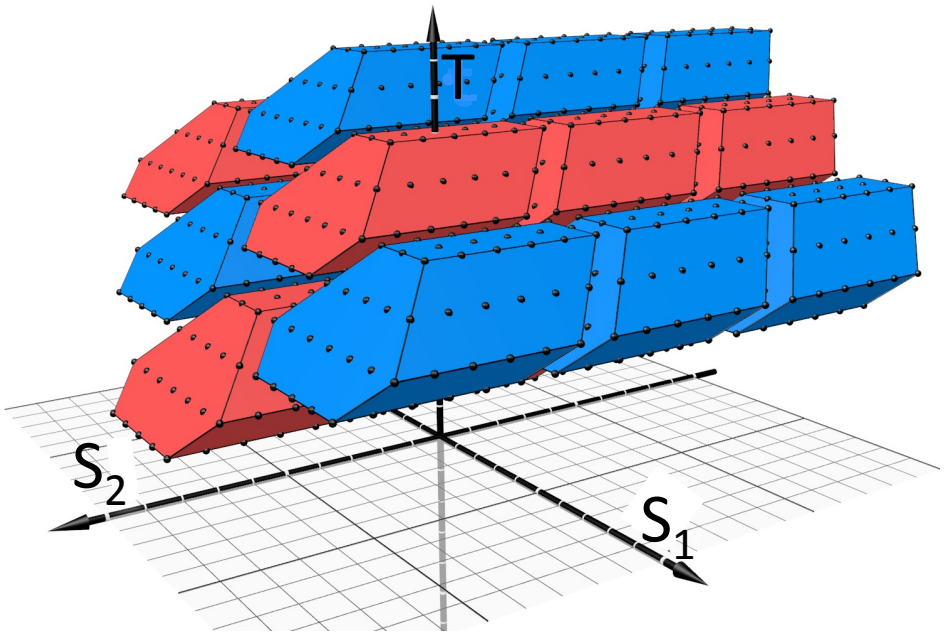


Figure A.3: Hybrid-hexagonal tiling for 2D stencils.

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.

A.4 Execution Time Model

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 A.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.

Table A.1: List of the parameters. E/C stand for elementary/composite, and S/H/P for software/hardware/problem type; M_{io} is measured in 4-byte words.

Name	Type	Description
$\mathtt{S_i}$	EP	i-th space dimension
$\mathtt{T}$	EP	time dimension
$t_{\mathrm{S_i}}$	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)
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)

A.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 (of the outer two loops), is tiled into hexagons with a base of size t_S and a height of size t_T (see Figure B.4.a). In the following we suppose that t_T is even, since the HHT compiler only supports this case. Wavefronts are visualized with red dashed horizontal lines. The dependences allow computing these wavefronts in a sequential manner from bottom to top. Figure B.4.b shows more details within the tiles.

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

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

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

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

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, $w_{\text{tile}}()$, 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 B.4.b). Adding these, we get

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

Furthermore, the distance between two consecutive tiles in a wavefront, called *pitch*, can be derived as

$$\text{pitch}() = w_{\text{tile}}() + t_S + 2 \tag{A.5}$$

Ignoring the fact that there may be one more or less tile in alternate wavefronts, the number of tiles in a wavefront (i.e., the width of a wavefront) is

$$w(i) = \left\lceil \frac{S}{\text{pitch}()} \right\rceil + \epsilon' \approx \left\lceil \frac{S}{2t_s + t_T} \right\rceil, \tag{A.6}$$

where ϵ' is 1 or 0. Since this is independent of i, the summation in (A.2) can be simplified to yield

$$T_{alg}() = N_w() T_{\text{tile}}(k) \left\lceil \frac{1}{n_{SM}} \left\lceil \frac{w()}{k} \right\rceil \right\rceil + N_w()\, T_{\text{sync}}. \tag{A.7}$$

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 B.4.b). Collectively, this data comes from from *two* other (blue) line segments in a neighboring south-east or south-west tile. Note also that these data are needed to compute the line segments situated most closely to the hexagon's south-east and south-west facets. This is visualized in Figure B.4.b where the points in red require the points in blue. 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 B.4.b. Therefore, for the total input/output tile memory footprint we obtain

$$m_{io}() = m_i() + m_o() = 2(t_S + 2t_T). \tag{A.8}$$

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

$$m'() = m_{io}()L + 2\tau_{sync} = 2(t_S + 2t_T)L + 2\tau_{sync}. \tag{A.9}$$

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).

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

$$\begin{aligned} c() &= 2 \sum_{x=t_S, step=2}^{w_{\text{tile}}()} \left(\left\lceil \frac{x}{n_V} \right\rceil C_{iter} + \tau_{sync} \right) \\ &= 2C_{iter} \sum_{x=t_S, step=2}^{w_{\text{tile}}()} \left\lceil \frac{x}{n_V} \right\rceil + t_T \tau_{sync}. \end{aligned} \tag{A.10}$$

Combining (A.9) and (A.10), the total time to process a tile is

$$T_{\text{tile}}() = m'() + c(). \tag{A.11}$$

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 $c() = t_T(C_{iter} + \tau_{sync})$. 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, $n_V \ll w_{\text{tile}}()$.

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:

$$1 < k \leq \min\left\{ \left\lfloor \frac{R_{SM}}{R_{\text{tile}}} \right\rfloor, \left\lfloor \frac{M_{SM}}{M_{\text{tile}}} \right\rfloor \right\}. \tag{A.12}$$

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

$$T_{\text{tile}}(k) = m'() + c() + (k-1)\max\{m'(), c()\}. \tag{A.13}$$

A.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.

Total execution time of Jacobi 2D

As illustrated in Figure A.3, each hexagon from Figure B.4 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 A.3, A.4, A.6 and A.7 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}.

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 A.3). 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 A.3). 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 (A.8) multiplied by the tile's length t_{S_2}. Hence

$$m_i() = m_o() = t_{S_2}(t_{S_1} + 2t_T). \tag{A.14}$$

For the corresponding time we obtain respectively

$$m^{'}() = (m_i() + m_o())L + 2\tau_{\text{sync}} \tag{A.15}$$

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

$$\begin{aligned} c() &= 2 \sum_{x=t_{S_1},step=2}^{w_{\text{tile}}()} \left(\left\lceil \frac{xt_{S_2}}{n_v} \right\rceil C_{iter} + \tau_{\text{sync}} \right) \\ &= 2C_{iter} \sum_{x=t_{S_1},step=2}^{w_{\text{tile}}()} \left\lceil \frac{xt_{S_2}}{n_v} \right\rceil + t_T \tau_{\text{sync}}. \end{aligned} \tag{A.16}$$

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,

$$T_{prism}(k) = \begin{cases} k = 1 \quad : & \left(m^{'}() + c()\right) \left\lceil \frac{S_2 + t_T}{t_{S_2}} \right\rceil \\ k > 1 \quad : & m^{'}() + k\zeta() \left\lceil \frac{S_2 + t_T}{t_{S_2}} \right\rceil . \end{cases} \tag{A.17}$$

Plugging this into (A.2) and simplifying we get

$$T_{\text{alg}}() = N_w T_{\text{sync}} + N_w T_{\text{prism}} \left\lceil \frac{1}{n_{SM}} \left\lceil \frac{w()}{k} \right\rceil \right\rceil . \tag{A.18}$$

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

$$m_i() = t_{S_2}(t_{S_1} + 2t_T) \tag{A.19}$$

$$M_{\text{tile}}() = 2(t_{S_1} + t_T + 1)(t_{S_2} + t_T + 1) \tag{A.20}$$

A.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.

Total execution time of Jacobi 3D

Each hexagon from Figure B.4 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 A.3 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 respect to the t_{S_1}-axis (w), are identical to the Jacobi 1D case, and respectively are

$$N_w() \approx 2 \left\lceil \frac{T}{t_T} \right\rceil, \tag{A.21}$$

$$w_{\text{tile}}() = t_{S_1} + t_T - 2, \tag{A.22}$$

and

$$w() \approx \left\lceil \frac{S_{s_1}}{t_{S_1} + t_T} \right\rceil. \tag{A.23}$$

The total execution time is hence similar to the one of Jacobi 1D A.7 with the unique difference that the term T_{tile} is substitute now by T_{slab}.

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 A.3) 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

$$N_{sslabs} = \left\lceil \left(\frac{S_2 + t_T}{t_{S_2}} \right) \left(\frac{S_3 + t_T}{t_{S_3}} \right) \right\rceil \tag{A.24}$$

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 (A.8) multiplied by the tile's length in the S_2 and S_3 axes. Hence

$$m_i() = m_o() = t_{S_2} t_{S_3} (t_{S_1} + 2t_T). \tag{A.25}$$

For the corresponding time we obtain respectively

$$m^{'}() = (m_i() + m_o())L + 2\tau_{\text{sync}}. \tag{A.26}$$

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

$$V_{\text{tile}}() = t_{S_2} t_{S_3} \frac{t_T (w_{\text{tile}}() + t_{S_1})}{2}. \tag{A.27}$$

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

$$\begin{aligned} c() &= 2 \sum_{x=t_{S_1}, step=2}^{w_{\text{tile}}()} \left(\left\lceil \frac{x t_{S_2} t_{S_3}}{n_V} \right\rceil C_{iter} + \tau_{\text{sync}} \right) \\ &= 2 C_{iter} \sum_{x=t_{S_1}, step=2}^{w_{\text{tile}}()} \left\lceil \frac{x t_{S_2} t_{S_3}}{n_V} \right\rceil + t_T \tau_{\text{sync}}. \end{aligned} \tag{A.28}$$

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

$$T_{\text{slab}}(1) = (m^{'}() + c())N_{sslabs}. \tag{A.29}$$

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

$$T_{slab}(k) = m^{'}() + k \max(m'(), c())N_{sslabs}. \tag{A.30}$$

Plugging this into (A.2) and simplifying yields

$$T_{\text{alg}}() = N_w T_{\text{sync}} + N_w T_{\text{slab}}(k) \left\lceil \frac{1}{n_{SM}} \left\lceil \frac{w()}{k} \right\rceil \right\rceil. \tag{A.31}$$

All the equations developed here hold true for all stencil codes generated by HHC compiler. However, the parameter,, C_{iter}, that 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 A.7 holds for wavefront parallel codes.

A.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*.

A.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, 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 requirements 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 $n_{\mathrm{thr,i}}$. Each unique combination of an experiment with the parameters t_{T}, t_{S_1}, t_{S_2} and $n_{\mathrm{thr,i}}$ is called a *data point*. Hence, our baseline-experiment set contains 850 data points for each experiment.

Table A.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 A.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}

The HHC compiler generates a separate program (code) for every data point (it cannot produce codes with parametric tile sizes), a total of $850 * (80 + 48) = 108,800$ 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 A.2 shows such parameters for our two platforms. We also need values of the remaining parameters L, τ_{sync}, T_{sync} and C_{iter}, that are not available. We conduct the following microbenchmark experiments to gather these values.

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 A.3.

Table A.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}

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 size and tile size parameters 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 for C_{iter} for a benchmark-machine combination. The resulting values are given in Table A.4.

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

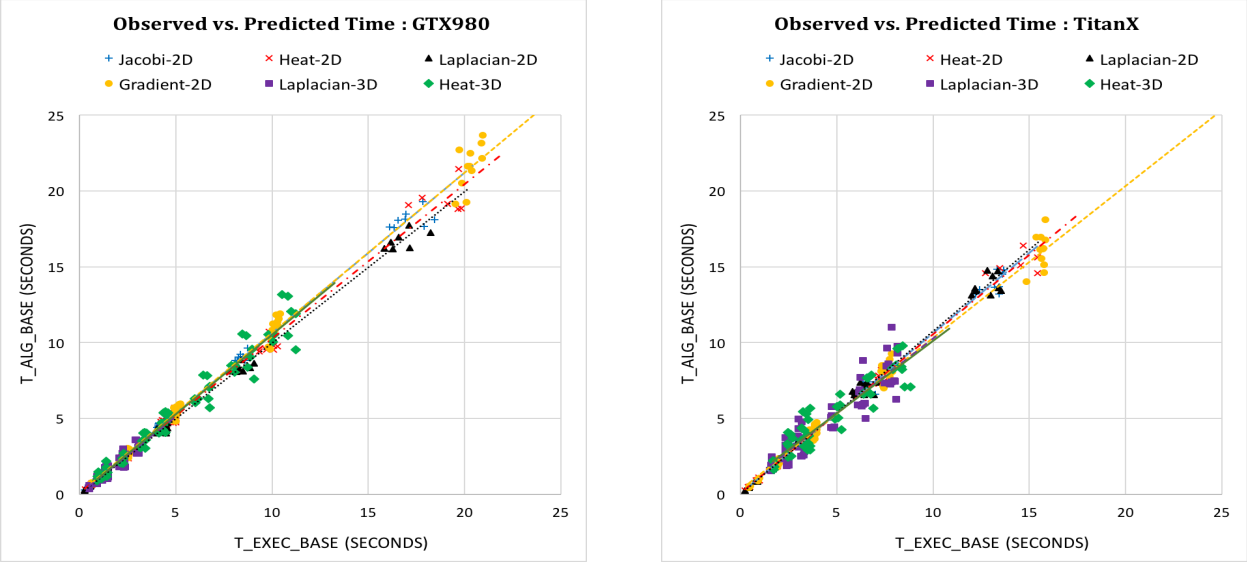


Figure A.4: Observed execution time vs. model predicted time on GTX 980 and on Titan X, where $T_{\text{alg\_base}}$ denotes the model predicted time and $T_{\text{exec\_base}}$ denotes the measured execution time for the baseline experiments.

A.5.2 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 results in 8500 points. We compute the root mean square error (RMSE) and observe that the RMSE ranges within 45%–200% over this whole set. However, the model is designed such that it makes optimistic prediction of execution time. Hence, the inaccuracies were expected and inevitable given the fact that our model is not designed to predict the performance of inefficient codes.

However, restricting 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 in model prediction is less than 10% for all stencils on both GPUs. Figure A.4 shows the correlation between the predicted and the measured time over the top performing points.

A.6 Tile size optimization

The model we developed in Section A.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.

A.6.1 The optimization problem

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

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

where M_{tile} and k are functions of the tile sizes. We require t_{T} to be even, as necessary for hybrid-hexagonal tiling [19], 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 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

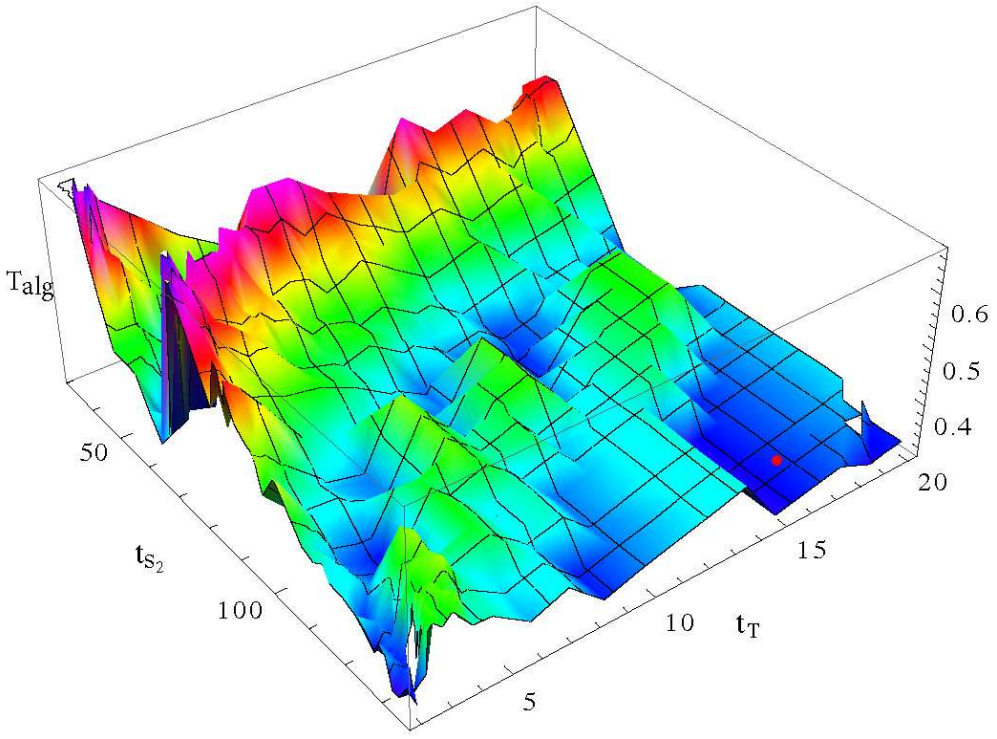


Figure A.5: T_{alg} for Heat2D and GTX 980 as a function of t_{T} and t_{S_2} and with t_{S_1} fixed at 8. The red dot shows $T_{\text{alg\_min}}$, the point of minimum over all T_{alg}.

function with respect to $\lceil x \rceil$. Figure A.5 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 [123] and solved it using several non-linear solvers, including commercial ones. The best results were obtained using the open-source solver Bonmin [124]. 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 don't 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 A.32 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 A.32
- 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.

A.6.2 Validation of the results

For each combination of an algorithm (Jacobi2D, Heat2D, Lapacian2D, or Gradient2D), hardware (GTX 980 or TitanX), and values of $S \in \{4096, 8192\}$ and $T \in \{1024, 2048, 4096, 8192, 16384\}$ we solved the problem (A.32) for these values in order to find tile sizes that minimize the running times, solving in total 80 optimization problems. We then measured the running times for these best predicted parameters and compared them with the best running times that we have found using manual (but not exhaustive) exploitation of the search space by running almost $60,000$ instances of the codes in total. For the i-th optimization problem we define the optimization error as $\mathrm{opt\_er}_i = T_{\mathrm{exec}}(opt_i) - T_{\mathrm{exec}}(exp_i)$, where opt_i and exp_i refer to the running times corresponding to the optimal tile sizes as predicted by the model and to the best known tile combination found using experiments, respectively. We then define two values for the total error,

$$\mathrm{Err}_{\max} = \sum_i \max\{0, \mathrm{opt\_er}_i\}$$

and

$$\mathrm{Err}_{\mathrm{sign}} = \sum_i \mathrm{opt\_er}_i.$$

A.6.3 Model Predicted tile sizes

The search space for model prediction experiments consists of 18,000 data points each for a benchmark-GPU combination. This search space is build such that it subsumes all the 850 points in the baseline experiments.

Our analytical model is implemented in a script for tile size selection. Input to this script includes machine parameters (table 1 and 2), C_{iter} value of the benchmark and problem size parameters S and T. The equations (equation numbers) are implemented to give T_{alg} value for a data point. Using these input values, the script first finds the predicted optimal data point in the search space. Precisely, we keep track of the minimum of T_{alg} possible over all the points in the search space, call this $T_{\text{alg\_min}}$. The script then searches for more data points in the vicinity of predicted optimal using $T_{\text{alg\_min}}$. This gives us a set of predicted optimal tile sizes and their respective run-time predictions, $T_{\text{alg\_pred}}$.

A.6.4 Experimental results

We observe that the set of new tile sizes perform better than those obtained in baseline. Figure A.6 shows execution and model predicted times for Gradient-2D stencil with a problem size of $S_1 = 8192$, $S_2 = 8192$ and $T = 8192$. Clearly, the model predicts optimal tile sizes that outperform the best of baseline. 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 17% improvement in performance. This model predicted tile size was not explored in our set of 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 A.7 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

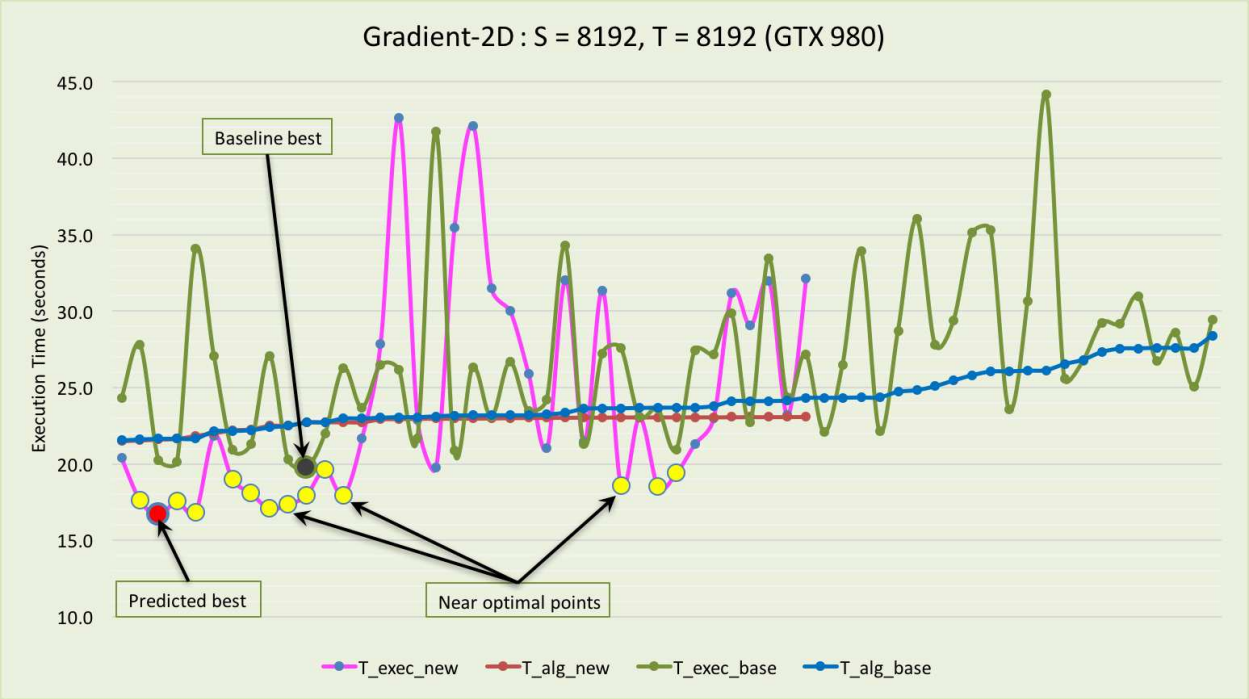


Figure A.6: Predicted tile size performance of Gradient-2D for $S_1 = 8192$, $S_2 = 8192$ and $T = 8192$ on GTX 980.

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.

A.7 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.

Our results also suggest that we should revisit the "conventional wisdom" that says that an optimal strategy of a tiling is to choose the "largest possible tile size that fits" i.e., its memory footprint matches the available capacity. First of all, this falls into the trap that it precludes overlapping of computation and communication (the "hyperthreading effect"). But this can be avoided

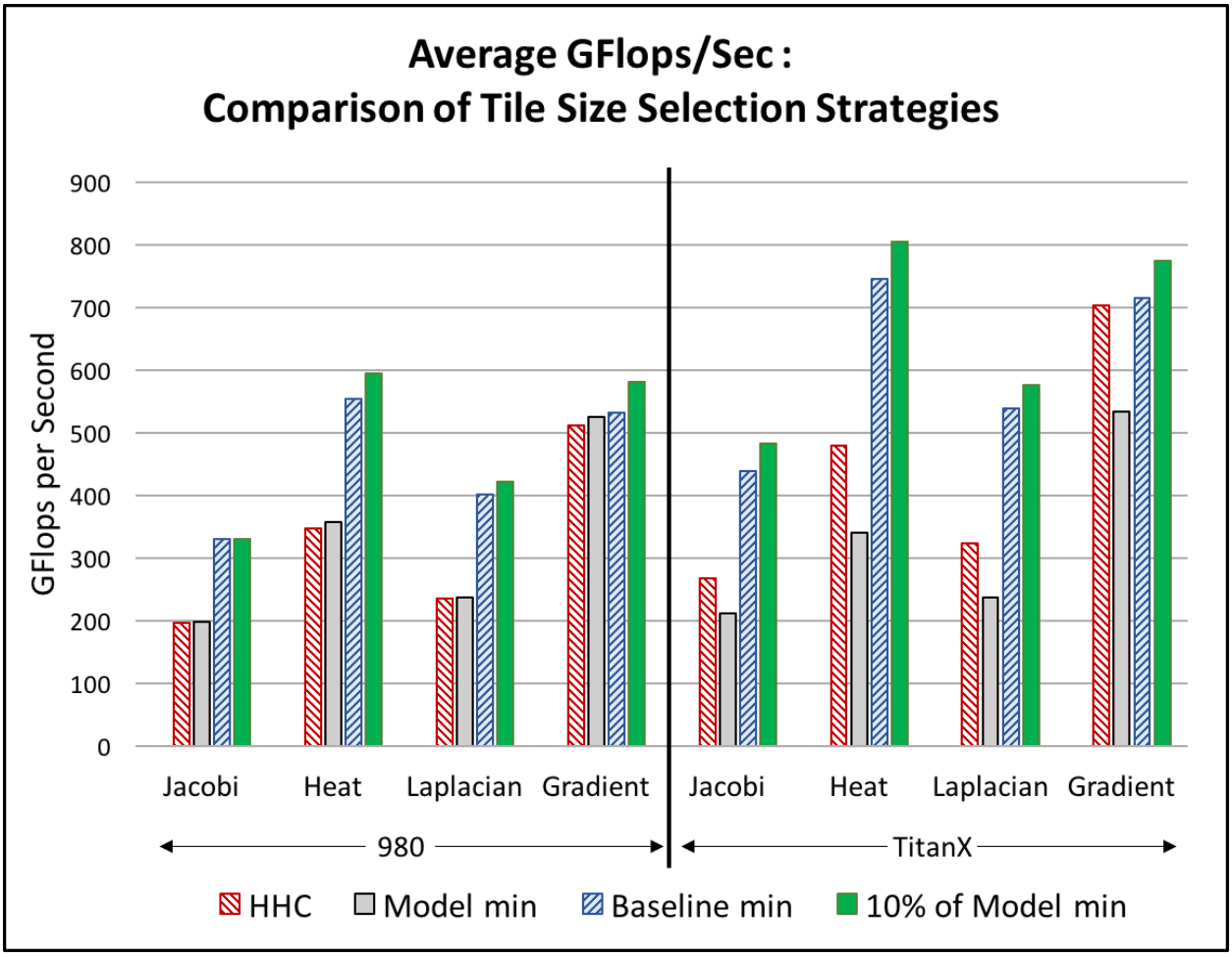


Figure A.7: Average (over 10 problem sizes) GFlops/Sec achieved by different tile size selection strategies for 2D stencils.

by modifying explicitly accounting for hyperthreading. Indeed, our GPU platforms preclude such large size by disallowing the data footprint of a thread block to exceed *half* the shared memory capacity.

Thus the hyperthreading-adjusted "conventional wisdom" would still seek to maximize tile volume subject to the half-capacity constraint—the best strategy is the largest tile volume for the given footprint. Our model and experimental data suggests otherwise—an even higher hyperthreading factor is turning out to yield the best performance. We still don't know why, and it is the subject of our ongoing investigation.

A.7.1 More Analysis/ Open Questions

Some of the questions that can be answered using such analytical models are:

- What happens when input parameters change?
- What happens when different number of processors is used?
- What is the largest possible problem size on a given architecture?
- When does efficiency drop?

Appendix B

Energy Modeling and Optimization for Tiled Nested-Loop Codes

We develop a methodology for modeling the energy efficiency of tiled nested-loop codes running on a graphics processing unit (GPU) and use it for energy efficiency optimization. We assume that a highly optimized and parametrized version of a tiled nested-loop code—either written by an expert programmer or automatically produced by a polyhedral compilation tool—is given to us as an input. We then model the energy consumption as an analytical function of a set of parameters characterizing the software and the GPU hardware. Most previous attempts at GPU energy modeling were based on low-level machine models that were then used to model whole programs through simulations, or were analytical models that required low level details. In contrast, our approach develops analytical models based on (i) machine and architecture parameters, (ii) program size parameters as found in the polyhedral model and (iii) tiling parameters, such as those that are chosen by auto- or manual tuners. Our model therefore allows efficient optimization of the energy efficiency with respect to a set of parameters of interest. We illustrate the framework on three nested-loop codes: Smith-Waterman, and one-dimensional and two-dimensional Jacobi stencils, and analyze the accuracy of the resulting models. With optimal choice of model parameters the RMS error is less than 4%. Two factors allow us to attain this high accuracy. The first is domain-specificity: we focus only on tile-able nested-loop codes. The second is that we decouple the energy model from a model of the execution time, a known hard problem.

B.1 Introduction

Energy efficiency has been recognized as one of the biggest challenges in the roadmap to higher performance (exascale) systems for a number of reasons including cost, reliability, energy conservation, and environmental impact. The most powerful computers today consume megawatts

of power, enough to power small towns, and at cost of millions per year. And those estimations do not include the cost of cooling, which might be almost as high as the cost of computing itself [113]. In addition, the cost of building a power provisioning facility ranges at $10-22 per deployed IT watt [114] and every $10\,^{\circ}\mathrm{C}$ temperature-increase results in a doubling of the system failure rate, thereby reducing the reliability of HPC systems [115]. Designing accurate models for energy efficiency can help better predict the power and energy requirements of an application and aid developers optimize the parameters of their codes for better energy efficiency on HPC systems.

The goal of this paper is to introduce a new approach for modeling the energy cost as an analytical function of tunable software parameters in a way that is both simple and accurate. Having such a model will allow the energy efficiency to be optimized with respect to (a subset of) the tunable parameters by solving the corresponding analytical optimization problem.

We target with our modeling approach tiled nested-loop code segments, which are the most compute-intensive portions of many application codes and which also allow a high degree of parallelism. In order to be more specific, we focus in our analysis on a subclass of the tiled nested-loop codes called *dense stencils*, which occur frequently in the numerical solution of PDEs and in many other contexts such as high-end graphics, signal and image processing, numerical simulation, scientific computing, and bioinformatics. We chose stencils for our case studies since that would allow us to model the entire class in a hierarchical way with a single generic model representing the whole class, while model parameters that are stencil-dependent have to be separately specified for each stencil of interest to complete its model. (However, the approach is applicable to any other class of nested-loop codes that allows tiling.) We will completely develop and validate the detailed models (including the stencil-dependent parameters) of three specific stencils. Models for other stencils can be developed in a similar way with relatively small amount of extra work.

In order to efficiently optimize stencils on accelerators, we aim to represent the amount of energy consumed as an analytic function of the software parameters. We assume that the input codes have been analyzed and optimized with respect to parallelism and data-access efficiency by appropriate skewing and tiling transformations, say by a polyhedral code generator.

Our specific contributions are as follows.

- Our energy model predicts energy efficiency by analyzing source code only, unlike other approaches [93, 95] that rely on parameters computed by running benchmarks for each individual code. We do use micro-benchmarks, but they are used to characterize hardware, rather than codes.

- We are not aware of any previous work combining the polyhedral method with energy modeling. Our approach allows optimization of codes that are already very efficient having been significantly improved by applying the polyhedral method and by using advanced tiling strategies such as hexagonal and hybrid tilings [19].

- Our model is very accurate (one version with $\mathrm{RMS\,error} \leq 17.14\%$ and another with $\mathrm{RMS\,error} \leq 4\%$), with similar or higher precision than alternative existing models, e.g., GPUSimPow [116], which are simulation based.

The paper is organized as follows. Section C.5 reviews relevant previous work. Section B.3, provides an overview of the approach. The GPU implementations of tiled nested-loop codes are described in Section B.5. Section B.6 introduces our energy model, and Section B.7 validates it. We conclude with a discussion of future work in Section B.9.

B.2 Related work

GPU power/energy model is a very active area: a recent survey article on the topic [90] cites almost 150 references. We only discuss the relvant work here. The model we present complements Mittal and Vetter [90] by enabling us to find the optimal parameters (i.e., tile sizes) for the energy efficient execution of stencil like programs.

Hong and Kim [91] present a GPU power model to predict the number of optimal GPU cores to achieve the peak memory bandwidth for a kernel. An analytical model is used to predict the execution time [92] which has enabled prediction of the power consumption statically. However, they have predicted the minimum number of cores required for a program to achieve the peak memory

bandwidth of GPU. While this approach may work for memory bandwidth bound programs, it is unlikely to produce better results for compute-bound programs like tiled stencil computations. Our model is much simpler, because our model does not depend on warp and thread level parameters and number of ptx instructions.

Nagasaka et al [93] model GPU power of kernels using performance counters. Lim et al. [94], GPUWattch [95] and GPUSimPow [96] are simulation based power models. McPAT [97] is the basis for Lim et al [94] and GPUWattch [95] uses GPGPUSim [98] to simulate execution time. Simulation and performance counters-based models require execution (or simulation) of the program to predict the power consumption. In contrast, we run our micro-benchmarks only to determine parameters of a GPU architecture, while the power consumption can be predicted for a given program statically without running the program.

There are studies [99, 100] focused on reducing the energy for both CPU and GPU by balancing the load among CPU and GPU. Our study is only focused on modeling the energy consumption of GPUs.

B.3 Stencil Computations

In this section, we describe stencil computations and illustrate them with a couple of simple examples.

In stencil codes, values of array elements are updated iteratively using the values of some of their neighbors according to a fixed pattern. More formally, we will consider stencil codes of the following kind. Let $\mathcal{S} = \{(i_1, \ldots, i_k) \mid s_j \leq i_j \leq S_j, i_j \in \mathbb{N} \text{ for } j = 1, \ldots, k\}$ is a k-dimensional *space index set* and $\mathcal{T} = \{t, \ldots, T\}$ is a *time index set*. Then, given two sets $\mathcal{N}$ and $\mathcal{N}'$ defining the "neighborhood" of any point in terms of a pattern of relative coordinates and a coefficient w_a associated each element $a \in N \cup N'$, a stencil code defines an iterative evaluation of the following weighted sums

$$A^t(s) = \Big(\sum_{a \in \mathcal{N}} w_a \otimes A^t(s+a) \Big) \oplus \Big(\sum_{a \in \mathcal{N}'} w_a \otimes A^{t-1}(s+a) \Big) \oplus \; c, \text{ if } t > 0, \tag{B.1}$$

$s \in \mathcal{S}$, $t \in \mathcal{T}$, where the summation is with respect to an associative commutative operation $\oplus$, which forms a semiring together with the multiplicative operation $\otimes$, e.g., $(\oplus, \otimes)$ can be $(+, \times)$ or $(\max, +)$, and c is an optional constant. It is further assumed that appropriate values are given for the "initial value," (when $t = 0$) and the "boundary values" (when the points $s + a$ in either of the summations above fall outside $\mathcal{S}$). Furthermore, the elements of the set $\mathcal{N}$ must be such that the computation is "explicitly defined" (see [122]). The definition may be extended to (i) allow a computation to depend on values at more than the previous time step, and (ii) allow the coefficients to be computed values, rather than compile-time constants, provided the evaluation of these coefficients is also described by a stencil computation. Such stencils are usually implemented as nested loops with the loop body evaluating the rhs of (C.3) and storing it in a data array. We call equation (C.3) *kernel* of the stencil.

In this paper we consider two speical cases: the Smith-Waterman stencil (where $\mathcal{N} = \{-1\}$ and $\mathcal{N}' = \{-1, 0\}$). and one dimensional and two dimensional Jacobi stencils (where $\mathcal{N}$ is empty and the elements of $\mathcal{N}'$ are vectors with components in $\{-1, 0, 1\}$), as described later in this section.

B.3.1 Smith-Waterman Algorithm (SW)

Given two molecular sequences $A = a_1 a_2 \ldots a_N$ and $B = b_1 b_2 \ldots b_M$, the Smith-Waterman algorithm (SW) solves the problem of finding an optimal alignment of A and B with respect to a given similarity measure $s(a_i, b_t)$. Using the dynamic programming methodology, SW constructs an $N \times M$ table H that identifies the optimum alignment. Since the alignment problem does not have an explicit time dimension, we have considered one of the two dimensions of the dynamic programming table, say M, as a time dimension. Then, $0 \leq i \leq N$, $0 \leq t \leq M$ and a kernel

$$H^t(i) = \max\left\{0, H^{t-1}(i-1) + s(a_i, b_t), H^{t-1}(i) + W, \right.$$
$$\left. H^t(i-1) + W\right\},$$

where W provides a score which depend on whether it's a new gap or extension of an existing gap. SW is tiled using a simple strategy called *orthogonal tiling* [125].

B.3.2 Jacobi 1D Stencil (J1D)

Our next example is of Jacobi 1-dimensional 3-point stencil, referred to as J1D, which is defined by a recurrence where the value at each point at a given time instance is evaluated by taking the average of its neighboring elements along with itself from the previous time step. Specifically, the space index set is $\mathcal{S} = \{1, \ldots, S\}$, the time index set is $\mathcal{T} = \{1, \ldots, T\}$, and the kernel is

$$A^t(i) = \frac{1}{3} \times \left(A^{t-1}(i-1) + A^{t-1}(i) + A^{t-1}(i+1)\right).$$

There are number of ways to tile J1D. We tile J1D using time skewing [126, 127, 56, 57]

B.3.3 Jacobi 2D Stencil (J2D)

The J2D stencil we consider is a 2D equivalent of J1D. It is a 5-point stencil with a space index set $\mathcal{S} = \{(i, j) \mid 1 \leq i \leq S_x,\ 1 \leq j \leq S_y\}$, a time index set $\mathcal{T} = \{1, \ldots, T\}$, and a kernel

$$A^t(i,j) = 0.2 \times (A^{t-1}(i,j) + A^{t-1}(i-1,j) +$$
$$A^{t-1}(i+1,j) + A^{t-1}(i,j-1) + A^{t-1}(i,j+1)).$$

The iteration space is a cuboid where a plane of values is computed using the values from the previous plane. J2D can be tiled using various techniques, we use the state of the art *hybrid hexagonal-classical* tiling (HHC) [19].

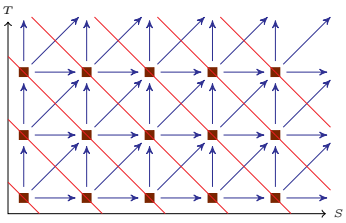


Figure B.1: Orthogonal tiling for SW stencil. Small dark red squares represent tiles. Data flow of the tiles are shown in blue arrows. Red lines along the northwest-southeast diagonals represent wavefronts. The tiles that belong to the same wavefront can be processed in parallel. Wavefronts are executed sequentially from southwest to northeast corner.

B.4 Tiling of Stencils

We now describe the standard parallelization strategies for stencil programs, illustrating them on the three common dependence patterns in our examples: SW, J1D and J2D. Some of these strategies have been implemented in sophisticated olyhedral compilers [51, 19] and possibly in other automated tools. However, our goal is in modeling the energy costs of such strategies, not in generating codes.

In order to increase the amount of parallelism and improve the locality of memory accesses, it is often useful to divide the set of all computations of a code into groups called *tiles*. Tiles are considered atomic, i.e., all the data needed to perform the computations in a tile can be made available before the computation begins and there are no cyclic data dependencies between tiles (tiling is *legal*). We also note that in order to obtain the best performance, tiling is often applied hierarchically at multiple levels.

Next we describe tiling strategies for SW, J1D, and J2D.

B.4.1 Orthogonal tiling of the SW stencil

A cell in the dynamic programming table H of SW is data-dependent on the west, south and southwest cells. Hence, *orthogonal tiling* (one with tiles shaped as rectangles, and whose boundaries are orthogonal to the canonic axes) is legal and will be used for this dependence pattern. All the tiles along a northwest-southeast wavefront, see Figure B.1, can be computed in parallel. A tile depends on the neighboring tiles to the west, south, and southwest. The wavefronts are ex-

ecuted sequentially from southwest to northeast. This execution order of wavefronts may suffer from pipeline fill-flush stages, especially if tiling is applied at multiple levels. This is inherent to SW due to the nature of the dependences (the tile or iteration point at the southwest corner must be computed before computing any other tile). Therefore, more sophisticated strategies like diamond tiling [72] and hexagonal tiling [19] cannot be used for SW.

B.4.2 Tiling and parallelization of the J1D stencil

To implement Jacobi 1D efficiently, we cannot use orthogonal tiling, but there are several, well-known strategies. The first of these is the common strategy of tiling only the space dimensions, and performing a "halo exchange" at every iteration of the time loop. Although widely used, it is bit slow—it makes the code memory bound, although the original program does $\Theta(NT)$ operations that touch $\Theta(N)$ memory locations, which is clearly compute bound. Indeed, there is a common misconception even among many "experts" that stencil computations are inherently memory bound. Many strategies are available for making stencil computations compute bound: *time skewing*, *diamond* tiling, *hexagonal* tiling, or even more recent extensions to this called *hybrid* hexagonal-classical (HHC) tiling [126, 127, 56, 57, 51, 72, 19]. We describe these below.

Time skewing

Time skewing is a special case of parallelpiped tiling [126] proposed by Wonnacott [127, 56] and studied by Andonov et al. [57]. It is now implemented as an automatic transformation in PLUTO [51].

The original iteration space (Figure B.2, left) does not allow tiling into orthogonal tiles, because each tile would need some elements from the tile above it and some from the tile below it, leading to a cyclic dependence, thereby violating the legality condition.

Using a skewing transformation $(i, t) \rightarrow (i + t, t)$, one can transform the iteration space (see Figure B.2, right) in order to make it tileable. As a result of this transformation, each tile is dependent only on the tile to its left and on the tile below it, breaking the cyclic dependency. Now, all the tiles along the 45^o northwest-southeast *wavefront* can be computed simultaneously, i.e., each

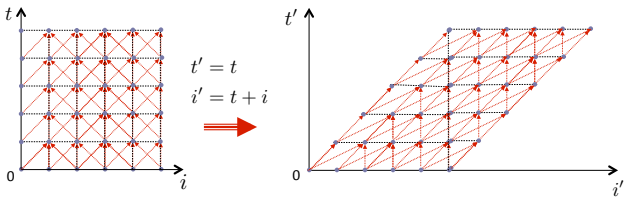


Figure B.2: The original computation space (left) and the same space after the chosen transformation (right).

kernel call launches a set of tiles that are along the wave-front, thus reducing the number of kernel calls.

There are many transformations that can result in tilable codes, and they can be further distinguished by the amount of data communication and the amount of parallelism they offer. One can find an optimal transformation using the polyhedral method and some of the publicly available tools based on it. We used PLUTO [51], a polyhedral compiler, to find the optimal transformation that minimizes the communication between the tiles for J1D and which transforms the iteration space shape from rectangle to parallelogram.

Hexagonal

Time skewing suffers from potentially large pipeline fill-flush latency, and in order to overcome this, some authors proposed an alternate strategy called *diamond* tiling [72]. However, we will not discuss it here since it is a special case of the more sophisticated HHC tiling [19].

In this approach, the original iteration space (Figure B.2, left) is directly tiled into hexagonal tiles (Figure B.4). The wavefronts are horizontal rows of hexagons and the i^{th} kernel computes the tiles from the i^{th} such wavefront (shown as red dashed lines). Parallelization is again by wavefronts where the i^{th} wavefront consists of the tiles that intersect the i^{th} horizontal (dashed red) line. The dependencies between the iteration points are visualized by black arrows, while the dependencies between tiles are illustrated by thick arrows. These dependencies allow to compute the wavefronts in a sequential manner from bottom to top. Note the alternation–in odd indexed wavefronts only yellow colored tiles are executed, while in even indexed wavefronts only green colored tiles are executed. Although the special shape may indicate that there is a high control overhead, the authors show that this is currently the best approach for GPU parallelization of stencils.

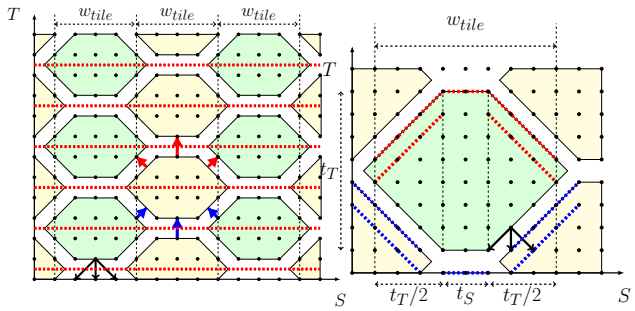


Figure B.3: The computational space in hexagonal tiling is partinionned in hexagons (left). On the right side we zoom on and visualize the input/output data for a given tile. The data read from the global memory is blue colored, while the data written to the global memory is red colored.

B.4.3 Tiling and parallelization of the J2D stencil

When a Jacobi stencil has additional dimensions, we have two choices, based on what is done to the outer two loops—the time and the first space dimension. The first option is to tile the outer two dimensions using time skewing, and the second is to use hexagonal tiling for the two outer loops. The tiling this yields "prisms" whose "cross-section" is either the rectangular tile of Figure B.2 or the hexagonal tiles of Figure B.4. The third dimension (inner space loop) comes out of the plane, and broken into smaller tiles. Because of the nature of the dependences, it is necessary to use time skewing of the inner space dimension. To visualize this, imagine either Figure B.2 or Figure B.4 as the cross section of a grid of prisms coming out of the paper. We then skew the hexagonal faces of the prism "out of the plane" at an angle dictated by the dependences. Grosser et al [19] present a very convincing argument that HHC tiling outperforms the simple time skewing applied to all the three loops. For this reason, we only analyze this strategy for J2D.

B.5 GPU implementation of tiled stencils

We now describe how the above stencils are implemented on GPUs.

B.5.1 Overview of GPU architecture

A GPU has an array of Streaming Multiprocessors (SMs) each one with a set of SIMD Vector Processors (CUDA cores). The number of CUDA cores can vary from 8 to 192 depending on

the GPU. All SMs share an off-chip global memory and have a private memory (ironically called "shared" memory) and a register file. The programming model consists of a *grid* of *thread blocks*. The threads of a thread block execute concurrently on one SM and multiple thread blocks may execute concurrently on one SM (depending on available shared memory and register resources). As thread blocks terminate, new thread blocks are launched [128]. The computations in a thread block are expected to be independent of those in other thread blocks. Hence, the execution order of the thread blocks in a grid can be arbitrary. Within a thread block, all the threads can access shared memory collaboratively and explicitly synchronize to have a uniform view of shared memory data among threads.

B.5.2 GPU implementation

We have seen above that diverse strategies for parallelization are available for different stencil computations, and this topic is the subject of active research. Regardless of the specific tiling and parallelization strategy chosen, the GPU implementation has some common structure. (Note that the scope of this paper is restricted to a single-GPU implementation.)

The iteration space tiles may have many possible shapes: rectangles (as in orthogonal tiling), parallelograms (time skewing), rhombi (diamond tiling), or hexagons, or higher dimensional extensions of these shapes. Regardless, legality condition for tiling ensures that a parallel wavefront schedule can be chosen. Sometimes the wavefronts are perpendicular to one of the canonic dimensions, at others, the tiles need to be skewed (oblique wavefronts). We set l as the number of wavefronts and let w_i denote the set of tiles in the i-th wavefront. Its cardinality $|w_i|$ is the number of tiles in the i-th wavefront.

The code, as described in Algorithm 1, may be written manually or generated automatically, and consists primarily of a sequential loop with exactly l iterations executing on the host. Outside this loop there is, of course, a copy of the input data to the GPU and of the output data from the GPU.

Algorithm 1: GPU Execution of a tiled stencil computation

The iteration space is tiled, the tiles are partitioned into l wavefronts $w_1, \ldots, w_l$ such that all tiles in a wavefront can be executed independently in parallel.

1. The host copies the input data from the CPU to GPU's global memory.
2. The host executes a loop with l iterations, the i-th iteration making a kernel call to execute $|w_i|$ tiles. The GPU runtime system schedules these for execution on the physical GPU resources. Computing one tile involves three steps:
   (a) transferring data from global to shared memory,
   (b) computation, and
   (c) writing the result back from shared to global memory.

   If multiple tiles are executing "simultaneously" on the same SM, then computation (b) of one tile can overlap with data transfers (a) and (c) for another. Our cost model accounts for this.
3. At the end of the l-the kernel call, the result is copied from the GPU's global memory to the CPU.

In the i-th iteration, a GPU kernel is launched to execute the w_i tiles of that wavefront. The code that executes each tile (Step 2.b) will have fine grain, thread-level parallelism, corresponding to the fine-grain SPs units within each SM. This parallelization can be done in one of three ways. First, one or more of the loops iterating over the points in a tile may be parallel, and can be directly mapped to the threads. Second, a skewing transformation may make one or more loops parallel. Third, we could use hierarchical tiling (subtiling).

Subtiles (if any) are executed in a similar fashion as tiles, except that data transfers are between shared memory and registers, for subtiles, compared to transfers between global and shared memory, for tiles.

A note about HHC tiling

For this strategy, we use a slightly different interpretation of the terminology of Algorithm 1. Here, the "tiles" in each of the l kernel do not have a bounded size—their memory footprint is a function of the size of the iteration space, and may be larger than the available shared memory

resources of the GPU. As described by the authors, these tiles are actually executed with a special form of subtiling. The tile is partitioned into subtiles, each of which has a small enough footprint, and these subtiles are executed through a set of *sequential* loops executed by each thread, in an "outer-sequential, inner parallel manner." In order to retain consistent terminology, we refer to these "subtiles" as "tiles" in the remainder of this paper.

B.6 Energy model

Our energy model uses parameters of several types as shown in Table B.1. We first describe a generic model applicable to all stencil computations, and then show how to specialize it for the three instances analyzed here. Note that we model the total energy consumed by the co-CPU accelerator itself, accounting neither for the energy to transfer data to/from the host, nor the energy on the host. We assume that the decision to use an accelerator has already been made, and in most uses of our model, these costs are invariant.

B.6.1 Generic model

The total energy consumed by the accelerator in executing a kernel consists of *static energy*, which is the energy consumed even when the device is on but idle, and *dynamic energy*, which is the energy needed to actually do the computation, including data transfers. Clearly the static energy is proportional to the static power and the execution time, hence, $E_{\text{stat}} = P_{\text{stat}} T_{\text{alg}}$.

The dynamic energy is the product of the number of tiles, N_{tiles}, and the amount of energy consumed per tile E_{tile}, i.e., $E_{\text{dyn}} = N_{\text{tiles}} E_{\text{tile}}$, which itself can be subdivided into energy for memory transfers between global and shared memory, between shared memory and registers, energy for doing arithmetic, and energy consumed for synchronization, hence, $E_{\text{tile}} = E_{\text{gs}} + E_{\text{sr}} + E_{\text{ar}} + E_{\text{sync}}$. Parameters E_{gs}, E_{sr}, and E_{ar} are problem dependent and calculated separately for each stencil kernel. During validation of the model, we noticed that E_{sync} is negligible and dropped from the formula. Using the parameters defined in Table B.1,

Table B.1: List of the parameters. S/H/P stand for software/hardware/problem type, D/I stand for problem dependent/independent, and E/C—for elementary/composite; MA stands for multiply-add operation. M_{io} is measured in 4-byte words. Also, for 1-dimensional problems, the subscript on the space dimension may be dropped.

Name	Type	Description
$\mathtt{S_i}$	EP	i-th space dimension
$\mathtt{T}$	EP	time dimension
$\mathtt{op_j}$	EP	# of operations of type $\mathtt{j}$ in loop body
$t_{\mathrm{S_i}}$	ES	tile size along the i-th space dimension
t_T	ES	tile size along time dimension
$s_{\mathrm{S_i}}$	ES	subtile size along the i-th space dimension
P_{stat}	EH	static power of device
e_{gr}	EH	Energy per word of global↔register transfer
e_{sr}	EH	Energy per word of shared↔register transfer
e_{gs}	EH	Energy per word of global↔shared transfer
e_{sync}	EH	energy for a single synchronization
e_{j}	EH	energy per operation of type j
E_{iter}	CD	energy for a single iteration computation
E_{gs}	CD	Energy per tile for global↔shared transfers
E_{sr}	CD	Energy per tile for shared↔register transfers
E_{ar}	CD	Energy per tile for arithmetic
E_{sync}	CD	Energy per tile for synchronizations
M_{io}	CSD	I/O volume per tile (global↔shared)
V_{tile}	CSI	volume of a tile (# of iteration points)
N_{tiles}	CSI	total number of tiles
T_{alg}	CI	total execution time of stencil
E_{obs}	CI	total observed/experimental energy consumption
E_{stat}	CI	total static energy
E_{dyn}	CI	total dynamic energy
E_{alg}	CI	total energy

$$E_{\text{tile}} \approx E_{\text{gs}} + E_{\text{sr}} + E_{\text{ar}} = e_{\text{gs}} M_{\text{io}} + E_{\text{sr}} + E_{\text{iter}} V_{\text{tile}} \tag{B.2}$$

and the total energy is just the sum of E_{stat} and E_{dyn}, $E_{\text{alg}} = P_{\text{stat}} T_{\text{alg}} + N_{\text{tiles}} E_{\text{tile}}$.

For the number of tiles we could use a simple approximation that $N_{\text{tiles}} = \frac{V_{\text{alg}}}{V_{\text{tile}}}$, where V_{alg} is the volume of the entire iteration space. In some cases, e.g., in the case of orthogonal tiling as applied to SW, this is exact. In other cases, it may not be, and we use a more kernel specific model that accounts for the dependences, the shape of the iteration domain, etc.[8]

B.6.2 Energy model for the 1D kernels

We now specialize the generic model described above for SW and J1D.

SW model

As noted earlier, SW uses orthogonal tiling, and also subtiling and $V_{\text{tile}} = t_S t_T$. The number of global-to-shared transfers per tile is proportional to its perimeter. Specifically, $M_{\text{io}} = 4.25(t_S + t_T)$. The number of shared-to-register transfers per subtile is proportional to one of its sides—note that the other side of the subtile remains in registers because the same thread executes the next subtile in a row of subtiles. We multiply the latter by the number of subtiles in a tile, $\frac{t_S t_T}{s_S s_T}$ to get E_{sr}. The loop body perfroms five add-compare operations. Putting this all together and simplifying, we get

$$\begin{aligned} E_{\text{tile}} = E_{\text{gs}} + E_{\text{sr}} + E_{\text{ar}} &= 4.25\,(t_{\text{S}} + t_T)\, e_{\text{gs}} \\ &+ \left(V_{\text{tile}} + 5 t_T \frac{t_{\text{S}}}{s_{\text{S}}} \right) e_{\text{sr}} + 5 V_{\text{tile}}\,(e_{\text{add}} + e_{\text{max}}) \end{aligned}$$

J1D model

By a similar reasoning as for SW, and accounting for the specificities of J1D we get

$$E_{tile} = 2e_{gs}(2t_T + t_S) + 4e_{sr}\left(2t_S + 4t_T \frac{t_S}{s_S}\right) + E_{iter} V_{tile},$$

[8] Exact formulas available in spreadsheets at http://www.cs.colostate.edu/~waruna/energy_paper/

where V_{tile} is $t_S t_T$, as in SW, and E_{iter} can be modeled as the product of the number of multiply-add operations in the loop body and the energy required to do one multiply-add, i.e.,

$$E_{iter} = \sum_{j \in \text{add,mul}} \texttt{op}_\texttt{j} e_j$$

B.6.3 Energy model for the 2D kernels

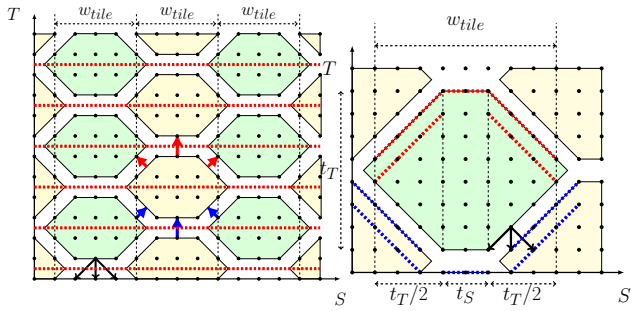


Figure B.4: The computational space is partitioned in hexagons (left). On the right side we zoom in and visualize the I/O data for a given tile. Blue - data read from global memory. Red - data written to global memory.

J2D uses HHC tiling, see Figure B.4. The "area" of a hexagon is $\frac{t_T}{2}(2t_{S_1} + t_T - 2)$ and its "I/O perimeter" is $2(t_{S_1} + 2t_T)$. (These formulas assume that t_T is even. They change slightly when it is odd.) We can obtain the number of hexagons by multiplying the number of rows or wavefronts, which is simply $2\frac{T}{t_T}$—accounting for both colors, by the number of hexagons per row. Since the pitch between successive hexagons on a row is $t_S + w_\text{tile}$, where w_tile is the width of the tile, see Fig. B.4, this latter term is $\frac{S}{t_S + w_\text{tile}}$. Note that it needs to be adjusted to account for the fact that the first and last tiles on alternate rows are partial, even if we assume that the pitch evenly divides S.

To model the energy of J2D, we first add a sub-subscript to the terms involving the first spatial dimension, and make the following observations. After time-skewing the s_2 dimension and breaking the prisms up into tiles of size t_{S_2}, the total number of tiles in a prism is $\frac{S_2 + t_T}{t_{S_2}}$. Of these, the first (resp. last) one is a "partial tile," and has an oblique upper bound (resp. lower bound), but we ignore this and assume that all tiles are "full." Hence we can get V_tile (respectively, M_io) by simply

multiplying the above computed area (resp. "I/O perimeter") by t_{S_2}. Similarly, we can obtain N_{tile} by multiplying the total number of hexagons, by $\frac{S_2+t_T}{t_{S_2}}$. This yields

$$M_{\text{io}} = 2t_{S_2}(t_{S_1} + 2t_T) \tag{B.3}$$

$$V_{\text{tile}} = \frac{t_{S_2}t_T(2t_{S_1} + t_T - 2)}{2} \tag{B.4}$$

$$N_{\text{tile}} = \frac{2TS_1(S_2 + t_T)}{t_T t_{S_2}(t_{S_1} + w_{\text{tile}})} \tag{B.5}$$

We can now rewrite the general formula (B.2) as follows

$$E_{\text{tile}} = e_{\text{gs}}M_{\text{io}} + \mu_{\text{sr}}e_{\text{sr}}V_{\text{tile}} + E_{\text{iter}}V_{\text{tile}} \tag{B.6}$$

where M_{io} and V_{tile} are given respectively by (B.3) and (B.4), while μ_{sr} is a stencil dependent coefficient related to the corresponding shared↔register transfers and equals 6 in the case of J2D.

B.7 Model validation

We first determine values for the missing elementary hardware (EH) parameters from Table B.1, denoted by $\mathcal{X}$, and then compare energy values predicted by the model with ones found through measurement.

B.7.1 Computing energy parameters

We use NVIDIA K20c GPU to experimentally validate our model. We will use two methods: benchmarking and regression.

Benchmarking Approach

NVIDIA has not disclosed energy or power data for memory and compute operations for K20c [129]. We used micro-benchmarks—short pieces of code each of which stresses an operation of interest—and the NVIDIA NVML library to measure instantaneous power at any time of

the execution of a micro-benchmark. The total consumed energy is then the product of the average measured power and the execution time of the micro-benchmark.

We designed micro-benchmarks for each parameter in $\mathcal{X}$ to determine the value of that parameter experimentally. The micro-benchmarks were implemented in such a way that the execution time and total energy consumption is dominated by the operation of interest.

We obtain instantaneous power readings at different points of time during the micro-benchmark execution. The average of power readings taken during the steady state temperature is considered as the average power of the micro-benchmark. Then, the total consumed energy is computed by multiplying the average power by the length of the time period for which the average power is computed. Finally, the energy $\mathrm{e_j}$ for operation j can be calculated as follows

$$e_{\mathrm{j}} = \frac{E_{\mathrm{dyn}}}{\mathrm{op_j}} = \frac{E_{\mathrm{alg}} - E_{\mathrm{stat}}}{\mathrm{op_j}} = \frac{E_{\mathrm{alg}} - P_{\mathrm{stat}} T_{\mathrm{alg}}}{\mathrm{op_j}}$$

where the parameters in the formula are as defined in Table B.1.

The NVIDIA NVML library was used to obtain power readings of the GPU. The library reads power once every 16ms interval for NVIDIA Kepler GPUs. We used the approach proposed by Lang et al. [130] to determine the interval.

The static power (P_{stat}) is measured while the GPU is idling but operating in its highest performance state. We carefully implemented the micro-benchmarks in such a way that there are no shared memory bank conflicts, and all global memory accesses are coalesced. The body of the benchmark is repeated to make the operation of interest the dominant in both execution time and total energy consumption. The resulting parameter values are given and discussed in Section B.7.2.

Regression Approach

We describe here an alternative method to benchmarking based on nonlinear regression, which we can use to estimate the values of the model parameters. We create a set of combinations of elementary program (EP) and elementary software dependent (ES) parameters (Table B.1) and, for each such combination, experimentally measure the energy consumed by the corresponding sten-

cil. We then plug in the values of the known parameters into the formula for the energy derived in the previous section and get, for each experiment, an equation with the values of the elementary hardware parameters as unknowns. The left-hand side of the resulting system of equations will contain the energy values predicted by the model and the right-hand side will contain the corresponding measured values.

We can then use an optimization method to find values of the unknown parameters that minimize the error defined as a norm of the difference between the left and right-hand sides of the equations. In other words, we are finding values of the parameters that result in a best fit of the model to the observed energy values. The observed error in the new method will be due to the model accuracy and inaccuracy in measuring the *total* energy only and will exclude errors resulting from parameter measurements.

Specifically, let $E^i_{\text{alg}}(\mathcal{X})$ and E^i_{meas} denote the energy predicted by the model (as a function of the parameters in $\mathcal{X}$) and the measured energy for the i-th experiment (combination of parameter values), respectively. Then, in our implementation, we minimize the objective function

$$Err(\mathcal{X}) = \sum_i |E^i_{\text{alg}}(\mathcal{X}) - E^i_{\text{meas}}(\mathcal{X})|,$$

which, while being nonlinear, can be implemented as a linear program using known techniques. We encoded the resulting linear program in the modeling language AMPL [123] and solved it using the open source solver Cbc [131].

B.7.2 Energy parameter values and error analysis

Table B.2 contains the values of the energy parameters as evaluated by the two methods. We measured the time and energy for 298 instances of SW with different values of the space and time dimensions and the tile sizes, 265 instances of J1D, and 125 of J2D. For the micro-benchmark approach, we used the values of the parameters found through micro-benchmarking, which are listed in the second column of Table B.2, to build an energy model, which we refer to as Model A. Then we compared the energy values predicted by Model A with the measured ones and computed

Table B.2: Table of the energy parameter values from the micro-benchmark approach and the regression approach as described in Section B.7.2.

Parameter Name [unit]	Micro-benchmarks	Regression
P_{stat} [W]	48	53
e_{gs} [J]	2.2×10^{-9}	3.17×10^{-09}
e_{sr} [J]	2.23×10^{-10}	1.84×10^{-10}
e_{fadd} [J]	5.3×10^{-11}	5.02×10^{-11}
$e_{\text{fmultiply}}$ [J]	3.7×10^{-11}	3.51×10^{-11}
e_{iadd} [J]	7.2×10^{-11}	0
e_{imax} [J]	4.8×10^{-11}	0
error SW [RMSE]	12.77%	2.59%
error J1D [RMSE]	17.14%	2.35%
error J2D [RMSE]	8.30%	3.95%

the root-mean-square error (RMSE) for the SW set and the J1D set. The errors, as shown in Table B.2, are 12.77% for SW, 16.35% for J1D, and 8.30% for J2D[9].

For the regression method, we used the same experimental data, which we divided into two sets. The first "training" set, containing a sample of 10% of all data points, was used for computing optimal (with respect to the sample) values for the parameters. Those values were used in an energy model, referred to as Model B, which was applied to predict the values of the energy for the remaining 90% of the instances, i.e., the ones not in the training set. The resulting RMSEs are 2.59% for SW, 2.35% for J1D, and 3.95% for J2D. The optimal parameter values are quite close to the ones found through benchmarking, except for the values of e_{iadd} and e_{imax}. Note that those values are used in SW only. Since those values are clearly positive in reality, the fact that they were estimated to be zeros implies that it may be possible to make the formulas in the SW model even more accurate. But even the errors reported in Table B.2 are quite low. The main factor which allows us to attain such high accuracy is domain-specificity: we focus only on stencils.

Clearly, model B is more accurate than model A, and that is to be expected since model B is based on optimization. However, model A has the advantage that its parameters are determined

[9]More details on validation are available at http://www.cs.colostate.edu/AlphaZ/HPPAC15/

in a stencil-independent way, while for model B they are more tightly coupled with the stencils studied.

Another observation is that, in Model A, J2D is more accurate than SW and J1D, while it is less accurate in Model B. The reason for the former is probably that SW and J1D use subtiling and a more complex register reuse strategy, hence being more difficult to model, while J2D doesn't. In Model B, optimization is done with respect to the sum of the errors for all experimental points, and SW and J1D, each having more points and, hence, larger contribution to the objective function than J2D, come out better optimized.

B.8 Optimizing energy efficiency

In Section B.6 we developed a model for energy consumption that expresses the amount of consumed energy as an analytical function of software and hardware parameters. In the previous section we discussed how regression analysis leads to an optimization problem that can be used to determine some unknown hardware parameters. In this section we use the model for a more typical task—finding a combination of software parameters that minimizes the energy consumption.

B.8.1 The optimization problem

Let the space of all elementary program (EP) input parameters be Γ and the space of all tunable elementary software (ES) parameters be Δ. Let $E_{\text{alg}}(\gamma, \delta)$ and $E_{\text{meas}}(\gamma, \delta)$ be the energy predicted by the model and the measured energy as functions of $\gamma \in \Gamma$ and $\delta \in \Delta$, respectively. Then we have to solve an optimization problem of the type:

$$\text{Find} \quad \min_{\delta \in \Delta} E_{\text{alg}}(\gamma, \delta) \text{ for a given } \gamma \in \Gamma, \tag{B.7}$$

subject to feasibility constraints. The objective function $E_{\text{alg}}(\gamma, \delta)$ is nonlinear with respect to δ, but unlike the one from the optimization problem from Section B.7.1, it cannot be linearized. But the optimization problem is of small size and we could solve it efficiently using the nonlinear solver Knitro [132].

Table B.3: Energy optimization error

Stencil	# Experim.	# Wrong	RMSE	Max error
SW	298	2	0.05%	0.75%
J1D	265	1	0.02%	0.4%
J2D	125	0	0.00%	0.0%

B.8.2 Validation of the results

Although relatively small, there is some accuracy error in our energy model. Hence we wanted to measure its effect on the optimization results. Let Γ' be the subset of Γ corresponding to the experiments discussed in Section B.7.2. Let $\delta^{\min}_{\text{alg}}(\gamma)$ be the value of δ that minimizes (B.7) and let $\delta^{\min}_{\text{meas}}(\gamma)$ minimizes $E_{\text{meas}}(\gamma, \delta)$ over all $\delta \in \Delta$ such that $(\gamma, \delta) \in \mathcal{X}$. If $\delta^{\min}_{\text{alg}}(\gamma) = \delta^{\min}_{\text{meas}}(\gamma)$, then our model is accurately predicting tile sizes that consume minimum energy. We define the error in energy when we use the values of δ found by solving (B.7) compared to the optimal values as

$$E_{\text{err}}(\gamma) = E_{\text{meas}}(\gamma, \delta^{\min}_{\text{alg}}(\gamma)) - E_{\text{meas}}(\gamma, \delta^{\min}_{\text{meas}}(\gamma)), \gamma \in \Gamma'.$$

Table B.3 reports the number of input parameter combinations (experiments), optimal parameter mismatches, RMSE and max error for SW, J1D, and J2D. We used values of the hardware parameters determined by the benchmarking method. There are very few miss-predictions and the RMSE is negligible. The maximum percentage error out of all the experiments is less than 1%. The accuracy of our model in predicting optimal software parameters for SW, J1D and J2D is a positive indication of efficacy of our energy model in solving practical problems.

B.9 Conclusion

We developed an energy model for tiled nested-loop codes on a GPU and illustrated its application to optimizing software parameters for energy efficiency. Our validation analysis shows that the model is quite accurate with respect both to predicting the consumed energy and to the optimization accuracy. We believe that the approach we developed can be adapted and applied to other related problems. One extension is to combine the energy model with an execution-time

model (we have already developed such a time model, but its description and analysis would have almost doubled the size of this paper). We can also apply the approach to a wider class of nested-loop codes that includes but is not restricted to stencil codes and to platforms other than GPU. We can also model and optimize with respect to the maximum power instead of energy, or extend the model to allow dynamic voltage and frequency scaling optimization. We will study these and other applications of our approach in our future work.

Acknowledgment: We would like to thank Tobias Grosser for providing access to the HHC code generator, and helping us use it effectively. This research was supported in part by the National Science Foundation (Contract N. 1240991) and by the Air Force Office of Scientific Research (Contract No. FA9550-13-1-0064).

Appendix C

Optimization Approach to Accelerator Codesign

We propose an optimization approach for determining both hardware and software parameters for the efficient implementation of a (family of) applications called *dense stencil computations* on programmable GPGPUs. We first introduce a simple, analytical model for the silicon area usage of accelerator architectures and a workload characterization of stencil computations. We combine this characterization with a parametric execution-time model and formulate a mathematical optimization problem that seeks to maximize a common objective function of *all the hardware and software parameters*. The solution to this problem, therefore, "solves" the codesign problem: simultaneously choosing software-hardware parameters to optimize total performance.

We validate this approach by proposing architectural variants of the NVIDIA Maxwell GTX-980 (respectively, Titan X) specifically tuned to a predetermined workload of four common 2D stencils (Heat, Jacobi, Laplacian, and Gradient) and two 3D ones (Heat and Laplacian). Our model predicts that performance would potentially improve by 28% (respectively, 33%) with simple tweaks to the hardware parameters such as tuning the number of streaming multiprocessors, the number of compute cores each contains, and the size of shared memory. We also develop a number of insights about the optimal regions of the design landscape.

C.1 Introduction

Energy constraints are the driving force behind exascale systems [2]. Despite exponential transistor density growth driven by Moore's Law, these constraints impose the specter of "dark silicon," where an exponentially increasing fraction of exascale chips will need to be powered down or in low power mode [133], thus limiting the "performance delivered to the consumer." Consequently, highly specialized coprocessors will become common in the exascale era. Such specialized, energy-efficient accelerators are important for a diverse range of systems, from HPC through general-purpose workstations, gaming machines, tablets, other media devices, to phones,

things, and sensors. In order to attain exascale-level performance, accelerators have to become even more energy-efficient, and experts anticipate that a large part of this must come through even more specialization [134, 2].

Future high-performance computing (HPC) systems are expected to be increasingly heterogeneous, consisting of several multi-core CPUs and a large number of accelerators, special-purpose hardware that will increase the computing power of the system in a very energy-efficient way [134]. Consequently, highly specialized coprocessors are already becoming common in exascale [135, 136, 137].

Software-hardware codesign is one of the proposed enabling technologies for exascale computing and beyond [7], yet hardware and software are currently designed largely separately. For example, HPC hardware manufacturers design and produce a system with great computing potential and deliver it to customers, who then try to adapt their application codes to run on the new system. But because of the typical mismatch between hardware and software structure and parameters, such codes are often only able to run at a small fraction of the total performance the new hardware can reach. Hence, optimizing both the hardware and software parameters *simultaneously* during hardware design is a promising way to achieve better hardware usage efficiency and thereby enable leadership-class HPC availability at more manageable cost and energy efficiency.

Codesign is particularly important for embedded systems. In many uses of these systems (e.g., medical imaging, smart cameras, self-driving cars, and cyber-physical systems) general purpose platforms based on standard CPUs deliver inadequate "performance" on a combination of many cost metrics: speed/throughput, power/energy, weight, size, and manufacturing/fabrication cost, especially in volumes that the market can sustain. As a result, specialized hardware is essential.

Hardware platforms for embedded systems are usually heterogeneous, with (instruction-set) programmable processors (CPUs and microcontrollers), accelerators that are either instruction-set programmable (e.g., GPUs), or "hardware programmable" ones like FPGAs and reconfigurable logic, as well as Application Specific Integrated Circuits (ASICs). For HPC systems, ASICs are usually not a designer option.

In both HPC and embedded systems design, the platforms have specialized, highly parallel, often fine-grain, components like FPGAs, GPUs, or DSPs, called *accelerators*. They are typically used as targets for the (parts of) applications that are well suited to exploit their fine-grain parallelism, typically programs with regular, repetitive computations.

Despite the appeal of such an approach to simultaneously optimize software and hardware, its implementation represents a formidable challenge because of the huge search space. Previous approaches [119, 120, 121], pick a hardware model $\mathcal{H}$ from the hardware design space, a software model $\mathcal{S}$ from the software design space, map $\mathcal{S}$ onto $\mathcal{H}$, estimate the performance of the mapping, and iterate until a desirable quality is achieved. However, not only can each of the software and hardware design spaces be huge, but each iteration also takes a long time, since finding a good mapping of $\mathcal{S}$ onto $\mathcal{H}$ and estimating the performance of the resulting implementation are themselves challenging computational problems.

In the context of embedded systems, software-hardware codesign refer to the problem of determining both the optimal software, $\mathcal{S}$ and the hardware, $\mathcal{H}$ for a single application or program. The problem we tackle in this paper generalizes this classic codesign problem to a precisely defined set of programs, drawn from a relatively narrow range of programs that are expected to execute on the accelerator (sub) system.

C.1.1 Overall approach

We *simultaneously optimize compilation and hardware/architectural parameters for compiling stencil computations to GPU-like vector-parallel accelerators* by formulating an optimization problem as follows. A given class of programs, with some *problem parameters*, $\overrightarrow{p} \in \mathcal{P}$ (e.g., size of the iteration space, number of array variables) is compiled to an accelerator with some *hardware parameters* $\overrightarrow{h} \in \mathcal{H}$ (e.g., number of cores, number of vector units, cache size, etc.) The compiler itself may use some well-defined optimization strategies (e.g., tiling, skewing, parallelization, etc.), and may have some *software parameters*, notably tile sizes ($\overrightarrow{s}$).

We develop an analytical function, $\mathcal{T}(\overrightarrow{p}, \overrightarrow{h}, \overrightarrow{s})$ that predicts the execution time of the target program as a function of these parameters. The values of the parameters must satisfy a set of *feasibility constraints* (e.g., tile sizes must be such that the data footprint of a tile must fit in the available scratchpad memory) denoted by $\mathcal{F}(\overrightarrow{p}, \overrightarrow{h}, \overrightarrow{s})$.

With such a model, a classic compilation problem like optimal tile size selection [125] can be formulated as solving the following problem:

$$\begin{aligned} \underset{\overrightarrow{s}}{\text{minimize}} \quad & \mathcal{T}(\overrightarrow{p}, \overrightarrow{h}, \overrightarrow{s}) \\ \text{subject to:} \quad & \overrightarrow{s} \in \mathcal{F}(\overrightarrow{p}, \overrightarrow{h}, \overrightarrow{s}) \end{aligned} \tag{C.1}$$

Here, the unknown variables are $\overrightarrow{s}$. The hardware and program parameters are considered as constants—compiling a different program and/or compiling to a different target machine entails solving a new optimization problem. Extending this to codesign requires the following steps.

- *Workload characterization* We first pick a set $\mathcal{W}$ of representative program instances, called *benchmarks*. For each one, we use profiling to pick the probability/frequency with which it occurs in the workload. Each benchmark may be executed with a range of program parameters, and we also have a frequency with which of these appear in the workload. So, $\overrightarrow{p_{i,j}}$ are the program parameters of the i-th instance of the j-th benchmark, and the set $\{\overrightarrow{p_{i,j}} \mid i \in \text{Instances}, j \in \text{Benchmarks}\}$ is $\mathcal{P}$, the range of program parameters.

- *Codesign Optimization* Now we formulate a simple extension of (C.2). Instead of leaving the hardware parameters as constants, we allow them to be unknown variables of the optimization problem, which now becomes

$$\begin{aligned} \underset{\langle \overrightarrow{s}, \overrightarrow{h} \rangle}{\text{minimize}} \quad & \mathcal{T}(\overrightarrow{p}, \overrightarrow{h}, \overrightarrow{s}) \\ \text{subject to:} \quad & \overrightarrow{s} \in \mathcal{F}(\overrightarrow{p}, \overrightarrow{h}, \overrightarrow{s}) \\ & \overrightarrow{h} \in \mathcal{F}_h(\overrightarrow{h}) \end{aligned} \tag{C.2}$$

- *Area Model* Since $\overrightarrow{h}$ is now an unknown, we must formulate its feasible space, $\mathcal{F}_h(\overrightarrow{h})$ precisely. For this, we develop an analytical model of the area of the accelerator. We assume that we are given an area budget within which the accelerator must fit, and solve the optimization problems over the resulting feasible space.

This seems (deceptively) simple, but the devil is in the detail. The resulting optimization problem has many hundreds of variables, and has non-convex constraints and objective functions, making it computationally intractable. We tackle this by exploiting *domain specificity*, as explained below.

C.1.2 Domain Specificity

There are three ways in which we use domain specificity: in the program, in the architecture, and in the compiler.

First of all, we tackle a specific (family of) computations that are nevertheless very important in many embedded systems. This class of computations, called *dense stencils*, includes the compute-intensive parts of many image processing kernels, simulation of physical systems relevant to realistic visualization, as well as the solution of partial differential equations (PDEs) that arise in many cyber-physical systems such as automobile control and avionics.

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[10] 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.

$$A^t(s) = \Big(\sum_{a \in \mathcal{N}} w_a * A^{t-1}(s + a) \Big) + c \tag{C.3}$$

[10]Stencils where some updates may use values from the current time step are not included in the definition.

$s \in \mathcal{S}$, $t \in \mathcal{T}$, 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 (C.3) and storing it in a data array.

Second, we target GPU-like *vector-parallel programmable accelerators*. Such components are now becoming a de-facto standard in most embedded platforms and MPSoCs since they provide lightweight parallelism and energy/power efficiency. We further argue that they will become ubiquitous for the following reasons. Any device on the market today that has a screen (essentially, *any device*, period) has to render images. GPUs are natural platforms for this processing (for speed and efficiency). So all systems will have an accelerator, by default. If the system now needs any additional dense stencil computations, the natural target for performing it in the most speed/power/energy efficient manner is on the accelerator.

The third element of domain specificity that we exploit is a formalism called the *polyhedral model* as the tool to map dense stencil computations to GPU accelerators. Developed over the past thirty years [14, 15, 16, 17, 18], it has matured into a powerful technology, now incorporated into `gcc`, `llvm` and in commercial compilers such as Rstream. Tools targeting GPUs are also available [19, 138].

Such domain specificity was combined by Prajapati et al. [22] who developed a simple analytical model for the execution time of tiled stencil codes on GPUs. They showed that for a set of stencils, the model is very accurate over the range of interest (highly tuned codes): the reported error is below 10%. As a result, they also showed how the model could be used to solve for optimal tile size selection, leading to a further 9% improvement in total performance. Because of these properties, we build off their results.

C.1.3 Contributions

The main focus of the paper is not on making specific design recommendations, but rather on the methodology. Our specific contributions are:

- We propose a new approach (outlined in Section C.1.1) to software-hardware codesign.
- We develop a simple, analytical model for the silicon area (Section C.2) of programmable accelerator architectures, and calibrate it using the platform that was used by Prajapati et al. [22], namely NVIDIA Maxwell class GPUs.
- We combine this area model with a workload characterization of stencil codes, and the Prajapati et al. time model [22] to formulate a mathematical optimization problem that maximizes a common objective function of the hardware and software parameters (see Section C.3).
- Our analysis (Section C.4) provides several interesting insights:
  - We produce a Pareto set of designs that represent the optimal combination of hardware and compiler parameters. The cardinality of the Pareto set is two orders of magnitude smaller than that of the feasible space of possible architectures. Our analysis also predicts that for say, a die area comparable to that of the NVIDIA Maxwell Titan, our specialized design could provide more than 120% improvement in the GFLOPs/sec attained by the accelerator.
  - The optimal design is very sensitive to the benchmark.
  - In general, the optimal architectures for 3D stencils need more shared memory than those for 2D stencils.
  - We see specific patterns in how resources should be shared between memory and compute units for 2D vs 3D stencils.

Designers, with access to more accurate area models and more extensive and precise workloads, could use our methodology to rapidly explore the design space, before expensive investment into hardware design, simulation and fabrication need to be made.

C.2 Area Model

We now develop an analytic model for the total silicon area of a GPU accelerator. We faced some difficulties in deriving an acceptable analytical model, as silicon data had to be reverse engineered from extremely limited public domain resources. As a general observation, within each GPU family, there is little diversity in the parameter configurations. For the Maxwell family of GPUs, the GTX980 and Titan X chips were chosen as two sufficiently distinct points to calibrate our analytical models. The calibration itself was performed by evaluating die photomicrographs, publicly available information about the NVIDIA GTX-980 (Maxwell series) GPU, and other generally accepted memory architecture models. The model validation was done by comparing the predictions with known data on the Maxwell series Titan X GPU. We found the model prediction to be accurate to within, 2%, though this number is not significant.[11]

C.2.1 Analytical Model for GPU Area

The area model is a function of the main parameters that we want to select optimally in the codesign problem: the number of SMs, n_{SM}, the number of vector units or cores in each SM, n_{V}, and the respective sizes of the register file and shared memory size of the SM, R_{SM}, and M_{SM}. Table C.1 shows a summary of the various parameters used in developing the area model.

$$\mathcal{A}_{\mathrm{tot}}() = n_{\mathrm{SM}}\mathcal{A}_{\mathrm{SM}} + \mathcal{A}_{\mathrm{cache}} + \mathcal{A}_{\mathrm{oh}} \tag{C.4}$$

where $\mathcal{A}_{\mathrm{SM}}$ is the cost of each SM, $\mathcal{A}_{\mathrm{cache}}$ is the total area of the on-chip L1 and L2 caches and $\mathcal{A}_{\mathrm{oh}}$ is the overall overhead area cost. The term $\mathcal{A}_{\mathrm{oh}}$ accounts for several components which are of only peripheral interest to us, such as metal-layer routing overheads, I/O pads, buffers, and clock-signal global distribution trees, gigathread scheduler, PCI express interface, raster engine, and memory controller. Based on our current understanding of the NVIDIA architecture, a best effort annotation of various functional blocks is shown overlaid on a die photomicrograph of GTX980

[11] Although many configurations of any family of GPUs is spaced out, they come from binning only a small number of distinct dies. We ended up calibrating our model on one die and validating it on only another one.

Table C.1: Area model parameters. The top two groups are elementary parameters, and the third one is composite (some function of the elementary parameters). Of the elementary ones, the second one are treated as variables in our optimization formulation.

Name	Description
α_R	overhead area per kB of register-memory per vector-unit
α_M	overhead area per kB of shared memory per SM
α_{L1}	L1 cache overhead area per SM-pair
α_{L2}	L2 cache overhead area
α_{oh}	common overhead area (I/O, global routing etc) per SM
β_R	area per register-file-bank per kB per vector-unit
β_M	area per shared memory bank per kB per SM
β_{L1}	L1 cache area per kB per SM-pair
β_{L2}	L2 cache area per kB
β_{VU}	core-logic area within a vector-unit
n_{SM}	total number of SM on the GPU chip
n_{V}	number of vector-units per SM
R_{VU}	kB of register files per vector-unit
M_{SM}	kB of shared memory per SM
$L1_{\mathrm{SMpair}}$	kB of L1 cache per SM-pair
$L2_{\mathrm{SM}}$	kB of L2 cache
$\mathcal{A}_{\mathrm{tot}}$	total GPU chip die area
$\mathcal{A}_{\mathrm{SM}}$	total die area per SM
$\mathcal{A}_{\mathrm{cache}}$	total cache die area
$\mathcal{A}_{\mathrm{oh}}$	total on-chip overhead die area
$\mathcal{A}_{\mathrm{LSU}}$	total load-store unit die area
$\mathcal{A}_{\mathrm{SFU}}$	total special-function unit die area
$\mathcal{A}_{\mathrm{FDU}}$	total fetch-decode unit die area
$\mathcal{A}_{\mathrm{Icache}}$	total instruction-cache die area
$\mathcal{A}_{\mathrm{LSUperSM}}$	load-store unit die area per SM
$\mathcal{A}_{\mathrm{LSUperV}}$	load-store unit die area per vector-unit
$\mathcal{A}_{\mathrm{MperSM}}$	memory die area per SM
$\mathcal{A}_{\mathrm{SFUperV}}$	special-function unit die area per vector-unit

chip in Figure C.1. The term $\mathcal{A}_{\mathrm{SM}}$ denotes the area of a single SM, and we develop it next. These micro-architectural elements form an SM:

- Individual vector units: The area cost of this is $n_{\mathrm{V}}\beta_{VU}$.
- Load-store unit (LSUs): Every core may issue independent memory requests, to either the shared memory or to global memory. Therefore the LSU has a component that is replicated per core, and also a shared component that (eventually) collects together all requests from a single warp and interfaces to the (global) memory hierarchy. Therefore $\mathcal{A}_{\mathrm{LSU}} = n_{\mathrm{V}}\mathcal{A}_{\mathrm{LSUperV}} + \mathcal{A}_{\mathrm{LSUperSM}}$.
- Special function units (SFUs): These are dedicated functional units that are used for common graphics functions, transcendentals, etc. Their number is not always exactly equal to n_{V}, (e.g., in the NVIDIA GTX-980, there is one SFU every 8 cores) but we model it as a linear function of n_{V}, i.e., $\mathcal{A}_{\mathrm{SFU}} = n_{\mathrm{V}}\mathcal{A}_{\mathrm{SFUperV}}$.
- Instruction fetch-decode unit (FDU): Like the I-cache, there is a single FDU per SM, so its cost is simply a constant $\mathcal{A}_{\mathrm{FDU}}$
- Various specialized "memories:" the shared memory used as scratchpad, the register file, the instruction cache, the texture, and other special caches. We call this term $\mathcal{A}_{\mathrm{MperSM}}$ and develop it separately.

Of these, the memories other than register files and the FDU are shared by the entire SM, while the register files, SFU and LSU are accounted for on a per vector unit basis. Hence,

$$\begin{aligned}\mathcal{A}_{\mathrm{SM}}() &= (\mathcal{A}_{\mathrm{FDU}} + \mathcal{A}_{\mathrm{Icache}} + \mathcal{A}_{\mathrm{LSUperSM}}) + \mathcal{A}_{\mathrm{MperSM}} \\ &\quad + n_{\mathrm{V}}(\mathcal{A}_{\mathrm{SFUperV}} + \mathcal{A}_{\mathrm{core}} + \mathcal{A}_{\mathrm{LSUperV}}) \\ &= \alpha' + \mathcal{A}_{\mathrm{MperSM}} + \beta n_{\mathrm{V}} \end{aligned} \tag{C.5}$$

where $\alpha' = \mathcal{A}_{\mathrm{FDU}} + \mathcal{A}_{\mathrm{Icache}} + \mathcal{A}_{\mathrm{LSUperSM}}$, and $\mathcal{A}_{\mathrm{MperSM}}$ depends on the capacity of each of the individual memories. Of these, we are interested in modeling the area costs for the register file,

shared (scratchpad) memory, L1 cache and L2 cache; so the others are treated as constants absorbed in α'. The L1 cache is shared by a pair of SM units, while the L2 cache is shared by all the SM units on the chip. The register files are organized so that a few registers (512 registers each 32 bits wide in GTX980) are exclusively accessible by each vector-unit. The shared memory is accessible by all vector-units in an SM. Assuming independent linear cost models for each of these four memory types, therefore, $\mathcal{A}_{\mathrm{MperSM}}() = n_{\mathrm{V}}\left(\beta_{\mathrm{R}}R_{\mathrm{VU}} + \alpha_{\mathrm{R}}\right) + \left(\beta_{\mathrm{M}}M_{\mathrm{SM}} + \alpha_{\mathrm{M}}\right) + \frac{n_{\mathrm{SM}}}{2}\left(\beta_{\mathrm{L1}}L1_{\mathrm{SMpair}} + \alpha_{\mathrm{L1}}\right) + \left(\beta_{\mathrm{L2}}L2_{\mathrm{SM}} + \alpha_{\mathrm{L2}}\right)$. Here, we have subscribed to a design philosophy where the size of the L2 cache is not proportional to the number of SM units, it is a constant. This choice seems to be the norm, in that the GTX980 chip with 16 SMs has an L2 cache of 2MB, whereas the Titan X with 24 SMs has an L2 cache of 3MB. Another design choice we make is the following – the common area overhead $\mathcal{A}_{oh}$, comprising the I/O pads, buffers, routing, gigathread engine, PCI and memory controllers, is also assumed to be proportional to the number of SMs[12], that is, $\mathcal{A}_{oh} = n_{\mathrm{SM}}\alpha_{oh}$. Substituting this result in the above equations, collecting all common overhead area contributions including α' into α_{oh}, and further simplifying, we get,

$$\begin{aligned}\mathcal{A}_{\mathrm{tot}}() \;=\; & n_{\mathrm{SM}}n_{\mathrm{V}}\beta_{VU} + n_{\mathrm{SM}}n_{\mathrm{V}}\left(\beta_{\mathrm{R}}R_{\mathrm{VU}} + \alpha_{\mathrm{R}}\right) \\ & + n_{\mathrm{SM}}\left(\beta_{\mathrm{M}}M_{\mathrm{SM}} + \alpha_{\mathrm{M}}\right) + \tfrac{n_{\mathrm{SM}}}{2}\left(\beta_{\mathrm{L1}}L1_{\mathrm{SMpair}} + \alpha_{\mathrm{L1}}\right) \\ & + \left(\beta_{\mathrm{L2}}L2_{\mathrm{SM}} + \alpha_{\mathrm{L2}}\right) + n_{\mathrm{SM}}\alpha_{oh}\end{aligned} \tag{C.6}$$

C.2.2 Calibrating the Model

Since the overall area model involves non-linear terms, we used an incremental fitting approach to calibrate our model. We first developed linear regression models for the area contribution due to the memory elements. We then incorporated the memory element area model into a measurement based linear model for the area due to the SM core vector units.

[12]This design choice is by no means the only possible one – another possibility is also to have an additional constant term, though one would need more than two GPUs in the Maxwell family with different die areas to calibrate such a model.

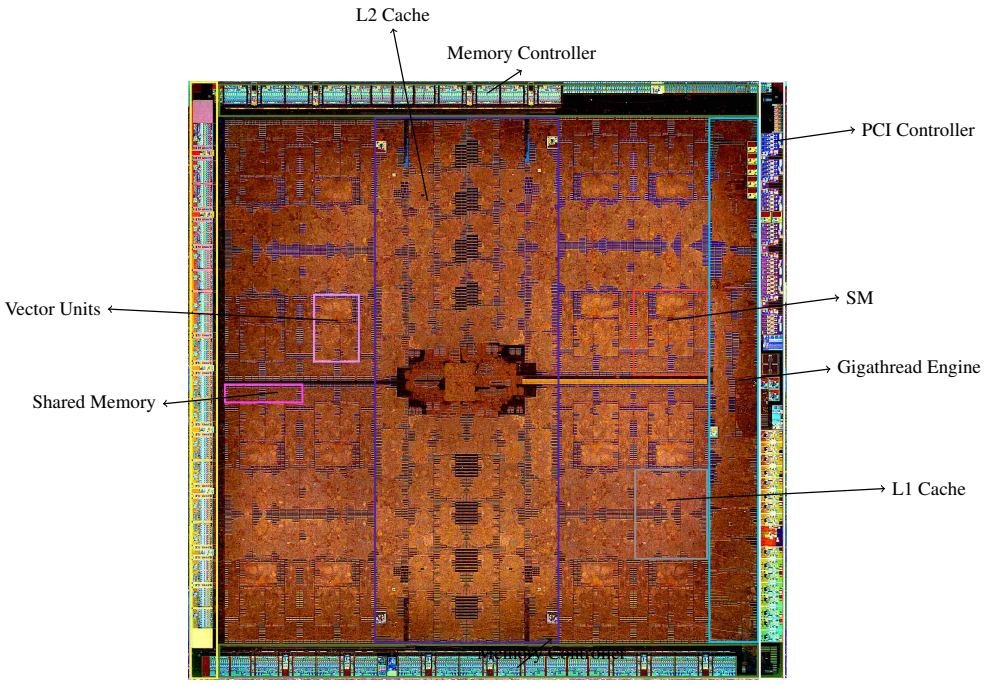


Figure C.1: Best effort annotation of functional blocks in a GTX980 die photograph [1] based on current understanding of the NVIDIA Maxwell architecture and chip layout.

The Maxwell family was fabricated using the TSMC 28nm process. For this process, the typical SRAM bit cell has an area of only 0.127 to 0.155 μm^2 [139]. The total area used by memory banks, be it the register files, shared memory, or the caches, involve the much more substantial additional area used for routing, addressing, read, write and bit-sense amplification logic. The register file, shared memory, and cache area models should take this overhead into account, by accounting for the bit width, number of addresses, and number of read and write ports. We used the open source Cacti 6.5 RAM and cache estimation tool from HP labs [140] to optimize and calibrate all our memory models:

- Register files: We modeled the register file at the vector unit level with a data bus of 32 bits. For example, the GTX980 chip has 512 such registers per vector unit. For configuring the register file in Cacti, we assumed a 'RAM' model which is direct-mapped, with 2 exclusive single-ended read ports and 1 exclusive write port per vector unit. Our Cacti design objective was to aggressively minimize area, as register files are placed physically close to the logic core where space is at an extra premium.

- Shared memory units: The shared memory bank is organized per SM. The GTX980 chip has 96kB of shared memory per SM. The user is allowed to optionally dynamically configure a part of the shared memory in an SM as cache. We assumed, for the purpose of Cacti tool configuration, that it is a direct mapped 'RAM' type memory with 32 bit wide data bus, on each of its 8 read-write ports per SM. Our design objective was to minimize area, with a secondary objective of minimizing propagation delays.
- L1 cache: The L1 cache is shared by two adjacent SM units. The GTX980 has 48kB of L1 cache per SM. Our Cacti model for L1 cache is a 'cache' type memory with a line-size of 128 bytes, and a data width of 32 bits. We also chose a conservative setting of full associativity for this cache level. There are 8 exclusive read ports for downstream access from the logic cores, and 8 exclusive write ports for upstream access from the L2 cache. The design was tailored for speed and the optimization objective was primarily propagation delay and secondarily power dissipation.
- L2 cache: For the NVIDIA Maxwell architecture, the L2 cache is shared by all the SM units on the chip. For this GPU family, the L2 cache is on average 128kB per SM. Our Cacti configuration assumes a 'cache' type memory with a line-size of 128 bytes, and a data width of 256 bits on each of its 8 exclusive read ports feeding into every L1 cache on the chip downstream. There is also an exclusive read-write port for upstream interface with the memory controllers. The design objective was a weighted mix of delay and area.

Using these Cacti models, we obtained area estimates for per vector unit register file banks of $512, 1024, 2048, 4096$, and 8192 bytes each. We then derived a linear fit model for the area given the bank size. The coefficients of this model were $\beta_R = 0.004305$ and $\alpha_R = 0.001947$.

Similarly, we obtained area estimates for per SM shared memory banks of $24, 48, 96, 192$, and 384 kilobytes size each, and derived a linear fit model. The coefficients of this linear model were $\beta_M = 0.01565$ and $\alpha_M = 0.09281$.

The L1 and L2 cache models were similarly calibrated by performing optimizations using our Cacti models as described earlier. The L1 linear fit model was obtained with per SM-pair sizes

of $3, 6, 12, 24, 48$, and 96 kilobytes each. The model parameters obtained were $\beta_{L1} = 0.1604$ and $\alpha_{L1} = 0.08204$. The L2 linear fit model was obtained with per SM sizes of $32, 64, 128, 256$, and 512 kilobytes each. The parameters obtained were $\beta_{L2} = 0.04197$ and $\alpha_{L2} = 0.7685$. The linear regression models obtained as above are shown in Figure C.2.

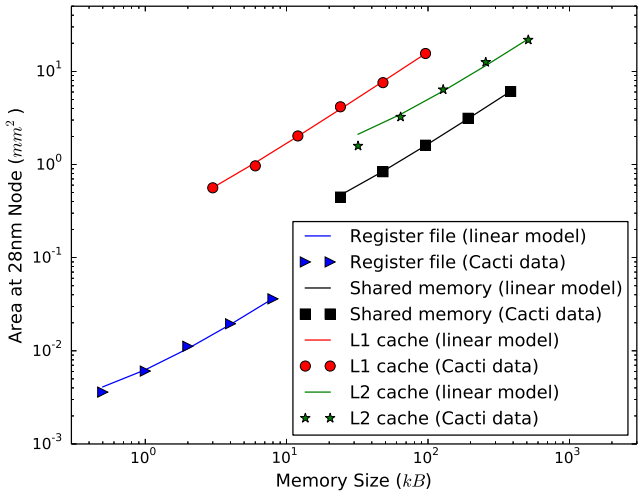


Figure C.2: Linear regression models for various memory types on a Maxwell series GPU. The Cacti predicted area used to obtain the linear model are also shown.

NVIDIA publishes obfuscated *die-shots* of their GPU chips, though these photographs are sometimes not to scale. The community of GPU enthusiasts and researchers have also attempted to decap packaged GPU chips and photograph the silicon dies independently. For example, Fritzchens Fritz [1] has a vast collection of NVIDIA die photographs which we have used in association with the obfuscated die shots published by NVIDIA. The NVIDIA official "die-shots" tend to show demarcated *logical* functional blocks - for SMs, this includes not only the SM core logic regions, but also the associated control, and thread-scheduler logic areas too. We have taken these aspects into account while making our area measurements. The area measurements were initially made in terms of pixels and later normalized to mm^2 using the total die area from the GPU's datasheet. As a verification of the memory area model calibration, we first measured the area of the following memory blocks on the die photograph – L2: $105\,\text{mm}^2$, L1: $7.34\,\text{mm}^2$, and shared memory: $1.27\,\text{mm}^2$.

These measured areas matched quite well with the linear model predictions – L2: $98.25\,\text{mm}^2$, L1: $7.78\,\text{mm}^2$, and shared memory: $1.59\,\text{mm}^2$. Furthermore, from the die photomicrographs for GTX980 [1, 141], area per vector unit logic core was measured to be $\beta_{VU} = 0.04282\,\text{mm}^2$, excluding the register-file area. Similarly, measurements on GTX980 die photograph gave an area estimate of $102.65\,\text{mm}^2$ for the overhead region containing the I/O pads, buffers, memory controllers, gigathread and raster engines and PCI controller, which gives an $\alpha_{oh} = 6.4156\,\text{mm}^2$ per SM. The total published die area for GTX980 is also known to be $\mathcal{A}_{\text{tot}} = 398\,\text{mm}^2$. For the Maxwell family, our overall area model is therefore given by:

$$\begin{aligned}\mathcal{A}_{\text{tot}} = 0.0447 n_{\text{SM}} n_{\text{V}} + 0.0043 R_{\text{VU}} n_{\text{SM}} n_{\text{V}} \\ +0.015 M_{\text{SM}} n_{\text{SM}} + 0.08 L1_{\text{SMpair}} n_{\text{SM}} \\ +0.041 L2_{\text{kB}} + 7.317 n_{\text{SM}}\end{aligned} \tag{C.7}$$

C.2.3 Validating the Model

In order to validate our area model, we applied it to another member of the Maxwell family, the Titan X for which the total die area is known. Our model predicts a total area of $589.2\,\text{mm}^2$ which is within an error of 1.96% of the published die area of $601\,\text{mm}^2$.

Our area model above is only valid for the functional blocks as implemented for the specific TSMC 28-nm process used to fabricate the NVIDIA Maxwell chips. The newer Pascal family of NVIDIA GPUs are fabricated on a 16-nm technology node. If each of the constituent functional block's chip-level layouts were to remain the same across technology nodes and were only optically reduced down by a certain shrinkage factor, then the area model can be similarly rescaled and reused. However, the shift from 28-nm to 16-nm involves a non-trivial move from planar CMOS to 3D-FINFET technology, and consequently, a simple area rescaling would clearly not be sufficient to predict the area parameters of Pascal family chips. However, the overall methodology still remains applicable to any technology node or device family.

C.3 Optimizing software and hardware parameters

We now show how to use the analytical chip area model and a previous execution time model [22] to formulate and solve an optimization problem for simultaneously finding optimal hardware and software parameters. Note that both components of the optimization problem, the area and time models, are relatively accurate, with errors 2% and 10% [22], respectively, implying the accuracy and amount of error expected in the results of the optimization.

C.3.1 Problem formulation

The parameters we are going to optimize are the elementary software (ES) parameters t_{S_1}, t_{S_2}, and t_{T} ($t_{\mathrm{S}_\mathrm{i}}$ represents tile size in the i^{th} space dimension and t_{T} represents tile size in the time dimension) and the elementary hardware (EH) parameters n_{SM}, n_{V}, and M_{SM}. For each combination of EH parameters, the corresponding running time in the objective function will be defined as the optimal time for that hardware configuration and for the stencil of interest over all possible choices of tile sizes.

This results in the following optimization problem, which finds the best EH parameters for a given stencil and a given bound on the chip area denoted by $chip\_area$

$$\underset{n_{\mathrm{SM}},n_{\mathrm{V}},M_{\mathrm{SM}},t_{\mathrm{S}_1},t_{\mathrm{S}_2},t_{\mathrm{T}}}{\text{minimize}} \quad T_{\mathrm{alg}}(n_{\mathrm{SM}}, n_{\mathrm{V}}, M_{\mathrm{SM}}, t_{\mathrm{S}_1}, t_{\mathrm{S}_2}, t_{\mathrm{T}}) \tag{C.8}$$

subject to:

$$\mathcal{A}_{\mathrm{tot}}(n_{\mathrm{SM}}, n_{\mathrm{V}}, M_{\mathrm{SM}}) \leq \mathit{chip\_area} \tag{C.9}$$

$$M_{\mathrm{tile}} \leq M_{\mathrm{SM/threadblock}} \tag{C.10}$$

$$k \leq MTB_{\mathrm{SM}} \tag{C.11}$$

$$k * M_{\mathrm{tile}} \leq M_{\mathrm{SM}} \tag{C.12}$$

$$k, t_{\mathrm{S}_1} \text{ – integers} \tag{C.13}$$

$$n_{\mathrm{V}}, t_{\mathrm{S}_2} \text{ – multiple of 32} \tag{C.14}$$

$$M_{\mathrm{SM}} \text{ – multiple of 48 k} \tag{C.15}$$

$$n_{\mathrm{SM}}, t_{\mathrm{T}} \text{ – even} \tag{C.16}$$

Note that T_{alg} (modeled execution time as in [22]) in (C.8) is in fact a function not only of the parameters n_{SM}, n_{V}, M_{SM}, t_{S_1}, t_{S_2}, and t_{T}, but also of the other parameters detailed elsewhere [22]. Also, for 3D stencils, there is an additional parameter to function (C.8) which is the tile size in the third space dimension t_{S_3}. We are only explicitly using those T_{alg} parameters that are needed in the specific context. Constraints encode the following requirements.

- Constraint (C.9) guarantees that the total area for the current hardware configuration, according to the area model, does not exceed the optimization parameter *chip_area*.
- Constraint (C.10) asks the memory required for each tile (M_{tile}, the memory footprint of a tile) not to exceed $M_{\mathrm{SM/threadblock}}$, the shared memory available for a threadblock.
- Constraints (C.11) and (C.12) restrict k, the number of tiles simultaneously executed by an SM (using hyperthreading), to be no greater than the max threadblocks an SM can handle and such that the total memory of such tiles is at most M_{SM}.
- The values of k and t_{S_1} can be only integers, by (C.13).

- t_{S_2} being a multiple of 32 (C.14) ensures that neighboring threads in S_2 dimension can fit in a number of full warps, where each warp is a group of 32 threads.

- We require n_V to be multiple of 32 (C.14), M_{SM} to be multiple of 48k (C.15), and n_{SM} to be even (C.16) in order to be consistent with the patterns for these parameters chosen by the manufacturers in the existing NVIDIA GPUs.

- For the tile sizes, we want t_T to be even (C.16) as it is a requirement for hybrid-hexagonal tiling [19].

The optimization problem (C.8)–(C.16) allows, given a stencil of interest (say Jacobi 2D) and specific values of the problem size parameters S_i (i^{th} space dimension) and T (time), to find a combination of software and hardware parameters that results in optimal (smallest) run time. However, it is unlikely that one may need a hardware that will be used for just a single problem size. A more relevant problem is to optimize the hardware over a set of problem sizes. Given a set of sizes SZ and a frequency function fr such that $\mathrm{fr}(\mathrm{Sz}_i)$ is the expected frequency the size $\mathrm{Sz}_i \in \mathrm{SZ}$ will be encountered for the intended application, the corresponding new objective becomes

$$\underset{n_{SM},n_V,M_{SM},SP}{\text{minimize}} \; T^{J}_{alg} = \sum_{s \in SZ} \mathrm{fr}(s) T_{alg}(s), \tag{C.17}$$

where SP is the set of tile sizes, with each tile size given as a triple (t_{S_1}, t_{S_2}, t_T) for 2D stencils (additionally t_{S_3} for 3D stencils), and T^{J}_{alg} (J is for Jacobi) is the time-model function, whose EH parameters are n_{SM}, n_V, M_{SM}, and whose ES parameters are SP. Hence, the optimization problem (C.17) has $|SP| + 2 = 3|SZ| + 3$ decision variables.

In our experiments, we use as values for S the set $\mathrm{SZ}_S = \{4096, 8192, 12228, 16384\}$ and for T the set $\mathrm{SZ}_T = \{1024, 2048, 4096, 8192, 16384\}$ and define $\mathrm{SZ} = \{(S,T) \mid S \in \mathrm{SZ}_S, T \in \mathrm{SZ}_T, T \leq S\}$. We use $T \leq S$ since it is known from practice that no more than S iterations are needed for convergence. One can check that $|\mathrm{SZ}| = 16$ and hence the number of variables for problem (C.17) is $16 * 3 + 3 = 51$. While this is a small number of variables for optimization problems with a nice structure, e.g., linear or convex, in our case the problem is of a difficult (non-

convex) type, with all variables being integer and the objective function and the constraints being rational functions [22]. Because of the floor and ceiling functions used in the time model [22], our objective and constraints are not even continuous, although this can be handled by introducing one new integer variable per floor or ceiling function. This however further increases the number of variables, which becomes $16 * 10 + 2 = 162$.

Regardless of the fact that an optimization problem with 162 integer variables and non-convex constraints and objective is in most cases computationally infeasible, we will consider an even larger version of (C.17), and then show how both problems can be simplified and solved accurately and in reasonable time by exploring the separability of the objective.

Let us first describe a problem setting that is more relevant from practical point of view. While (C.17) can be used to determine *stencil-optimal* architecture parameters, in many cases, people would be interested in having a special-purpose hardware that is optimized for an application whose computation time is dominated by a set of often-used kernels. In our case, we assume that an application Apl is using the six stencils studied in this paper and by studying a profiling data we have determined how often each stencil is used and a distribution of problem dimension sizes for each application. In our experiments, we use four 2D stencils: Jacobi-2D, Heat-2D, Laplacian-2D, and Gradient-2D, all first order stencils, and two 3D stencils: Heat-3D and Laplacian-3D. All 2D stencils have two space dimensions and one time dimension, while the 3D stencils have three space dimensions and one time dimension. We then extend the definition of the frequency function so that, for each code $c \in \mathrm{Cd} := \{$Jacobi-2D, Heat-2D, Laplacian-2D, Gradient-2D, Heat-3D, Laplacian-3D$\}$, $\mathrm{fr}(c)$ denotes the frequency of using c in application Apl. Moreover, given $c \in \mathrm{Cd}$ and $\mathrm{Sz} \in \mathrm{SZ}$, let $\mathrm{fr}(c, \mathrm{Sz})$ denote the frequency with which problem size Sz has been used for stencil c. The updated objective of the resulting optimization problem is:

$$\underset{HP,SP}{\text{minimize}}\, T^{\mathrm{Cd}}_{\mathrm{alg}} = \sum_{c\in\mathrm{Cd},\mathrm{Sz}\in\mathrm{SZ}} \mathrm{fr}(c)\mathrm{fr}(c,\mathrm{Sz})T^{c}_{\mathrm{alg}}(HP, SP, \mathrm{Sz}). \tag{C.18}$$

The set SP here includes tile sizes for each $c \in \mathrm{Cd}$ and each $\mathrm{Sz} \in \mathrm{SZ}$. The set HP includes the hardware parameters in consideration. Hence the number of integer variables for problem (C.18)

is $10|\mathrm{Cd}||\mathrm{SZ}|+2 = 10 \cdot 4 \cdot 16 + 2 = 642$, which is too large to be solved by existing solvers, given that the problem is nonlinear, nonconvex, and with integer variables. Fortunately, the problem can be made feasible by dividing the variables into two sets and combining exhaustive search on the first set with nonlinear programming optimization on the second.

C.3.2 Solving the optimization problem

The main observation that helps us solve problem (C.18) more efficiently is the fact that, if we knew the optimal value hp_{opt} of parameters HP, then the objective (C.18) would become separable since $T^c_{\mathrm{alg}}(hp_{opt}, SP, \mathrm{Sz})$ can be minimized with respect to the tile sizes independently for each c and Sz. Formally, we are transforming (C.18) into the following equivalent problem

$$\underset{hp \in HP}{\text{minimize}} \sum_{c \in \mathrm{Cd}, \mathrm{Sz} \in \mathrm{SZ}} \mathrm{fr}(c)\mathrm{fr}(c,s) \min_{SP(c,s)} T^c_{\mathrm{alg}}(hp, SP, \mathrm{Sz}). \tag{C.19}$$

Since we don't know hp_{opt}, then we replace hp in (C.19) with all feasible values of HP and, for each such value, we run a separate optimization problem with respect to SP for each combination of $c \in \mathrm{Cd}$ and $\mathrm{Sz} \in \mathrm{SZ}$. As a result, instead of solving one large problem (C.18) with 642 variables, we do exhaustive search on the space $HP \times \mathrm{Cd} \times \mathrm{SZ}$ and, for each point in that space, we solve using a nonlinear solver an optimization problem with only 10 integer variables. (While the variables we are interested in are only three, the optimization problem includes additional internal variables such as the optimal number of tiles for hyperthreading and variables used to simplify some nonlinearities, resulting in a total of 10.)

To further reduce the number of these problems, we will fix the range of values of the parameters as follows: $2 \le n_{\mathrm{SM}} \le 32$ and is even, $32 \le n_{\mathrm{V}} \le 2048$ and is a multiple of 32, which come from the constraints of (C.8)–(C.16). $48k \le M_{\mathrm{SM}} \le 480k$ with the requirement (C.15) to be multiple of 48k and positive. In addition, we explore the sizes $12k$, $24k$ and $36k$ for M_{SM}.

We ran these experiments using chip areas in the range $200 - 650\mathrm{mm}^2$. Since we haven't tied the experiments to a specific application, we assumed all six stencils are equally likely, and that each size combination is also equally likely; i.e., we set all coefficients in (C.18) to 1. For solving

the optimization problems, we used the open source solver Bonmin [124]. The average solution time per optimization instance is 19 sec (although it varies a lot from instance to instance), so the total solution time varies between 7 and 24 hours depending on the chip area. Also note that in the execution time model we use a parameter C_{iter}, the execution time of a single iteration on one thread. For optimal tile size selection, [22] empirically measured this parameter for the different stencils. Although there was a small difference between the two platforms (GTX-980 and Titan X) we used the former value for our experiments.

C.4 Insights

C.4.1 Insight 1: Pruning the Design Space

Given a range of area budgets between 200 and 650 mm^2, we enumerate all feasible architectures and solve the codesign optimization problem for each one of them. We separate the workload into two classes, 2D stencils, and 3D stencils. Each illustrates interesting features. For each class, we assume a uniform frequency (each instance is equally likely, and within each instance, each problem size is also equally likely). Figure C.3 shows our results. The first observation is that only about 1% of the thousands of feasible hardware designs (the Pareto optimal designs shown in blue) are worth exploring further, leading to a significant pruning of the design space.

The same figure also shows the performance of two standard design points, NVIDIA's GTX980 and Titan X architectures. For the comparable area, we can improve performance by 104% (resp., 69% wrt. Titan X) for 2D stencils, and by 123% (resp., 126%) for 3D stencils. A large part of the gains come from the fact that our proposed designs do not have caches because the HHC compiler for which our model is specialized [19] generates codes to perform data transfers explicitly rather than relying on caches[13]. To compensate for this, we could "delete" the caches from the GTX980 and Titan X die areas, which reduces their areas to respectively, 237mm^2 and 356mm^2. The performance of the corresponding Pareto optimal designs for those reduced areas is 9.34% (resp.,

[13]For all polyhedral programs, the off-chip data transfers can be statically predicted and tuned using models for tile sizes. Therefore, the ideal architecture for polyhedral programs are the ones that do not have caches.

Table C.2: Workload sensitivity. The optimal architecture configuration for a single benchmark varies significantly.

Code	n_{SM}	n_{V}	M_{SM}	Area	GFLOPs/S
Jacobi 2D	32	128	24	438	2059
Heat 2D	22	256	12	447	3017
Gradient 2D	28	160	24	431	4963
Laplacian 2D	28	160	12	426	2549
Heat 3D	18	288	192	447	3600
Laplacian 3D	8	896	96	446	1427

28.44% for the cache-less Titan X) better for 2D stencils and 9.22% (resp., 33.15% for the cache-less Titan X) better for 3D stencils.

C.4.2 Insight 2: Workload Sensitivity

Eqn. (C.19) partitions the design space into a number of independent optimization problems. This allows us to explore other hypothetical scenarios "for free." As an example, changing frequencies of the different programs in the benchmark suite requires us to simply recompute new weighted sums of optimal values of minimization problems that we have already solved. A specific example would be to hypothetically set the frequency for one of the benchmarks as one (and zero for the others) thereby allowing us to explore designs optimized for a single kernel. It also helps determine whether the chosen suite is representative of the mix that occurs in practice. Table C.2 illustrates the architectural parameters for the best performing designs for each of the six benchmarks, for an area budget between 425–450 mm^2. Observe how the parameters of the best architecture are significantly different. There are also differences in the achieved performance for each benchmark, but that is to be expected since the main computation in the stencil loop body has different numbers of operations across the benchmarks.

C.4.3 Insight 3: Shared Memory Requirements

We can also observe that there are marked differences between the optimal architecture configurations for 2D and 3D stencils in Table C.2. 3D stencils seem to require larger shared memory

($\geq$ 96 kB / SM) compared to 2D stencils ($\leq$ 24 kB / SM). Indeed, for designs with lower than 48kB, the performance was nowhere near the optimal for 3D stencil programs. Comparing the optimal configurations for $Heat\ 2D$ stencil with that of $Heat\ 3D$ stencil (both have exactly same area of $447\ mm^2$), we see that the amount of shared memory required of $Heat\ 3D$ stencil is 16 times more than $Heat\ 2D$ stencil. Also note, 3D stencils require a higher number of vector units per SM for optimal performance.

C.4.4 Insight 4: Resource Allocation

Another interesting perspective is seen in Figure C.4 which plots the same design space as in Figure C.3 but this time the axes are different, showing the relative percentages of the chip area devoted to memory, and to vector units. We notice that the optimal designs (blue points) lie in a relative cluster for 3D stencils. This clustering of Pareto points suggests that 60-70% of die area should be dedicated to the vector units and no more than 15% area should be dedicated to shared memory. Ignoring the outliers for 2D stencils, 40-70% area should be allocated to vector units and very little area (less than 5%) should be allocated to the shared memory.

C.5 Related work

Chip Reverse Engineering Chip area modeling can be formally considered a branch of semiconductor reverse engineering, which is a well researched subject area. Torrence et al. [86] gives an overview of the various techniques used for chip reverse engineering. The packaged chips are usually decapped and the wafer die within is photographed layer by layer. The layers are exposed in reverse order after physical or chemical exfoliation. *Degate* [87], for example, is a well known open source software that can help in analyzing die photographs layer by layer. The reverse engineering process can be coarse-grained to identify just the functional macro-blocks. Sometimes, the process can be very fine-grained, in order to identify standard-cell interconnections, and hence, actual logic-gate netlists. Degate is often used in association with catalogs of known standard cell gate layouts, such as those compiled by *Silicon Zoo* [88]. Courbon et al. [89] provides a case study

of how a modern flash memory chip can be reverse engineered using a targeted scanning electron microscope imagery. For chip area modeling, one is only interested in the relatively easier task of demarcating the interesting functional blocks.

Performance Modeling There have been various works on time modeling and performance optimization. Pochoir [39] presents a cache-oblivious approach to avoid performance modeling on CPUs all together. Polymage [77] provides a stencil graph DSL and pairs it with a simple analytical performance model for the automatic computation of optimal tile sizes and fusion choices. With MODESTO [78] an analytical performance model has been proposed that allows modeling multiple cache levels and fusion strategies for both GPUs and CPUs as they arise in the context of Stella. For stencil GPU code generation strategies that use redundant computations in combination with ghost zones an analytical performance model has been proposed [79] that allows to automatically derive "optimal" code generation parameters. Yotov et al. [80] showed already more than ten years ago that an analytical performance model for matrix multiplication kernels allows generating code that is performance-wise competitive to empirically tuned code generated by ATLAS [81], but at this point no stencil computations have been considered. Shirako et al. [82] 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. Grebhahn et al. [142] combine ideas from a product line research with function-learning to predict the performance of a multigrid solver. Bao et al. [143] present an analytical model for the execution of affine programs on set associative caches. In contrast to tools for tuning, Hong and Kim [83] present a precise GPU performance model which shares many of the GPU parameters we use. It is highly accurate, low level, and requires analyzing the PTX assembly code. For stencil GPU code generation strategies that use redundant computations in combination with ghost zones, an analytical performance model has been proposed [79] that allows deriving "optimal" code generation parameters automatically.

Hardware/Software Codesign Application codesign is a well-established discipline and has seen active research for well over two decades [101, 102, 103, 104, 105]. The essential idea is to start with a program (or a program representation, say in the form of a CFDG—Control Data Flow Graph) and then map it to an abstract hardware description, often represented as a graph of operators and storage elements. The challenge that makes codesign significantly harder than compilation is that the hardware is not fixed, but is also to be synthesized. Most systems involve a search over a design space of feasible solutions, and various techniques are used to solve this optimization problem: tabu search and simulated annealing [106, 107] integer linear programming [108].

There are some recent studies on accurately modeling the design space, especially for regular, or *affine control* programs [144, 25, 26]. However, all current approaches solve the optimization problem for a single program at a time. To the best of our knowledge, no one has previously considered the *generalized application codesign* problem, seeking a solution for a *suite of programs*. The research presented by Lee et al. [145] is closest to our work and uses GPUs as the target architecture. The SESH framework [145] takes the source code and workload as input, transforms the code for better performance and later explores different hardware configurations to find optimal performance per area. This process is repeated for each benchmark. Whereas in our work, we consider an application suite with multiple benchmarks as the workload and formulate a mathematical optimization to find the optimal architecture. The work presented in [146, 147] proposes design space exploration as in [145] but for FPGAs.

There are multiple publications on codesign related to exascale computing, but they focus on different aspects. For instance, Dosanji et al. [109] focus on methodological aspects of exploring the design space, including architectural testbeds, choice of mini-applications to represent applications codes, and tools. The ExaSAT framework [110] was developed to automatically extract parameterized performance models from source code using compiler analysis techniques. Performance analysis techniques and tools targeting exascale and codesign are discussed in [7].

There is also a large area of domain-specific accelerator which includes machine learning [148], image processing [149], and linear algebra [150]. While all approaches design custom accelerators none of these approaches has been reported to use non-linear solvers to find optimal design points for performance trade-offs in the accelerator design.

C.6 Conclusion and Discussion

Codesign—the simultaneous design of hardware and software—has two common interpretations. *System codesign* is the problem of simultaneously designing hardware, runtime system, compilers, and programming environments of entire computing systems, typically in the context of HPC systems [7]. *Application codesign*, also called *hardware-software codesign*, is the problem of systematically and simultaneously designing a dedicated hardware platform and the software to execute a single application (program). The approach proposed in this paper is applicable to both contexts. For system codesign, we *specialized* the problem. For application codesign, we *generalize* it in the same domain-specific manner.

We developed a framework for software-hardware codesign for simultaneous optimization of software and hardware parameters. It assumes having analytical models for performance (execution time), and cost (chip area). We use the execution time model from Prajapati et al. [22] for stencil programs. We develop an analytical chip area model for designs from the Maxwell GPU architecture.

We formulated a codesign optimization problem using the time model and our area model for optimizing the compiler and architecture parameters simultaneously. We predict an improvement in the performance of 2D stencils by (104% and 69%) and 3D stencils by (123% and 126%) over existing Maxwell (GTX980 and Titan X) architectures.

The analyses from this paper indicate that for the chosen performance, cost criteria, application profile and specific compiler used, we should remove caches completely and use the area (previously devoted to caches) to add more cores on the chip. If the special purpose system is used as hypothesized, then using part of the chip for cache is not optimal. For other uses and for

general-purpose processing, cache might be useful or even critical element of the design. But our result is not totally unexpected for efficiency-optimized stencil computations. For any program in Polybench [151], data movement can be counted statically. In short, all polyhedral programs do not need caches. With these techniques of using models for tile sizes, all data transfers can be statically predicted and tuned. Therefore, ideal architecture for polyhedral programs are the ones that do not have caches.

Our goal is to design accelerator for a suite of applications (stencils). We do not target a single benchmark. Therefore, we need the programmability and ASICs are not programmable. The fact that schedulers are able to statically predict the data transfers makes caches unfruitful. Cache-less architecture are ideal for polyhedral programs. Moreover, the advantage of using GPUs in our approach is that we just need to predict optimal values for several parameters; designing an ASIC is a much more complex and involved task. It is about “how to design *instruction-set-programmable* accelerators." We use GPUs as accelerators unlike other works that design FPGA based system. In an FPGA based accelerator design, it is essential that the accelerator must be “redesigned" for each application, and sometimes for each changing size parameter of the input program. Hence FPGA based design space exploration is forced to solve a new optimization problem for each application. Using GPU-like instruction set programmable targets is one of the novelties of our work.

However, our models are approximate and other parameters play an important role in, e.g., designers may seek to optimize for both execution time as well as power simultaneously. Our approach can be extended to consider energy/power consumption into account. If the energy consumption details of the individual components of the architecture are known, then the objective function can be updated to be the argmin of the weighted execution times and energy components where we seek to maximize the performance. Such an optimization function can be formulated to solve power-gating problems where the designer wants to turn off certain parts of the chip.

The execution time model in [22] does not account for register usage, known to be critical for performance because the size of the register file constrains the optimal tile sizes. In our area model, we fix the register file size and do not include register files as constraints of the optimization.

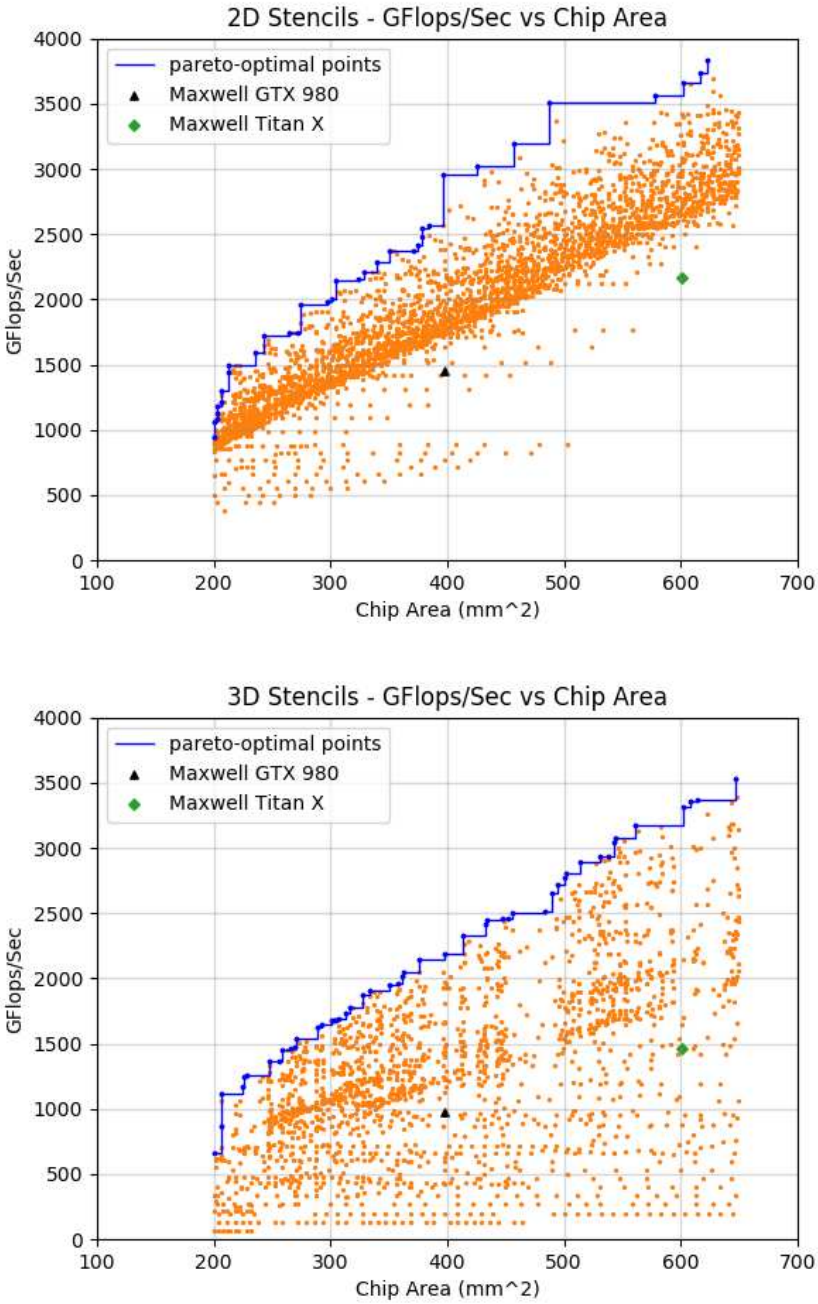


Figure C.3: Optimal performance (in GFLOPs/Sec) of each feasible design point as a function of total chip area for 2D stencils (left) and 3D stencils (right). Each benchmark and input size combination is assumed to occur with equal frequency. Of the many thousands of points ($\approx$ 3000 for 2D stencils and $\approx$ 2000 for 3D) many are *dominated* by other design choices–a different architecture with smaller area yields better performance. So, only a few tens of design points (the Pareto optimal points, shown in blue) need to be explored further, a nearly 100-fold savings in design cost. In addition, the performance of the existing NVIDIA Maxwell GTX980 and Titan X are also shown. The optimized designs for comparable area budget could improve performance of 2D stencils by as much as 104% (c.f. 69%) relative to the GTX980 (c.f., the TitanX). The performance of 3D stencils can be improved by as much as 123% (c.f. 126%) relative to the GTX980 (c.f., the TitanX).

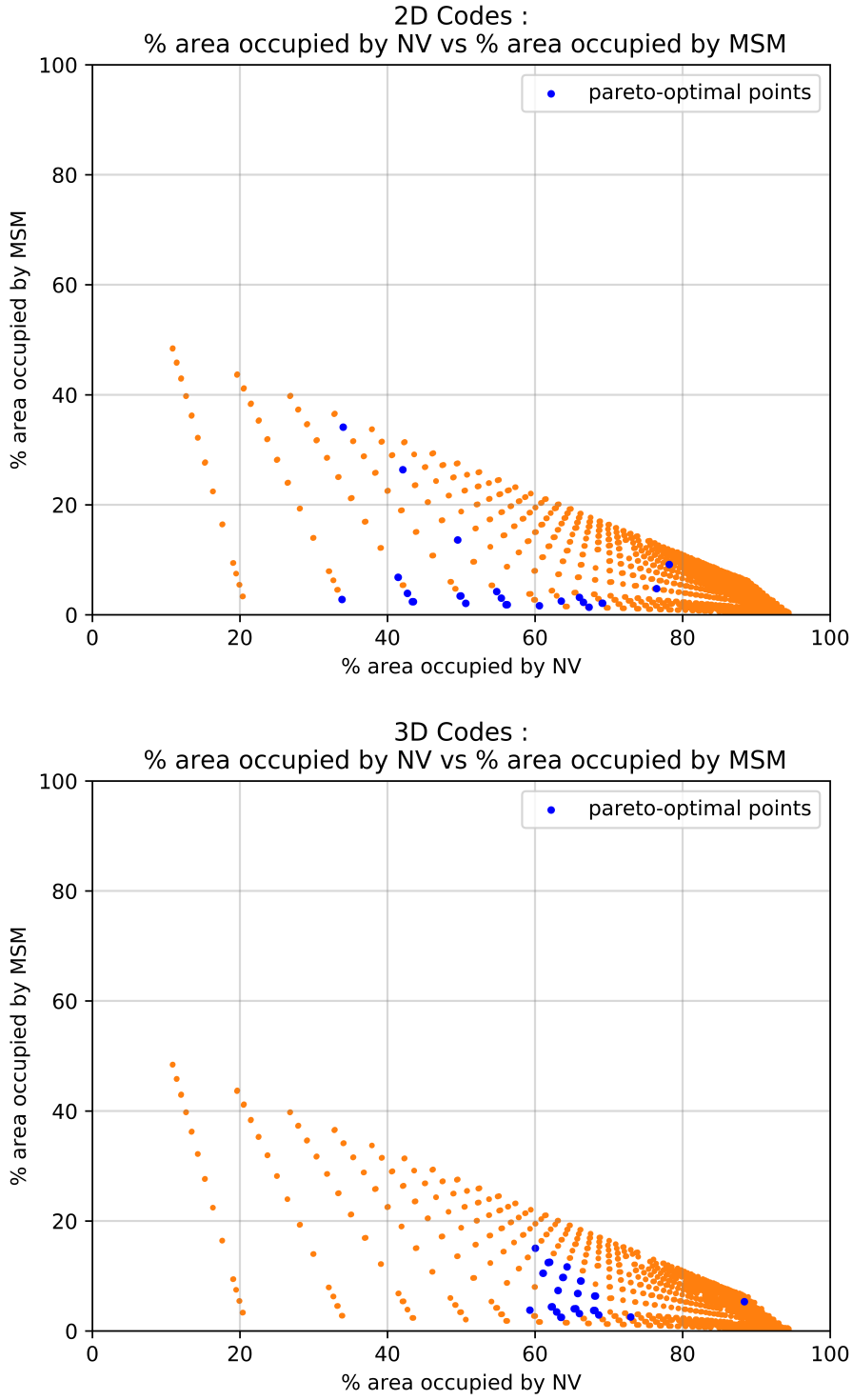


Figure C.4: Resource Allocation.

Appendix D

Minimizing Data Movement for Matrix Chain Products

Data Movement is expensive, in terms of both, execution time and energy consumption of computing systems. Moving data across memory hierarchy consumes a large portion of system energy. For consumer devices such as smart phones, tablets, and wearable devices, the data movement between main memory and the compute units consumes, on average, 62.7% of the total system energy [152]. The energy consumption due to data movement is also one of the exascale challenges [2, 3, 4, 5, 6]. Energy consumption of any application can be divided into two major components: *static energy* and *dynamic energy*. The static energy component varies with the execution time performance. The dynamic energy component can be further divided into two high level components: computation and communication. In this work, we seek to minimize the total energy consumption for matrix chain products. First, we use the classic matrix chain ordering algorithm [61] to minimize the total op-count (reduce the static as well as dynamic energy) for a given matrix sequence. Next, we develop a *hyper-node parenthesization* algorithm to minimize the off-chip communication over the minimum op-count tree, in turn reducing the dynamic component of energy consumption.

The *hyper-node parenthesization* algorithm builds over data movement costs of a single matrix multiplication and a fused two matrix multiplication. We assume that all matrices are too large to fit on-chip and that the memory hierarchy consists of two levels: fast and slow. Matrix multiplication for those matrices that do not fit into the fast memory capacity of an architecture involves blocking of the input and output matrices. The input and output matrices are transferred between fast and slow memory in blocks. The size of these blocks dictate the total data transfer cost [153, 154, 155] of matrix multiplication. We revisit closed form solutions for these block sizes that minimize data transfers between slow and fast memory, given the fast memory capacity. For matrix chain

products, two matrices are multiplied to produce an intermediate result matrix and this intermediate result is multiplied with another matrix (or intermediate result). A reduction in data movement is possible if the intermediate result is not written to the slow memory, but is consumed as soon as it is produced. We call this, *Two MM*. We develop closed form solutions for block sizes that minimize data movement for *Two MM*. Next, we build a recurrence that uses these closed form solutions to minimize global data movement for matrix chains.

The main contributions are:

- ***Two MM*** closed form solutions for block sizes to minimize off-chip data movement for two matrix multiplications.
- ***Hyper-node Parenthesization*** a recurrence that minimizes data movement over a minimum op-count tree for matrix chain products.

We apply ***Hyper-node Parenthesization*** recurrence over a set of $19,000$ matrix sequences of lengths $\{2, 3, 4, \ldots, 20\}$ and show that off-chip data transfers can be reduced by 9% (and 13%) on average for architectures with 64KB (and 96KB) of on-chip memory capacity.

The rest of the chapter is organized as follows. Section D discusses related work. Section D.1 develops closed form solutions for block sizes to minimize off-chip data transfers for single matrix multiplication. Section D.2 develops closed form solutions for block sizes that minimize off-chip data transfers for *Two MM*. Section D.3.2 discusses the classic matrix chain ordering algorithm [61] that minimizes the total number of computations. Section D.4 produces a recurrence to minimize data transfers for a given minimum op count tree. Section D.5 further reduces the data transfers for a minimum op count tree using loop fusion. Finally, section D.7 concludes the chapter.

D.1 Related Work

In 1981, Hong and Kung [153] showed that a sequential matrix multiplication algorithm transfers $\Omega(n^3/\sqrt{M})$ words between fast and slow memory (M is the size of the slow memory and n is the size of matrices). This lower bound is attained by blocking/tiling algorithms that slice the three

matrices into square blocks of size $\sqrt{M/3}$, which means that three faces (one of each matrix) of a cubic tile occupy the fast memory. Irony et al. [156] extended this algorithm for distributed memory machines and showed that for $2D$ algorithms such as Cannon [154] and SUMMA [157], the communication lower bounds are $\Omega(n^3/\sqrt{P})$ where each processor P stores at least $M = O(n^2/P)$ words. Irony et al. [156] also showed that for $3D$ algorithms such as Agarwal et al. [155] and Johnsson [158], the communication lower bounds are $\Omega(n^2/P^{2/3})$ where each processor P stores at least $M = O(n^2/P^{2/3})$ words. For sequential BLAS computations, Frigo et al. [159] developed cache oblivious algorithms and showed that the blocked versions of the naive algorithms attain the lower bound in the two-level memory model for matrix multiplication. In 2011, Solomonik and Demmel [160] came up with a $2.5D$ communication-avoiding algorithm that stores c/p data in memory, where c is a tunable parameter ranging between 1 to $\sqrt{3p}$, and allows the algorithm to interpolate between $2D$ and $3D$ cases. McColl and Tiskin [161], Solomonik and Demmel [160] and Demmel et al. [162] proposed parallel classical algorithms attaining communication lower bounds with no assumption on local memory use. These algorithms have computational cost $\theta(n^3/P)$ and may use local memory of size more than $\theta(n^2/P)$. Ballard et al. [163] provide an excellent survey of methods for finding communication lower bounds, communication avoiding algorithms, and cache-oblivious blocking techniques for numerical linear algebra. For single matrix-multiplication, we provide a simple derivation of well known closed form solutions for tile sizes that minimize off-chip communication. These solutions can be further refined to maximally exploit available hardware resources. For example, Kim et al. [164] developed cost models for efficient data movement (coalescing, register tiling, etc) of tensor contractions over GPUs.

Loop fusion is a transformation that merges multiple loops into single loop. It is used to improve data locality of programs. In 1993, Kennedy and McKinley [165] showed that applying fusion transformation for maximizing data locality is NP-hard and proposed a new algorithm for fusing a collection of sequential and parallel loops. Singhai and McKinley [166] developed a parameterized loop fusion algorithm and modeled data locality, parallelism, and register pressure. Qasem and Kennedy [167] combined loop fusion with loop tiling and presented a model-guided

empirical tuning strategy at different levels of memory hierarchy. Bondhugula et al. [168] presented a model that captures the utilization of hardware prefetch streams, loss of parallelism, constraints imposed by privatization and code expansion into a single convex optimization space. Unlike these strategies that can be applied to different types of problems, our work focuses on fusing two matrix multiplications. We develop closed form solutions that guarantee minimum off-chip data movement for fused *Two MM* kernels for two-level memory architectures.

Pouchet et al. [169] developed an automatic parallel framework that searches empirically the set of valid possibilities to perform fusion/code motion transformations, and relies on model-based mechanisms to perform tiling, vectorization and parallelization on the fusion-transformed program. They later introduced a fusibility concept [170] that generalizes legality conditions and enables fusion transformations. They developed cost models to tune for trade-off between parallelism and locality at different levels. The fusibility decision considers all possible loop permutations and minimizes the dependence distance. For matrix chain products, they explore the space of possible fusion/distribution structures and tiling but the input matrices were square and hence fusion and tiling of all dimensions is legal. Our work focuses on minimizing reuse distance for matrix chain products with rectangular matrix sizes. However, with rectangular matrices fusion of only the two outer-loops is possible as discussed in section D.2. Therefore, Pouchet et al. [170] has a larger optimization space compared to the space discussed in this work and they explore *k-fuse*: fusion of more than two matrix multiplications. Their decisions depend on the problem and the underlying architecture considering square (single problem size) matrices and are made statically (at compile-time). Our work takes as input the problem sizes and architecture parameters and minimizes the off-chip data movement over a minimum op-count tree dynamically (at run-time).

Rajbhandari et al. [171, 172] developed a compile-time approach to dependence characterization and program transformation to enable fusion across recursively specified tree-traversal operators on k-ary trees. They develop necessary and sufficient conditions for fusion across a collection (directed acyclic graph) of k-ary tree operators with producer-consumer relationships. They consider multiple types of operators for a k-ary tree and explore the fusion choices for multiple related

(parents, children, and sibling) nodes in the tree. For sequences of loops representing a producer-consumer relationship, Rajbhandari et al. [173] presented an approach to use data movement lower bounds to reason about the utility of loop fusion to reduce data movement and its application to find an optimal fusion choice for the four-index integral transform. Similar to our work, they consider two-level memory architecture. However, they consider a single input problem size.

Barthels et al. [174] extend the matrix chain multiplication problem to generalized matrix chain problem (GMCP) with inverse and transpose operations. The matrices can also be different shapes: lower and upper triangular, symmetric, diagonal, and symmetric positive-definite. Their parenthesization decision is based on the number of flops required considering matrix shape and operation. For this work, we consider only rectangular matrices without any inverse or transpose operators. Also, our cost function consists of both op-count as well as data movement. In the future we plan to extend our methods of minimizing off-chip data transfers to the GMCP problem.

D.2 Off-chip Data Movement for Single Matrix Multiplication

Consider the following matrix multiplication:

$$R = A_1 A_2 \tag{D.1}$$

where P_0, P_1, P_2 are the problem sizes. Our goal is to reduce off-chip data transfers. We consider three options: keep either of the R, A_1, or A_2 matrices in the on-chip memory. We develop the following for GPU architectures, with shared memory as the fast (on-chip) memory.

Our goal is to reduce global memory accesses (off-chip data transfers) for matrix multiplication equation D.1 where the matrices R, A_1, and A_2 are too large to fit into the shared memory capacity of a GPU. For such large problem sizes, one of the common techniques is to tile all the three dimensions of the iteration space [155, 158]. Consider tile sizes x, y, and z as shown in the Figure D.1. There are three options: retain one of the R, A_1, or A_2 matrices in the shared memory.

$$R_{ij} = \sum_{k} A_{1_{ik}} * A_{2_{kj}}$$

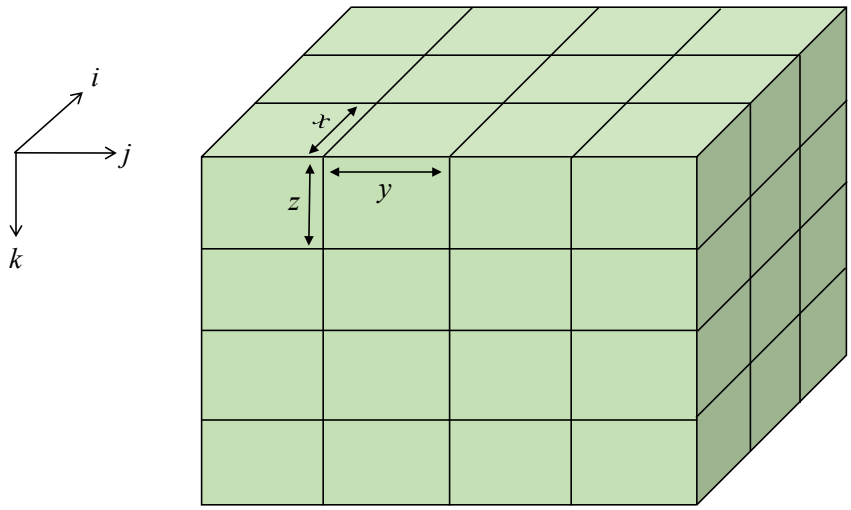


Figure D.1: 3D blocking of Matrix Multiplication

Case 1: Store R in the shared memory

In this case, a threadblock is responsible for executing all the tiles along k. The partial answers produced by the previous tile are consumed from shared memory and are overwritten. This means that the output (a patch of R) generated by a tile is retained in the shared memory and is written back to the global memory only after the final answer is computed. The shared memory required to store the result per tile is given by $x * y$. As shown in figure D.2, a $x * z$ patch of matrix A_1 and a $y * z$ patch of matrix A_2 is read from the global memory to compute $x * y$ patch of R.

The total number of off-chip data transfers can be calculated as

$$\begin{aligned} h_R &= \left(\frac{P_0}{x} * \frac{P_2}{y} * \frac{P_1}{z}\right) * (xz + yz) + P_0P_2 \\ &= P_0P_1P_2\left(\frac{x+y}{xy}\right) + P_0P_2 \end{aligned} \tag{D.2}$$

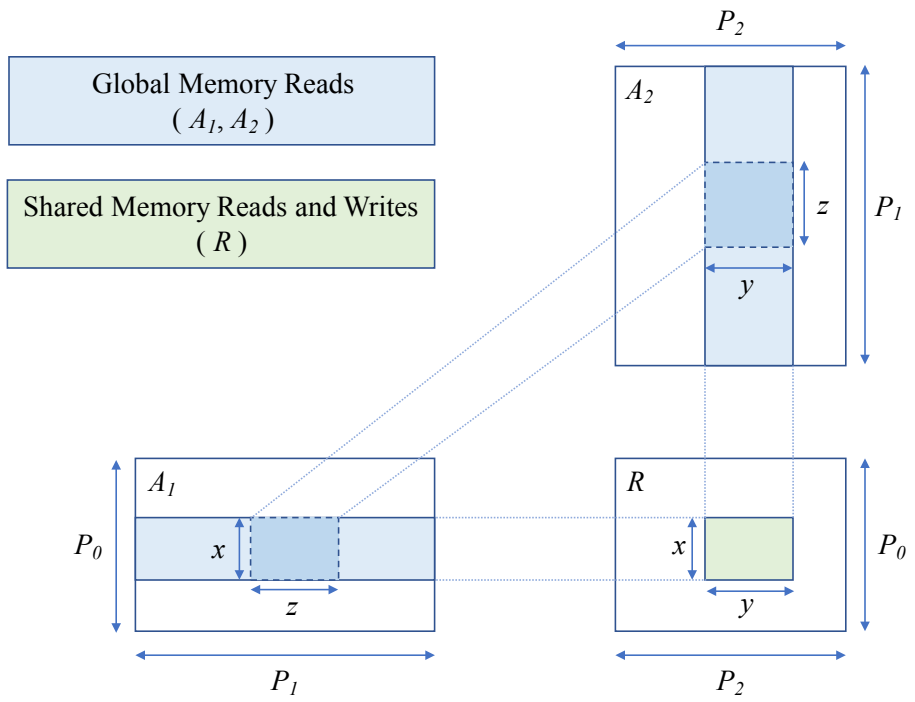


Figure D.2: Retain the output matrix R in the shared memory.

Note, z cancels out in the equation. The matrix R (of size P_0P_2) is written to off-chip memory only once, which is after the final answer is computed. In order to reduce the off-chip data transfers in h_R, the term $(x+y)/xy$ should be minimized. If $xy = c$ is a constant, the value of $(x+y)/xy$ decreases as the value of $x+y$ decreases, and the value of $x+y$ is minimum when $x = y$. The constraints on the tile sizes, x and y, are $0 < x \leqslant P_0$ and $0 < y \leqslant P_2$. Also, if the shared memory capacity (per threadblock) is M, then xy must fit into M. Figure D.3 shows the $xy = M$ hyperbola. The minimum value of $(x+y)/xy$ is found at the point where the line $x = y$ intersects with $xy = M$ hyperbola.

The on-chip memory capacity is M, and $xy = x^2$ must fit into M. i.e., $x = \sqrt{M}$ where $x \leqslant P_0, P_2$, call this x^*. For the example in figure D.3, the bounds on x and y are $P_0 = 60$ and $P_2 = 50$. The on-chip memory capacity is 64 and $x = y = \sqrt{M} = 8$. Substituting $x = y$ in equation (D.2) and using the closed form solution for tile sizes $x = y = x^* = \sqrt{M}$, we get h_R^*, the minimum off-chip data transfers required in the case where the output matrix R is stored on-chip, as

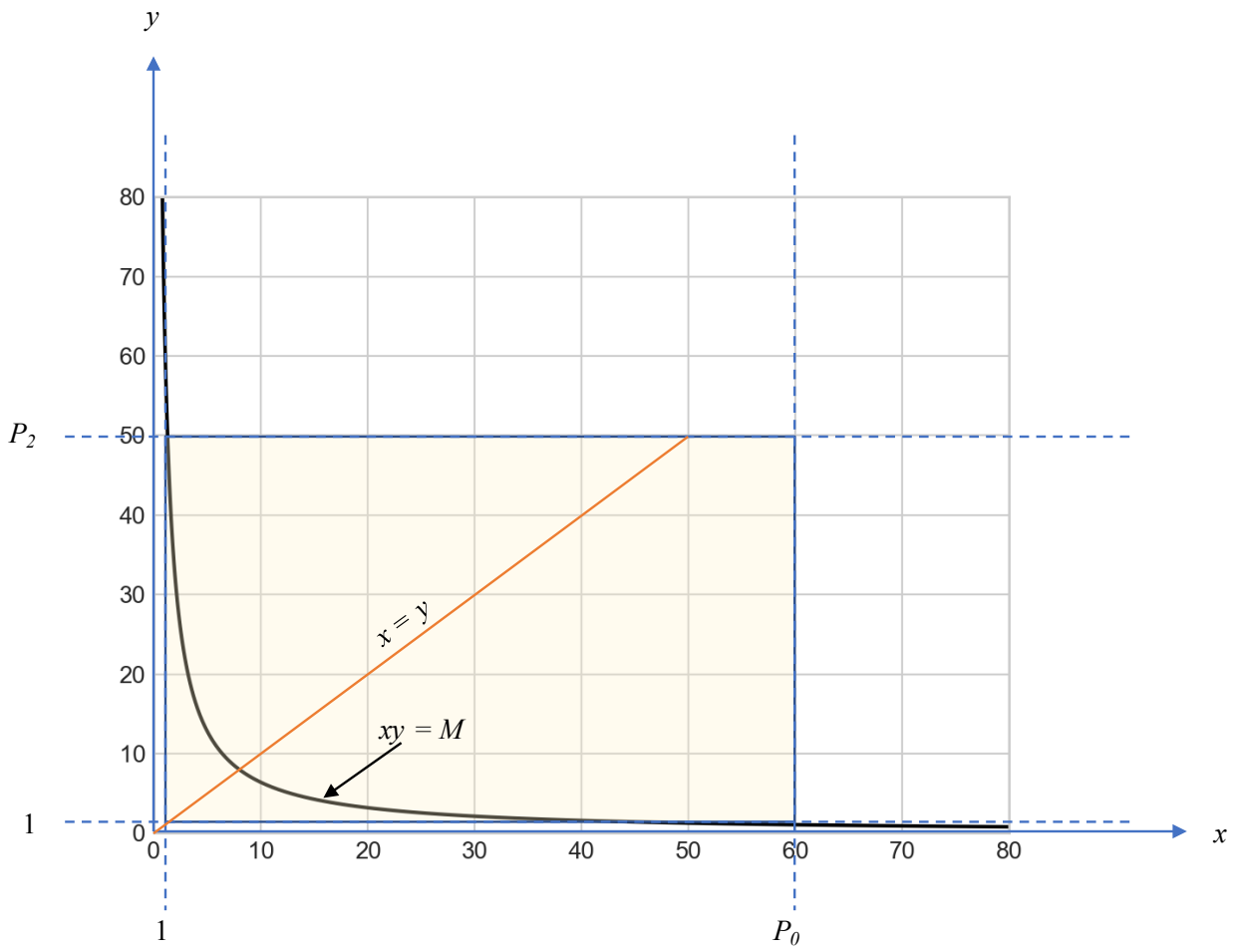


Figure D.3: Minimizing the value of $(x + y)/xy$ for the case when matrix R is stored on-chip.

$$\begin{aligned} h_R^* &= \frac{2P_0P_1P_2}{x^*} + P_0P_2 \\ &= \frac{2P_0P_1P_2}{\sqrt{M}} + P_0P_2 \end{aligned} \tag{D.3}$$

Case 2: Store A_1

Alternatively, the matrix A_1 can be stored in the shared memory (i.e., A_1 is read exactly once from the global memory), matrix $A2$ can be read, and the matrix R can be read & written from global memory. In this case, a threadblock computes all tiles in the $i - th$ row of R. The shared memory requirement per tile is given by $x * z$. As shown in Figure D.4, a $x * z$ patch of matrix A_1 stays in the shared memory. A patch of matrix A_2 of size $y * z$ is read from the global memory to compute $x * y$ patch of R. The values of $x * y$ patch of R are read and written.

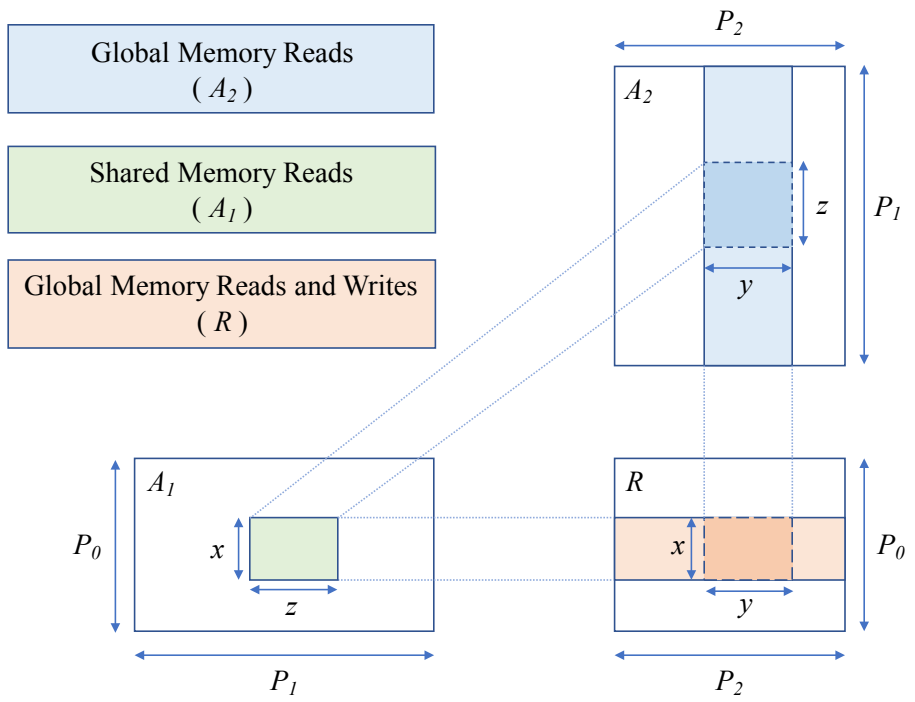


Figure D.4: Retain the input matrix A_1 in the shared memory.

The total number of off-chip data transfers is given by

$$\begin{aligned} h_A &= \left(\frac{P_0}{x} * \frac{P_2}{y} * \frac{P_1}{z}\right) * (2xy + yz) + P_0P_1 - P_0P_2 \\ &= P_0P_1P_2\left(\frac{2x+z}{xz}\right) + P_0P_1 - P_0P_2 \end{aligned} \tag{D.4}$$

Note the term P_0P_1, i.e., matrix A_1, is read only once. We also subtract the term P_0P_2 because we assume that the first read of matrix R is not needed because it is initialized to 0. To reduce the off-chip data transfers h_{A_1}, the term $(2x+z)/xz$ should be minimized. The constraints on x and z are $0 < x \leqslant P_0$ and $0 < z \leqslant P_1$. The objective is to minimize $(2x+z)/xz$ such that $x * z$ fits into the shared memory M. The value of $(2x+z)/xz$ decreases as the value of x increases. Similar to the analysis in Case 1, the minimum value of $(2x+z)/xz$ is found at $z = 2x$.

The shared memory capacity is M, $xz = 2x^2$ must fit into M. i.e., $x = \sqrt{M/2}$ where $x \leqslant P_0$ and $z = 2x = \sqrt{2M}$ where $z \leqslant P_1$. Substituting $z = 2x$ in equation(D.4) and using the maximum value of x given by $x^* = \sqrt{M/2}$, we get $h^*_{A_1}$, the minimum off-chip data transfers required in the case where the output matrix R is stored on-chip, as

$$h^*_{A_1} = 2\sqrt{2}P_0P_1P_2\frac{1}{\sqrt{M}} + P_0P_1 - P_0P_2 \tag{D.5}$$

Case 3: Store A_2

One can also choose to keep the matrix A_2 on-chip (shared memory), read matrix A_1 as needed (from off-chip memory), and read & write R (from off-chip memory). Similar to Case 2, we can obtain a closed form solution for the total amount of off-chip data transfers as

$$h^*_{A_2} = 2\sqrt{2}P_0P_1P_2\frac{1}{\sqrt{M}} + P_1P_2 - P_0P_2 \tag{D.6}$$

i.e., $x = \sqrt{M/2}$ where $x \leqslant P_0$, and $y = \sqrt{2M}$ where $y \leqslant P_2$.

We would like to reiterate the fact that our problem formulation assumes that the problem sizes are too large and the matrices do not fit in the on-chip memory. Therefore, note that the closed form solution, $h^*_{A_1}$ (and $h^*_{A_2}$), does not hold true for all problem sizes. Consider the case where $P_1 = 1$. The cubic term in $h^*_{A_1}$ becomes quadratic and depending on the value of $\sqrt{M}$, the value of $h^*_{A_1}$ might become negative. This situation is not possible in our scenario where the problem sizes are too large to fit matrices in the on-chip memory. To be more precise, we add a constraint that the problem sizes must be greater than $\sqrt{M}$. This guarantees that $h^*_{A_1}$, and $h^*_{A_2}$ are positive.

In the next section we extend this analysis to the case of multiplying 3 matrices.

D.3 Off-chip Data Movement for Two Matrix Multiplications

Let $R = A_1A_2A_3$. By associativity, there are two ways to multiply these matrices: $(A_1A_2)A_3$ and $A_1(A_2A_3)$. Note that there are two matrix multiplications involved and the closed form solutions developed in the previous section can be applied to the first matrix multiplication to produce an intermediate result and then to the second matrix multiplication. This method involves writing the intermediate result to the off-chip memory. We develop solutions that avoid off-chip data movement for the intermediate result.

D.3.1 Option 1: $R = (A_1 A_2) A_3$

Let $T = A_1 A_2$ be the intermediate result produced by multiplying the matrices A_1 and A_2. Conventionally, this intermediate result is stored in the off-chip memory. However, this result can be consumed as soon as it is produced — it is not necessary to write it to off-chip memory. Let us see how.

The code structure for multiplying the matrices A_1 and A_2 to produce the intermediate result T is shown in the Listing 2. The code structure for multiplying the intermediate result T with input matrix A_3 to produce the final answer R is shown in the Listing 3. We permute the loops j_2 and k_2 as shown in the Listing 4. The outermost two loops in the Listings 2 and 4 can be fused to form the code structure as shown in the Listing 5. The indices i_1 in Listing 2 and i_2 in the Listing 4 are referred as i in Listing 5. Similarly, the indices j_1 in Listing 2 and k_2 in the Listing 4 are referred as j in Listing 5.

Algorithm 2: $T = A_1 * A_2$

```
for i1 ∈ 1, ..., P0 do
    for j1 ∈ 1, ..., P2 do
        for k1 ∈ 1, ..., P1 do
            T[i1][j1] += A1[i1][k1] * A2[k1][j1];
        end
    end
end
```

Algorithm 3: $R = T * A_3$

```
for i2 ∈ 1, ..., P0 do
    for j2 ∈ 1, ..., P3 do
        for k2 ∈ 1, ..., P2 do
            R[i2][j2] += T[i2][k2] * A3[k2][j2];
        end
    end
end
```

Algorithm 4: $R = T * A_3$ (loop permutation)

```
for i2 ∈ 1, ..., P0 do
    for k2 ∈ 1, ..., P2 do
        for j2 ∈ 1, ..., P3 do
            R[i2][j2]+ = T[i2][k2] * A3[k2][j2];
        end
    end
end
```

Algorithm 5: $R = (A_1 * A_2) * A_3$ (loop fusion)

```
for i ∈ 1, ..., P0 do
    for j ∈ 1, ..., P2 do
        for k1 ∈ 1, ..., P1 do
            T[i][j]+ = A1[i][k1] * A2[k1][j];
        end
        for j2 ∈ 1, ..., P3 do
            R[i][j2]+ = T[i][j] * A3[j][j2];
        end
    end
end
```

Figure D.5 shows the data movement requirements for the fused code in Listing 5. Consider the tile sizes x, y, z, and z' as shown in the figure D.5. The idea is to retain the $x * y$ patch of the intermediate matrix T in the shared memory and consume it as soon as it is produced. This means that we avoid writing the matrix T of size P_0P_2 to the global memory. The matrices A_1, A_2, and A_3 are read from the global memory. The output matrix R is read and written from the global memory multiple times.

The number of data transfers required for computing the $x * y$ tile of the intermediate matrix T is given by $\frac{P_0}{x} * \frac{P_2}{y} * \frac{P_1}{z} * (xz + yz)$. And the number of data transfers required to compute the output matrix R is given by $\frac{P_0}{x} * \frac{P_2}{y} * \frac{P_3}{z'} * (2xz' + yz') - P_0P_3$. Therefore, total data movement cost of computing the output $R = (A_1A_2)A_3$ via fusion is

$$h_{lc} = P_0P_1P_2\frac{x+y}{xy} + P_0P_2P_3\frac{2x+y}{xy} - P_0P_3 \tag{D.7}$$

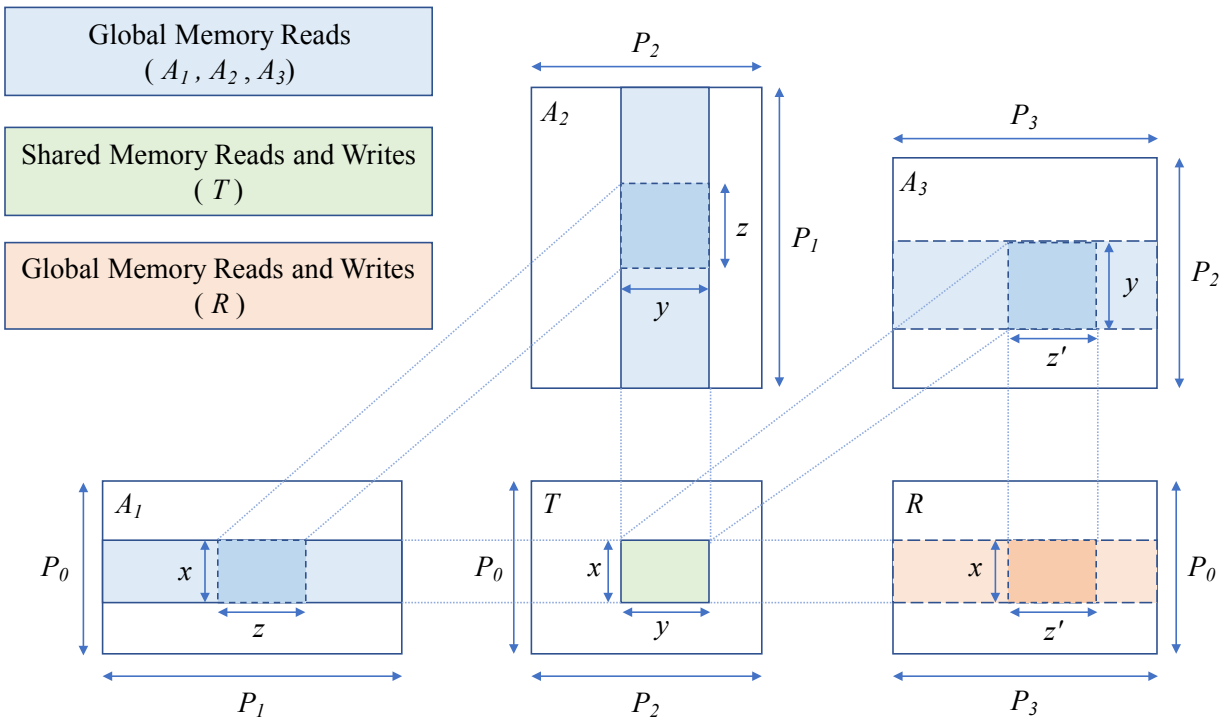


Figure D.5: Data Movement for fused code: $R = (A_1 A_2) A_3$

Note that the terms z and z' cancel out. Let $P_3/P_1 = \alpha$. The problem sizes P_3 and P_1 are greater than 0. Therefore, $\alpha > 0$. Substituting $P_3/P_1 = \alpha$, we get

$$\begin{aligned} h_{lc} &= P_0 P_1 P_2 \frac{x + y + 2\alpha x + \alpha y}{xy} - P_0 P_3 \\ &= P_0 P_1 P_2 \frac{(1 + 2\alpha)x + (1 + \alpha)y}{xy} - P_0 P_3 \end{aligned} \tag{D.8}$$

Let $(1 + 2\alpha)/(1 + \alpha) = \alpha'$ and observe that $1 < \alpha' < 2$, sine $\alpha > 0$. Substituting this in the above equation

$$h_{lc} = P_0 P_1 P_2 (1 + \alpha) \frac{\alpha' x + y}{xy} - P_0 P_3 \tag{D.9}$$

In order to minimize the value of h_{lc}, the term $(\alpha' x + y)/xy$ must be minimized. The constraints are $1 \leqslant \alpha' \leqslant 2$, $1 \leqslant x \leqslant P_0$, and $1 \leqslant y \leqslant P_2$. The objective function $(\alpha' x + y)/xy$ is minimum at $\alpha' x = y$, where xy fits the shared memory capacity of M, Therefore, $x * y = x(\alpha' x) = \alpha' x^2$

must fit into the shard memory capacity (per threadblock), M, of the GPU. $\alpha' x^2 = M$, which gives $x = \sqrt{M/\alpha'}$ and $y = \sqrt{M\alpha'}$. Substituting these in the above equation D.9

$$h^*_{lc} = \frac{2P_0P_1P_2(1+\alpha)\sqrt{\alpha'}}{\sqrt{M}} - P_0P_3 \tag{D.10}$$

This gives us the closed form solution for minimum data movement requirement for fusing the computation of *left child* $T = (A_1A_1)$ with the second matrix multiplication with matrix A_3. If any of the problem sizes P_1 or P_2 are less than $\sqrt{M}$, the value for h^*_{lc} can be negative. Therefore, the closed form solution h^*_{lc} holds true only for the case where problem sizes that are larger than $\sqrt{M}$. Next we show how the *right child* $T = (A_2A_3)$ can be fused with the output R.

D.3.2 Option 2: $R = A_1(A_2A_3)$

Let $T = A_2A_3$ be the intermediate result produced by multiplying the matrices A_2 and A_3. We apply loop permutation and fusion techniques similar to the case of $R = (A_1A_2)A_3$. The code structure for multiplying the matrices A_2 and A_3 to produce the intermediate result T is shown in the Listing 6. The code structure for multiplying the input matrix A_3 with the intermediate matrix T to produce the final output matrix R is shown in the Listing 7. We permute the loops i_2 and k_2 as shown in the Listing 8. The outermost two loops in the Listings 6 and 8 can be fused to form the code structure as shown in the Listing 9. The indices i_1 in Listing 6 and k_2 in the Listing 8 are referred as i in Listing 5. Similarly, the indices j_1 in Listing 6 and j_2 in the Listing 8 are referred as j in Listing 9.

Algorithm 6: $T = A_2 * A_3$

```
for i1 ∈ 1, ..., P1 do
    for j1 ∈ 1, ..., P3 do
        for k1 ∈ 1, ..., P2 do
            T[i1][j1]+ = A2[i1][k1] ∗ A3[k1][j1];
        end
    end
end
```

Algorithm 7: $R = A_1 * T$

```
for i2 ∈ 1, ..., P0 do
    for j2 ∈ 1, ..., P3 do
        for k2 ∈ 1, ..., P1 do
            R[i2][j2]+ = A1[i2][k2] ∗ T[k2][j2];
        end
    end
end
```

Algorithm 8: $R = A_1 * T$ (loop permutation)

```
for k2 ∈ 1, ..., P1 do
    for j2 ∈ 1, ..., P3 do
        for i2 ∈ 1, ..., P0 do
            R[i2][j2]+ = A1[i2][k2] ∗ T[k2][j2];
        end
    end
end
```

Algorithm 9: $R = A_1 * (A_2 * A_3)$ (loop fusion)

```
for i ∈ 1, ..., P1 do
    for j ∈ 1, ..., P3 do
        for k1 ∈ 1, ..., P2 do
            T[i][j]+ = A2[i][k1] ∗ A3[k1][j];
        end
        for k2 ∈ 1, ..., P0 do
            R[k2][j]+ = A1[k2][i] ∗ T[i][j];
        end
    end
end
```

Figure D.6 shows the data movement requirements for the fused code in 9. Consider the tile sizes x, y, z, and z' as shown in the figure D.6. The idea is to retain the $x * y$ patch of the intermediate matrix T in the shared memory and consume it as soon as it is produced. This means that we avoid writing the matrix T of size P_1P_3 to the global memory. The matrices A_1, A_2, and

A_3 are read from the global memory. The output matrix R is read and written from the global memory.

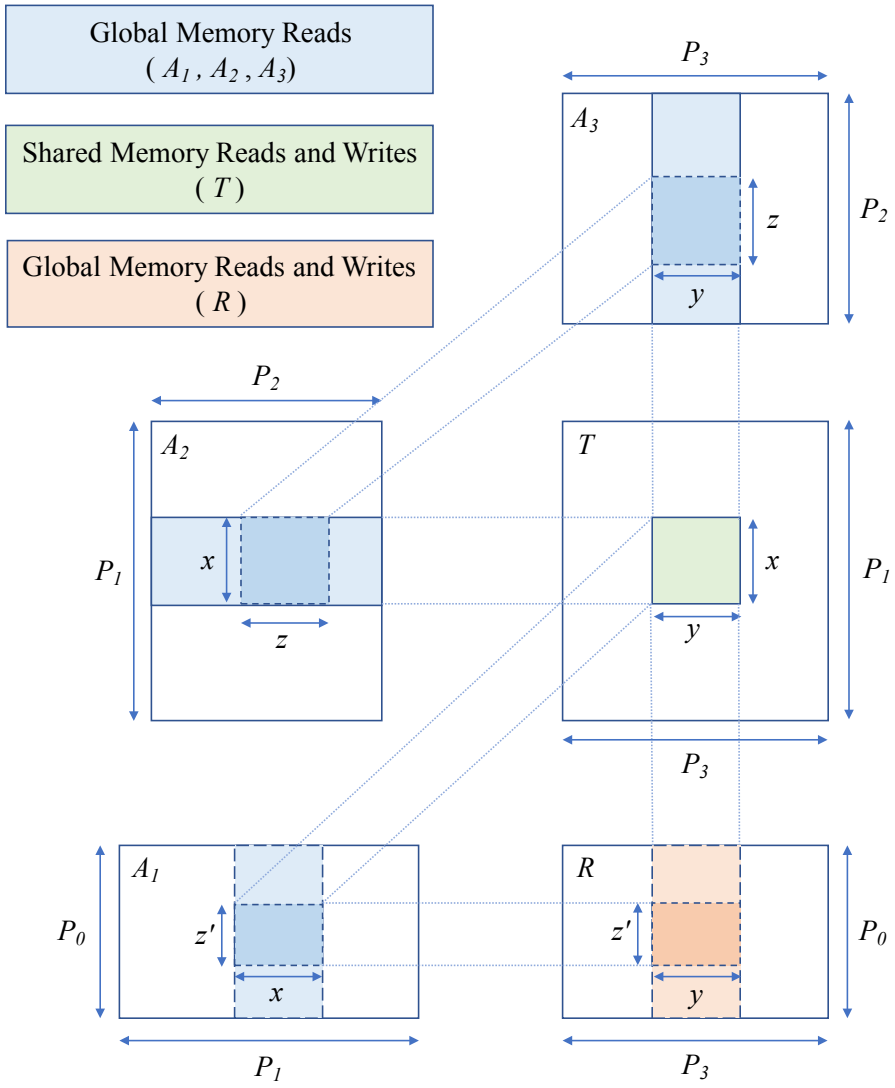


Figure D.6: Data Movement for fused code: $R = A_1(A_2A_3)$

The amount of data transfer required for computing the $x * y$ tile of the intermediate matrix T is given by $\frac{P_1}{x} * \frac{P_3}{y} * \frac{P_2}{z} * (xz + yz)$. And the amount of data transfers required to compute the output matrix R is given by $\frac{P_1}{x} * \frac{P_3}{y} * \frac{P_0}{z'} * (2yz' + xz') - P_0P_3$. Therefore, total data movement cost of computing the output $R = (A_1A_2)A_3$ via fusion is

$$h_{rc} = P_1 P_2 P_3 \frac{x+y}{xy} + P_0 P_1 P_3 \frac{x+2y}{xy} - P_0 P_3 \tag{D.11}$$

Note that the terms z and z' cancel out. Let $P_0/P_1 = \beta$. The problem sizes P_0 and P_1 are greater than 0. Therefore, $\beta > 0$. Substituting $P_0/P_1 = \beta$, we get

$$\begin{aligned} h_{rc} &= P_1 P_2 P_3 \frac{x+y+\beta x+\beta y}{xy} - P_0 P_3 \\ &= P_1 P_2 P_3 \frac{(1+\beta)x+(1+2\beta)y}{xy} - P_0 P_3 \end{aligned} \tag{D.12}$$

Let $(1+2\beta)/(1+\beta) = \beta'$ and observe that β' lies between 1 and 2, since $\beta > 0$. Substituting this in the above equation

$$h_{rc} = P_1 P_2 P_3 (1+\beta) \frac{x+\beta' y}{xy} - P_0 P_3 \tag{D.13}$$

In order to minimize the value of h_{rc}, the term $(x+\beta' y)/xy$ must be minimized. The constraints are $1 \leqslant \beta' \leqslant 2$, $1 \leqslant x \leqslant P_1$, and $1 \leqslant y \leqslant P_3$. The value of this hyperbola is minimized at $x = \beta' y$. The objective function is given by $(\beta' y + \beta' y)/y(\beta' y) = 2/y$. The value of $2/y$ is minimized when y is maximized. Therefore, we maximize the value of y such that $x * y = (\beta' y) * y = \beta' y^2$ fits into the shard memory capacity (per threadblock), M, of the GPU. $\beta' y^2 = M$, which gives $y = \sqrt{M/\beta'}$ and $x = \sqrt{M\beta'}$. Substituting in the above equation D.13

$$h^*_{rc} = \frac{2 P_1 P_2 P_3 (1+\beta)\sqrt{\beta'}}{\sqrt{M}} - P_0 P_3 \tag{D.14}$$

This gives us the closed form solution for fusing the *right child* $T = (A_2 A_3)$ with the output R. In the next section, we discuss the state-of-the-art matrix chain ordering algorithm that minimizes the total number of computations. The matrix chain ordering algorithm produces a minimum op-count tree. We apply the technique of fusion to minimize data movement for this tree.

D.4 Optimal Parenthesization: Op Count

A sequence of matrices $A_1, A_2, A_3 \ldots, A_n$ with problem sizes $P_0, P_1, P_2 \ldots, P_n$, are to be multiplied. Matrix multiplication is associative, therefore, there are many ways to parenthesize the chain. The classic textbook dynamic programming algorithm seeks to find the parenthesization that minimizes the total number of computations [61]. We refer to this algorithm as ***OP-opcount***. The algorithm is shown in Figure 10.

Algorithm 10: OP-opcount

Input : $P[0...n]$: matrix sizes
Output: $opcount[i][j]$: minimum number of computations required to compute the subsequence $A_{i...j}$
$split[i][j]$: the value k that determines an optimal split for $A_{i...j}$

```
for i ∈ 1, ..., n do
    opcount[i][i] = 0;
end
for l ∈ 1, ..., n − 1 do
    for i ∈ 1, ..., n − l do
        j = i + l;
        opcount[i][j] = ∞;
        for k ∈ i, ..., j − 1 do
            min_count = opcount[i][k] + opcount[k + 1][j] + P[i − 1] · P[k] · P[j];
            if min_count < opcount[i][j] then
                opcount[i][j] = min_count;
                split[i][j] = k;
            end
        end
    end
end
```

The ***OP-opcount*** algorithm builds a dynamic programming table that keeps the count of total number of operations required to multiply the matrices. First, all the diagonal entries are set to 0 (i.e., $opcount[i][i] = 0, 1 \leqslant i \leqslant n$). The cost of multiplying two consecutive matrices is computed, $opcount[i][i+1], 1 \leqslant i \leqslant n-1$. The cost is $opcount[i][i+1] = opcount[i][i] + opcount[i+1][i+1] + P[i-1] \cdot P[i] \cdot P[i+1]$, where the first two terms are 0. Using this information the minimum

cost of multiplying three consecutive matrices is computed as

$$opcount[i][i+2] = min\{opcount[i][i] + opcount[i+1][i+2] + P[i-1] \cdot P[i] \cdot P[i+2],$$
$$opcount[i][i+1] + opcount[i+2][i+2] + P[i-1] \cdot P[i+1] \cdot P[i+2]\}.$$

This is repeated for all chains of sizes 1 to n. The value $opcount[1][n]$ gives the minimum number of computation required to multiply n matrices. The main observation is that $opcount[i][j]$ depends on $opcount[i][k]$ and $opcount[k+1][j]$ for all $i \leqslant k < j$.

OP-opcount seeks to minimize the total number of computations required for multiplying a sequence of matrices. To the best of our knowledge we are not aware of any algorithm that minimizes the cost of data transfers. In the next section we show how the closed form solution described in Section D.1 can be used to minimize the total number of data transfers required for matrix chain products.

D.5 Minimizing Data Movement for Minimum Op Count Tree

The *Op-opcount* algorithm minimizes the total number of computations (*Op Count*) required for a given matrix chain product. After minimizing the *Op Count*, we want to minimize the total number of off-chip data transfers to reduce energy consumption. Consider the following matrix multiplication:

$$R = A_1 A_2 A_3 \tag{D.15}$$

with P_0, P_1, P_2, P_3 as the problem sizes. This can be parenthesized in two ways: $(A_1A_2)A_3$ and $A_1(A_2A_3)$, because matrix multiplication is associative. Figure D.7 shows the data movement requirements for the case $R = (A_1A_2)A_3$. First the matrices A_1 and A_2 are multiplied, that produces an intermediate matrix T. T is then multiplied with A_3 to produce the output matrix R. Again, from the solutions h^*_R, $h^*_{A_1}$, and $h^*_{A_2}$, it is clear that the total off-chip data movement is minimized in the case where the output of the matrix multiplication is stored on-chip. Therefore,

it is beneficial to store the intermediate matrix T and the output matrix R in the shared memory. The total number of off-chip data transfers for $R = (A_1A_2)A_3$ can be calculated as follows:

$$h_{(A_1A_2)A_3} = \frac{2P_0P_1P_2}{\sqrt{M}} + P_0P_2 + \frac{2P_0P_2P_3}{\sqrt{M}} + P_0P_3 \tag{D.16}$$

Similarly, for $R = A_1(A_2A_3)$ parenthesization the data movement is given by

$$h_{A_1(A_2A_3)} = \frac{2P_1P_2P_3}{\sqrt{M}} + P_1P_3 + \frac{2P_0P_1P_3}{\sqrt{M}} + P_0P_3 \tag{D.17}$$

The total data transfers in D.16 and D.17 can vary based on the problem sizes. We now formulate an optimal parenthesization problem with data movement as the cost metric for matrix product chains. Our formulation is similar to the classic dynamic programming formulation explained in Section D.3.2 with op count as the cost metric.

For a sequence of matrices $A_1, A_2, A_3, \ldots, A_n$ to be multiplied, given their problem sizes $P_0, P_1, P_2, \ldots, P_n$, we first use the ***Op-opcount*** algorithm to find a parenthesization that minimizes the total number of computations. Let this tree be called ***OpTree***. We apply the following objective function to this tree to minimize the total off-chip data transfers.

$$S(i,j) = \begin{cases} 0, & \text{if i = j} \\ S(i,k) + S(k+1,j) + \dfrac{2P[i-1]P[k]P[j]}{\sqrt{M}} + P[i-1] \cdot P[j], & \text{otherwise} \end{cases} \tag{D.18}$$

The optimal split value of k for subsequence $A_{i\ldots j}$ is obtained from $split[i][j]$ table produced by ***Op-opcount*** algorithm. We build an algorithm, ***OP-DM*** (OP refers to Optimal Parenthesization and DM refers to Data Movement), that calculates the total number of off-chip data transfers using the equation D.18.

The ***OP-DM*** algorithm (see algorithm 11) builds a table that keeps the count of minimum number of data transfers required to multiply the matrices. First, all the diagonal entries are set to 0 (i.e., $table[i][i] = 0, 1 \leqslant i \leqslant n$). Then the cost of multiplying two consecutive matrices is

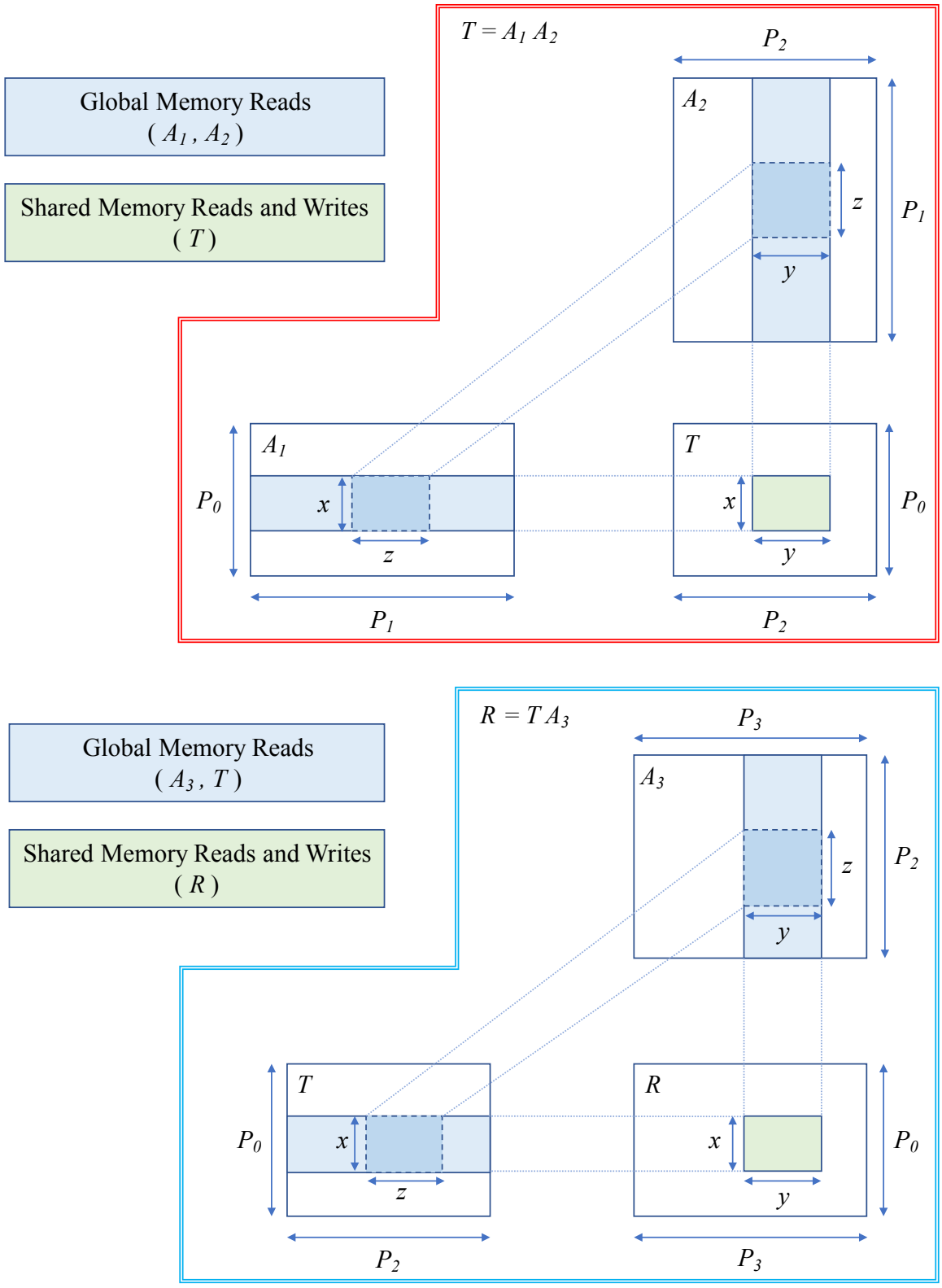


Figure D.7: Data Movement for Matrix Product Chains, $R = (A_1 A_2) A_3$

computed, $table[i][i+1], 1 \leqslant i \leqslant n-1$. The cost is $table[i][i+1] = table[i][i]+table[i+1][i+1]+\frac{2P[i-1]\cdot P[i]\cdot P[i+1]}{\sqrt{M}} + P[i-1]\cdot P[i+1]$. The cost of $table[i][i] + table[i+1][i+1]$ is 0. So the total data movement cost for multiplying any two adjacent matrices(A_i and A_{i+1}) is given by $\frac{2P[i-1]\cdot P[i]\cdot P[i+1]}{\sqrt{M}} + P[i-1]\cdot P[i+1]$. Using this information the cost of multiplying three consecutive matrices is computed using the values from $split[i][j]$ table. This process is repeated for all chains of sizes 1 to n. The value at $table[1][n]$ gives the minimum number of data transfers required to multiply n matrices.

Algorithm 11: OP-DM

Input : $P[0...n]$: matrix sizes
M : size of on-chip memory
$split[i][j]$: the value k that determines optimal split for $A_{i...j}$ using *Op-opcount*
Output: $table[i][j]$: minimum number of off-chip data transfers required to compute the subsequence $A_{i...j}$

```
for i ∈ 1, ..., n do
    table[i][i] = 0;
end
for i ∈ 1, ..., n − 1 do
    for j ∈ 1, ..., n − 1 do
        k = split[i][j];
        table[i][j] = table[i][k] + table[k + 1][j] + 2P[i − 1] · P[k] · P[j] / √M + P[i − 1] · P[j];
    end
end
```

A further reduction in the data movement is possible for matrix product chains. The savings can be obtained if the intermediate result is not stored in the off-chip memory. Next section describes this in detail.

D.6 Hyper-node Parenthesization of Minimum Op Count Tree

The *Op-opcount* algorithm minimizes the total number of computations (*Op Count*) required for a given matrix chain product. After minimizing the *Op Count*, we want to minimize the total number of off-chip data transfers to reduce energy consumption. With the code structures in

Listings 5 and 9, it is possible to consume the intermediate values of $T[i][j]$ as soon as they are produced and bypass writing the intermediate matrix T to the off-chip memory. We apply these *loop fusion* techniques to a given *Minimum Op Count Tree* for matrix chain products and formulate an algorithm to minimize the global cost of data movement. This involves identifying two matrix multiplications (Two MM) that produce fused hyper-nodes in a Minimum Op Count Tree.

Given a *Minimum Op Count Tree* for a sequence of matrix products $A_1, A_2, A_3, \ldots, A_n$ with problem sizes $P_0, P_1, P_2, \ldots, P_n$, the problem is to minimize the off-chip data movement for this given tree. Let this *Minimum Op Count Tree* be called *OpTree*.

At every intermediate node in this *OpTree*, we have three choices: either fuse the node with its left child, or fuse the node with its right child, or do not fuse at all. We call this *Local Decision*. Recall, fusion means we consume the left/right child (intermediate matrix) as soon as it is computed and that the intermediate matrix is never written to the off-chip memory. Note that the fusion technique can be applied only to the intermediate nodes. The leaf nodes are the input matrices and have to be read from the off-chip memory. We develop a set of three local costs (one for each case: no fusion, fuse left, fuse right) for every intermediate node in the *OpTree*.

Let $w(i, j)$ be the cost of writing the node $\langle i, j \rangle$ to the off-chip memory. While making a local decision, we do not want to consider the cost of writing the node $\langle i, j \rangle$ to the off-chip memory. Therefore, $w(i, j)$ can be computed as

$$w(i,j) = \begin{cases} 0, & \text{if i = j} \\ P_{i-1} \cdot P_j, & \text{otherwise} \end{cases} \tag{D.19}$$

When $i = j$, the node is a leaf, that means it is an input matrix which is not written to the off-chip memory.

The data transfer cost for the case where an intermediate node $\langle i, j \rangle$ is not fused with either of its children is

$$h_p(i,j) = w(i,k) + w(k+1,j) + \frac{2P_{i-1} \cdot P_k \cdot P_j}{\sqrt{M}} \tag{D.20}$$

where, the first term is the cost of writing the node $\langle i, k\rangle$ to the off-chip memory. Similarly, the second term is the cost of writing the node $\langle k+1, j\rangle$ to the off-chip memory. And, the third term is the data movement cost of computing $\langle i, j\rangle$ but not writing it out to the off-chip memory. The third term is derived from the closed form solution of equation D.3.

Figure D.8 shows fusion options. The black nodes represent the nodes that are written to (and read from) the off-chip memory. A red edge represents the fusion edge. This means the child stays on chip.

The data movement cost for the case when the node $\langle i, j\rangle$ is fused with its left child $\langle i, k\rangle$ is

$$h_l(i,j) = w(i,k_1) + w(k_1+1,k) + w(k+1,j) + \frac{2P_{i-1}\cdot P_{k_1}\cdot P_k(1+\alpha)\sqrt{\alpha'}}{\sqrt{M}} - 2w(i,j) \tag{D.21}$$

where $\alpha = P_j/P_{k_1}$ and $\alpha' = (1+2\alpha)/(1+\alpha)$. The first three terms are the cost of writing the nodes $\langle i, k_1\rangle$, $\langle k_1+1, k\rangle$, and $\langle k+1, j\rangle$ to the off-chip memory. The fourth and the fifth terms are derived from the closed form solution D.10 from the Section D.2. Note that we subtract $w(i,j)$ twice. We subtract the cost of initial read of the intermediate node $\langle i, j\rangle$ because we assume it is initialized to 0. See Figure D.8(b), the nodes $\langle i, k_1\rangle$, $\langle k_1+1, k\rangle$ and $\langle k, j\rangle$ are read from the off-chip memory.

Similarly, the data movement cost for the case(See Figure D.8(c)) when the node $\langle i, j\rangle$ is fused with the right child $\langle k+1, j\rangle$ is given by

$$h_r(i,j) = w(k+1,k_2) + w(k_2+1,j) + w(i,k) + \frac{2P_k\cdot P_{k_2}\cdot P_j(1+\beta)\sqrt{\beta'}}{\sqrt{M}} - 2w(i,j) \tag{D.22}$$

where $\beta = P_{i-1}/P_k$ and $\beta' = (1+2\beta)/(1+\beta)$. The first three terms are the cost of writing the nodes $\langle k+1, k_2\rangle$, $\langle k_2+1, j\rangle$, and $\langle i, k\rangle$ to the off-chip memory. The fourth and the fifth terms are derived from the closed form solution in equation D.14. Again, we subtract $w(i,j)$ twice because we do not consider the cost of writing the node $\langle i, j\rangle$ to the off-chip memory in the local cost. See Figure D.8(c), the nodes $\langle i, k\rangle$, $\langle k, k_2\rangle$ and $\langle k_2+1, j\rangle$ are read from the off-chip memory.

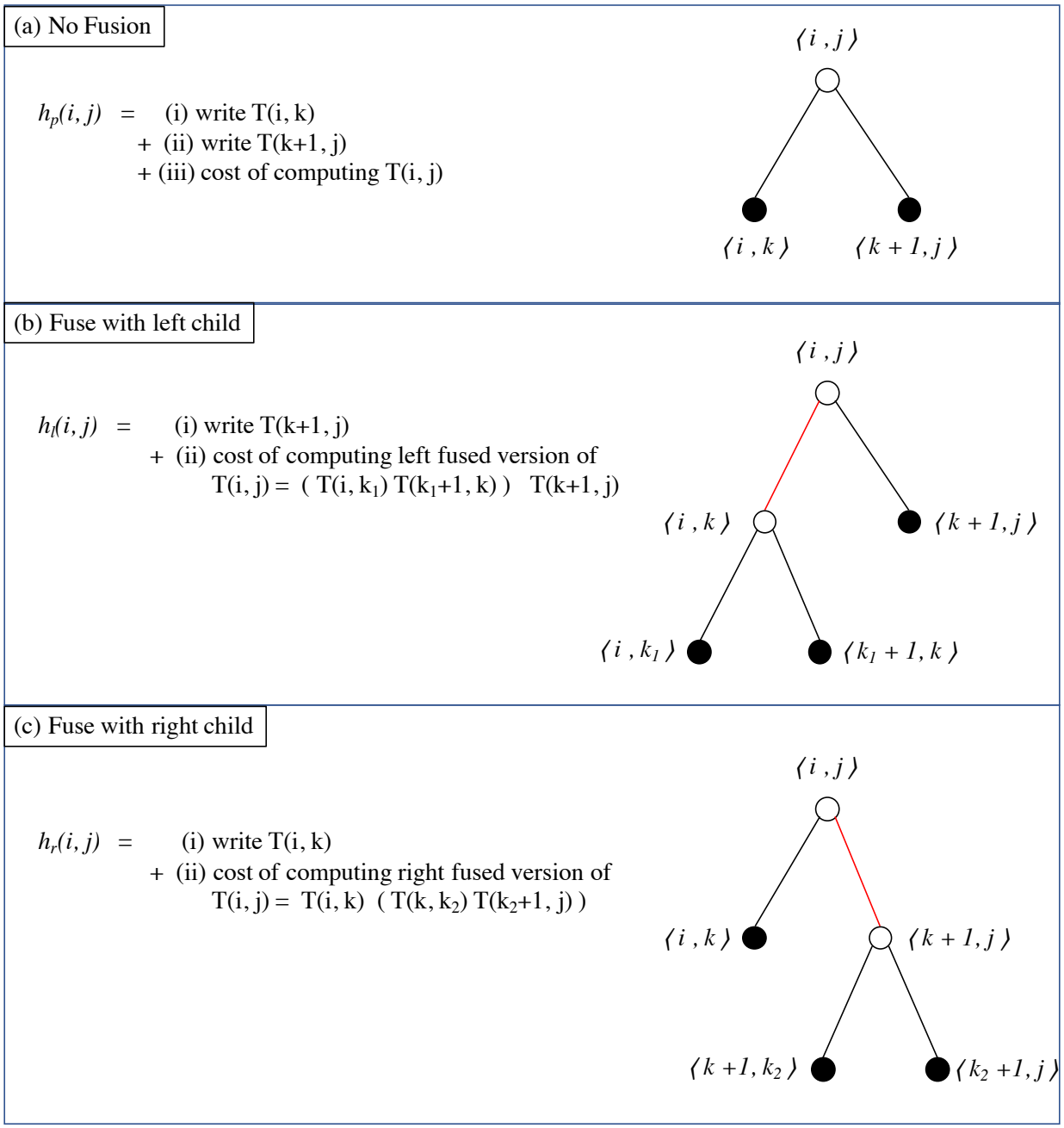


Figure D.8: Local data movement cost of computing the intermediate node $T\langle i, j\rangle$.

Next, we calculate the total data movement cost of computing an intermediate node $T\langle i, j\rangle$. The goal is to minimize the total data movement cost for this sub-tree. Consider a sub-tree (of seven nodes) of a given *OpTree*, as shown in figure D.9. For this sub-tree, we have the following options: (a) do not fuse any edge, (b) fuse one of the $1-6$ edges, or (c) fuse two of the $1-6$ edges. In the case where we want to fuse two edges, all combinations of $1-6$ are not legal. If we

decide to fuse an edge then no adjacent edge can be fused. For e.g., it is not legal to fuse the edges 1 and 5. Fusing both 1 and 5 means that we have three matrix multiplications to be fused, and is not legal as discussed in Section D.2. Therefore, the only legal choices are $\{1,3\}$, $\{1,4\}$, $\{1,6\}$, $\{2,3\}$, $\{2,4\}$, $\{2,6\}$, $\{3,5\}$, and $\{4,5\}$. Moreover, it is not legal to fuse three or more edges in $1-6$.

For an intermediate node $\langle i,j\rangle$, we develop a set of three total costs (i.e., for the entire sub-tree rooted at $\langle i,j\rangle$) as follows. The total off-chip data movement cost, $F_p(i,j)$, when node $T\langle i,j\rangle$ is not fused, the total data movement cost, $F_l(i,j)$, when node $T\langle i,j\rangle$ is fused with its left child, and the total data movement cost, $F_r(i,j)$, when node $T\langle i,j\rangle$ is fused with its right child. And finally, let $F(i,j)$ be the minimum global cost of computing the subsequence $A_{i...j}$, defined as

$$F(i,j) = \begin{cases} 0, & \text{if i = j} \\ F_p(i,j), & \text{if i = k and k+1 = j} \\ min(F_p(i,j), F_r(i,j)), & \text{if i = k} \\ min(F_p(i,j), F_l(i,j)), & \text{if k+1 = j} \\ min(F_p(i,j), F_l(i,j), F_r(i,j)), & \text{otherwise} \end{cases} \tag{D.23}$$

We now set up recurrences for each of these functions. The value of $F_p(i,j)$ is computed as

$$F_p(i,j) = F(i,k) + F(k+1,j) + h_p(i,j) \tag{D.24}$$

Equation D.24 computes the minimum cost of data movement considering all the possible cases when the node $\langle i,j\rangle$ is not fused with any of its two children. It considers all the possibilities where the edges 5 and 6 are not marked as fused.

Consider the case when the edge 5 is marked as fused. In this case, the node $\langle i,k\rangle$ is not available for fusion with its children and must be free. However, the node $\langle k+1,j\rangle$ is available for fusion. The minimum total data movement cost for when edge 5 is fused can be computed as

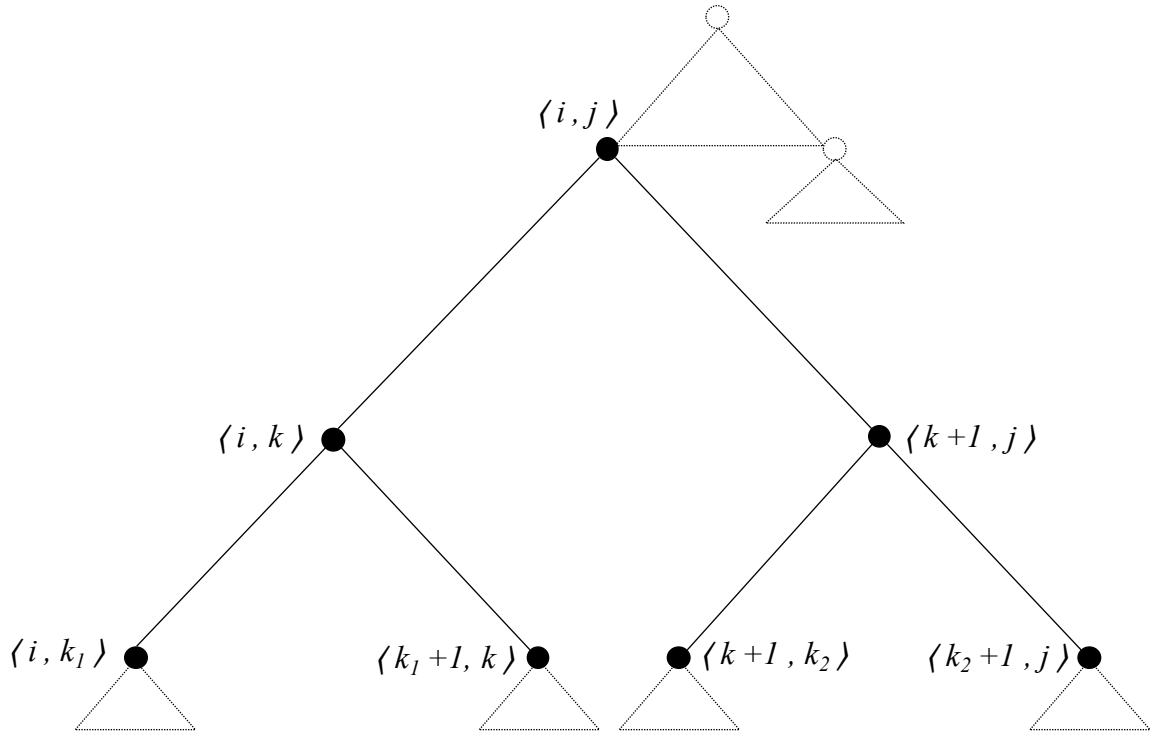


Figure D.9: Global Data Movement Cost.

$$F_l(i,j) = F(i,k_1) + F(k_1+1,k) + F(k+1,j) + h_l(i,j) \tag{D.25}$$

Similarly, the minimum total cost of data movement for when edge 6 is marked as fused can be computed as

$$F_r(i,j) = F(i,k) + F(k+1,k_2) + F(k_2+1,j) + h_r(i,j) \tag{D.26}$$

The root of the *OpTree* is at $i = j = n$ and is the final output that is written to the off-chip memory. Therefore, the global data movement cost will be $F(1,n) += P_0 P_n$.

D.7 Reduction in Data Movement

Consider the matrix sequence $\{936, 1008, 552, 368, 1016, 616, 544\}$ with $n = 6$ matrices and shared memory capacity of $M = 65536$. ***OP-opcount*** algorithm generates the parenthesization $((A[1](A[2]A[3]))((A[4]A[5])A[6]))$ with minimum op-count of 1092977664. The tree is shown in

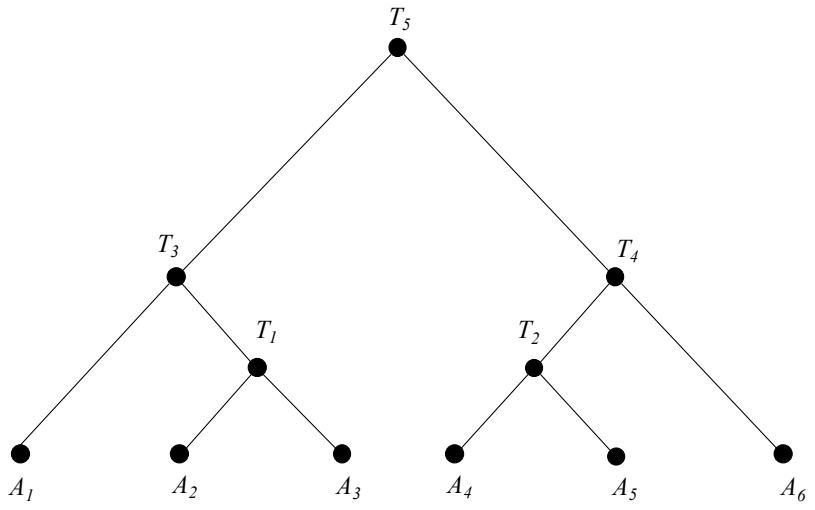


Figure D.10: Minimum Op Count Tree for the example.

	h_p	h_l	h_r	F_p	F_l	F_r	F	*Fusion*	*tile size x*	*tile size y*
T_1	16	–	–	16	–	–	16	–	256	256
T_2	18	–	–	18	–	–	18	–	256	256
T_3	31	–	31	47	–	31	31	fuse-right ($T_1 - T_3$)	312	210
T_4	12	28	–	30	28	–	28	fuse-left ($T_2 - T_4$)	220	296
T_5	20	44	40	79	88	89	79	no fuse	256	256

Figure D.11: Calculating data movement cost of the intermediate nodes of the minimum Op-Count Tree in figure D.10. All values(except tile sizes) are multiples of 10^4.

figure D.10. For this Op-Count Tree, the total data movement cost of single matrix multiplication recurrence S in equation D.18 is $10,190,344$. For this Op-Count Tree, let us apply the fused two matrix multiplication recurrence $F(i,j)$. The values of h_p, h_l, h_r, F_p, F_l, F_r, and F are shown in the table D.11. All the values are multiples of 10^4. The value F_p cost of the node T_5(highlighted in red) is given by the F cost of T_3, F cost of T_4, and h_p cost of T_5, i.e., $F_p\langle T_5\rangle = 31+28+20 = 79$. The value F_l cost of the node T_5(highlighted in green) is given by the F cost of T_1, F cost of T_4, and h_l cost of T_5, i.e., $F_l\langle T_5\rangle = 16+28+44 = 88$. Similarly, the F_r of T_5 is calculated as $18+31+40 = 89$. The total data movement cost of $F(1,6)$ is $7,882,874+509,184 = 8,392,058$.

The reduction in total off-chip data transfers using $F(1,6)$ compared to $S(1,6)$(recurrence D.18) is ***17.6%***.

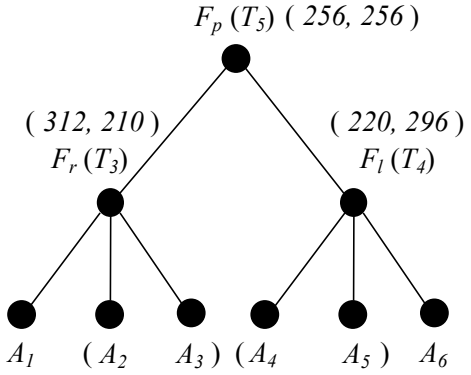


Figure D.12: Minimum Op Count Tree with hyper-nodes (fused nodes) for the example.

The minimum op-count tree with fused nodes is shown in Figure D.12. Note, there are only 3 intermediate nodes in this hyper-node tree. The nodes T_3 and T_4 are the hyper-nodes with *right-fuse* and *left-fuse* Two MM operations. In general, every node in the hyper-node tree can have two or three children. Any node with three children cannot have a child with more than two children. The tile sizes used for computing the intermediate nodes(including the root) are as follows. For the nodes with the minimum cost $F(i,j) = F_p(i,j)$, the tile sizes are $x = y = \sqrt{M} = 256$. For the hyper-nodes that have minimum cost $F(i,j) = F_r(i,j)$ or $F(i,j) = F_l(i,j)$, the tile sizes are based on the values of α' and β'. For our example, the tile sizes for nodes T_3 and T_4 are $\{x = 312, y = 210\}$ and $\{x = 220, y = 296\}$ respectively.

For lengths $n = \{2, 3, 4, \ldots, 20\}$, we generate $1,000$ instances for each length of matrix chains. We apply the recurrences $S(1,n)$(equation D.18) and $F(1,n)$(equation D.23) to produce the total cost of data transfers. For on-chip memory capacities of $M = 64KB$ and $M = 96KB$, the average % improvement in the data transfer cost of F over S is 9.44% and 13.27% respectively. The average, minimum, and maximum $\%$ improvement in the data transfer cost across chains of different lengths are listed in Table D.1. Observe that the $\%$ reduction in the data movement cost

Table D.1: Reduction(%) in total off-chip data transfers for matrix sequences.

	M = 64KB			M = 96KB		
n	**Average**	**Min**	**Max**	**Average**	**Min**	**Max**
2	0.00	0.00	0.00	0.00	0.00	0.00
3	6.58	0.00	46.93	10.30	0.00	50.11
4	7.67	0.00	47.36	10.76	0.00	41.48
5	8.31	0.00	33.65	12.16	0.00	36.00
6	8.81	0.00	37.63	12.68	1.32	38.20
7	8.97	0.00	33.47	12.68	2.36	32.71
8	9.57	0.35	31.22	13.21	3.20	31.64
9	9.49	0.49	28.76	13.45	1.75	32.30
10	9.55	0.63	27.37	13.59	3.10	31.01
11	9.86	1.26	25.78	13.73	3.65	30.75
12	9.90	1.03	25.43	13.95	4.04	30.39
13	9.87	1.39	24.20	13.78	4.80	31.61
14	9.80	1.74	27.68	13.88	4.30	30.83
15	10.14	1.48	26.45	14.09	4.88	29.95
16	10.26	1.65	24.00	14.23	4.77	28.54
17	10.15	1.11	24.45	14.17	4.14	32.21
18	10.29	1.72	25.38	13.81	4.93	27.10
19	10.20	2.30	24.21	14.32	5.86	30.94
20	10.53	2.43	23.97	14.14	5.65	28.26
	9.44	0.00	47.36	**13.27**	0.00	50.11

Table D.2: Tile sizes and closed form solutions for Single MM.

On-chip	tile size 1	tile size 2	data transfers
Output	$x = \sqrt{M}$	$y = \sqrt{M}$	$h^*_R = \frac{2P_0P_1P_2}{\sqrt{M}} + P_0P_2$
Input 1	$x = \sqrt{M/2}$	$z = \sqrt{2M}$	$h^*_{A_1} = \frac{2\sqrt{2}P_0P_1P_2}{\sqrt{M}} + P_0P_1 - P_0P_2$
Input 2	$x = \sqrt{M/2}$	$y = \sqrt{2M}$	$h^*_{A_2} = \frac{2\sqrt{2}P_0P_1P_2}{\sqrt{M}} + P_1P_2 - P_0P_2$

Table D.3: Tile sizes and closed form solutions for Two MM.

	tile size x	tile size y	data transfers
left fuse	$\sqrt{M/\alpha'}$	$\sqrt{M\alpha'}$	$h^*_{lc} = \frac{2P_0P_1P_2(1+\alpha)\sqrt{\alpha'}}{\sqrt{M}} - P_0P_3$
right fuse	$\sqrt{M\beta'}$	$\sqrt{M/\beta'}$	$h^*_{rc} = \frac{2P_1P_2P_3(1+\beta)\sqrt{\beta'}}{\sqrt{M}} - P_0P_3$

decreases as the chain length increases. Also, the higher the on-chip memory capacity the more the savings in data transfers(expected).

D.8 Conclusion

We present techniques that minimize off-chip data transfers for single matrix multiplication and fused two matrix multiplications. These techniques consist of closed form solutions that minimize off-chip data transfers. Problem sizes and on-chip memory capacity are the inputs. The outputs are respective tile sizes and total off-chip data transfers. Table D.2 and D.3 show these tile sizes and closed form solutions for Single MM and Two MM.

We also present two ways to minimize the global data movement for a matrix chain products. Using the closed form solutions for Single MM and Two MM, the recurrences $S(i, j)$ (equation D.18) and $F(i, j)$ (equation D.23) minimize the off-chip data transfers for a given minimum op-count tree. Using these recurrences, we show that off-chip data transfers can be reduced by 9% (and 13%) on average for architectures with 64KB (and 96KB) of on-chip memory capacity.

Acknowledgments: We would like to thank NVIDIA for collaborating on this project. Nirmal Prajapati interned at NVIDIA in 2018 and started working on the topic of minimizing data movement cost for matrix chain products. At NVIDIA, she formulated mathematical optimization problem to minimize off-chip data transfers for matrix chain products with square matrices. The work presented in this chapter for rectangular matrices was developed at CSU.