LA-UR-18-21279
Approved for public release; distribution is unlimited.

Title: Analytical Cost Metrics : Days of Future Past

Author(s): Prajapati, Nirmal
Rajopadhye, Sanjay
Djidjev, Hristo Nikolov

Intended for: arXiv preprint

Issued: 2018-02-20

ANALYTICAL COST METRICS : DAYS OF FUTURE PAST

February 2018

Nirmal Prajapati and Sanjay Rajopadhye
Department of Computer Science
Colorado State University, Fort Collins, Colorado

Hristo Djidjev
Los Alamos National Laboratory
LANL, Los Alamos, New Mexico

Chapter 1

Introduction

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 [1–5] 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 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 [1, 6]. *System codesign* [7, 8] 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* [9–11], 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 [1, 2, 4, 7] 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/Power/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 [12] 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 recently 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 Tensor-Flow 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* [1]. 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 [13–17], it has matured into a powerful technology, now incorporated into `gcc`, `llvm` and in commercial compilers [Rstream, IBM]. Tools targeting GPUs are also available [18, 19].

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

Previously, we presented [20] an approach to solve the above problem as follows:

1. **Develop Models**

   (a) **Time Model [21]** 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 [20]** 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 [22]** 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 [20]** We combine the proposed execution time model [21] and the area model [20] 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 [23–25]. 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 [5]. Consequently, highly specialized coprocessors will become much more common in exascale than in the current HPC systems. Such 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. 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 [1].

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

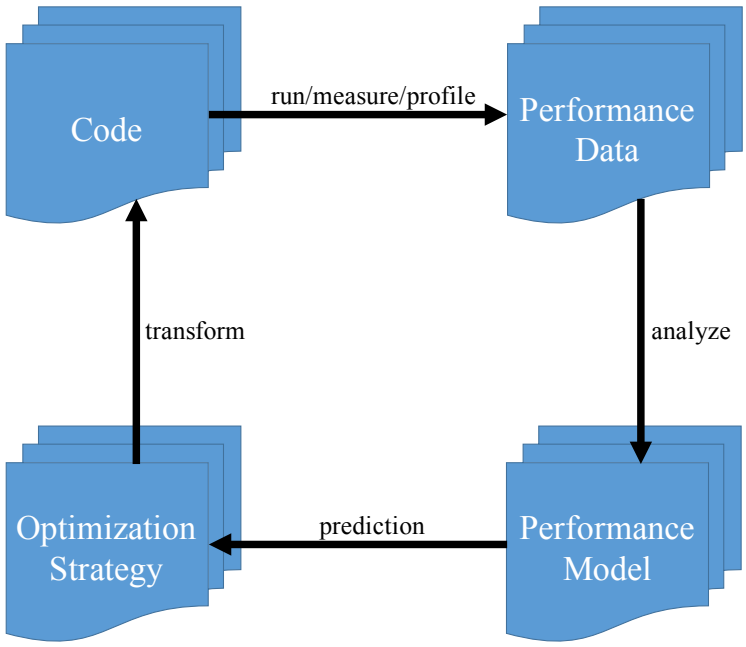


Figure 1.1: HPC application performance improvement cycle. The layers represent different architectures.

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.

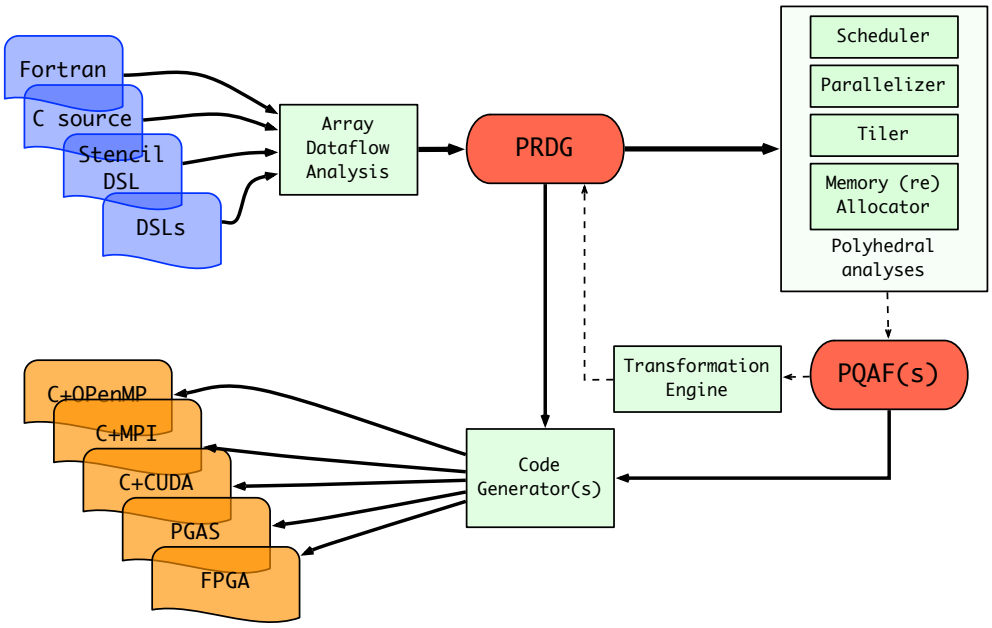


Figure 1.2: Polyhedral Compilation

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 performance models for polyhedral analysis, in turn, breaking the cycle and reducing the time to find optimal solutions.

Novelty

1. The main novelty of our work comes out as a consequence of some of the exascale challenges [1]. For exascale system design, various architectures, programs and transformation strategies are to be explored simultaneously in order to find the optimal. We add performance 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).

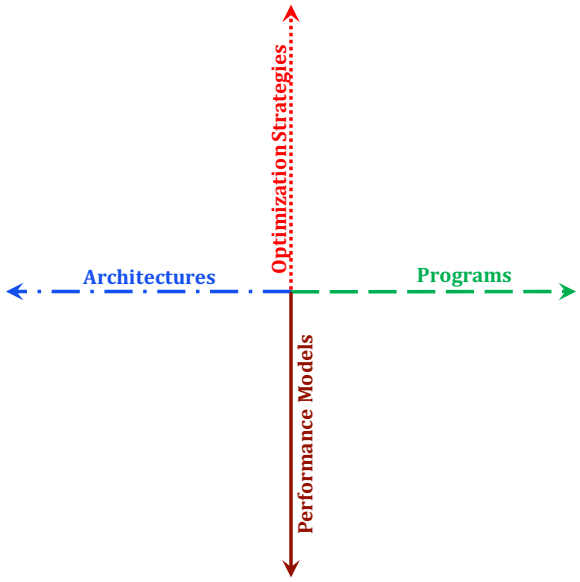


Figure 1.3: Unified View of the Design Space.

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. Chapters 3, 4, and 5 discuss the work that has been accomplished. Bottleneck Analysis is explained in details in Chapter 6. Chapter 7 discusses the relevant literature. Finally, Chapter 8 concludes the work.

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 [26], 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 [27–39]. 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 [13–17, 48–50], and many tools are widely available, e.g., PPCG, developed by the group at ENS, Paris [51]. 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. [18] 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 [50], 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

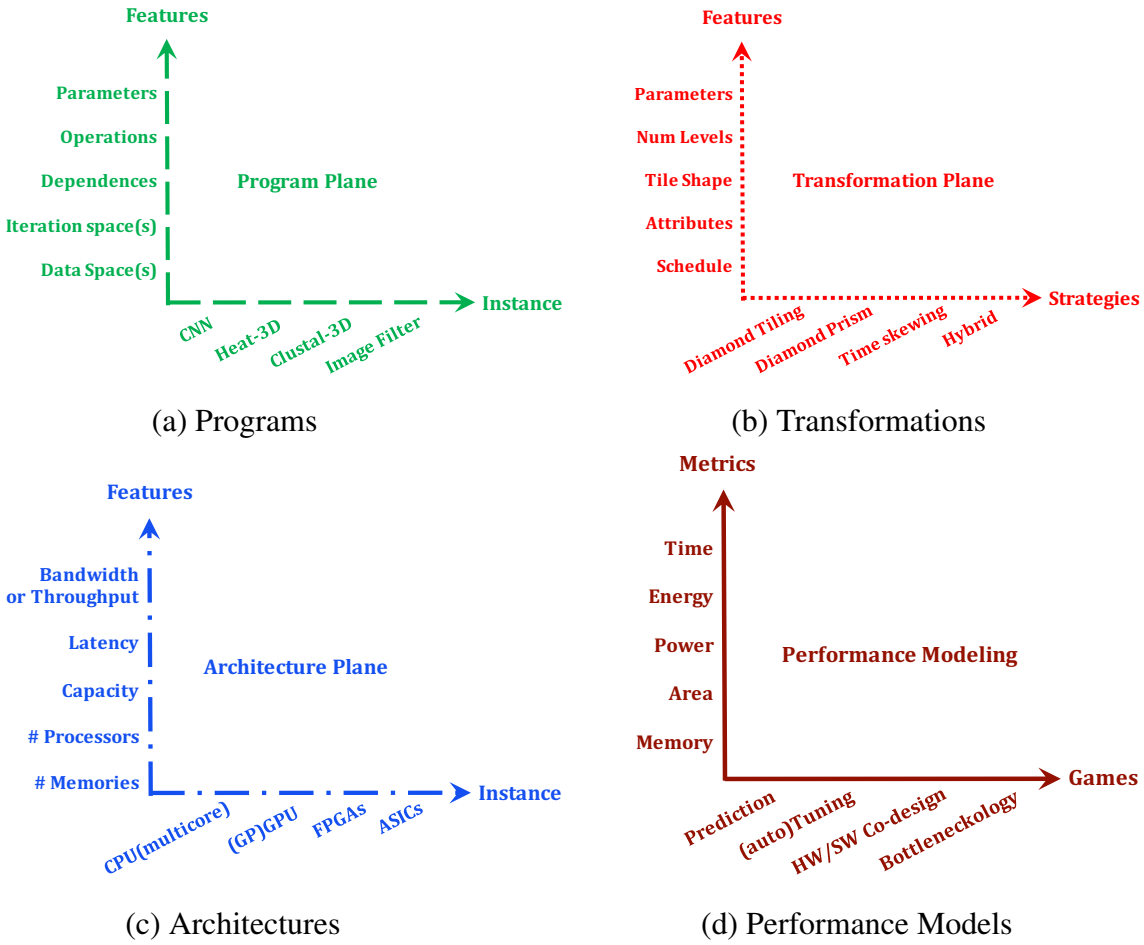
Consider how polyhedral compilation has recently evolved. Bondhugula et al. [52] proposed an extension of PLuTO for periodic stencil computations, and Bandishti et al. [53] developed another extension to allow concurrent starts. Since the objective functions in these strategies are all surrogates for 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 [54], 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. [18] 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) Performance 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 performance models (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 com-

putational 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 [55, 56], diamond tiling [53], diamond prisms [36, 57], or hybrid hexagonal-classic (HHC) tiling [18,58,59]. 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.

We combine this six-dimensional design space with the ***performance modeling plane*** (Fig. 2.1(d)) which gives us a high level picture of various performance metrics and the games that we can play with these performance models. ***This unified view of the landscape is the main novelty of our***

[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.

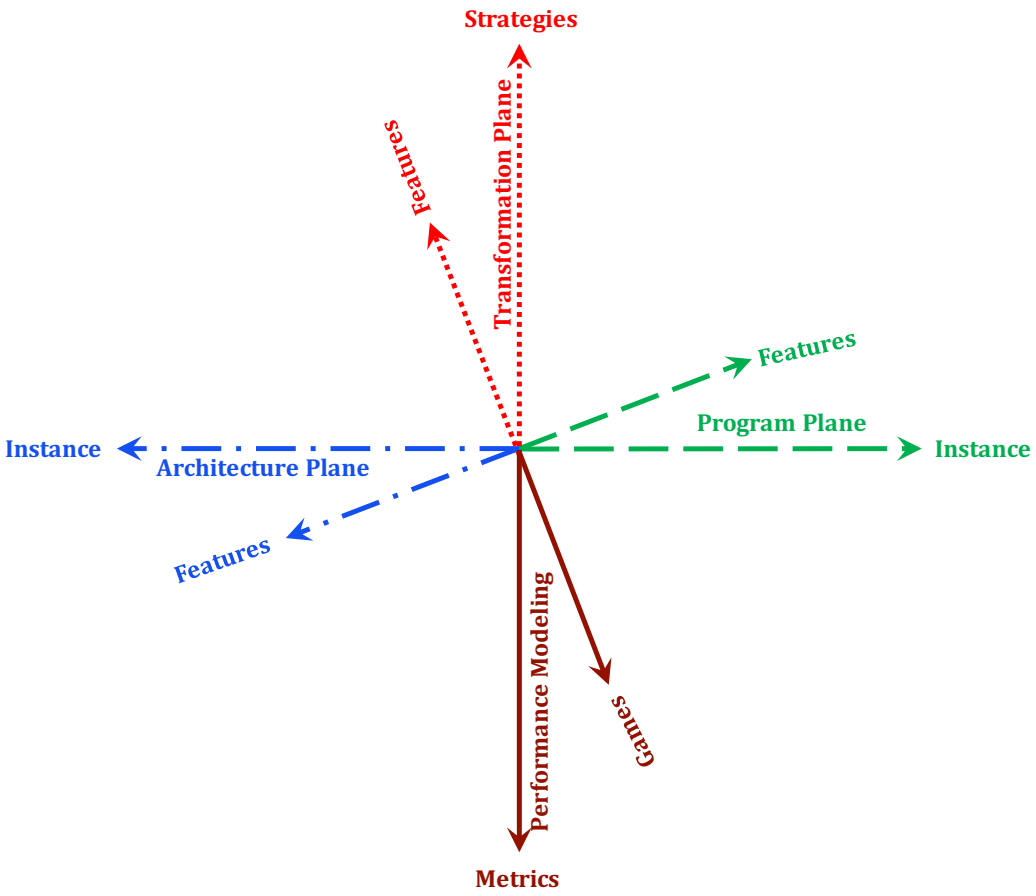


Figure 2.2: The main novelty of our work! The Domain-Specific Design Landscape that we must navigate is thus the cross-product of the four planes in Fig. 2.1.

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

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

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 ac-

curacy 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.

Area Model and Chip Area Prediction of GPUs

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].

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 *ob-*

[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.

jective 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)*. We will discuss two such common objective functions, $\mathcal{M}_T$ for execution time and $\mathcal{M}_E$ for energy (expressed as GOPs/sec or GOPs/joule) in details in the following chapters.

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) Minimize $\min(M_{T_1}(\vec{P}, \vec{S_1}, \vec{A}), M_{T_2}(\vec{P}, \vec{S_2}, \vec{A}))$, 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, $\mathcal{S} = \{S_i\}$.

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 [16, 17, 50] 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***. We study this in three ways: (i) investigate codesign-tradeoffs, (ii) perform overhead analysis, and (iii) explore the effect of hyperthreading. More details are provided in Chapter 6.

In the next chapters (3, 4, 5, and 6), we discussed our work in more details.

Chapter 3

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.

3.1 Execution Time Model for GPGPU Stencils

We develop an execution time models for GPGPU stencils 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.

3.2 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 [60]. In addition, the cost of building a power provisioning facility ranges at $10-22 per deployed IT watt [61] and every $10\,^{\circ}\mathrm{C}$ temperature-increase results in a doubling of the system failure rate, thereby reducing the reliability of HPC systems [62]. 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 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 [63, 64] 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 [18].

- 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 [65], which are simulation based.

3.3 Memory Access Model for GPGPU Stencils

We develop Memory Access models [21, 22] 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 appear 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 [20]. 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.

3.4 Silicon Area Model for GPUs

We 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.[5]

In the next chapter, we will show the use of these cost models for tile size selection.

[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.

Chapter 4

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" [66]. 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 [67] 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.

4.1 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. 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. [21] 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.

4.2 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. [22] describes our energy models, the optimization methods, the results and experimental validation in details.

Chapter 5

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 [6]. 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 [68–70], 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 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 [21] 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 [21] 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 6

Bottleneckology

We 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.

6.1 Codesign Trade-offs

Workload Sensitivity Table 6.1 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 6.1. 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

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

Table 6.1: Workload sensitivity. The optimal architecture configuration for a single benchmark varies significantly.

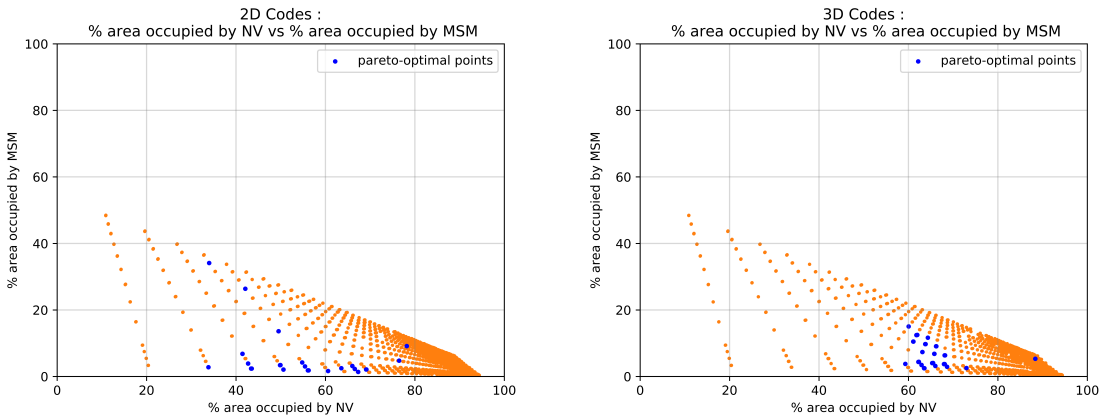


Figure 6.1: Resource Allocation.

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.

Resource Allocation Another interesting perspective is seen in Figure 6.1 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 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.

6.2 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 [21] guides such a choice. This execution time model is a simple set of analytical functions that predicts the execution time of the HHC [18] 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 [18]. 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. [21] 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. [21] for a given 2D stencil is formulated as follows:

$$minimize \quad T_{alg}(t_{S_1}, t_{S_2}, t_T) \tag{6.1}$$

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:

$$T_{\text{alg}} = N_w T_{\text{sync}} + N_w T_{\text{prism}} \left\lceil \frac{1}{n_{SM}} \left\lceil \frac{w}{k} \right\rceil \right\rceil . \tag{6.2}$$

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. [21].

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:

$$T_{\text{ideal}} = \frac{S_1 S_2 T}{n_{SM} n_V} C_{iter} \tag{6.3}$$

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

$$T_{\text{overhead}} = T_{\text{alg}} - T_{\text{ideal}} \tag{6.4}$$

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

$$T_{\text{overhead}} = C_{iter} S_1 T \frac{1}{t_{S_1} + \frac{t_T}{2}} \tag{6.5}$$

Instead of minimizing the execution time(as in the equation 6.1), we can now minimize the overhead. To minimize equation 6.5, 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 6.5.

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.

6.3 The effect of the hyper-threading

Our results in [21] 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.

Chapter 7

Related Work

Our research draws upon prior work in the following distinct areas.

7.1 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 [53, 71, 72]. One successful technique is called *time tiling* [53, 57, 73–78], an advanced form of loop tiling [73, 79, 80]. 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 [38] 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 [81], Hybrid-hexagonal tiling [18], and Overtile [82] 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 [83], 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.

7.2 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 [84] and Stella [85] 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 [86] 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 [87] 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 [88] that allows to automatically derive "optimal" code generation parameters. Yotov et. al [89] 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 [90], but at this point no stencil computations have been considered. Shirako et al. [91] 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 [92] 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 [88] that allows to automatically derive "optimal" code generation parameters. Patus [93] 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. [94] 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.

7.3 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. [95] 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* [96], 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* [97]. Courbon et. al. [98] 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.

7.4 Energy Modeling

GPU power/energy model is a very active area: a recent survey article on the topic [99] cites almost 150 references. We only discuss the relevant work here. The model we present complements Mittal and Vetter [99] by enabling us to find the optimal parameters (i.e., tile sizes) for the energy efficient execution of stencil like programs. Hong and Kim [100] 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 [101] 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 [63] model GPU power of kernels using performance counters. Lim et al. [102], GPUWattch [64] and GPUSimPow [103] are simulation based power models. McPAT [104] is the basis for Lim et al [102] and GPUWattch [64] uses GPGPUSim [105] 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 [106, 107] 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 [106, 107] 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.

7.5 Codesign

Application codesign is a well established discipline and has seen active research for well over two decades [108–112]. 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 [113, 114], integer linear programming [115].

There is some recent work on accurately modeling the design space, especially for regular, or *affine control* programs [23–25]. 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. [116] 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 [117] 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 [6].

Kuck et al. [118, 119] 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 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 3) for execution time, energy, memory access, and silicon area of a chip. Our models are reasonably accurate and help predict the associated cost. 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 ***performance tuning*** (Chapter 4) and ***accelerator codesign*** (Chapter 5). For GPGPU stencil computations and polyhedral code generator, we illustrate a proof of concept [20] and present a novel optimization approach to accelerator codesign.

8.1 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.

8.2 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] 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.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

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

[9] 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.

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

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

[12] 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.

[13] 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.

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

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

[16] 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.

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

[18] 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.

[19] 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.

[20] Nirmal Prajapati, Sanjay Rajopadhye, Hristo Djidjev, Nandkishore Santhi, Tobias Grosser, and Rumen Andonov. Accelerator Codesign as Non-Linear Optimization. *ArXiv e-prints*, December 2017. https://arxiv.org/abs/1712.04892.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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`.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

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

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

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

[41] 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.

[42] 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.

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

[44] 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.

[45] 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.

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

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

[48] 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. Swart-

zlander, editors, *International Conference on Application Specific Array Processing*, pages 100–110, Princeton, New Jersey, Sept 1990. IEEE Computer Society.

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

[50] 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.

[51] 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.

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

[53] 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.

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

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

[56] 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.

[57] 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.

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

[59] 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.

[60] 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.

[61] 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.

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

[63] 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.

[64] 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.

[65] 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.

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

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

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

[69] 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.

[70] 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.

[71] 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.

[72] 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.

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

[74] 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.

[75] 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.

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

[77] 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.

[78] 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.

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

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

[81] 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.

[82] 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.

[83] 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.

[84] 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.

[85] 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.

[86] 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.

[87] 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.

[88] 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.

[89] 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.

[90] 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.

[91] 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.

[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. Symposium on Computer Architecture*, ISCA '09, pages 152–163, New York, NY, USA, 2009. ACM.

[93] 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.

[94] 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.

[95] 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.

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

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

[98] 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.

[99] 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.

[100] 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.

[101] 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.

[102] 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.

[103] 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.

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

[105] 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.

[106] 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.

[107] 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.

[108] 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.

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

[110] 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.

[111] 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.

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

[113] 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.

[114] 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.

[115] 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.

[116] 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.

[117] 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.

[118] 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.

[119] 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.