DISS. ETH NO. 26410

DATA MOVEMENT OPTIMIZATION FOR HIGH-PERFORMANCE COMPUTING

A dissertation submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by

TOBIAS GYSI
Dipl. Informatik-Ing. ETH
ETH Zurich

born on the 7th of July 1980
citizen of Buchs AG, Switzerland

accepted on the recommendation of

Prof. Dr. Torsten Hoefler, examiner
Prof. Dr. Thomas C. Schulthess, co-examiner
Dr. Albert Cohen, co-examiner

2019

ABSTRACT

Tuning codes to make efficient use of high-performance computing systems is known to be hard. Programmers have to schedule their computations to thousands of compute cores having the compute and data movement costs in mind. The necessary code transformations – for example, to overlap computation and inter-node communication – are well known. But the complex interplay of hardware and software often prevents programmers from identifying performance bottlenecks and selecting good code transformations. This dissertation introduces compilation frameworks, performance tools, and programming models to tackle these programmability challenges.

Over the last decades, the compute performance improved at a much faster pace than memory performance. Data-movement optimizations to reduce the communication and memory access costs thus became much more pressing. We address the problem by automating the selection of data-locality transformations (ABSINTHE) and by adapting the programming model (DCUDA) to overlap computation and inter-node communication automatically. The performance models needed to automate the tuning (ABSINTHE and HAYSTACK) also provide the programmers with valuable insights when optimizing codes manually.

An important algorithmic motif in high-performance computing is the sequential execution of multiple but different stencils. Our compilation framework (ABSINTHE) automates the selection of data-locality transformations for such stencil programs. It has three main components: 1) a transformation algebra, 2) a performance model, and 3) an optimizer. The transformation algebra (MODESTO) defines the space of possible code transformations and the learned performance model (ABSINTHE) guides the selection of good code transformations.

In summary, this dissertation contributes compilation frameworks, performance tools, and programming models to foster the application of data movement optimizations in high-performance computing. In particular, we automate the selection of data-locality transformations for stencil programs. We believe our work lays the foundation for future compilation frameworks that support even broader application domains.

ZUSAMMENFASSUNG

Es ist bekannt, dass die Optimierung von Codes zur effizienten Nutzung von Hochleistungsrechnern schwierig ist. Programmierer müssen, unter Berücksichtigung von Rechenaufwand und Datenübertragungskosten, ihre Berechnungen auf Tausende von Rechenkernen verteilen. Die notwendigen Code-Transformationen – zum Beispiel zur Überlappung der Kommunikation zwischen den Rechenknoten mit den Berechnungen – sind allgemein bekannt. Das komplexe Zusammenspiel von Hardware und Software hindert Programmierer jedoch häufig daran, Performance-Probleme zu identifizieren und gute Code-Transformationen auszuwählen. In dieser Dissertation werden Kompilierungs-Frameworks, Performance-Tools und Programmiermodelle vorgestellt, um die Programmierbarkeit von Hochleistungsrechnern zu verbessern.

In den letzten Jahrzehnten hat sich die Rechenleistung deutlich schneller verbessert als die Speicherleistung. Optimierungen zur Reduzierung der Kommunikations- und Speicherzugriffskosten wurden daher dringlicher. Wir gehen das Problem an, indem wir die Auswahl von Transformationen zur Verbesserung der Datenlokalität automatisieren (ABSINTHE) und das Programmiermodell anpassen (DCUDA), um die Kommunikation zwischen den Rechenknoten automatisch mit den Berechnungen zu überlappen. Die zur Automatisierung erforderlichen Performance-Modelle (ABSINTHE und HAYSTACK) bieten den Programmierern zudem wertvolle Einblicke bei der manuellen Code Optimierung.

Ein wichtiges algorithmisches Motiv im Hochleistungsrechnen ist die sequentielle Ausführung mehrerer, aber unterschiedlicher Stencil Berechnungen. Unser Kompilierungs-Framework (ABSINTHE) automatisiert die Auswahl von Code-Transformationen zur Verbesserung der Datenlokalität von Stencil Programmen. Es besteht aus drei Hauptkomponenten: 1) einer Transformations-Algebra, 2) einem Performance-Modell und 3) einem Optimierer. Die Transformations-Algebra (MODESTO) definiert den Raum möglicher Code-Transformationen und das erlernte Performance-Modell (ABSINTHE) steuert die Auswahl guter Code-Transformationen.

Zusammenfassend leistet diese Dissertation einen Beitrag zur Entwicklung von Kompilierungs-Frameworks, Performance-Tools und Programmiermodellen, um die Anwendung von Datenlokalitätsoptimierungen im Hochleistungsrechnen zu fördern. Insbesondere automatisieren wir die

Optimierung von Stencil Programmen. Wir glauben, dass unsere Arbeit den Grundstein für zukünftige Kompilierungs-Frameworks legt, die noch breitere Anwendungsbereiche unterstützen.

ACKNOWLEDGEMENTS

I want to thank Torsten Hoefler for supervising my Ph.D. studies here at ETH Zurich. His scientific guidance was essential to address the right research questions and helped to widen the scope of my work. I am also grateful to Torsten Hoefler and Thomas Schulthess for providing me with the opportunity to return to academia after spending multiple years in the industry. I furthermore want to thank my co-examiners, Albert Cohen and Thomas Schulthess, for their effort and their valuable feedback. Special thanks go to Tobias Grosser who was co-supervising my Ph.D. studies and contributed key ideas to my research.

I value the contributions of all my co-authors, collaborators, and students. I especially enjoyed working with Tobias Grosser, Jeremia Bär, Laurin Brandner, Grzegorz Kwasniewski, Aditya Konduri, Siddharth Bhat, and Alain Denzler on published and yet to be published works. The contributions of my co-authors were essential for the success of my projects.

I also want to thank the entire group for the fun birthday parties at the lake and many other memorable moments. Last but not least, I want to thank my family for their support and my dance friends for many good moments and great dances that were a welcome change to my research work.

PUBLICATIONS

Publications that form the basis of this thesis:

- Tobias Gysi, Tobias Grosser, and Torsten Hoefler.
  "MODESTO: Data-centric Analytic Optimization of Complex Stencil Programs on Heterogeneous Architectures."
  ICS 2015. [1].

- Tobias Gysi, Tobias Grosser, and Torsten Hoefler.
  "Absinthe: Learning an Analytical Performance Model to Fuse and Tile Stencil Codes in One Shot."
  PACT 2019. [2].

- Tobias Gysi, Tobias Grosser, Laurin Brandner, and Torsten Hoefler.
  "A Fast Analytical Model of Fully Associative Caches."
  PLDI 2019. [3].

- Tobias Gysi, Jeremia Bär, and Torsten Hoefler.
  "dCUDA: Hardware Supported Overlap of Computation and Communication."
  SC 2016. [4].

Additional publications not part of this thesis:

- 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."
  SC 2015. [5].

- Oliver Fuhrer, Carlos Osuna, Xavier Lapillonne, Tobias Gysi, Ben Cumming, Mauro Bianco, Andrea Arteaga, and Thomas C. Schulthess.
  "Towards a Performance Portable, Architecture Agnostic Implementation Strategy for Weather and Climate Models."
  Supercomputing Frontiers and Innovations. April 2014. [6].

CONTENTS

1 INTRODUCTION

Over time the cost of data movement steadily gained importance and started to dominate the overall computational cost – both in terms of execution time and energy consumption. Analyzing and reducing the cost of data movement became an important concern in high-performance computing [7]. But applying data movement optimizations [8–10] increases the code complexity and often requires non-trivial parameterization. Domain-specific tools [5, 11–13] or compilers [14, 15] may hide the increased code complexity, but choosing optimal parameters remains hard. Examples for the necessary parameterization are tile size and fusion choices or the ratio of the inner to the outer domain when overlapping computation and inter-node communication.

The hardware landscape in high-performance at the same time got much more diverse. Heterogeneous systems equipped with accelerators more and more supersede the previously dominating multi-core machines. The resulting diversity complicates the development of single source code that performs well on today's and tomorrow's systems. Especially since data-locality transformations are very much system-specific. As a result, the model of having expert programmers that in a heroic effort tune the codes for every new hardware generation does not scale anymore.

In this thesis, we use the COSMO atmospheric model [16] as our primary motivational example. The code is used for operational weather forecasting [17] and climate modeling [18]. COSMO consists of more than 300,000 lines of code that mostly implement stencil computations (finite-differences). Its low arithmetic intensity in combination with the sheer size makes the real-world application a good testbed for data-locality transformations.

The current version of COSMO already targets both CPU and GPU systems [6] using a single source code. To this end, the dynamical core of COSMO was rewritten in the domain-specific language STELLA [5] that implements target specific data-locality transformations for both CPU and GPU. STELLA automates the code generation but still requires manual interventions to select the tile sizes and to fuse stencils. The goal of this thesis is to enable the development of domain-specific languages that provide true performance portability by automatically selecting good data-locality transformations.

Auto-tuning is the swiss-army knife for the selection and parameterization of good code transformations. It relies on the empiric evaluation of different implementation variants to select good code transformations. Existing auto-tuning frameworks [19–22] implement ready-to-use search strategies that enable the efficient search space exploration. Yet, compiling and running different implementation variants requires target system access during compilation and can become prohibitively expensive depending on the size of the search space.

An alternative are optimization frameworks [13, 23–25] relying on heuristics and analytical models. In this thesis, we focus on the development of analytical performance models and optimization strategies for selecting data-locality transformations. We show that analytical performance models enable the fast search space exploration while being accurate enough to guide the optimization. Model-based optimization is a promising approach that in many cases provides a good set of initial transformations that if required may be further refined using auto-tuning.

1.1 AUTOMATIC DATA-LOCALITY OPTIMIZATION

All programming models and domain-specific languages that aim at performance portability have to apply target system-specific code transformations. Loop fusion and tiling [8, 26, 27] are examples of important data-locality transformations in high-performance computing. The fusion space alone is exponential in the number of loop nests, and for every fused loop nest, a range of possible tile sizes exists. The large search space and the interdependence of the parameter choices – fusion and tile size need to be considered in tandem – make the optimization challenging.

An automatic optimization framework has to formalize the space of possible code transformations and select the most beneficial ones. Several approaches to describe the search space for polyhedral programs [28–30], high-level programs written using algorithmic skeletons [31], and domain-specific programs [12, 32, 33] exist. Possible techniques to select code transformations are heuristics [34], performance models [24, 35], and empiric evaluation [19, 20]. We focus on the performance model guided selection of data-locality transformations. The mathematical structure and the fast evaluation of a well-designed performance model can considerably accelerate the search space exploration compared to empirical tuning. A downside of the model-based approach is the limited model accuracy that prevents us from guaranteeing optimality for the selected code transformations.

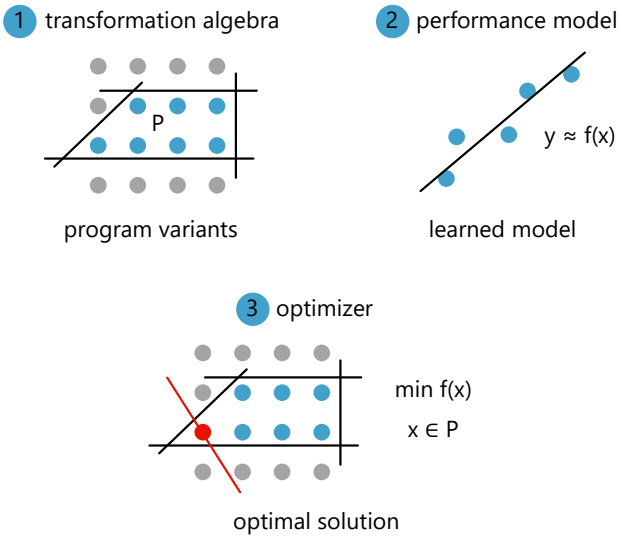


FIGURE 1.1: An optimization problem that implements transformation algebra and performance model enables the automatic selection of data-locality transformations.

Data-locality transformations adapt the program schedule to improve the spatial and temporal locality of the computation. After the optimization computations that access the same data are ideally scheduled to the same compute core and the shared data is stored in cache or registers. At the same time, increasing data-locality often reduces the available parallelism. We thus need to balance data-locality and parallelism [36, 37] while scheduling the computations.

Figure 1.1 shows the main components of a typical automatic optimization framework: 1) an algebra that defines the space of possible code transformations, 2) a performance model to evaluate the effects of the selected code transformations, and 3) an optimizer to explore the search space. All components need to interface with the code generation to automate the optimization process.

TRANSFORMATION ALGEBRA The core of every automatic optimization framework is a transformation algebra that formalizes the space of possible code transformations. We defined the stencil algebra MODESTO to enumerate the space of possible data-locality transformations for stencil

programs. The algebra specifies the execution order and the fusion and tile size choices for all stencils of the program. Every element of the algebra represents a program variant that performs the same computation but has different performance characteristics.

PERFORMANCE MODEL The performance model estimates the cost of the different program variants. We developed the stencil program optimizer ABSINTHE and the analytical cache model HAYSTACK. Both of them can be used to select code transformations. ABSINTHE models the latency and throughput of the stencil loop nests. The simple linear model enables the efficient program optimization using linear integer programming. HAYSTACK models fully associative caches with a least recently used replacement policy. The model is more complex but can compute the cache misses for arbitrary polyhedral programs.

OPTIMIZER An automatic optimization framework searches the program variant that minimizes the cost, for example, by integrating the transformation algebra and the performance model into a single optimization problem. ABSINTHE uses linear integer programming to select optimal code transformations with respect to the performance model. We thereby rely on an existing state-of-the-art solver [38] that is guaranteed to find the optimal solution.

In general, data-locality transformations such as tiling and fusion are interdependent. For example, the optimal tile size of a fused loop nest typically decreases due to the larger memory footprint after fusion. The optimization problem formulated by ABSINTHE performs single shot fusion and tile size choices to account for the interdependence of the different data-locality transformations. The simplicity of both transformation algebra and performance model additionally enables the formulation of a linear optimization problem that can be solved efficiently. We thus believe the development of an automatic optimization framework requires a holistic view of transformation algebra, performance model, and optimization method.

1.2 USER-GUIDED DATA MOVEMENT OPTIMIZATION

Automatic tuning is only available for some application domains, and the results may be suboptimal. In these cases, programmers have to fall back to manual optimization. But the large space of possible data-locality

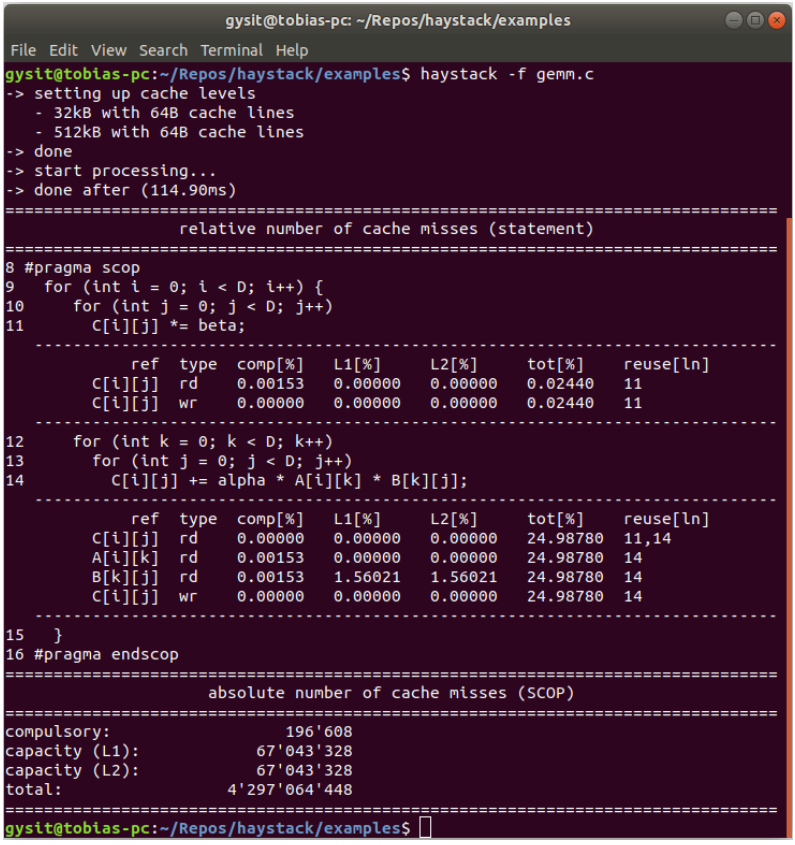


FIGURE 1.2: Screenshot of HAYSTACK analyzing matrix multiplication.

transformations and the complexity and heterogeneity of the available hardware architectures make the manual tuning challenging. Tools that help programmers to identify performance bottlenecks and to evaluate the effectiveness of their code transformations are thus an important concern.

Almost all modern processors rely on caches to reduce data movement and to hide memory access latencies. Caches not only improve performance but unfortunately also make understanding the memory access cost hard. Hence, programmers need to know the state of the cache to estimate the memory access cost. As a result, the cost of data movement depends on global state and does not compose.

The state of the cache depends on the exact memory access history. Minor changes of the memory access history can have significant effects on the cache efficiency. Let us assume a least recently used replacement policy. If a program implements two identical loop nests that access an array with the size of the cache, then all access of the second loop nest are cache hits. But if the program performs an additional memory access between the two loop nests, then all memory accesses of the second loop nest are cache misses. This example illustrates that understanding the cache state requires an exact rather than an approximate understanding of the memory access history.

Programmers may have a notion of the relative cost of different implementation variants. But having the exact memory access history in mind is hard. We thus developed the analytical cache model HAYSTACK that computes exact cache miss information to provide programmers the means to estimate the cost of data movement.

The screenshot in Figure 1.2 shows the output of HAYSTACK for generalized matrix multiplication. The tool analyzes the cache misses for every memory access of the program and prints the percentage of compulsory and capacity misses relative to the total number of memory accesses. This fine-grained analysis allows programmers to estimate the memory access cost of individual loop nests and statements. The tool also computes the total number of compulsory and capacity misses. These absolute numbers are helpful when comparing different implementation variants. For example, we may determine if a tiling is effective by running the tool on a tiled and an untiled implementation variant of the same program.

1.3 IMPORTANCE OF THE PROGRAMMING MODEL

A high-performance computing programming model ideally abstracts low-level implementation details of the target system to improve performance portability. At the same time, the programming model should provide the necessary control to attain optimal performance. These two targets are conflicting and sometimes result in design choices that make specific hardware features inaccessible.

Overlapping computation and inter-node communication [10, 39] is an important data movement optimization in distributed memory computing. We can manually implement it by splitting the compute domain of every node into an inner and an outer domain. We may then overlap the inter-node communication with the computation on the inner domain. The manual application of this optimization is tedious, and its effectiveness depends on the size of the two domains. The computation on the inner domain has to take long enough to overlap the communication. At the same time, both the inner and the outer domain have to be large enough to avoid performance penalties due to the reduced parallelism.

In GPU computing, sufficient amounts of parallelism are a prerequisite to attain optimal performance [40]. Splitting the compute domain to overlap computation and inter-node communication may thus harm the overall performance. Instead, it seems desirable to use the built-in hardware latency hiding – over-subscription and hardware threading – to hide the inter-node communication. But the existing GPU programming models do not provide communication primitives that benefit from this hardware latency hiding.

We developed the DCUDA programming model that implements device-side communication primitives to take advantage of the hardware latency hiding. If a thread waits for incoming data, the GPU immediately proceeds with the execution of another thread that is ready for execution and thus automatically hides the inter-node communication latency. The project demonstrates the importance of an expressive programming model that provides access to all relevant hardware features.

1.4 THESIS CONTRIBUTIONS

This thesis makes the following main contributions:

- In Chapter 2, we present the stencil algebra MODESTO that defines the space of possible data-locality transformations for stencil programs. A

stencil program executes a sequence of different stencils that depend on each other.

- In Chapter 3, we introduce the stencil program optimizer ABSINTHE that learns a performance model to select good data-locality transformations. A single integer linear program optimizes loop fusion and loop tiling in tandem.
- In Chapter 4, we discuss the analytical cache model HAYSTACK that computes the cache misses for polyhedral programs. The model provides both programmers and compilers with exact cache miss information to support the choice of suitable data-locality optimizations.
- In Chapter 5, we introduce the programming model DCUDA that enhances the CUDA programming model with device-side inter-node communication. This extension enables the automatic overlap of communication and computation.

We believe that optimizing the data movement of high-performance computing codes requires better tools, compilers, and programming models. This thesis contributes to all three domains and also motivates further research. The ultimate goal is the development of code generators that learn the performance of the target system and then lower domain-specific code to optimized code that makes optimal use of the hardware. ABSINTHE demonstrates the feasibility of this vision for the stencil domain.

2

A STENCIL ALGEBRA

Stencil computations on regular domains are one of the most important algorithmic motifs in embedded, high-performance, and scientific computing. Applications range from climate modeling [41], seismic imaging [42], fluid dynamics, heat diffusion and electromagnetic simulations [43] through image processing [12] to machine learning. Given their importance, numerous optimization strategies [44–46] and domain-specific languages [11, 12, 22] exist. Yet, most of these schemes consider the optimization of a single stencil in isolation. Many applications, however, require nested stencils [35] that are applied in succession. The data dependencies of these nestings can form complex directed acyclic *stencil graphs* where multiple stencils need to be optimized in tandem to achieve highest performance.

Stencils programs perform element-wise computations on a fixed neighborhood called the stencil. Such stencils often have low arithmetic intensity because they have a fixed number of operations per loaded value. The biggest challenge is to map stencil programs to modern architectures with a growing gap between memory and compute bandwidth. Such architectures require *data-centric optimizations* that arrange data accesses to efficiently use the available memory bandwidth. Complex stencil graphs can be optimized using various techniques such as loop fusion, tiling, and various communication strategies on subgraphs. We model all possible combinations of optimizations for a particular stencil program (graph) using a *stencil algebra* and apply mathematical optimization techniques to find the best combination specific to an abstract hierarchical machine model.

Since typical stencil programs contain hundreds of stencils arranged in paths with dozens of stages and several input arrays, manual tuning of all options is infeasible. In fact, the number of stencil program variants is usually exponential in the number of stencils. In addition, the optimal stencil program variant is specific to each architecture. We show how to fully automate the optimization and implement it in our open-source tool MODESTO, a model driven stencil optimization framework. We demonstrate the efficacy of our method using the real-world application COSMO [41], a numerical weather prediction and regional climate model used by more than 10 national weather services and many other institutions. The dynamical core of COSMO, a central part of its implementation, applies more than

150 stencils, each operating on 13 arrays on average. This most performance-critical code has been rewritten using the STELLA library and was carefully tuned by experts for optimal performance. MODESTO-optimized stencil graphs match or improve upon the expert-tuned code by a factor of 1.0-1.8x. This demonstrates how our technique enables next generation stencil libraries that completely abstract optimizations from the library interface. Hence, we are able to improve usability as well as performance portability compared to state-of-the-art stencil libraries such as STELLA [6] or Halide [12]. In summary, we make the following contributions:

- We introduce a set of data-centric code transformations, an algebraic formulation of the transformation space, and a compile-time performance model that enables the automatic optimization of complex stencil programs.
- We evaluate our approach by modeling the optimization of stencil codes written using the STELLA library and successfully tune kernels of a real-world application.
- We formulate the automatic tuning of stencil programs as a mathematical optimization problem and solve it using dynamic programming techniques.

2.1 STENCIL ALGEBRA

Although the stencil motif appears in a wide range of codes from various application domains, common patterns can be identified. Using them, we introduce a stencil algebra that formalizes stencil computations and facilitates their analysis and optimization.

2.1.1 *Definition of a Stencil Program*

The following core elements describe a stencil program:

A *field* F defines a dense, multi-dimensional and commonly hyperrectangular set of data values, which can be read and modified.

A *stencil* S is a computation that derives a value located in an output field from a set of input field values located within bounded distance to the output value. It is described by the triple (*ops*, *out*, *in*). The first element, *ops*, specifies the (possibly approximated) computational cost of executing this stencil. The second element, *out*, is the output field of the

stencil. The third element, *in*, is a set that defines the input elements of the stencil. The elements of the input set are so-called "named vectors" that are named according to the field the input is read from and the vector itself describes the location of the input element as a relative offset to the position of the element the stencil computes. The set of input elements *in* is not allowed to contain elements of the output field. We define an example stencil s that computes the value $F_0(i,j)$ from the inputs $F_1(i,j)$, $F_1(i,j+1)$ and $F_2(i,j)$ with 5 computational operations using the following notation: $s := (5, F_0, \{F_1(0,0), F_1(0,1), F_2(0,0)\})$

A *stencil program* $P = T \cup O$ consists of a set of temporary stencils T as well as a set of output stencils O, where the results computed by the output stencils form the result of the stencil program, but the results of the temporary stencils are not made available outside of the stencil program. All stencils and fields have the same dimensionality.

The program definition just introduced is formulated minimalistic way and with a strong focus on stencil graphs. As a consequence, it omits aspects that in the context of this work are of limited importance, e.g., boundary conditions, variable input field dimensionality, as well as complex dynamic control flow. However, programs that use such concepts can, in many cases, still be modeled. For example, stencils with varying input sets, due to the use of special boundary conditions, can often be modeled with an over-approximated input set and iterative stencil computations can be modeled by (partially) unrolling the relevant time loop.

2.1.2 *Example*

We now present an example stencil program which is derived from a horizontal diffusion kernel used by the COSMO atmospheric model [41]. We define the stencil program P_{hd} in terms of the temporary stencils s_{lap}, s_{fli}, and s_{flj} necessary to evaluate the output stencil s_{out}. A data dependency either refers to an input field loaded by the stencil program, such as *in* or

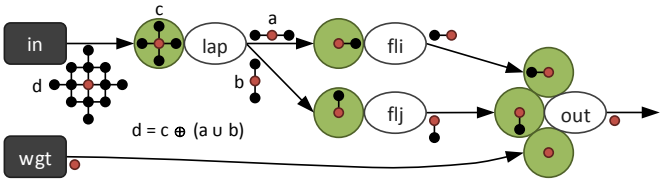


FIGURE 2.1: Horizontal diffusion dependency graph annotated with stencil (c) and stencil program (a, b, and d) access patterns

wgt, or to a temporary field computed by the corresponding temporary stencil, such as fli, flj, or lap:

$$
\begin{aligned}
s_{lap} &:= (5, lap, \{in(-1,0), in(1,0), in(0,-1), in(0,1), in(0,0)\}) \\
s_{fli} &:= (1, fli, \{lap(1,0), lap(0,0)\}) \\
s_{flj} &:= (1, flj, \{lap(0,1), lap(0,0)\}) \\
s_{out} &:= (5, out, \{fli(-1,0), fli(0,0), flj(0,-1), flj(0,0), wgt(0,0)\}) \\
P_{hd} &:= \{s_{lap}, s_{fli}, s_{flj}\} \cup \{s_{out}\}
\end{aligned}
$$

Figure 2.1 illustrates the data flow of the stencil program using a directed graph, whose black and white nodes represent input fields and stencils, respectively. Arrows that do not point to a node and consequently exit the stencil graph model the outputs of the stencil program. A directed edge in the graph corresponds to a flow dependency between two nodes. We annotate each incoming edge of a stencil with the access pattern necessary for a single stencil evaluation. For instance, a single evaluation of the lap stencil accesses the in field at the five offsets shown by c. In addition, we annotate all outgoing edges of a stencil or an input field with the accumulated access pattern necessary to evaluate the out stencil at a single position. E.g., the lap stencil is evaluated at the positions defined by the union of the sets a and b. We compute the accumulated in field access pattern d as the Minkowski sum $d = (a \cup b) \oplus c$, with $a \oplus b = \{a' + b' \mid a' \in a, b' \in b\}$. Figure 2.2 shows a naive implementation of the horizontal diffusion kernel, which executes each stencil using a separate loop nest. While such an implementation may be straightforward to write, it is not efficient in terms of data locality, memory usage, or parallelism.

```
1  // allocate temporary storage
2  Field[double] lap(ibegin,iend), fli(ibegin,iend), flj(ibegin,iend);
3  // apply the lap stencil
4  for(int i=ibegin-1; i<iend+1; ++i)
5    for(int j=jbegin-1; j<jend+1; ++j)
6      lap(i,j) = -4.0 * in(i,j) +
7        in(i-1,j) + in(i+1,j) + in(i,j-1) + in(i,j+1);
8  // apply the fli stencil
9  for(int i=ibegin-1; i<iend; ++i)
10   for(int j=jbegin; j<jend; ++j)
11     fli(i,j) = lap(i+1,j) - lap(i,j);
12 // apply the flj stencil
13 for(int i=ibegin; i<iend; ++i)
14   for(int j=jbegin-1; j<jend; ++j)
15     flj(i,j) = lap(i,j+1) - lap(i,j);
16 // apply the out stencil
17 for(int i=ibegin; i<iend; ++i)
18   for(int j=jbegin; j<jend; ++j)
19     out(i,j) = wgt(i,j) *
20       (fli(i-1,j) - fli(i,j) + flj(i,j-1) - flj(i,j));
```

FIGURE 2.2: Naive implementation of the simplified horizontal diffusion example used by the COSMO [41] atmospheric model

2.1.3 *Data Locality Transformations*

To improve the data locality of stencil programs, we discuss code transformations that combine loop tiling and loop fusion. While tiling sub-divides the loop domain into typically hyperrectangular tiles of limited size, fusion substitutes a sequence of loops by a single loop. Applied to stencil codes, we divide the stencil evaluation domain into tiles and apply multiple stencils tile-by-tile. Consequently, we can store temporary values in smaller buffers that hold the working set of a single tile instead of the full evaluation domain.

While tiling increases the data locality, it causes additional synchronization efforts at the tile boundaries. As shown in Figure 2.1, a single stencil evaluation depends on one or more input or temporary fields accessed in a local neighborhood. When combining multiple stencils the neighborhoods grow depending on the stencil access patterns and the longest path in the dependency graph. We call all dependencies outside of the tile domain the *halo points* of a tile. In addition, we suggest three halo strategies that trade off parallelism against computation. Figure 2.3 shows the iteration space

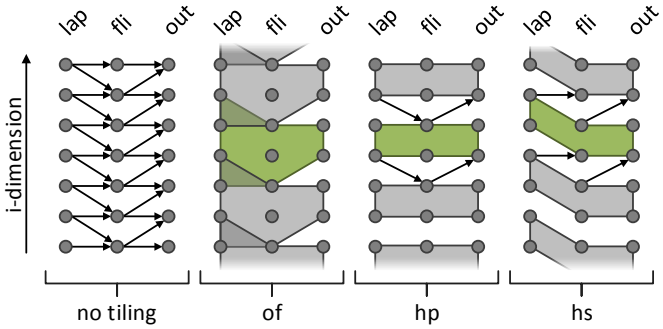


FIGURE 2.3: Tile shapes (shaded) for different tilings applied to a subset of the horizontal diffusion example projected to the *i*-dimension

of one dependency path in the horizontal diffusion example, once without any tiling and then with different tiles as they result from the suggested halo strategies. Shaded regions mark the points that belong to a specific tile.

COMPUTATION ON-THE-FLY (OF) satisfies all halo point dependencies using redundant computation at the tile boundaries. Hence, we load input fields and evaluate temporary stencils in an extended domain covering the tile itself as well as its halo points. Using computation on-the-fly, we can update different tiles independently postponing synchronization at the cost of additional computation. As shown by Figure 2.3, computation on-the-fly results in overlapping tiles and is therefore often referred to as overlapped tiling [8, 9, 12].

HALO EXCHANGE PARALLEL (HP) satisfies all halo point dependencies using communication with neighboring tiles. More precisely, we update all tiles in parallel and perform at least one halo exchange communication per edge in the longest dependency chain of the stencil dependency graph. Hence, halo exchange parallel avoids redundant computation at the cost of additional synchronizations.

HALO EXCHANGE SEQUENTIAL (HS) modifies the tile shape such that all unsatisfied halo point dependencies point in one direction. By iterating over the tiles in reverse dependency direction, we can update all tiles sequentially using a single sweep. While halo exchange sequential in general applies to one-dimensional tilings only, we can complement it with other halo

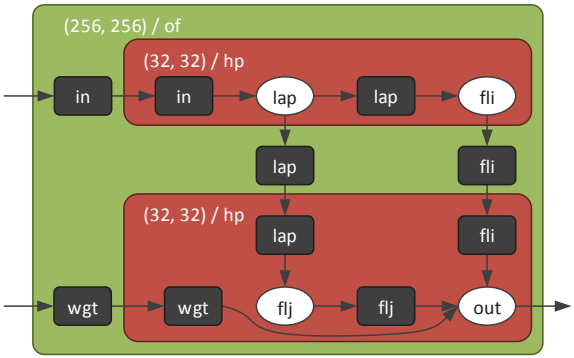


FIGURE 2.4: Stencil dependency graph of the horizontal diffusion example annotated with two tiling hierarchy levels.

strategies to support higher dimensional tilings. In summary, halo exchange sequential avoids redundant computation and synchronizations at the cost of being sequential.

As the surface to volume ratio decreases with increasing tile size, we preferably update small tiles using halo exchange communication and large tiles using computation on-the-fly. Depending on the hardware architecture high synchronization costs make computation on-the-fly attractive. Overall, choosing the optimal data locality transformations is not straight forward and motivates the use of a performance model.

2.1.4 *Stencil Algebra Definition*

Using the data locality transformations introduced in the previous section, we are able to generate a large number of stencil program implementation variants. In particular, we can repeatedly apply our tiling transformations to obtain a hierarchical tiling that leverages multiple levels of the memory hierarchy. By combining our data locality transformations, we are therefore able to cover most of the established stencil implementation techniques. Next, we formally define a stencil algebra whose elements express different stencil program implementation variants and show how to enumerate them. Figure 2.4 shows an implementation variant of the horizontal diffusion example, introduced in Section 2.1.2, annotated with two tiling hierarchy levels. Each white node corresponds to a stencil and each black node to a storage region that buffers either an input or a temporary field. We extend

the dependency graph with boxes that represent the tiling hierarchy. More precisely, the boxes form a tiling tree where each box corresponds to a tiling that executes all contained boxes respectively stencils. Finally, we annotate each box with the tile size and the halo strategy of the tiling. In Figure 2.4 we employ an on-the-fly tiling at the bottom of the tiling hierarchy with two nested halo exchange parallel tilings.

In order to specify an element of our stencil algebra, we initially define a tiling hierarchy. More precisely, we define a tile size $t^l \in \mathbb{Z}^n$ for each level l of the tiling hierarchy. In case of the horizontal diffusion example we define two tiling hierarchy levels:

$$t^1_{hd} = (256, 256) \qquad t^2_{hd} = (32, 32)$$

Next, we specify a stencil program implementation variant as a bracket expression. We put all stencils that correspond to a specific tiling hierarchy level into brackets. A hierarchical tiling thus results in a nested bracket expression with the outermost bracket term representing the bottom of the tiling hierarchy. We can define the horizontal diffusion implementation variant shown by Figure 2.4 using a twofold nested bracket expression.

$$[[s_{lap}, s_{fli}], [s_{flj}, s_{out}]]$$

In the following, we call each bracket term representing a tiling hierarchy a stencil group. A stencil group can be seen as a node of the tiling tree containing nested stencils or stencil groups that as a whole define the stencil program implementation variant.

Let g be a stencil group, then $g.child$ is the set of all children of the stencil group g, where a child is either a stencil or a nested stencil group. In addition, $g.sten$ is the set of all stencils in the subtree defined by the stencil group g. Finally, $g.in$ and $g.out$ define the input and output sets of a stencil group g, where an input and an output correspond to an incoming respectively to an outgoing data dependency. As an example, we provide the stencil properties of the horizontal diffusion example shown in Figure 2.4.

$$g_0 = [g_1, g_2] \qquad g_1 = [s_{lap}, s_{fli}] \qquad g_2 = [s_{flj}, s_{out}]$$

First, we define the tree properties.

$$g_0.child = \{g_1, g_2\} \qquad g_0.sten = \{s_{lap}, s_{fli}, s_{flj}, s_{out}\}$$

Next, we define the external data dependencies.

$$g_0.in = \{in, wgt\} \qquad g_0.out = \{out\}$$

We enumerate all stencil program implementation variants using two operations: 1) shuffle the stencils respecting their topological order and 2) group stencils on different tiling hierarchy levels.

2.1.5 *Performance Modeling*

In order to understand the performance characteristic of a stencil program implementation variant, we next introduce a performance model. Similar to the Roofline model [47], we estimate the execution time based on the peak compute and communication throughput of the target hardware. In addition, we do not only distinguish between cached and global memory accesses but model additional memory hierarchy levels.

To model our target hardware we use an abstract machine that is built around a processing unit that performs computations on a limited set of local registers. All data is by default stored in a global memory (e.g., DRAM) with limited bandwidth to the processing unit. Data is transferred from global memory to local registers before any computation is performed and the results of a computation are transferred back to global memory before becoming externally visible. Between global memory and local registers there is a set of additional hierarchically organized memory levels, each with limited size, but increasing bandwidth to the processing unit.

When mapping a parallel hardware architecture to our model, the bandwidth of a given memory hierarchy level is the combined bandwidth of all (possibly multiple) memories at this level. The size of a memory hierarchy level is not the combined size, but the size of an individual memory at this level. E.g., assuming there are multiple L1 caches, we consider the size of a single L1 cache. Finally, assuming sufficient parallelism to simultaneously use all processing resources, the compute throughput of our model is the combined peak compute throughput of the hardware architecture.

We now consider again Figure 2.4, an illustration of a stencil program implementation variant with two tiling hierarchy levels that was introduced in the previous section. Each tiling hierarchy targets one specific level of the memory hierarchy, such as the DDR memory or the L1 cache of a CPU. We assume all input and temporary values of a stencil group are stored in the associated memory hierarchy level. Whenever a stencil program communicates data from one tiling hierarchy level to the next higher one, we model the communication time using the bandwidth of the associated memory hierarchy level. Therefore, we define a communication bandwidth $V^l \in \mathbb{R}$ as well as a memory capacity $M^l \in \mathbb{Z}$ for each level

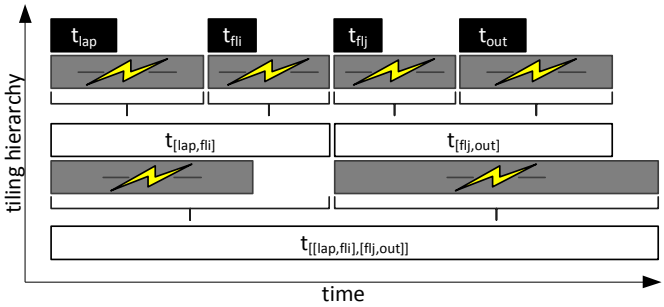


FIGURE 2.5: The time estimation for the horizontal diffusion example

l of the tiling hierarchy. In addition to this vertical communication, a stencil code might also perform lateral halo exchange communication between neighboring tiles of the tiling hierarchy. Hence, we define a lateral communication bandwidth $L^l \in \mathbb{R}$ for each level l of the tiling hierarchy. Typical representatives of lateral communication links are interconnect networks or the scratch pad memory of a GPU. Finally, we define the compute throughput $C \in \mathbb{Z}$ of the target architecture. Thereby, we define storage sizes in terms of floating point values instead of bytes. In case two nested tiling hierarchy levels are associated to the same memory hierarchy level, we set the vertical communication bandwidth to infinity. Just like the Roofline model, we assume that we can overlap communication and computation on all communication links respectively compute units of the system.

When modeling the performance of a stencil program, we assume that the arithmetic intensity remains constant during the execution of a single stencil. On the other hand, the arithmetic intensities of different stencils might vary. Figure 2.5 illustrates the time estimation for the horizontal diffusion implementation variant shown by Figure 2.4. At the top of the tiling hierarchy, black boxes denote the stencil execution times. Below, gray boxes (with flashes) denote the communication times between parents and children in the tiling hierarchy. Furthermore, white boxes denote the stencil group execution times computed as the sum of the maximum between stencil execution times and communication times.

In particular, we estimate the execution time t_s of a stencil s that performs c_s floating point operations as the time needed to compute the stencil without considering any communication cost.

$$t_s = c_s / C$$

Using the child execution time t_c of a child stencil or stencil group c that causes v_c vertical and $l_c^1, \ldots, l_c^l$ lateral data movements, we compute the execution time t_g of a stencil group g that corresponds to level l of the tiling hierarchy as the sum of the maximum of the child execution times, the vertical communication between the stencil group and its children, and the lateral communication necessary to update the halo points of the temporary fields. We thereby optimistically assume the lateral communication overlaps with the child execution, which assumes the later communication is sufficiently balanced over the stencil group execution.

$$t_g = \sum_{c \in g.child} \max(t_c, v_c / V^l, l_c^1 / L^1, \ldots, l_c^l / L^l)$$

We model the performance of an entire stencil program as the estimated execution time of the stencil group at the bottom of the tiling hierarchy. Furthermore, we complement the performance estimation with a feasibility check that compares the storage requirements of the stencil program to the available memory capacity on all tiling hierarchy levels.

2.1.6 *Stencil Program Analysis*

In order to evaluate our performance model, we analyze stencil programs using the mathematical concept of affine sets and affine maps. In particular, we show how to count the number of floating point operations, data movements, and storage locations required during the stencil program execution. Using the performance model introduced in Section 2.1.5, our analysis finally allows us to estimate the execution time and the feasibility of a stencil program.

2.1.6.1 *Affine Sets and Maps*

An affine set $S = \{\vec{i} \mid \vec{i} \in \mathbb{Z}^n \wedge \text{cons}(\vec{i})\}$ is a set of n-dimensional integer vectors, where the elements of the set are constrained by a Presburger formula $\text{cons}(\vec{i})$. Presburger formulas consist of comparisons ($<, \leq, =, \neq, \geq, >$) between expressions (quasi-)affine in vector dimensions and external parameters that are combined by Boolean operations ($\wedge, \vee, \neg$). For affine sets set operations such as union, intersection, subtraction, projection as well as cardinality are defined.

An affine map $M = \{\vec{i} \to \vec{j} \mid \vec{i} \in \mathbb{Z}^n, \vec{j} \in \mathbb{Z}^m \wedge \text{cons}(\vec{i}, \vec{j})\}$ is a relation, that relates n-dimensional input (domain) vectors with m-dimensional output (range) vectors. The elements are again constraint by a Presburger formula

$\text{cons}(\vec{i}, \vec{j})$. Besides the normal set operations, there exist map-specific operations such as the application of a map m on a set s ($m(s)$), the composition of two maps ($m_0 \circ m_1$), or the inverse of a map (m^{-1}), which switches input and output of a map. We define the following set of important map operations in more detail.
The range product of two maps R_1 and R_2 is defined as:

$$R_1 \times_{\text{ran}} R_2 = \{ \vec{i} \to (\vec{j_1}, \vec{j_2}) \mid \vec{i} \to \vec{j_1} \in R_1 \wedge \vec{i} \to \vec{j_2} \in R_2 \}$$

The range intersection of a map R with a set S is:

$$R \cap_{\text{ran}} S = \{ \vec{i} \to \vec{j} \mid \vec{i} \to \vec{j} \in R \wedge \vec{j} \in S \}$$

The range-projection of a map R projects the n output dimensions of a map onto the first $k+1$ output dimensions:

$$\mathcal{P}^{\text{ran}}_{[0-k]}(R) = \{ \vec{i} \to (j_0, \ldots, j_k) \mid \exists x_{k+1}, \ldots, x_{n-1} \in \mathbb{Z} : \vec{i} \to (j_0, \ldots, j_k, x_{k+1}, \ldots, x_{n-1}) \in R \}$$

R^+ is the transitive closure of R:

$$R^+ = \{ \vec{i} \to \vec{j} \mid \exists m \geq 0 : \vec{j} = (\underbrace{R \circ \cdots \circ R}_{m \text{ times}})(\vec{i}) \}$$

We use $|S|$ to specify the cardinality of a set and $|R|$ to specify the cardinality of a map, where the cardinality of a map is defined as the number of related domain and range pairs.

We also define named sets and named maps as affine sets and maps that contain so-called "named vectors". The elements of these sets can either be written as tuples of a string and a vector, for example $\{(\text{“}A\text{”}, \vec{i}), (\text{“}B\text{”}, \vec{j}) \mid \vec{i} \in \mathbb{Z}^n, \vec{j} \in \mathbb{Z}^m\}$, or as named vectors $\{A(\vec{i}), B(\vec{j}) \mid \vec{i} \in \mathbb{Z}^n, \vec{j} \in \mathbb{Z}^m\}$. Named sets (maps) allow differently named elements to have vectors of different dimensionality. On named sets and maps the operations introduced above are applied individually to subsets or submaps that share the same name and dimensionality. To extract a set from a named set S, we define a bracket operator $S[\text{“}x\text{”}] = \{(\text{“}x\text{”}, \vec{i}) \mid (\text{“}x\text{”}, \vec{i}) \in S\}$. The bracket operator applied on a map, filters the maps according to the name of their domains $R[\text{“}x\text{”}] = \{(\text{“}x\text{”}, \vec{i}) \to (name, \vec{j}) \mid (\text{“}x\text{”}, \vec{i}) \to (name, \vec{j}) \in R\}$.

Computations on integer sets can be performed with isl [48] and counting of integer sets is possible using the Barvinok algorithm [49].

2.1.6.2 *Data Dependencies*

Given a stencil program P the set of flow dependencies in P can be derived from the stencil data dependencies. To obtain them, we define for each stencil $s \in P$ a map D_s that associates the stencil evaluations to the corresponding input data dependencies.

$$D_s = \{s.out(\vec{u}) \rightarrow d(\vec{u} + \vec{v}) \mid d(\vec{v}) \in s.in\}$$

Next, we define the union of all stencil data dependencies.

$$D = \bigcup_{s \in P} D_s$$

2.1.6.3 *Stencil Tiling Maps*

We model the tiling transformations discussed in Section 2.1.3 using affine maps that relate the stencil evaluation domain to the tile domain. More precisely, we define for each stencil a tiling map that maps each point in the n-dimensional stencil evaluation domain to an n-dimensional tile identifier, such that all points that belong to the same tile are associated with a common tile identifier. We initially consider only a single tiling level and later generalize the concept to nested tilings.

Given a multi-dimensional tile size vector $\vec{t} = (t_0, \ldots, t_{n-1}) \in \mathbb{Z}^n$, we define a hyperrectangular tiling of a single stencil s as a named map $T_s^{\square}$ that associates each point $\vec{i} = (i_0, \ldots, i_{n-1}) \in \mathbb{Z}^n$ of the stencil evaluation domain with exactly one tile identifier.

$$T_s^{\square} = \{(s, \vec{i}) \rightarrow (\lfloor i_0/t_0 \rfloor, \ldots, \lfloor i_{n-1}/t_{n-1} \rfloor)\}$$

Depending on size and alignment of tiles and stencil evaluation domains, such a tiling may yield truncated tiles at the stencil evaluation domain boundaries. In case a given dimension of the stencil evaluation domain should not be tiled (indicated by tile size ∞), the corresponding dimension of the tile identifiers is set to zero.

We represent the tiling of a stencil group g by computing a named map that contains a tiling map for each stencil of the stencil group. We distinguish here between the three halo strategies introduced in Section 2.1.3.

Computation on-the-fly satisfies halo point dependencies using redundant computation. The corresponding tiling map is therefore a relation which maps the halo point stencil evaluations at the tile boundaries to multiple overlapping tiles. Given a stencil group g, we construct a tile map T_g

in two steps. First, all output stencils of g are tiled with a rectangular tiling map. This does not yet introduce any redundant computation. Next, we compute for each tile all stencil evaluations that are required to compute the output points already assign to this tile. We do this by first defining the set of data dependencies D_g that are local to g and then composing the inverse transitive hull of D_g with the tiling map already defined for the output stencils. The resulting map connects the temporary stencil evaluations via the dependent output stencil evaluation to the corresponding tile identifier. This map may now possibly relate one temporary stencil evaluation to multiple tiles and can consequently introduce redundant computation.

$$T_g = \bigcup_{s \in g.out} T_s^{\square} \circ (D_g^+)^{-1}$$

Halo exchange parallel satisfies halo point dependencies using communication. We therefore assign each point in the stencil evaluation domain to exactly one tile and use tiles of identical size, shape and alignment for all stencils in our stencil group. The tiling map T_g describes such a tiling for a stencil group g.

$$T_g = \bigcup_{s \in g.sten} T_s^{\square}$$

Halo exchange sequential is a variant of halo exchange parallel, whose tiling map is constructed accordingly. In contrast to halo exchange parallel, we shift the stencil tiling maps such that all unsatisfied halo point dependencies between tiles point in one direction. Figure 2.3 illustrates the tile shape of shifted stencil tiling maps and their halo point dependencies. We define a shifted tiling map by subtracting the shift offset form the stencil evaluation domain before computing the associated tile identifiers.

NESTED TILINGS We now describe the construction of nested tilings, tilings that result from recursively applying the previously introduced tiling transformations. To give a first intuition of such tilings, Figure 2.6 shows the different nested tilings that can be constructed from combining on-the-fly and halo exchange parallel tiling on two tiling levels. It shows for each combination one full outer tile, one full inner tile, and, using dashed lines, the remaining inner tiles placed inside the outer tile. Most combinations are rather straightforward, but it is interesting to note, that in case of on-the-fly tiling being nested inside halo exchange parallel tiling, the redundant computation of the on-the-fly tiles may require the computation of points located outside of the surrounding tile.

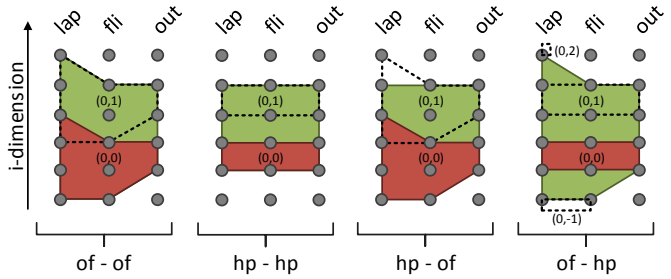


FIGURE 2.6: Tile shapes (shaded) for a nested tiling applied to a subset of the horizontal diffusion example projected to the i-dimension

As visible in the illustration just discussed, we identify each nested tile with a tile vector whose first and second entry correspond to the tile identifiers of the first and second tiling level, respectively. Hence, we can model a nested tiling with l tiling hierarchy levels with a tiling map that relates each point in the n-dimensional stencil evaluation domain to a tile identifier with $n \cdot l$ dimensions. To construct such a map for a given stencil group g nested in another stencil group p we first define tiling maps for the output stencils of g. These tiling maps are formed by combining for each stencil the tiling map $T_p[s]$ that we derive for this stencil from p (not considering any nested groups) with an additional hyperrectangular tiling that uses the tile sizes specified for g. We define the tiling map $T_{g,s}$ of such a stencil s as the range product of the tiling map $T_s^{\square}$ with the recursively computed parent tiling map $T_p[s]$.

$$T_{g,s} = T_p[s] \times_{\text{ran}} T_s^{\square}$$

When computing the tiling map of a nested stencil group T_g, we adapt the previously introduced on-the-fly and halo exchange tiling maps to use $T_{g,s}$ instead of $T_s^{\square}$. The resulting tiling maps for halo exchange parallel and on-the-fly tiling are

$$T_g = \bigcup_{s \in g.sten} T_{g,s} \qquad \text{and} \qquad T_g = \bigcup_{s \in g.out} T_{g,s} \circ (D_g^+)^{-1}.$$

We can now define for each stencil a tiling map T_s that maps each evaluation of this stencil to a tile identifier with $l \cdot n$ dimensions, that identifies for all levels of the tiling hierarchy the tiles the stencil evaluation is assigned to.

We obtain T_s by extracting the tile map that corresponds to s from the tile map of the stencil group g at the top of the tiling hierarchy that contains s.

$$T_s = T_g[s]$$

When constructing hierarchical tilings that involve halo exchange sequential, we inherit the shift offsets introduced by the sequential execution to all nested tiling hierarchy levels. Thereby, we align the nested tiles to the parent tile boundaries.

2.1.6.4 *I/O Maps*

While the tiling maps alone allow the analysis of computational aspects, we introduce auxiliary maps that support the analysis of data movements and storage usage.

First, we define for each stencil s an input map I_s that relates a set of inputs (stencil evaluations or input fields) used by a certain evaluation of s to the tile(s) this evaluation is assigned to. The construction of I_s is similar to the construction of the on-the-fly tiling. We compose the stencil tiling map T_s with the reversed stencil data dependencies D_s^{-1}. Furthermore, we define the input map of an entire stencil group g as the union of all nested stencil input maps.

$$I_s = T_s \circ D_s^{-1} \qquad I_g = \bigcup_{s \in g.sten} I_s$$

Second, we define for each child stencil or stencil group c an output map O_c that relates the set of outputs written by the child to the tiles they are assigned to. In case the parent stencil group applies halo exchange communication, we define the output map O_c as the union of the child output stencil tiling maps.

$$O_c = \bigcup_{s \in c.out} T_s$$

In case the parent stencil group applies computation on-the-fly, we compute the output map by following the data dependencies starting from the parent stencil group output stencils. While this construction is similar to the computation of the on-the-fly stencil evaluation tiling map, it differs by the fact that we only consider the data dependencies of the stencils executed after the child stencil or stencil group. Thereby, we make sure we do not consider internal dependencies between the output stencils of the

child stencil group. Initially, we define the partial input map $I_{p,c}$ of a parent stencil group p and a child stencil or stencil group c considering all input dependencies of children executed after the child c.

$$I_{p,c} = \bigcup_{\substack{c_i \in p.child \\ c < c_i}} I_{c_i}$$

Then the output map O_c of a child stencil or stencil group is the union of all partial input and parent output dependencies.

$$O_c = \bigcup_{s \in c.out} \left(I_{p,c}[s] \cup \left(\bigcup_{o \in p.out} T_{p,o} \right)[s] \right)$$

2.1.6.5 *Tile Selection*

We analyze the characteristics of a stencil program by counting stencil evaluations, data movements, or storage requirements on a limited domain. As we are interested in the relative rather than the absolute performance and as our performance model does not consider low hardware utilization due to strong scaling, we can choose an arbitrary but limited domain size. We therefore perform our analysis on the origin tile of the lowest tiling hierarchy level. Assuming m tiling hierarchy levels, we select the origin tile of the lowest tiling hierarchy level using the tile selection set S that contains all tile identifiers with the first n-dimensions fixed to zero.

$$S = \{(x_0, \ldots, x_{n-1}, y_n, \ldots, y_{nm}) \mid x_i = 0 \wedge y_j \in \mathbb{Z}\}$$

When analyzing the storage requirements, we want to make sure a single tile fits the memory capacity of the corresponding memory hierarchy level. We thus define an additional tile selection set S^* that selects the origin tile on all levels of the tiling hierarchy.

$$S^* = \{(x_0, \ldots, x_{nm}) \mid x_i = 0\}$$

In order to limit the domain of a tiling map, we finally intersect the range of the tiling map with a selection set.

2.1.6.6 *Analysis*

Relying on the previously introduced stencil program formulation, we now discuss the analyses we use to obtain the program properties needed for

evaluating the performance model introduced in Section 2.1.5. Using the previously introduced maps, we count the points that correspond to the number of stencils evaluations, the amount of data moved, and the amount of storage used when evaluating a given stencil program on a limited domain.

COMPUTATION In order to analyze the amount of computation performed by a stencil program, we count the stencil evaluations associated to the origin tile of the lowest tiling hierarchy level. We obtain these evaluations by intersecting the range of the stencil evaluation tiling map with the origin tile selection set S. We then count all stencil evaluations associated to the remaining tile identifiers. Hence, we define the amount of computation c_s performed by a stencil s as the cardinality of the constraint tiling map times the number of floating point operations performed by a single stencil evaluation.

$$c_s = |T_s \cap_{\text{ran}} S| \cdot s.ops$$

VERTICAL COMMUNICATION As discussed in Section 2.1.5, vertical communication refers to the data movements between a parent stencil group and its child stencils or stencil groups. We therefore analyze the number of loads and stores performed by a child stencil or stencil group when executed by a parent stencil group. We analyze the vertical communication on a restricted domain that corresponds to the origin tile of the lowest tiling hierarchy level.

In order to compute the number of loads performed by a stencil or stencil group c, we count the elements in the constraint input map of c. More precisely, we intersect the range with the origin tile selection set and project out any dimension above the parent stencil group tiling hierarchy level l. Due to the projection, the points in the resulting map describe all elements loaded by the child stencil or stencil group not considering redundant stencil evaluations on nested tiling hierarchy levels. Hence, we define the number of loads r_c performed by a child stencil or stencil group c as the cardinality of the constraint and projected child input map.

$$r_c = \sum_{s \in c.in} |\mathcal{P}^{\text{ran}}_{[0-nl]}(I_c[s] \cap_{\text{ran}} S)|$$

Accordingly, we define the number of stores w_c performed by a child stencil or stencil group c as the cardinality of the constraint and projected child output map.

$$w_c = \sum_{s \in c.out} |\mathcal{P}^{\text{ran}}_{[0-nl]}(O_c[s] \cap_{\text{ran}} S)|$$

Finally, we define the total amount of vertical communication of a child stencil or stencil group c as the sum of its loads and stores.

$$v_c = r_c + w_c$$

LATERAL COMMUNICATION Lateral communication refers to the halo exchange communication between neighboring tiles of the same tiling hierarchy level. We therefore compute the lateral communication performed by a stencil group as the difference between the amount of computed and the amount of consumed temporary values, which corresponds to the unsatisfied halo point dependencies between the children of the stencil group. We analyze the lateral communication on a restricted domain that corresponds to the origin tile of the lowest tiling hierarchy level.

We compute the amount of lateral communication necessary to update the outputs of a child stencil or stencil group, as the difference of the elements used by subsequent children and the elements written by the child itself. We thus intersect the range of this difference with the origin tile selection set and project out any dimensions above the parent stencil group tiling hierarchy level l. Hence, we define the amount of halo points l_c communicated by a child stencil or stencil group c as the cardinality of the difference between the projected and constraint partial input and output maps.

$$l_c = \sum_{s \in c.out} |\mathcal{P}^{\text{ran}}_{[0-nl]}((I_{p,c}[s] \setminus O_c[s]) \cap_{\text{ran}} S)|$$

In case multiple nested tiling hierarchy levels employ halo exchange communication, we possibly run lateral communication on all these levels. By projecting out one level after the other, we assign the lateral communication to the different levels of the tiling hierarchy. Thereby, we get the sum of the lateral communication on the remaining tiling hierarchy levels not yet projected out. By computing the difference of adjacent levels, we finally get the lateral communication assigned to exactly one level.

STORAGE REQUIREMENTS We analyze the feasibility of a stencil program by computing an upper bound for the storage necessary in order to execute a single tile on each level of the tiling hierarchy. We therefore analyze the storage requirements on a restricted domain that corresponds to the origin tile on all levels of the tiling hierarchy. In case the upper bound exceeds the capacity of one memory hierarchy level, we say a stencil program is infeasible.

We compute the storage requirement of a stencil group as the amount of storage necessary to evaluate the stencil group on a single tile. As shown by Figure 2.4, we reserve storage for each input and temporary field used during the evaluation of the stencil group. In contrast, output fields are immediately written to storage managed outside of the stencil group. We overestimate the storage requirement, for example, since the lifetime of some fields might allow sharing a common buffer. We evaluate the storage requirements using the input map intersected with the tile selection set S^*. Furthermore, we project out any dimension above the parent stencil group tiling hierarchy level l. Hence, we define the amount of storage m_p required by a parent stencil group p as the cardinality of the constraint and projected input maps.

$$m_p = \sum_{c \in p.child} \sum_{s \in c.in} |\mathcal{P}^{\text{ran}}_{[0-nl]}(I_p[s] \cap_{\text{ran}} S^*)|$$

In order to determine the feasibility of a stencil program, we compare the memory requirements of each stencil group to the available memory capacity.

2.2 CASE STUDY

We evaluate our approach using the real-world application COSMO. Its dynamical core was recently rewritten using the STELLA [6] stencil library, which exposes the possibility to manually fuse or split stencils on multiple tiling hierarchy levels. In this case study we show how to automatically tune STELLA programs.

2.2.1 *STELLA*

STELLA is a domain-specific embedded language for finite difference methods that is designed to separate the stencil specification from the hardware architecture specific implementation strategy. When executing a stencil program STELLA uses two levels of parallelism: 1) coarse-grained parallelization that decomposes the stencil evaluation domain into blocks executed on different processing units and 2) fine-grained parallelization that executes the individual blocks on a single processing unit possibly using vectorization and hardware threads. STELLA supports stencil fusion on three different tiling hierarchy levels. We can apply consecutive stencils using a single loop over a block, using multiple separate loops over a block, or using multiple separate loops over the full domain.

Hierarchy	Vertical	Tile Size	Strategy
1	DDR	$(256, 256, 64)$	of
2	L2	$(8, 8, 64)$	of

TABLE 2.1: CPU tiling hierarchy

Hierarchy	Vertical/Lateral	Tile Size	Strategy
1	GDDR/-	$(256, 256, 64)$	of
2	GDDR/-	$(64, 4, 64)$	of
3	Register/Register	$(\infty, \infty, 1)$	hs
4	Register/Shared	$(1, 1, 1)$	hp

TABLE 2.2: GPU tiling hierarchy

At compile-time, STELLA generates target architecture specific loop code using C++ template meta-programming. With two available backends, STELLA can currently target CPU and GPU architectures using the OpenMP and CUDA programming models, respectively. Thereby, STELLA employs a fixed but platform specific tiling hierarchy, which we will model using our stencil algebra.

We model the CPU backend of STELLA using the two tiling hierarchy levels shown by Table 2.1. As discussed in Section 2.1.6, we compute all stencil program performance characteristics for the origin tile of the base tiling hierarchy level. Therefore, we introduce a first tiling hierarchy level that represents the stencil program evaluation domain. A second tiling hierarchy level models the coarse-grained parallelism of STELLA. Currently, the CPU backend does not implement fine-grained parallelism. Hence, there is no need to model the third tiling hierarchy level of STELLA.

We model the GPU backend of STELLA using the four tiling hierarchy levels shown by Table 2.2. Just as in the case of the CPU backend, we introduce two tiling hierarchy levels to model the stencil program evaluation domain and the coarse-grained parallelism. We also add two additional tiling hierarchy levels to represent the fine-grained parallelism. The GPU backend allocates one thread per ij-position (tiling hierarchy level 4) that iterates over all points in the k-dimension (tiling hierarchy level 3). The different threads communicate via shared memory, while consecutive loop

iterations executed by the same thread communicate via registers. Tile size infinity indicates that there is no tiling in the corresponding dimension.

2.2.2 *Stencil Program Optimization*

When implementing a stencil program using STELLA, we have multiple degrees of freedom. As discussed in Section 2.1.4, we can change the stencil evaluation order and fuse or split the execution of successive stencils on multiple levels of the tiling hierarchy. We therefore split the optimization in two steps and apply different optimization methods: 1) we optimize the stencil evaluation order using brute force search 2) we optimize the tiling for a given stencil evaluation order using dynamic programming. During our optimization we do not consider tile size choices, but rely on the tile sizes that are used by COSMO and have proven robust for a wide range of stencil programs and their implementation variants.

In order to optimize the stencil evaluation order, we enumerate all topological sorts of the stencil dependency graph using brute force search. In general, a graph may have up to $\mathcal{O}(n!)$ valid topological orders. However, due to its data dependency chains a typical stencil dependency graph has less topological orders resulting in a much smaller search space.

In a second step, we search the optimal tiling given a stencil evaluation order. Using a tiling hierarchy and an abstract machine model, we search for a tiling with minimal estimated execution time and a storage requirement that fits all levels of the memory hierarchy. We estimate execution time and storage requirements using the analysis introduced in Section 2.1.6. In order to enumerate the search space, we fuse all pairs of subsequent stencils on all levels of the tiling hierarchy. Thereby, we assume the subsequent stencils are executed by nested stencil groups that represent the full tiling hierarchy. Given m tiling hierarchy levels and n stencils, up to m tiling hierarchies can be split between each pair of neighboring stencils. Overall, this means there are $\mathcal{O}(m^n)$ ways to split the stencil program. Given the set of stencil program implementation variants I and the functions $t(x)$ and $m^l(x)$ that estimate the execution time and the maximal storage requirement at the level l of the tiling hierarchy, respectively, we define the following optimization problem:

$$\begin{aligned} \underset{x \in I}{\text{minimize}} \quad & t(x) \\ \text{subject to} \quad & m^l(x) \leq M^l \qquad l = 1, \ldots, m \end{aligned}$$

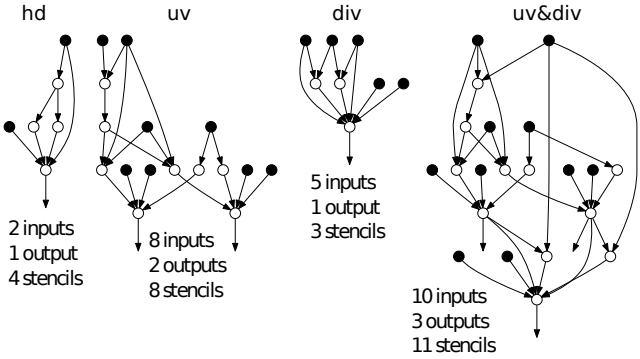


FIGURE 2.7: Example kernel stencil dependency graphs

We can either solve the optimization problem using brute force search or employ our dynamic programming approach reducing the search space from $\mathcal{O}(m^n)$ to $\mathcal{O}(mn^4)$ elements. We can apply dynamic programming as the problem has optimal substructure. In particular, we compute for each tiling hierarchy level an n^2 matrix that contains the optimal stencil group executing a continuous subset of the stencil program. Thereby, one matrix dimension corresponds to the start index and the other matrix dimension to the stop index of the subset. We compute a matrix entry using a second dynamic programming algorithm[1] that constructs the optimal stencil group using a combination of the previously computed optimal child stencil groups. More precisely, we compute the optimum for a given start and stop index either using the optimal child stencil group containing all stencils or using a child stencil group containing all stencils from an intermediate index to the stop index plus the recursively computed optimum from the start index to the intermediate index. By increasing the intermediate index step-by-step and storing partial solutions, we compute a single entry of our n^2 matrix using $\mathcal{O}(n^2)$ steps.

2.3 EVALUATION

We evaluated our framework using three example kernels from the COSMO atmospheric model. In addition to the horizontal diffusion kernel "hd" introduced in Section 2.1.2, we use two kernels that are part of the most

1 Our nested dynamic programming step is not guaranteed to find the optimal solution. For all four example kernels discussed in Section 2.3, exhaustive search based tests confirmed the optimality of the dynamic programming results for several stencil evaluation orders.

Hierarchy	Vertical (V)	Memory (M)
1	26 GB/s	∞
2	768 GB/s	512 KB

TABLE 2.3: Intel Core i5-3330

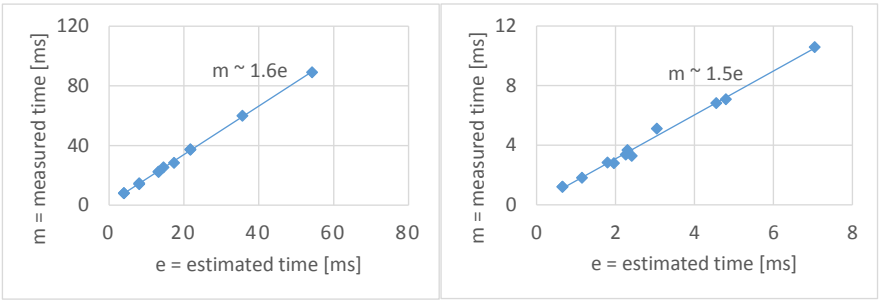
Hierarchy	Vertical (V)	Lateral (L)	Memory (M)
1	208 GB/s	-	∞
2	208 GB/s	-	∞
3	∞	1174 GB/s	4096 Registers
4	∞	∞	32 Registers

TABLE 2.4: Nvidia Tesla K20c

time-consuming component in COSMO, the sound wave forward integration. More precisely, the "uv" kernel updates the horizontal wind velocity components by computing the horizontal pressure gradient, whereas the "div" kernel computes the divergence of the three-dimensional wind field. Figure 2.7 illustrates all kernels used during the evaluation including a combination of the "uv" and "div" kernels.

We perform our experiments using adapted standalone kernels: 1) we replace divisions by multiplications to increase the numerical stability on random input data and 2) we replace one-dimensional constant fields by scalar constants as our framework does only support n-dimensional fields. We implement for each kernel three different variants: 1) "no fusion" refers to a naive implementation without loop fusion, 2) "hand-tuned" refers to a manually tuned implementation as used in production by COSMO, and 3) "optimized" refers to an automatically tuned version using MODESTO. All kernel variants are written using STELLA and therefore are parallel and employ tiling. Similar to the production configuration, we run our experiments using a $(256, 256, 64)$ point domain that provides sufficient parallelism to fully utilize the hardware.

We measure the performance of our example kernels using two target architectures: 1) an Intel Core i5-3330 CPU with a dual channel DDR3-1600 memory interface and 2) a Nvidia Tesla K20c GPU. Table 2.3 and Table 2.4 define the machine model of the target architectures for the STELLA tiling



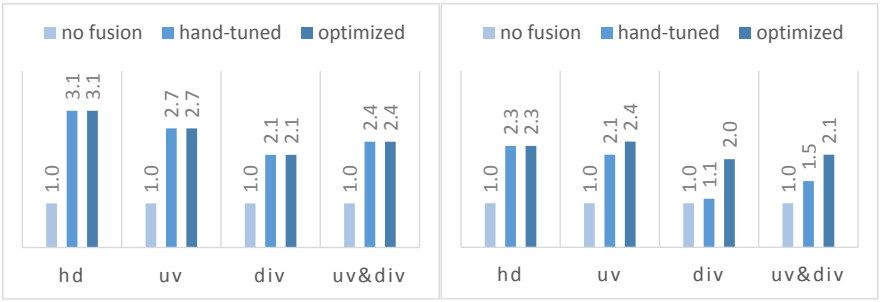
(a) CPU (b) GPU

FIGURE 2.8: Comparison of measured and estimated execution time

hierarchy discussed in Section 2.2.1. We thereby use the peak bandwidth of the individual memory hierarchy levels except for the register file and the shared memory used to buffer lateral communication. Since every lateral communication triggers a write and a read operation, we divide the peak bandwidth of these memories by two. We also underestimate the capacity of the GPU register file since it is not uniquely used to buffer lateral communication. We finally set the peak performance C of the target architectures to 48 Gflops and 585 Gflops, respectively (without fused multiply-add).

To evaluate the accuracy of our performance model, we compare the measured execution time of our example kernels to the modeled execution time. Figure 2.8 shows the accuracy of the model for both target architectures. Using linear regression, we fit trend lines that show a close correlation of modeled and measured performance. Hence, the relative performance of modeled and measured execution times for different kernels are in accordance, which is of key importance for our approach. However, we consistently overestimate the absolute performance as the kernels can not leverage the peak performance of both target architectures. Our performance model shows that our kernels are heavily memory bandwidth limited. Consequently, the correlation factors of 1.5x respectively 1.6x can be attributed to the fact that the kernels attain only a fraction of the peak main memory bandwidth.

Figure 2.9 shows the speedup of hand-tuned and automatically tuned implementation variants for both target architectures. As discussed in Section 2.2.2, MODESTO optimizes topological order and stencil fusion. Overall, MODESTO achieves the same or better performance compared to



(a) CPU (b) GPU

FIGURE 2.9: Speedup of hand-tuned and optimized kernels

the hand-tuned kernels used by COSMO. Starting from a naive STELLA implementation, we are able to improve the performance by a factor 2.0x–3.1x. The first three experiments achieve optimal performance by fusing all stencils on the highest level of the tiling hierarchy. In contrast, for the last experiment fusing all stencils exceeds the memory capacity. Hence, the optimization splits the stencils in two separate groups. To verify this decision, we implemented an additional variant of the last experiment that fuses all stencils. On CPU and GPU fusing all stencils results in a 10% and 8% performance reduction, respectively.

2.4 RELATED WORK

Optimal and close-to-optimal stencil arrangements have been investigated for several decades. Many approaches rely on empirical methods to derive efficient implementations. Datta et al. [50] optimize an example stencil for a wide range of hardware architectures using autotuning. Patus [22] is a DSL autotuning framework for single stencil computations on multi-core CPUs and single GPUs. Zhang et al. [51] present an iterative compilation approach for single stencil computations on single and multi GPU systems which focuses on deriving optimal block sizes.

Overtile [9] is a DSL code generator for iterative stencils that uses overlap tiling to generate efficient GPU code also relying on iterative compilation. There is also a cache-oblivious tiling strategy for iterative stencil computations [45] for which the number of expected cache misses has been analytically computed and empirically evaluated for single CPU systems and one caching level.

For stencil graphs, there is Halide [12], a DSL based approach focused on image processing. Halide uses again compilation based autotuning to choose stencil program implementation variants considering a set of tiling strategies and further optimizations. PolyMage [13] is an image processing DSL that guides the optimization using a model-driven heuristic. Basu et al. [44] perform loop fusion, overlapped tiling and wave front execution for optimizing a geometric multigrid stencil graph. They do not consider hierarchical tiling and do not use any analytical model. Olschanowsky et al. [46] optimize an iterative, but multi-kernel stencil computation resulting from solving partial differential equations and study different inter-loop optimizations using empirically evaluation on multi-core CPUs.

There has also been work that discusses analytical performance models. There is work not limited to stencil computations that provides lower bounds for tile sizes selection [52]. Renganarayana et al. [53] use geometric optimization to model tiling and related problems on one and multiple levels and to derive optimal tile sizes. Zhou et al. [8] present work on hierarchical overlapped tiling and optimize OpenCL programs for multi-core CPUs. They provide basic performance models for the number of stencils to fuse into one tile focusing on (possibly unrolled) kernels that process only one stencil repeatedly and do not consider varying tiling and fusion strategies. Finally, Wahib et al. [35] take arbitrary stencil graphs from larger scientific applications and present an analytical performance model for choosing an optimal execution strategy. Even though closely related, they limit themselves to kernel fusion using computation on-the-fly only considering shared memory and apply their work on NVIDIA GPUs only.

2.5 SUMMARY OF THE APPROACH

With MODESTO we have presented an approach for modeling and automatically selecting efficient implementation strategies for stencil programs. Focusing not only on single, possibly iterative applications of stencils, but on directed acyclic graphs of stencils we consider the effects of three different tiling strategies in combination with different fusion choices, all applied on possibly multiple hierarchy levels. We model the effects of these implementation strategies on the use of both lateral and vertical memory bandwidth, and estimate the cost of possibly redundant computation by using a analytical model that allows to predict the amount of data transfer and computation for a given stencil program implementation variant. In combination with a given CPU or GPU model we estimate the relative

performance of the different implementation variants and show using a combination of exhaustive search and dynamic programming how to choose the best implementation variant.

We evaluated MODESTO by means of the STELLA stencil library that implements different stencil program transformations for CPU and GPU architectures. In particular, we successfully model the tiling hierarchy of STELLA and automatically tune kernels of the COSMO atmospheric model. We thereby achieve speedups of 2.0–3.1x against naive and speedups of 1.0–1.8x against expert-tuned implementation variants.

3

A LEARNED PERFORMANCE MODEL

The cost of data movement in terms of energy and time has long exceeded the cost of computation. Thus, data locality recently became the most important optimization target for performance engineers [7]. Today, most programmers either rely on the compilation toolchain or manually optimize data locality by tiling and fusing loops. Manual loop optimizations are tedious and require a high porting effort to exploit different architectures efficiently because tiling and fusion parameters need to be adjusted for each target system. Various frameworks such as Halide [12] and Polymage [13] focus their tuning on this parameter selection, but they either apply heuristics or optimize tiling and fusion separately to control the exponential search space. However, fusion and tiling are inherently linked—for optimizing one, one needs to assume a specific configuration for the other. For example, the optimal tile size depends on the memory footprint of the loop, which changes with fusion. This missing *modularity* of the problem *requires* us to consider tiling and fusion in tandem.

Stencil computations on regular grids are ubiquitous in scientific computing applications such as climate modeling [16], seismic imaging [42], and electromagnetic simulations [43]. In this work, we use the COSMO atmospheric model [16], which is used in operational weather forecasting in most of Europe [17] as well as in large-scale climate modeling [18], as a motivating example. The 300,000 lines of code contain more than 16,000 loops, most of which implement single stencils. These stencils logically form complex producer-consumer relationships, called *stencil programs* [1]. We select three representative COSMO stencil programs to evaluate the effectiveness of our approach. Due to the very low arithmetic intensity of every single stencil, tiling and fusion are crucial for achieving good performance for stencil programs.

We show an example in Figure 3.1—COSMO's fastwaves stencil program which implements parts of the sound wave forward integration. The directed graphs show the data-flow (edges) between the stencils (nodes) of the fastwaves program. Our optimization framework, ABSINTHE, uses an automatically learned performance model to guide the program optimization. The figure plots the model prediction versus the measured execution

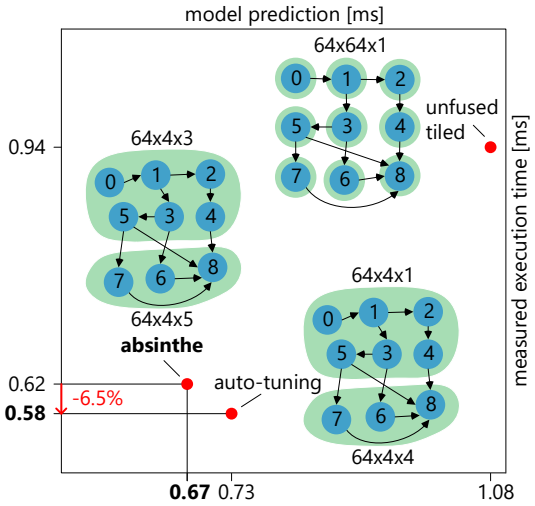


FIGURE 3.1: ABSINTHE optimization example

time for the tile size (annotated) and fusion (shaded shapes) choices of ABSINTHE compared to *auto-tuning* and an *unfused tiled* implementation.

ABSINTHE consists of three main pieces: (1) a model learner, (2) an optimizer, and (3) a code generator. The *model learner* generates a performance model specific to each target architecture. The *optimizer* derives an integer linear program encoding the structure of the stencil program and the performance model to tune tiling and fusion together. The *code generator* then emits an implementation with the optimal tiling and fusion parameters returned by the integer linear programming solver. In this way, ABSINTHE combines automated model learning with integer linear programming to control the exponential search space and to automatically find the best configuration for each target architecture.

In summary, we make the following key contributions:

- A linear formulation of parametric tiling for bound tile sizes (assumed to be non-linear in general).
- A linear performance model that learns the target system characteristics and enables the use of integer linear programming to explore the search space.

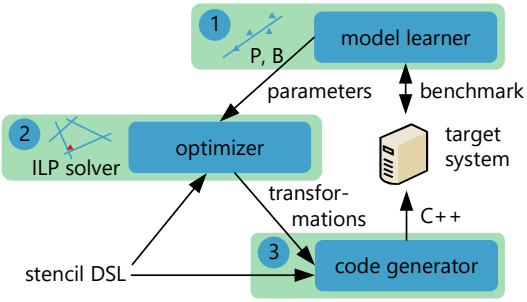


FIGURE 3.2: ABSINTHE architecture overview

- A single holistic optimization problem which applies the linear performance model to derive optimal fusion and tile size selection choices for stencil codes.

3.1 BACKGROUND

The execution of stencils in succession provides plenty of opportunities for data locality improvements.

3.1.1 *Architecture Overview*

ABSINTHE lowers stencil programs written in a high-level domain-specific language (DSL) to efficient C++ code. An automatically learned performance model drives the selection of target system-specific code transformations. Figure 3.2 shows the interplay of the ABSINTHE components.

The model learner (1) runs once for every target system to learn the model parameters. The optimizer (2) combines the model parameters with the memory access patterns of the stencil program to instantiate a target-specific performance model. An integer linear programming (ILP) solver searches the optimal data-locality transformations with respect to the performance model. The code generator (2) applies the optimal data-locality transformations to the high-level stencil program representation and generates tuned C++ code.

ABSINTHE targets three-dimensional stencil programs and optimizes them to utilize all processors of the target system, assuming exclusive system access. Our implementation has the following limitations: 1) we

```
1  for(int x=xbeg; x<=xend; ++x)
2   for(int y=ybeg; y<=yend; ++y)
3    for(int z=zbeg; z<=zend; ++z)
4     s0(x,y,z) = 0.5 * (i0(x+1,y,z) + i0(x,y,z));
5  for(int x=xbeg; x<=xend; ++x)
6   for(int y=ybeg; y<=yend; ++y)
7    for(int z=zbeg; z<=zend; ++z)
8     s1(x,y,z) = i1(x,y,z) * (s0(x,y+1,z) - s0(x,y-1,z));
```

FIGURE 3.3: Example stencil sequence with length $N = 2$

support only three-dimensional arrays, 2) we do not optimize the boundary conditions, and 3) we tile the codes only for one memory hierarchy level.

3.1.2 *Stencil Sequences*

A *stencil* is an element-wise computation with a position independent access pattern. Every stencil evaluation accesses the input arrays at fixed offsets relative to the updated output array element. We assume that every stencil writes a single array. We apply stencils to all array elements except for a constant width halo at the array boundary which prevents out-of-bounds accesses.

A *stencil sequence* is a program formed of several subsequent stencil applications. Figure 3.3 shows an example stencil sequence with length $N = 2$. The short example sequence allows us to illustrate our approach with less complexity compared to the fastwaves kernel introduced in Figure 3.1.

3.1.3 *Data-Locality Transformations*

ABSINTHE combines rectangular tiling with redundant computation at the tile boundaries to satisfy the data dependencies of fused stencils. This overlapped tiling [8] enables major performance improvements. The tuned fastwaves kernel shown by Figure 3.1 executes 1.5× faster compared to the unfused tiled implementation variant.

Loop tiling decomposes the domain into hyper-rectangular tiles of equal size. To increase the data-locality, we evaluate the stencil on the entire tile before proceeding with the next one. We thus introduce an additional outermost loop that iterates over all tiles. To support arbitrary domain sizes, we cut the tiles at the domain boundary.

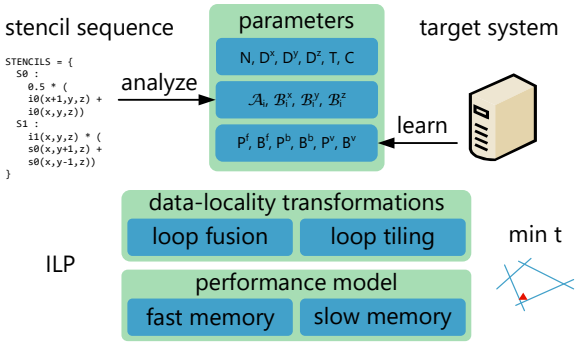


FIGURE 3.4: ABSINTHE ILP parameters and components

Loop fusion replaces the tile loops of consecutive stencils with a single tile loop that evaluates one stencil after another before proceeding with the next tile. After fusion, the data dependencies of producer-consumer stencils cross the tile boundaries. To enable the parallel tile execution, we extend the loop bounds of the producer stencils to perform redundant computation at the tile boundaries which satisfies all data dependencies locally.

The combination of *fusion and tiling* effectively increases the spatial and temporal locality for stencils with overlapping working sets. The *code generator* introduces one tile loop for every group of fused stencils and allocates temporary storage to buffer intra-tile data dependencies with minimal memory footprint.

3.2 MODELING

The *optimizer* automatically instantiates an integer linear program (ILP) to find good data-locality transformations. Figure 3.4 shows the main components of the ILP: the *parameters* component captures the stencil sequence and target system properties that provide the basis for the optimization, the *data-locality transformations* component defines the optimization variables that span the space of possible transformations, and the *performance model* component estimates the execution time for the selected code transformations. At optimization time, the ILP solver searches the code transformations with minimal estimated execution time.

We present the ILP for three-dimensional stencils, but the formulation generalizes to stencils with different dimensionality. If not mentioned otherwise, the variables are positive and integer-valued, while lowercase and

	constants
N	number of stencils in the stencil sequence
D^x, D^y, D^z	domain sizes
H^x, H^y, H^z	halo widths
T	number of processors
C	cache capacity
P^f, B^f	fast memory peel & body parameters
P^b, B^b	slow memory peel & body base parameters
P^v, B^v	slow memory peel & body variable parameters
	variables
g_i	group index
n_i^x, n_i^y, n_i^z	tile counts
p_i^f, b_i^f	fast memory peel & body cost
p_i^b, b_i^b	slow memory peel & body base cost
p_i^v, b_i^v	slow memory peel & body variable cost
$e_i^{x+}, e_i^{y+}, e_i^{z+}$	evaluation boundary widths (positive axis direction)
$e_i^{x-}, e_i^{y-}, e_i^{z-}$	evaluation boundary widths (negative axis direction)

TABLE 3.1: Important constants and variables

uppercase identifiers distinguish optimization variables and constants, respectively. Table 3.1 lists important constants and variables.

3.2.1 *Stencil Sequences*

The *optimizer* requires an analysis of the stencil access patterns to instantiate the ILP shown by Figure 3.4. The access patterns provide the basis to compute the data-flow and to estimate the performance of the stencil sequence.

We use positive indexes to number the stencils in execution order and negative indexes to identify the input arrays. For example, the indexes $[0, 1]$ refer to the stencils $[s0, s1]$ and the indexes $[-1, -2]$ to the input arrays $[i0, i1]$ of the example stencil sequence shown by Figure 3.3. The stencil indexes also map one-to-one to the output arrays since every stencil writes precisely one output. The resulting index space thus uniquely identifies the input and output arrays of the stencil sequence.

To specify the data access patterns, we define for every stencil i the access set $\mathcal{A}_i$ holding (index, offset) tuples that define the array and the three-dimensional relative offset of every input element access. The access sets

$$\mathcal{A}_0 = \{(-1, (1, 0, 0)), (-1, (0, 0, 0))\},$$
$$\mathcal{A}_1 = \{(-2, (0, 0, 0)), (0, (0, 1, 0)), (0, (0, -1, 0))\}$$

include all accesses of the example stencils. We also compute minimal bounding boxes that contain all access offsets. To represent the bounding boxes, we define for every stencil i and dimension d the bounds set $\mathcal{B}_i^d$ holding (index, range) tuples that specify the array and the minimal and maximal access offset along the dimension. The bounds sets

$$\mathcal{B}_0^x = \{(-1, (0, 1))\}, \quad \mathcal{B}_1^y = \{(-2, (0, 0)), (0, (-1, 1))\}$$

contain all accesses of the example stencils along the selected dimensions.

To execute the stencil sequence, we define for every dimension d the constant domain size D^d and the constant halo width H^d along the dimension. The domain sizes determine the stencil loop bounds, while the halo sizes together with the domain sizes specify the array allocation size. For example, we may execute the example stencils on the domain

$$D^x = 64, \quad D^y = 64, \quad D^z = 60$$

and select the halo widths

$$H^x = 1, \quad H^y = 1, \quad H^z = 0$$

to accommodate the transitive stencil access offsets, which results in the array allocation size $66 \times 66 \times 60$.

3.2.2 *Data-Locality Transformations*

The *optimizer* also defines the optimization variables that span the space of possible data-locality transformations and introduces constraints to exclude solutions that suffer from load imbalance or exceed the cache capacity.

To model *loop tiling*, we select for every stencil $i \in [0, N)$ and every dimension d the tile count n_i^d along the dimension from the range $\left[1, D^d\right]$. For example, the tile counts

$$n_0^x = 2, \quad n_0^y = 2, \quad n_0^z = 2$$

split the domain of the first stencil in the example stencil sequence into two tiles along every dimension.

To model *loop fusion*, we select for every stencil $i \in [0, N)$ the group index g_i and fuse stencils with the same group index. We set the group index of the first stencil to zero and increment the group index with every additional group along the stencil sequence. For every stencil, we thus have the choice to retain or increment the group index of the preceding stencil, which spans an exponential search space in the number of stencils. For example, the group index tuples

$$(g_0, g_1) \in \{(0, 0), (0, 1)\}$$

enumerate all possible group assignments for the example stencil sequence. The group indexes $g_0 = 0, g_1 = 0$ assign the stencils to the same group to model fusion while the group indexes $g_0 = 0, g_1 = 1$ assign the stencils to different groups that execute consecutively. To quantify the *redundant computation*, we also extend for every stencil $i \in [0, N)$ and for every dimension d the tile size with the evaluation boundary widths e_i^{d+} and e_i^{d-} along both directions. For example, the evaluation boundary widths

$$e_0^{y+} = 1, \quad e_0^{y-} = 1$$

extend the tile size of the first stencil to satisfy all data dependencies of the example stencil sequence locally. We define the tile sizes for every stencil,

but the stencils of each group share the same tile loop and tile size. We thus enforce tile count equality for succeeding stencils with the same group index.

To guarantee *data-locality*, we exclude tile sizes that exceed the cache capacity C (L2 cache). To estimate the cache utilization, we multiply for every stencil group the tile size with the number of accessed arrays. This approximation optimistically models a fully associative cache with a least recently used cache replacement policy and does not consider the accesses at the tile boundaries. We thus enforce the cache utilization for a single tile to be lower than one-third of the cache capacity. This choice compensates for our optimistic cache modeling and ensures that not only the current but also the next and the previous tile executed by the same processor mostly fit the cache. As a result, the data-locality improves since the overlapping boundaries of consecutive tiles stay in cache.

To guarantee *parallel efficiency*, we enforce a total number of tiles within 5% of an integer multiple of the number of processors T and for every dimension a tile count within 2% of an integer multiple of the domain size.

3.2.3 *Performance Model*

The *optimizer* finally instantiates the performance model based on the stencil sequence and target system parameters shown by Figure 3.4.

The performance model distinguishes two cost components: (1) the *peel cost* models the latency and (2) the *body cost* models the throughput of the innermost loop executions. In other words, the *peel cost* accounts for loop startup overheads – examples are the over fetch at the loop boundaries or the execution of scalar peel loops – while the *body cost* models the steady-state of the loop execution. For both components, we model the memory accesses for two memory hierarchy levels: (1) the *fast memory* (L2 cache) and (2) the *slow memory* (L3 cache or DDR memory). For every data element, we assume the *slow memory* handles the first and the *fast memory* all subsequent accesses during the tile execution. To estimate the execution time, the performance model multiplies the number of memory accesses with the learned model parameters.

The loop startup overheads make long tiles along the innermost loop dimension more efficient. To model this effect, we distinguish the *peel cost* proportional to the number of innermost loop executions (peel domain) and the *body cost* proportional to the number of innermost loop iterations (body domain). This separation allows us to assign a higher cost to memory

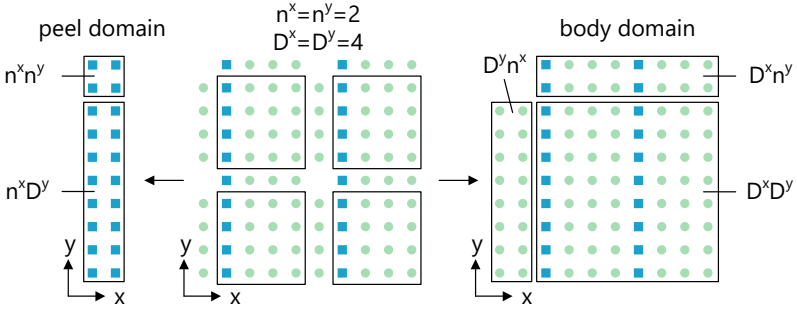


FIGURE 3.5: Illustration of the peel and body domain computation for domain size 8×8 split into 2×2 tiles with boundary width one along the positive axis directions.

accesses executed during the loop startup. The two cost components and the goal to employ efficient integer linear programming solvers result in linear cost functions of the form $Px + By$ that sum the peel cost Px and the body cost By. The variables x and y denote the number of memory accesses for the peel and body domains, respectively. The learned model parameters P and B convert the memory accesses to execution times. Section 3.2.4 details how the *model learner* determines the model parameters.

The performance model combines multiple cost functions to estimate the stencil sequence execution time. To define the cost functions, we next introduce the peel and body functions that compute the weighted size of the peel and body domains, respectively. Figure 3.5 shows the computation of the peel domain (left) and the body domain (right) for a simplified two-dimensional domain (middle) with all weights set to one. The peel domain counts the blue points (squares) while the body domain counts all points (squares and circles). The peel and body functions extend this computation with additional terms and factors to model our three-dimensional domain and parametric weights.

PEEL FUNCTION The *peel cost* is proportional to the number of innermost loop executions. Without loss of generality, we assume the innermost loops execute along the x-dimension, which means the number of innermost loop executions corresponds to the size of the tiles projected to the yz-plane.

The product D^yD^z of the domain sizes is equal to the sum of the tile domains and the products $D^zn_i^y$ and $D^yn_i^z$ of the tile counts with the perpendicular domain size approximate the tile boundaries. To sum the

tiles along the innermost loop dimension, we multiply the terms with the tile count along the x-dimension. This approximation is exact except for the tile corners. To evaluate the peel cost, we define for every stencil $i \in [0, N)$ the peel function

$$f_i^p(w, w^y, w^z) = n_i^x(D^y D^z w + D^z n_i^y w^y + D^y n_i^z w^z)$$

which scales the inner domain and the boundary terms with the inner weight w and the boundary weights w^y and w^z, respectively. For example, we set the inner weight to one and the boundary weights to the evaluation boundary widths to count the innermost loop executions.

BODY FUNCTION The *body cost* is proportional to the number of innermost loop iterations scaled with cost function-specific weights. The number of innermost loop iterations is equal to the sum of the tile volumes. To compute the volume of the overlapping tiles, we add the product $D^x D^y D^z$ of the domain sizes to the tile counts multiplied with the perpendicular domain sizes. This approximation again includes the tile domains and the tile boundaries without the tile corners. To evaluate the body cost, we define for every stencil $i \in [0, N)$ the body function

$$\begin{aligned} f_i^b(w, w^x, w^y, w^z) = {} & D^x D^y D^z w + D^y D^z n_i^x w^x + \\ & D^x D^z n_i^y w^y + D^x D^y n_i^z w^z \end{aligned}$$

which scales the inner domain and the boundary terms with the inner weight w and the boundary weights w^x, w^y, and w^z, respectively. For example, we set the inner weight to one and the boundary weights to the evaluation boundary widths to count the innermost loop iterations.

The peel and body functions next allow us to define the cost functions for the two memory hierarchy levels.

FAST MEMORY The fast memory model counts the memory accesses to estimate the stencil execution time. We assume that every evaluation of the stencil i loads the entire access set $\mathcal{A}_i$ and stores the result. The stencil i thus performs $1 + |\mathcal{A}_i|$ memory accesses per evaluation. To count the memory accesses, we set for every stencil $i \in [0, N)$ the weight

$$c_i = 1 + |\mathcal{A}_i|$$

to the number of memory accesses per stencil evaluation and for every dimension d the boundary weight

$$c_i^d = (1 + |\mathcal{A}_i|)(e_i^{d-} + e_i^{d+})$$

to the number of memory accesses per stencil evaluation scaled with the evaluation boundary widths. The multiplication reflects that the stencils are evaluated at every evaluation boundary line. We then set for every stencil $i \in [0, N)$ the peel cost p_i^f and the body cost b_i^f of the *fast memory* model to the products

$$p_i^f = P^f f_i^p(c_i, c_i^y, c_i^z), \quad b_i^f = B^f f_i^b(c_i, c_i^x, c_i^y, c_i^z)$$

which evaluate the peel and body functions to obtain the number of memory accesses for the peel and body domains, respectively. The learned model parameters P^f and B^f convert the memory accesses to execution times.

SLOW MEMORY The slow memory model determines the communication volume to estimate the execution time. We observe that the memory throughput improves with the number of parallel access streams. To model this behavior, we sum two cost functions that estimate the *base cost* and the *variable cost* with respect to the number of access streams. We assume that every stencil group loads and stores an array only once. Repeated accesses of the same array hit the fast memory and are not relevant for the slow memory model.

We compute the slow memory loads and stores based on the group indexes. A stencil only loads an array from slow memory if the group index of the stencil that accessed the array last differs. Otherwise, the array was already loaded to the fast memory. A stencil only stores an array to slow memory if the group index of the last stencil that accesses the array differs. Otherwise, the array is not used outside of the stencil group, and storing to slow memory is not necessary.

To estimate the *base cost*, we set for every stencil $i \in [0, N)$ the weight m_i to one if the stencil loads or stores at least one array and to zero otherwise. We also set for every dimension d the boundary weight

$$m_i^d = m_i(e_i^{d-} + e_i^{d+})$$

to the weight times the evaluation boundary widths. We then set for every stencil $i \in [0, N)$ the peel cost p_i^b and the body cost b_i^b of the *base cost* to the products

$$p_i^b = P^b f_i^p(m_i, m_i^y, m_i^z), \quad b_i^b = B^b f_i^b(m_i, m_i^x, m_i^y, m_i^z)$$

which evaluate the peel and body functions to obtain the number of stencil evaluations that access at least one array for the peel and body domains,

respectively. The learned model parameters P^b and B^b convert the stencil evaluations to execution times.

To estimate the *variable cost*, we set for every stencil $i \in [0, N)$ the weight s_i to the number of accessed arrays and for every dimension d the boundary weights s_i^d to the sum of the array access boundary widths along the dimension. We consider only arrays and boundary lines that have not been accessed by a preceding stencil of the same stencil group. To compute access boundary widths, we extend for every data dependency $(j, (B^-, B^+)) \in \mathcal{B}_i^d$ the evaluation boundary widths with the access bounds B^- and B^+. We then set for every stencil $i \in [0, N)$ the peel cost p_i^v and the body cost b_i^v of the *variable cost* to the products

$$p_i^v = P^v f_i^p(s_i, s_i^y, s_i^z), \quad b_i^v = B^v f_i^b(s_i, s_i^x, s_i^y, s_i^z)$$

which evaluate the peel and body functions to obtain the number of access streams for the peel and body domains, respectively. The learned model parameters P^v and B^v convert the access streams to execution times.

The *slow memory* model finally sums the *base cost* and the *variable cost* to estimate the execution time.

To estimate the overall stencil execution time, we assume that the *fast memory* and the *slow memory* accesses overlap. We thus compute for every stencil the maximum *peel cost* and the maximum *body cost* of the two memory hierarchy levels. The sum

$$\sum_{i=0}^{N-1} \max(p_i^f, p_i^b + p_i^v) + \max(b_i^f, b_i^b + b_i^v)$$

accumulates the individual stencil execution times to obtain the execution time of the entire stencil sequence. The term

$$\sum_{i=0}^{N-1} 2 \cdot 3(B^b + B^v) n_i^x n_i^y n_i^z$$

emulates the slow memory access cost to load the two precomputed tile loop bounds for all three dimensions to account for the tile execution overheads. We include this term in the estimated execution time to favor implementation variants with fewer tiles. Together, the estimated execution time and the tile execution overheads define the objective function of the integer linear program.

3.2.4 *Learning the Performance Model*

The *model learner* adapts the performance model parameters to the performance characteristics of the target system. To learn the parameters, we

implemented training stencils that either stress the slow or the fast memory and measure their execution time for different tile sizes. We then compute the model parameters using least absolute deviations (LAD) [54] regression, which compared to least squares regression has better outlier robustness.

As the performance depends on the tile shape, we benchmark the training stencils with tile sizes ranging from 10 to 80 elements along the x-dimension and from 1 to 55 elements along the other dimensions. We exclude tiles with a volume below 500 or above 2000 elements to ensure that the tiles fit the fast memory (L2 cache). In total, we run 103 tile size configurations.

When learning the *fast memory* model, the fast memory accesses have to dominate the execution times of the training stencils. We used three training stencils that access 12, 16, and 20 array positions. We always connect nine identical stencils that access the same input array to one training sequence. The repeated accesses of the same input array guarantee that the fast memory accesses dominate the execution time.

We benchmark the three training sequences for all tile size configurations. For every run $r \in [0, R)$, we collect the measured execution time t_r and compute the number of fast memory accesses x_r and y_r for the peel and body domain, respectively. The LAD regression

$$(P^f, B^f) = \underset{(P,B)\in\mathbb{R}^2}{\operatorname{argmin}} \sum_{r\in[0,R)} |(Px_r + By_r) - t_r|$$

then selects the *fast memory* model parameters P^f and B^f that minimize the L1-norm of the prediction error.

When learning the *slow memory* model, the slow memory accesses have to dominate the execution times. We used nine training stencils that access 1, 2, or 3 input arrays with access boundary width 0, 1, or 2. The stencils access the input arrays at three diagonal offsets to avoid unnecessary fast memory accesses. We always connect nine identical stencils that access different input and output arrays to one training sequence. The many loaded and stored arrays guarantee that the slow memory accesses dominate the execution time.

We benchmark the nine training sequences for all tile size configurations. For every run $r \in [0, R)$, we collect the measured execution time t_r. To learn the *base cost*, we compute the number of stencil evaluations x_r and y_r that perform slow memory accesses for the peel and body domain, respectively.

To learn the *variable cost*, we compute the number of slow memory accesses u_r and v_r for the peel and body domain, respectively. The LAD regression

$$(P^b, B^b, P^v, B^v) = \underset{(P',B',P'',B'')\in\mathbb{R}^4}{\operatorname{argmin}} \sum_{r\in[0,R)} |(P'x_r + B'y_r + P''u_r + B''v_r) - t_r|$$

then selects the *slow memory* model parameters P^b, B^b, P^v, and B^v that minimize the L1-norm of the prediction error.

All training sequences are synthetic and differ from the application kernels tuned in Section 3.4.4.

3.3 OPTIMIZATION

The number of possible data-locality transformations defined in Section 3.2.2 makes the manual tuning of *stencil programs* difficult. To automate the process, we could exhaustively search for the optimal data-locality transformations according to the performance model introduced in Section 3.2.3. However, for stencil sequences of length N the search space contains $\mathcal{O}(2^N N D^x D^y D^z)$ implementation variants which decompose into 2^N fusion choices multiplied with up to N stencil groups and $D^x D^y D^z$ tile size choices. This large search space motivates advanced optimization methods.

To explore the search space, we rely on the well established mixed-integer linear programming (MILP) approach, which finds or approximates the optimal solution within some predefined objective function gap. The *optimizer* translates the performance model and the space of data-locality transformations to an MILP that defines the optimization problem. We next detail the automatic translation of the performance model to linear constraints.

3.3.1 *Linearizing Multiplications*

The performance model multiplies the tile count variables with other variables. Linear programs cannot directly express the product of two integer variables. An implementation trick [55] nevertheless allows us to multiply two variables x and y with known upper bounds X and Y.

We first observe that the product of the binary variable b and the variable x with known upper bound X translates to three constraints. The constraint $0 \leq p \leq x$ limits the product p to the range $[0, x]$, while the constraints

$$p - Xb \leq 0 \quad \text{and} \quad p - x - Xb \geq -X$$

force the product to zero if b is zero and to x otherwise.

To express the product of two variables x and y with the known upper bounds X and Y, we next encode the variable y with the sum

$$y = \sum_{i=0}^{\lfloor \log_2(Y) \rfloor} 2^i y_i$$

where the binary variables y_i represent the digits of y. Then the product $p = xy$ corresponds to the sum

$$p = \sum_{i=0}^{\lfloor \log_2(Y) \rfloor} 2^i x y_i$$

of the binary products xy_i scaled with the power of two associated with the respective digit. All binary products are translated to the constraints introduced before.

The *optimizer* implements the performance model by introducing binary representations for all tile count variables and lowers the products as shown above. This solution works since we know that for every dimension d the range $[1, D^d]$ limits the tile count variables.

3.3.2 *Modeling Stencil Groups*

The number of stencil groups is an optimization variable not known during the generation of the optimization problem. Allocating one variable per stencil group to store group properties such as the tile count is thus not possible. Instead, we model stencil group properties with the help of stencil specific variables. At optimization time, the group index variables of Section 3.2.2 allow us to compute stencil group properties and to assign them to all stencil specific variables of the group.

The group indexes increase monotonically along the stencil sequence. The constraint $g_0 = 0$ sets the group index of the first stencil to zero. To limit the remaining group indexes, we define for every stencil $i \in [0, N-1)$ the constraint

$$0 \leq g_{i+1} - g_i \leq 1,$$

which sets the group index difference of succeeding stencils to zero or one for fusion and no fusion, respectively.

```
1 STENCILS = {
2 "s0":"auto␣res␣=␣0.5*(i0(x+1,y,z)+i0(x,y,z));",
3 "s1":"auto␣res␣=␣i1(x,y,z)*(s0(x,y+1,z)+s0(x,y-1,z));"}
```

FIGURE 3.6: ABSINTHE version of the example stencil sequence

With the help of the group indexes, we define constraints that apply to the stencil groups. For example, the tile counts have to be equal within the stencil group. To enforce equality, we define for every stencil $i \in [0, N-1)$ and for every dimension d the constraints

$$n^d_{i+1} - n^d_i + D^d(g_{i+1} - g_i) \geq 0,$$
$$n^d_{i+1} - n^d_i - D^d(g_{i+1} - g_i) \leq 0$$

which limit the tile count difference to zero if the stencils have the same group index. Otherwise, the group index difference $g_{i+1} - g_i$ is positive since the indexes increase along the stencil sequence. Then the group index difference multiplied with the upper bound D^d for the tile count difference $n^d_{i+1} - n^d_i$ disables the constraints for all possible tile count assignments. The upper bound follows from the observation that the tile counts range from one to the domain size D^d.

The *optimizer* uses the group index variables to model the tile counts, the cache utilization, and the number of slow memory accesses.

3.4 EVALUATION

To validate our approach, we learn the performance model for three target systems and compare application kernels tuned with ABSINTHE to heuristically tuned, hand-tuned, and auto-tuned implementation variants.

3.4.1 *Setup & Methodology*

The target systems feature Xeon E5-2695 v4, Xeon Phi 7210, and Power8NVL sockets. We configure the Xeon Phi sockets with two NUMA domains, each of them with 32 processors and three DDR channels, and run the experiments on one of the two NUMA domains. We optimize the linear programs with CPLEX 12.6.3 and compile the generated C++ codes with GCC 5.3 on the Xeon and Xeon Phi systems and with GCC 5.4 on the Power system.

```
#pragma omp parallel for schedule(static)
for(int idx = 0; idx < 1 * 3 * 12; ++idx) {
    // views of the input and output arrays
    loop_info l = _tiles_group0[idx];
    array_view_3d i1(&__i1(l.xbeg, l.ybeg, l.zbeg));
    array_view_3d i0(&__i0(l.xbeg, l.ybeg, l.zbeg));
    array_view_3d s1(&__s1(l.xbeg, l.ybeg, l.zbeg));
    // stack allocated temporary arrays
    tarray0_3d ___s0;
    tarray0_view_3d s0(&___s0(HX, HY, HZ));

    { // apply s0 stencil
    int xbeg = _loops_s0[idx].xbeg;
    int xend = _loops_s0[idx].xend;
    int ybeg = _loops_s0[idx] ybeg;
    int yend = _loops_s0[idx].yend;
    int zbeg = _loops_s0[idx].zbeg;
    int zend = _loops_s0[idx].zend;
    for(int z = zbeg; z < zend; ++z)
    for(int y = ybeg; y < yend; ++y)
    #pragma omp simd
    for(int x = xbeg; x < xend; ++x) {
      auto res = 0.5 (i0(x+1,y,z) + i0(x,y,z));
      s0(x,y,z) = res;
    }}

    { // apply s1 stencil
    int xbeg = _loops_s1[idx].xbeg;
    int xend = _loops_s1[idx].xend;
    int ybeg = _loops_s1[idx] ybeg;
    int yend = _loops_s1[idx].yend;
    int zbeg = _loops_s1[idx].zbeg;
    int zend = _loops_s1[idx].zend;
    for(int z = zbeg; z < zend; ++z)
    for(int y = ybeg; y < yend; ++y)
    #pragma omp simd
    for(int x = xbeg; x < xend; ++x) {
      auto res = i1(x,y,z) * (s0(x,y+1,z) + s0(x,y-1,z));
      s1(x,y,z) = res;
    }}
}
```

FIGURE 3.7: Optimized code for the example stencil sequence

To perform the experiments, we set the domain size to $64 \times 64 \times 60$ elements with $3 \times 3 \times 3$ halo elements similar to the COSMO [16] production configuration. All experiments are performed using double-precision floating-point numbers.

We set the number of processors to the available cores $T = 18$, $T = 32$, and $T = 10$ for the Xeon, Xeon Phi, and Power systems, respectively.

To measure the execution time, we repeat every experiment 64 times and discard the first 16 measurements to warmup the memory hierarchy. Before every run, except when learning the fast memory model, we flush the L1 and L2 caches with dummy data. As we assume exclusive system access, we run one thread per processor. We time only the stencil executions, which excludes the initialization logic and the boundary conditions. All plots show median values and nonparametric 95% confidence intervals [56] to visualize the distribution of the measurements.

3.4.2 *Implementation*

ABSINTHE provides a high-level stencil DSL to implement stencil programs. Figure 3.6 and Figure 3.7 show the DSL version and the generated code for the example stencil sequence introduced in Section 3.1.2, respectively. ABSINTHE parses the DSL to extract the accesses patterns. Based on this analysis, the *optimizer* derives the integer linear program and determines the optimal solution using the CPLEX solver [38]. After the optimization, the *code generator* emits C++ code that implements the fusion and tile size choices of the optimal solution.

The *code generator* performs overlapped tiling [8] with one tiling hierarchy level and periodic boundary conditions. In addition to the stencil sequence, we also generate the boilerplate necessary to execute, benchmark, and verify the stencil sequence. The verification compares the results of the parallel implementation to naive sequential code. The code generator utilizes the Jinja2 template engine to specialize a generic stencil sequence template with the program-specific logic.

3.4.3 *Learning the Target Systems*

ABSINTHE learns the performance model parameters once for every target system and then tunes all stencil programs using the same parameter set. Section 3.2.4 discusses the performance model learning. We next evaluate the quality of the learned model parameters.

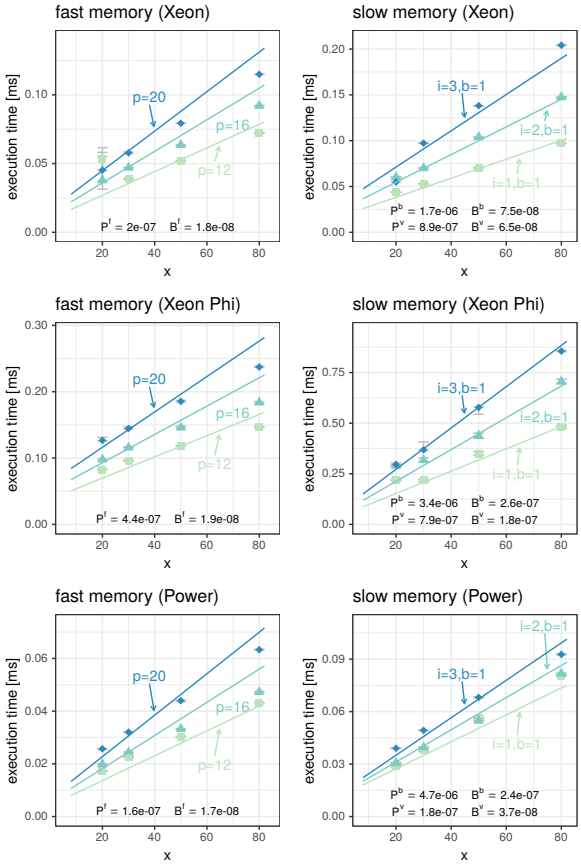


FIGURE 3.8: Measured (polygons) and estimated (lines) execution times for the fast memory (p=positions) and slow memory (i=input arrays and b=boundary width) training stencils and variable tile sizes along the x-dimension.

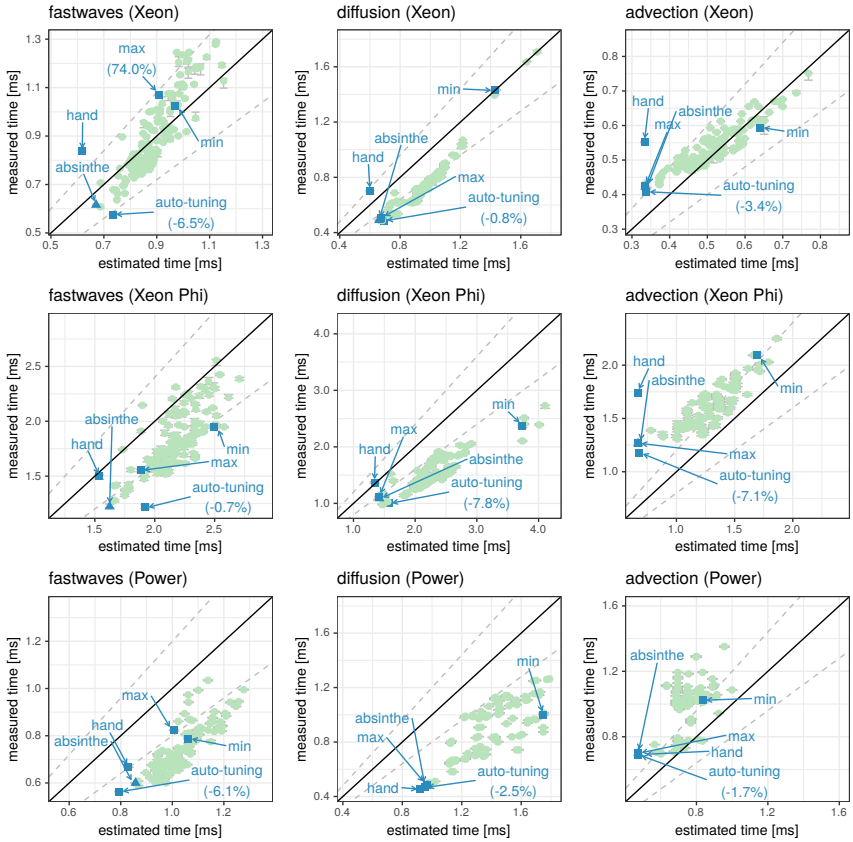


FIGURE 3.9: Measured and estimated execution times for the optimal (triangle), selected (squares), and random (dots) implementation variants of the fastwaves ($N = 9$), diffusion ($N = 16$), and advection ($N = 8$) kernels.

To improve the noise robustness, we use all 48 measurements per experiment when learning the model parameters using LAD regression [54]. We use the median of the repeated measurements when computing the R^2 values.

Figure 3.8 compares for the tile sizes $5 \times 5 \times x$ the measured execution times of the training stencils to the learned *fast memory* and *slow memory* models. We observe that for the shown tile sizes, the execution times increase almost linearly with the tile size along the x-dimension with model predictions close to the measured execution times of the training stencils. The annotations mark the different training stencils. For example, the annotation $p = 12$ refers to the training stencil that accesses twelve positions, and the annotation $i = 3, b = 1$ refers to the training stencil that accesses three input arrays with boundary width one.

The R^2 values of 0.87, 0.95, and 0.94 for the *fast memory* model and of 0.96, 0.96, and 0.90 for the *slow memory* model confirm the quality of the learned model parameters for the Xeon, Xeon Phi, and Power systems, respectively.

3.4.4 *Tuning the Application Kernels*

Existing benchmark suites such as PolyBench [57] often contain stencil programs that iterate only one stencil instead of multiple different stencils. To evaluate the quality of our fusion and tile size selection choices, we thus implement three stencil sequences from the COSMO atmospheric model [16]. These real-world benchmark kernels contain one-, two-, and three-dimensional stencils from first to fifth order. The fastwaves kernel consists of nine stencils that compute the pressure gradient, update the horizontal wind speeds, and compute the wind divergence. The diffusion kernel consists of sixteen stencils that update the pressure and the wind speeds. The advection kernel consists of eight stencils that transport the horizontal wind speeds. The two-dimensional advection and diffusion stencils access only neighbor elements in the horizontal xy-plane, while the fastwaves stencils perform three-dimensional accesses.

To perform the experiments, we adapt the COSMO stencils to match the current implementation of our *code generator*, which supports only three-dimensional arrays and periodic boundary conditions. We thus replace the original boundary conditions and remove accesses to lower-dimensional arrays.

Figure 3.9 shows the performance of ABSINTHE for all application kernels and target systems. We compare the measured and estimated execution

times of the optimal solution found by ABSINTHE to selected and random implementation variants with group index and tile size constraints. Data points close to the diagonal imply good model prediction. The dashed lines delimit the region with 20% prediction error. The timings include the stencil computation without boundary conditions.

Additionally, we add the following selected implementation variants: the *min* and *max* heuristics combine minimal and maximal fusion with the ABSINTHE tile size selection, the *hand* approach reproduces the hand-tuned fusion and tile size choices of the COSMO production code, and the *auto-tuning* approach combines tile size auto-tuning with the ABSINTHE fusion choices. As the *hand* and *auto-tuning* variants may violate the cache size or load imbalance constraints, their estimated execution times are possibly invalid.

The optimal solutions for the three application kernels contain at most four stencil groups. To sample random implementation variants, we select 20 random group index assignments with at most four groups and repeat the optimization with constraints that fix the group indexes. To examine different tile sizes, we also introduce tile size constraints that enforce smaller or larger tiles along one dimension. In total, we sample 60 random implementation variants.

The auto-tuning exhaustively searches for every stencil group the tile sizes 1, 2, 4, 12, 30, and 60 in the z-dimension and the powers of two in the xy-plane. The tuning of isolated stencil groups does not consider the cache reuse of consecutive stencil groups. However, the approach circumvents the joint evaluation of all stencil group tile size combinations. We use the ABSINTHE fusion strategy to avoid auto-tuning the exponential fusion search space.

FASTWAVES The optimal solution for all target systems splits the fastwaves kernel into two groups (Figure 3.1 shows the optimal solution for the Xeon system). The tile shapes reflect the three-dimensional access patterns detailed in Figure 3.10.

DIFFUSION The optimal solutions split the diffusion kernel into two, one, and four groups of equal sizes with tile size $64 \times 13 \times 1$, $64 \times 16 \times 1$, and $64 \times 32 \times 1$ for the Xeon, Xeon Phi, and Power systems, respectively. These choices reflect the two-dimensional access pattern of the stencils and the different L2 cache capacities. Most implementation variants perform better than expected. We attribute this bias to the peel cost of the slow memory

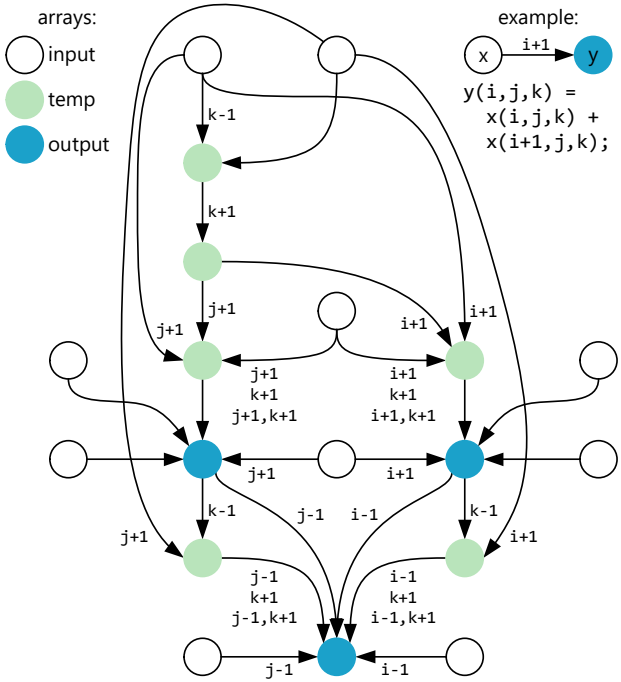


FIGURE 3.10: Data-flow graph of the fastwaves kernel. All edges are annotated with the non-center access offsets that the stencils read in addition to the center position (i,j,k).

model, which do not consider the cache reuse of consecutive innermost loop executions that span the full domain.

ADVECTION The optimal solution of the advection kernel fuses all stencils with tile size $64 \times 16 \times 1$, $64 \times 32 \times 1$, and $64 \times 32 \times 1$ for the Xeon, Xeon Phi, and Power systems, respectively. The fast memory model dominates the predicted execution time of the compute-intensive seven-point stencils. Especially for the Xeon Phi and Power systems, the fast memory model tends to underestimate the measured execution times. For example, since the cache accesses may not fully overlap with the actual stencil computation.

The auto-tuned versions of the fastwaves, diffusion, and advection kernels perform 6.5%, 0.8%, and 3.4% faster than ABSINTHE for the Xeon system, 0.7%, 7.8%, and 7.1% faster than ABSINTHE for the Xeon Phi system, and 6.1%, 2.5%, and 1.7% faster than ABSINTHE for the Power system, respectively. The small performance penalty compared to the much slower auto-tuning and the relative agreement of estimated and measured execution times demonstrate the effectiveness of our approach for different stencils and hardware architectures.

The hand-tuned kernels perform well, but the manual optimization of large codes is tedious and time-consuming. The combination of fusion heuristics with ABSINTHE demonstrates the challenge of independent fusion and tile size selection.

The auto-tuning approach always works best since it does not depend on the performance model assumptions. For example, the auto-tuned tile sizes violate the cache capacity constraints of the Power system, which means tiles fitting the L2 cache are not optimal for this architecture. Auto-tuning generates 277 implementation variants for every stencil group, which on the Xeon system results in 40 minutes search time for the diffusion kernel. Extending the auto-tuning to the 2^{15} fusion choices increases the search time beyond $10,000$ hours. ABSINTHE explores the full search space in 40 seconds.

3.4.5 *Comparison with Halide and Polymage*

To compare ABSINTHE, we implement the application kernels with Polymage [13] (git:a8a101b) and Halide [12] (git:3af2386). We optimize the stencil sequences with the built-in auto-schedulers [23, 58], compile the ABSINTHE and Polymage kernels with GCC 5.3, and adapt the scheduling parameters to match the processor count of the Xeon system.

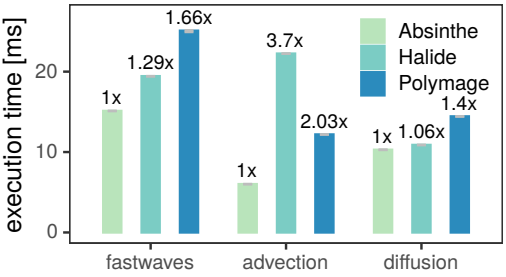


FIGURE 3.11: Execution times of the ABSINTHE, Halide, and Polymage tuned application kernels for domain size $256 \times 256 \times 60$ on the Xeon system (slowdowns relative to ABSINTHE).

Figure 3.11 compares the execution times for the ABSINTHE, Polymage, and Halide tuned application kernels. We set the domain size to $256 \times 256 \times 60$ elements since Halide and Polymage do not perform well for small domains. ABSINTHE and Polymage apply the same code transformations and use the same compiler, which makes the results comparable. Halide compiles with LLVM and performs loop reordering and stencil inlining, which reduces the significance of the results. ABSINTHE performs best for all kernels, which emphasizes the quality of the fusion and tile size selection choices.

3.5 RELATED WORK

Tile size selection is a well-researched topic with two main directions: purely analytic approaches [59–68] and empirical approaches [52, 69–71] that search different configurations for optimal performance. Yuki et al. [72] learn machine-specific static tile size selection models. Artificial neuronal networks are also effective for both instruction throughput prediction [73] and tile size selection [74].

D. Cociorva et al. [75] observe that program scheduling and tile size selection are intertwined and propose a dynamic programming based approach for combined scheduling and tile size selection specific for tensor sequences. Their work does not consider stencil computations. Quasem and Kennedy [76] propose a model guided empirical approach for loop fusion and tiling. Beaugnon et al. [77] also combine analytical modeling

and empirical search space exploration. They present an analytical model to compute a lower bound for the execution time of partially-specified program variants that allows them to prune the search space early-on. None of these works provide a linear programming formulation.

There exist several approaches for generating code for iterative stencils. Patus [22] is a code generator for iterative stencils on CPUs and GPUs. Henretty et al. [78] introduced a code generator for iterative multi-statement stencils that implements the DLT [79] data layout transformation. Both code generators rely on tile size auto-tuning. Pochoir [11] is an iterative stencil compiler that uses cache-oblivious tiling techniques to avoid the tile size selection problem. Prajapati et al. [80] manually derive tile size selection models for single statement stencils executed on GPUs. They require non-linear integer programming which takes hours to terminate and commonly does not guarantee optimal solutions.

STELLA [5] is a domain-specific language for climate modeling, while Halide [12] and Polymage [13] are domain-specific languages for image processing pipelines. All approaches support the optimization of stencil programs with data-locality transformations. MODESTO [1] is an analytic performance model to derive optimal fusion patterns for stencil programs based on memory bandwidth estimates. However, the model does not consider loop overheads and other metrics important for good tile size selection. For Polymage, Jangda and Bondhugula [58] employ dynamic programming to explore the space of fusion choices according to a cost function. During the optimization, a heuristic selects suitable tile sizes. For Halide, Liao et al. [81] and Mullapudi et al. [23] suggest cost functions and custom optimization strategies to perform automatic scheduling, which covers tile size selection. These solutions do not integrate the fusion and tile size selection choice in a single linear model. ABSINTHE thus provides the first holistic integer linear programming formulation that simultaneously schedules stencil programs and chooses matching tile sizes.

3.6 SUMMARY OF THE APPROACH

ABSINTHE instantiates an optimization problem that evaluates a learned performance model to select target system-specific data-locality transformations for stencil codes. Surprisingly six performance model parameters are sufficient to capture the relevant target system characteristics. The evaluation of the performance model is fast and requires no complex operations. We also demonstrate how to linearize the performance model for stencil

codes with known domain sizes of limited range. These properties facilitate the efficient exploration of the exponential search space with the help of powerful linear solvers. Our empirical evaluation provides strong evidence that learning a target-specific performance model is a competitive alternative to auto-tuning.

4

AN ANALYTICAL CACHE MODEL

Most programmers know the time complexity of their algorithms and tune codes by minimizing computation. Yet, ever increasing data-movement costs urge them to pay more attention to data-locality as a prerequisite for peak performance. When considering different implementation variants of an algorithm, we typically have a good understanding of which variant performs less computation or can be vectorized well. Selecting the optimal tile size or deciding which loop fusion choice is optimal is far less intuitive. Essentially, we lack a perception of the cache state that allows us to reason about data movement.

Data-locality optimizations are often pushed to the end of the development cycle when the code is available for benchmarking. But at this stage eliminating fundamental design flaws may be hard. We believe a cache model responsive enough to be part of the day-to-day workflow of a performance engineer can provide the necessary guidance to make good design choices upfront. After the completion of the development, the very same model could provide the necessary data for accurate model driven automatic memory tuning.

We present HAYSTACK the first cache model for fully associative caches with least recently used (LRU) replacement policy which is both fast and accurate. At the core of our model, we calculate the LRU stack distance [82] (also called reuse distance [83–85]) symbolically for each memory access. The stack distance counts the distinct memory accesses between two subsequent accesses of the same memory location. All memory accesses with distance shorter than the cache size hit a fully associative LRU cache.

We show in Figure 4.1 the scaling of HAYSTACK compared to the Dinero IV [86] cache simulator for increasing problem sizes. The simulation times are proportional to the problem size since simulators [86–89] enumerate all memory accesses. We use the Barvinok algorithm [49] to count the cache misses. The algorithm avoids explicit enumeration by deriving symbolic expressions that evaluate to the cardinality of the counted affine integer sets and maps. As demonstrated by the flat GEMM scaling curve, this symbolic counting makes the model execution time problem size independent. Even for Cholesky factorization, with its known

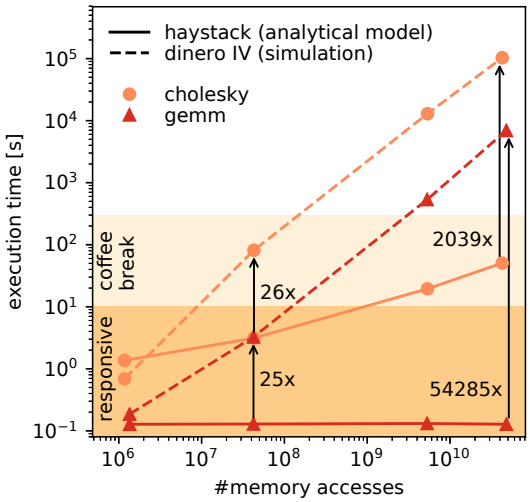


FIGURE 4.1: Scaling of the cache model compared to simulation.

non-linearities [90] that prevent full symbolic counting, the scaling of the execution time remains flat compared to simulation.

While computing stack distances for static control programs is a well known technique, reducing stack distance information for all dynamic memory accesses to a single cache miss count is difficult. Beyls et al. [90] show that stack distances in general are non-affine. The divisions introduced when modeling cache lines add even more non-affine constraints. While symbolic summation over affine constraint sets is possible with the Barvinok algorithm, symbolic counting over non-affine constraints is considered hard in general.

In this work, we show that this generally hard problem can in practice become surprisingly tractable if non-linearities are carefully eliminated by either specialization or partial enumeration. As a result we contribute:

- The first efficient cache model to accurately predict static affine programs on fully associative LRU caches.
- An efficient hybrid algorithm that combines symbolic counting with partial enumeration to reduce the asymptotic cost of the cache miss counting.

- A set of simplification techniques that exploit the regular patterns induced by the cache line structure to make the stack distance polynomials affine.
- An exhaustive evaluation which shows that our cache model performs well in practice with large speedups compared to existing approaches while achieving high accuracy compared to measurements on real hardware.

4.1 BACKGROUND

We first introduce our hardware model, provide background on cache misses, explain the concept of affine integer sets and maps, and discuss the set of considered programs.

4.1.1 *Hardware Model*

A cache implements various complex and sometimes undisclosed policies that define the exact behavior. We deliberately model a generic cache with full associativity and LRU replacement policy. When writing, we assume the caches allocate a cache line and load the memory reference if necessary (write-allocate) and then forward the write to all higher-level caches (write-through). We parametrize our cache model with the cache line size L and the cache size C in bytes. When modeling multiple cache hierarchy levels, we assume inclusive caches and specify the cache size for every hierarchy level. These design choices avoid an overly detailed model that is only correct in a very controlled environment with know data alignment and allocation. As shown by Section 4.3.2, we still model enough detail to produce actionable and accurate results in practice.

4.1.2 *Cache Misses*

We assume that the modeled programs run in isolation and that their execution starts with an empty cache. We count data accesses and ignore instruction fetches.

According to Hill [91], we distinguish three types of cache misses; 1) *compulsory misses* happen if a program accesses a cache line for the first time, 2) *capacity misses* happen if a program accesses too many distinct cache lines before accessing a cache line again, and 3) *conflict misses* happen if a

program accesses to many distinct cache lines that map to the same cache set of an associative cache before accessing a cache line again. We model fully associative caches and thus compute only compulsory and capacity misses.

Not every access of a program variable translates in a cache access as the compiler may place scalar variables in registers. Compiler and hardware techniques such as out-of-order execution also change the order of the memory accesses. We assume all scalar variables are buffered in registers and count only array accesses in the order provided by the compiler front end.

The cache misses measured when profiling a program depend on many factors generally unknown to an analytical cache model, for example, concurrent programs or the operating system may pollute the caches or the hardware prefetchers may load more data than necessary. We do not consider this system noise and instead provide an approximate but deterministic cache model.

4.1.3 *Integer Sets and Maps*

We use sets and maps of integer tuples to count the cache misses. We next define the relevant set and map operations necessary for the model implementation. These operations are a subset of the functionality provided by the integer set library (isl) [48].

An affine set

$$\mathbf{S} = \{(i_0, \ldots, i_n) : con(i_0, \ldots, i_n)\}$$

defines the subset of integer tuples $(i_0, \ldots, i_n) \in \mathbb{Z}^n$ that satisfy the constraints $con(i_0, \ldots, i_n)$. The constraints are Presburger formulas that combine affine expressions with comparison operators, boolean operators, and existential quantifiers. Presburger arithmetic [92] also admits floor division and modulo with a constant divisor.

An affine map

$$\mathbf{R} = \{(i_0, \ldots, i_n) \to (j_0, \ldots, j_m) : con(i_0, \ldots, i_n, j_0, \ldots, j_m)\}$$

defines the relation from integer tuples $(i_0, \ldots, i_n) \in \mathbb{Z}^n$ to integer tuples $(j_0, \ldots, j_m) \in \mathbb{Z}^m$ that satisfy the constraints $con(i_0, \ldots, i_n, j_0, \ldots, j_m)$ where the constraints have the same restrictions as the set constraints. The domain $\mathbf{R}_{dom}$ defines the set of the integer tuples $(i_0, \ldots, i_n)$ of the input dimensions for which a relation exists, and conversely the range $\mathbf{R}_{ran}$ defines the set

```
1        int sum = 0;
2        for(int i=0; i<4; ++i)
3   S0:   M[i] = i;
4        for(int j=0; j<4; ++j)
5   S1:   sum += M[3-j];
```

FIGURE 4.2: Example program used for illustration.

of integer tuples $(j_0, \ldots, j_m)$ of the output dimensions for which a relation exists.

Both sets and maps support the set operations intersection $\mathbf{S}_1 \cap \mathbf{S}_2$, union $\mathbf{S}_1 \cup \mathbf{S}_2$, projection, and cardinality $|\mathbf{S}|$. The domain intersection $\mathbf{R} \cap_{dom} \mathbf{S}$ intersects the domain of the map $\mathbf{R}$ with the set $\mathbf{S}$. Maps also support the map operations composition $\mathbf{R}_2 \circ \mathbf{R}_1$ and inversion $\mathbf{R}^{-1}$. The operator

$$\begin{aligned}\text{lexmin}(\mathbf{R}) = \{(i_0, \ldots, i_n) \to (m_0, \ldots, m_m) : \\ \nexists (i_0, \ldots, i_n) \to (j_0, \ldots, j_m) \in \mathbf{R}, \\ \text{s.t. } (j_0, \ldots, j_m) \prec (m_0, \ldots, m_m)\}\end{aligned}$$

computes for every input tuple $(i_0, \ldots, i_n)$ the lexicographic smallest output tuple $(m_0, \ldots, m_m)$ of all tuples $(j_0, \ldots, j_m)$ related to the input tuple.

A named set or map prefixes the integer tuples with names that convey semantic information. For example, we prefix the array element $\texttt{M}(2)$ with the array name and the statement instance $\texttt{S0}(1)$ with statement name. We use statement names starting with the letter `S` and array names starting with any other letter. The names are semantically equivalent to an additional tuple dimension.

4.1.4 *Static Control Programs*

Our cache model analyzes affine *static control programs* consisting of loop nests with known loop bounds that perform array accesses with affine index expressions. Figure 4.2 shows an example program with two statements: the statement `S0` initializes an array `M` and the statement `S1` accumulates the array elements. Before analyzing a program, we extract the sets and maps that specify the statement execution order and the memory access offsets.

The iteration domain

$$\mathbf{I} = \{\texttt{S0}(i) : 0 \leq i < 4; \texttt{S1}(j) : 0 \leq j < 4\}$$

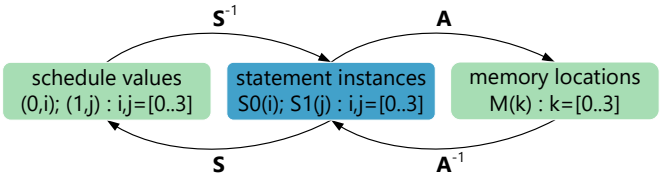


FIGURE 4.3: The statement instances and the related schedule values (schedule **S**) and memory accesses (access map **A**) are sufficient to compute the cache misses of a program.

defines the set of all executed statement instances. For the two statements of the example program, the loop variables i and j are limited to the range zero to three. To define the execution order, the schedule

$$\mathbf{S} = \{\mathtt{S0}(i) \to (0, i); \mathtt{S1}(j) \to (1, j)\} \cap_{dom} \mathbf{I}$$

maps the statement instances to a multi-dimensional schedule value. The statement instances then execute according to the lexicographic order of the schedule values. The intersection with the iteration domain **I** limits the schedule domain to the program loop bounds. The access map

$$\mathbf{A} = \{\mathtt{S0}(i) \to \mathtt{M}(i); \mathtt{S1}(j) \to \mathtt{M}(3 - j)\}$$

maps the array accesses of the statement instances to the accessed array elements. The iteration domain **I**, the schedule **S**, and the access map **A** capture all relevant program properties necessary to evaluate the cache model. Figure 4.3 shows how the schedule **S** and the access map **A** relate statement instances, schedule values, and memory locations.

4.2 CACHE MODEL

Our cache model computes for every memory access the stack distance parametric in the loop variables and counts the instances with a stack distance larger than the cache capacity to determine the capacity misses. All memory accesses with undefined backward stack distance access the cache line for the first time and count as compulsory misses.

Figure 4.4 shows the computation of the capacity misses for the example program introduced by Figure 4.2: (1) enumerates the statement instances according to the schedule **S** and (2) applies the access map **A** to the statement instances to compute the memory trace. Assuming the array element size is equal to the cache line size, the stack distance corresponds to the

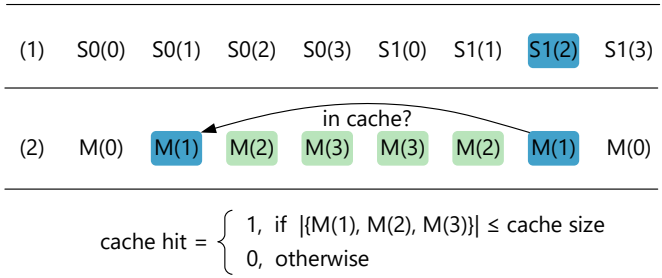


FIGURE 4.4: The (1) statement instance and the (2) memory access trace of the example program allow us to compute if the access $\texttt{M}(1)$ of the statement $\texttt{S1}(2)$ hits the cache.

cardinality of the set $\{\texttt{M}(1), \texttt{M}(2), \texttt{M}(3)\}$ which contains the array elements accessed between and including the two subsequent accesses of $\texttt{M}(1)$. The second access of $\texttt{M}(1)$ hits the cache if the cardinality of the set is lower than or equal to the cache capacity.

4.2.1 *Computing the Stack Distance*

The stack distance computation counts the number of distinct memory accesses between subsequent accesses of the same memory location. We determine for every memory reference the last access to the same memory location and count the set of memory accesses since this last access to obtain the stack distance parametric in the loop variables.

For our example program, the stack distance of the memory access in statement S1 is equal to the loop variable j plus one. We can thus express the stack distance of the memory access with the map

$$\mathbf{D} = \{\texttt{S1}(j) \to j + 1 : 0 \leq j < 4\}$$

limited to the statement iteration domain. As the statement S0 accesses all array elements for the first time its backward stack distance is undefined and the accesses count as compulsory misses.

Our discussion of the stack distance computation initially assumes that every statement performs at most one access of a one-dimensional array with an element size equal to the cache line size. At the end of this section, we show how to overcome these limitations.

The memory accesses execute according to the statement execution order defined by the schedule. The map

$$\begin{aligned}\mathbf{L}_{\prec} = \{&(i_0, \ldots, i_n) \to (j_0, \ldots, j_n) : \\ &(i_0, \ldots, i_n) \prec (j_0, \ldots, j_n) \wedge \\ &(i_0, \ldots, i_n), (j_0, \ldots, j_n) \in \mathbf{S}_{ran}\}\end{aligned}$$

relates the schedule values $(i_0, \ldots, i_n)$ to all lexicographically larger schedule values $(j_0, \ldots, j_n)$ and the map

$$\begin{aligned}\mathbf{L}_{\preceq} = \{&(i_0, \ldots, i_n) \to (j_0, \ldots, j_n) : \\ &(i_0, \ldots, i_n) \preceq (j_0, \ldots, j_n) \wedge \\ &(i_0, \ldots, i_n), (j_0, \ldots, j_n) \in \mathbf{S}_{ran}\}\end{aligned}$$

relates the schedule values $(i_0, \ldots, i_n)$ to all lexicographically larger or equal schedule values $(j_0, \ldots, j_n)$. Later on, we use these helper maps to filter relations by execution order.

The stack distance computation first identifies all accesses to the same array element. The equal map

$$\mathbf{E} = \mathbf{S} \circ \mathbf{A}^{-1} \circ \mathbf{A} \circ \mathbf{S}^{-1}$$

relates each schedule value to all schedule values that access the same array element. The concatenation $\mathbf{A} \circ \mathbf{S}^{-1}$ maps the schedule values to the accessed array elements and its reverse $\mathbf{S} \circ \mathbf{A}^{-1}$ maps the accesses back to the schedule values. For our example program, the composition

$$\begin{aligned}\mathbf{A} \circ \mathbf{S}^{-1} = \{&(0, i) \to \mathtt{M}(i) : 0 \leq i < 4; \\ &(1, j) \to \mathtt{M}(3 - j) : 0 \leq j < 4\}\end{aligned}$$

relates the schedule values to the accesses of the array `M`. The equal map then relates all schedule values that access the same array element. For example, the relations $(0, i) \to \mathtt{M}(i)$ and $(1, j) \to \mathtt{M}(3 - j)$ access the same array element if i is equal to $3 - j$. The resulting equal map

$$\begin{aligned}\mathbf{E} = \{&(0, i) \to (0, i) : 0 \leq i < 4; \\ &(1, j) \to (1, j) : 0 \leq j < 4; \\ &(0, i) \to (1, j) : j = 3 - i \wedge 0 \leq i < 4; \\ &(1, j) \to (0, i) : i = 3 - j \wedge 0 \leq j < 4\}\end{aligned}$$

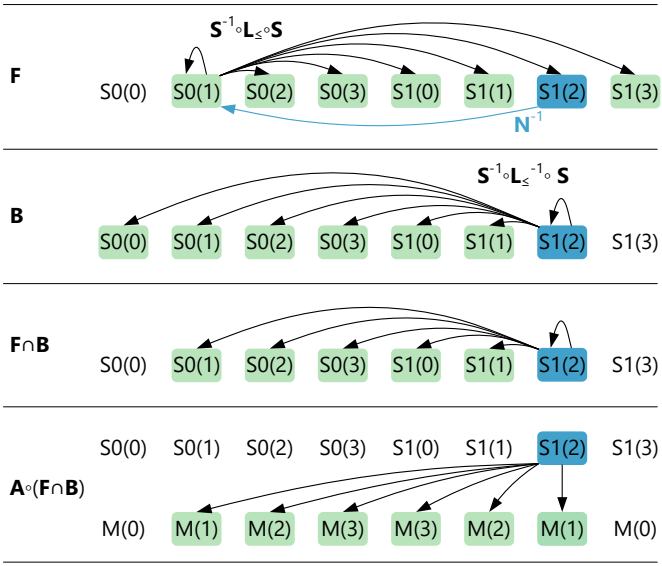


FIGURE 4.5: The relations of the forward map **F** and the backward map **B** for the statement instance S1(2) of the example program (the forward map **F** corresponds to the concatenation of the blue backward arrow and the black forward arrows). The map intersection defines the statement instance between and including the two accesses of M(1). The concatenation with the map **A** yields the related memory accesses.

contains the relation $(0, i) \to (1, j)$ with $j = 3 - i$ and its reverse but also the self relations of the schedule values.

The lexicographically shortest relations of the equal map denote the subsequent accesses to the same array element which are closest in time. The next map

$$\mathbf{N} = \mathbf{S}^{-1} \circ \text{lexmin}(\mathbf{L}_{\prec} \cap \mathbf{E}) \circ \mathbf{S}$$

intersects the equal map $\mathbf{E}$ with the map $\mathbf{L}_{\prec}$ to filter out all backward in time and self relations and the lexmin operator removes all forward in time relations except for the shortest ones. We compose the result with $\mathbf{S}$ and $\mathbf{S}^{-1}$ to convert the schedule values to statement instances. The next map consequently relates every statement instance to the next statement instance that accesses the same array element. For our example program, the equal map contains only the forward relation $(0, i) \to (1, j)$ which means the lexmin operator has no effect since there is only one relation per statement instance. The next map

$$\mathbf{N} = \{\texttt{S0}(i) \to \texttt{S1}(j) : j = 3 - i \wedge 0 \leq i < 4\}$$

thus relates the instances of statement `S0` to the instances of statement `S1` that access the same array element.

The next map contains subsequent statement instances that access the same array element but not the statement instances executed in between. To compute them, we intersect the set of statement instances executed after the first access with the set of statement instances executed before the second access of the same array element. Figure 4.5 illustrates this intersection. The backward map

$$\mathbf{B} = \mathbf{S}^{-1} \circ \mathbf{L}_{\preceq}^{-1} \circ \mathbf{S}$$

relates the statement instances to all statement instances with lexicographically smaller or equal schedule value. The maps $\mathbf{S}$ and $\mathbf{S}^{-1}$ convert from statement instances to schedule values and back. The forward map

$$\mathbf{F} = (\mathbf{S}^{-1} \circ \mathbf{L}_{\preceq} \circ \mathbf{S}) \circ \mathbf{N}^{-1}$$

relates the statement instances to all statement instances with lexicographically larger or equal schedule value than the statement instance that last accessed the same array element. We reverse the next map $\mathbf{N}$ to compute the statement instance that accessed the array element last. The intersection of the forward map and the backward map contains all statement instances executed between subsequent accesses of the same array element.

Figure 4.5 shows the forward and backward map relations for the statement instance $\texttt{S1}(2)$ of the example program that accesses the array element $\texttt{M}(1)$. The forward map $\mathbf{F}$ corresponds to the concatenation of the blue backward arrow and the black forward arrows. The intersection of the two maps contains the statement instances executed between the subsequent accesses of the array element $\texttt{M}(1)$. We finally concatenate this intersection with the access map $\mathbf{A}$ to obtain the stack distance map that relates every statement instance to the array accesses performed since the last access of the same array element.

The number of related array elements defines the stack distance of the statement instances in the stack distance map. We use the isl [48] implementation of the Barvinok algorithm [49] to count the relations symbolically. The algorithm computes the map cardinality by counting the points of the range related to every point of the domain. The result of the computation are quasi polynomials parametric in the input dimensions of the map that evaluate to the number of related range points. As the domain is not always homogeneous, the algorithm splits the map domain into pieces that consist of a quasi polynomial and the subdomain of the map domain where the polynomial is valid. After counting the stack distance map, the distance set

$$\mathbf{D} = \{|\mathbf{A} \circ (\mathbf{F} \cap \mathbf{B})|\}$$

contains pieces with quasi polynomials parametric in the schedule input dimensions that for a subdomain of the iteration domain evaluate to the stack distance. The pieces do not overlap and together cover the full iteration domain. For our example program, the distance set

$$\mathbf{D} = \{\texttt{S1}(j) \to j + 1 : 0 \leq j < 4\}$$

contains one piece with the polynomial $\texttt{S1}(j) \to j + 1$ and the domain $0 \leq j < 4$ covering the entire iteration domain.

CACHE LINES AND MULTI-DIMENSIONAL ARRAYS An adapted access map $\mathbf{A}$ that relates statement instances to cache lines instead of array elements suffices to support cache lines and multi-dimensional arrays. Let us assume our example program initializes the diagonal elements of a two-dimensional array $\texttt{M}(i, i)$. Then the access map

$$\mathbf{A} = \{\texttt{S0}(i) \to \texttt{M}(i, c = \lfloor i * E/L \rfloor)\}$$

models the accessed cache lines given the size of the array elements E and cache line size L in bytes. We replace the innermost dimension of the array

access with the cache line index c, which multiplies the array index with the element size and divides the result by the cache line size. As a result, accesses of neighboring array elements map to the same cache line. The outer dimensions of the array index remain unchanged since we assume the innermost dimension is cache line aligned and padded to an integer multiple of the cache line size. This restriction can be lifted at the expense of a more complex formulation.

MULTIPLE MEMORY ACCESSES PER STATEMENT An extension of the schedule $\mathbf{S}$ and the access map $\mathbf{A}$ with an additional schedule dimension that orders the memory accesses of the statements allows us to model more than one memory access per statement. Let us assume the statement `S0` of the example program reads the array element $\mathtt{I}(i)$ and writes the result to the array element $\mathtt{M}(i)$. We then extend the schedule

$$\mathbf{S} = \{\mathtt{S0}(i,a) \to (0,i,a); \mathtt{S1}(j,a) \to (1,j,a)\}$$

with the access dimension a that orders the memory accesses of the statement. Then the access map

$$\mathbf{A} = \{\mathtt{S0}(i,0) \to \mathtt{I}(i); \mathtt{S0}(i,1) \to \mathtt{M}(i); \mathtt{S1}(j,0) \to \mathtt{M}(3-j)\}$$

assigns every array access to a unique statement instance since the access dimension enumerates the array accesses of every statement in the order provided by the compiler front end. The extended schedule executes only one array access per statement instance and thus requires no further modifications of the stack distance computation.

The output of the stack distance computation is a set of polynomials that defines the backward stack distance for every array access of the static control program.

4.2.2 *Counting the Capacity Misses*

All memory accesses with stack distance larger then the cache size count as capacity miss. As discussed in Section 4.2.1, the stack distance computation splits the iteration domain into pieces. Each piece defines the stack distance for a subdomain of the iteration domain. To obtain the capacity misses, we count for every piece the points of the subdomain for which the polynomial evaluates to a stack distance larger than the cache size.

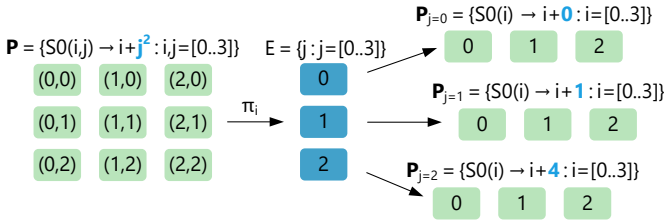


FIGURE 4.6: To count the non-affine piece $\mathbf{P}$, we project out the affine i-dimension to obtain the enumeration domain $\mathbf{E}$. We next bind the j-dimension of the piece $\mathbf{P}$ to the j-values in the enumeration domain and separately count the cache misses for the resulting affine pieces $\mathbf{P}_{j=0}$, $\mathbf{P}_{j=1}$, and $\mathbf{P}_{j=2}$.

The piece with polynomial $\texttt{S1}(j) \to j+1$ and domain $0 \leq j < 4$ defines the stack distance for the entire iteration domain of our example program. The cache miss set

$$\mathbf{M} = \{\texttt{S1}(j) : j+1 > C \wedge 0 \leq j < 4\}$$

contains all points of the piece with stack distance larger than cache size C which means the cardinality of the cache miss set $|\mathbf{M}|$ is equal to the number of capacity misses. Assuming cache size two, the cache miss set contains the statement instances $\texttt{S1}(2)$ and $\texttt{S1}(3)$ that cause two capacity misses.

The distance set specifies the stack distance for all program statements. To count the capacity misses per statement, we split the distance set by statement and compute the cache misses separately. Without loss of generality, we discuss the cache miss computation for a statement `S0`.

The Barvinok algorithm also computes the set cardinality by counting the points symbolically. We use the algorithm to count affine cache miss sets and resort to explicit enumeration for non-affine sets. As explicit enumeration is expensive, we only enumerate the non-affine polynomial dimensions and count the affine dimensions symbolically. This *partial enumeration* technique splits cache miss sets into pieces with affine lower-dimensional polynomials. Figure 4.6 demonstrates the technique for an example polynomial with non-affine j-dimension. Section 4.2.3 discusses further techniques to split non-affine pieces into multiple affine pieces.

Algorithm 1: counting the capacity misses

input : **D** distance set of pieces
output : T total number of cache misses
parameter: C cache size

```
1  T ← 0
2  foreach P in D do
3      if isPieceAffine(P) then
4          T ← T + countAffinePiece(P, C)
5      else
6          E ←getNonAffineDomain(P)
7          foreach pt in E do
8              P_pt ←bindNonAffineDimensions(P, pt)
9              T ← T + countAffinePiece(P_pt, C)
10         end
11     end
12 end
13 return T
```

Algorithm 1 counts the total number of cache misses T given the distance set **D** of the program. The algorithm enumerates all pieces **P** of the distance set (lines 2-12). Every piece **P** consists of a polynomial and a domain that define the stack distance of a memory access for a subdomain of the iteration domain. If the polynomial of the piece **P** is affine we count the cache misses symbolically (lines 3-4), otherwise the *partial enumeration* projects the non-affine dimensions out of the domain of the piece **P** and enumerates all points of the resulting non-affine enumeration domain **E** (lines 6-9). For every such point pt, we bind the non-affine dimensions of the piece **P** to the coordinates of the point pt and count the cache misses of the affine piece $\mathbf{P}_{\text{pt}}$ symbolically. Figure 4.6 illustrates the splitting of non-affine pieces (lines 6-9).

The method *countAffinePiece* counts the cache misses of the piece **P** with affine stack distance polynomial. A polynomial is affine if its degree is zero or one. We first compute the cache miss set

$$\mathbf{M} = \{\texttt{S0}(i_0, \ldots, i_n) : \mathbf{P}_p(i_0, \ldots, i_n) > C \wedge (i_0, \ldots, i_n) \in \mathbf{P}_D\}$$

where $\mathbf{P}_p$ denotes the polynomial and $\mathbf{P}_D$ the domain of the piece **P**. The cache miss set contains all memory accesses with stack distance larger than

cache size C. To count the cache misses, we compute the cardinality $|\mathbf{M}|$ using the Barvinok algorithm.

The method *getNonAffineDomain* projects all points of the piece $\mathbf{P}$ to the non-affine dimensions to obtain the enumeration domain $\mathbf{E}$. For example, Figure 4.6 projects the piece

$$\mathbf{P} = \{\texttt{S0}(i,j) \to i + j^2 : 0 \leq i < 3 \wedge 0 \leq j < 3\}$$

which contains the quadratic term j^2. We project the points to the non-affine j-dimension to compute the enumeration domain $\mathbf{E} = \{j : 0 \leq j < 3\}$. The enumeration always spans all dimensions with degree larger than one. But the polynomial may also contain product terms with multiple dimensions. We then greedily select the dimensions that conflict with most other dimensions. For example, if the polynomial contains the products ij and ik we enumerate the i-dimension since it conflicts with both other dimensions.

The method *bindNonAffineDimensions* binds the non-affine dimensions of the piece $\mathbf{P}$ to the values of the point pt. For example, Figure 4.6 binds the j-dimension of the piece

$$\mathbf{P} = \{\texttt{S0}(i,j) \to i + j^2 : 0 \leq i < 3 \wedge 0 \leq j < 3\}$$

to the value two and obtains the piece

$$\mathbf{P}_{j=2} = \{\texttt{S0}(i) \to i + 4 : 0 \leq i < 3\}$$

which we can count with the method *countAffinePiece*.

The counting algorithm works for all static control programs and avoids complete enumeration except all dimensions are non-affine.

4.2.3 *Eliminating Non-Affine Terms*

Many stack distance polynomials contain non-affine terms that prevent fast symbolic counting. We develop rewrite strategies that eliminate non-affine terms containing floor expressions. The floor expressions themselves are quasi-affine but often appear in products with other non-constant operands modeling effects such as the stack distance variation for different cache line offsets. We specialize the stack distance polynomials for different cache line offsets to make them affine which enables the efficient symbolic counting.

The floor expressions of some polynomials differ only by a constant offset. For example, the piece

$$\mathbf{P} = \{\texttt{S0}(i,j) \leftarrow (\lfloor (1+i)/3 \rfloor - \lfloor i/3 \rfloor) j : 0 \leq i < 3 \wedge 0 \leq j < 2\}$$

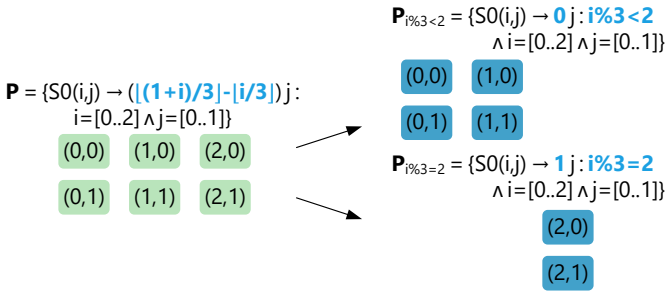


FIGURE 4.7: *Equalization* replaces the non-affine piece $\mathbf{P}$ with the affine pieces $\mathbf{P}_{i\%3<2}$ and $\mathbf{P}_{i\%3=2}$ to model a stack distance that varies at the last cache line offset.

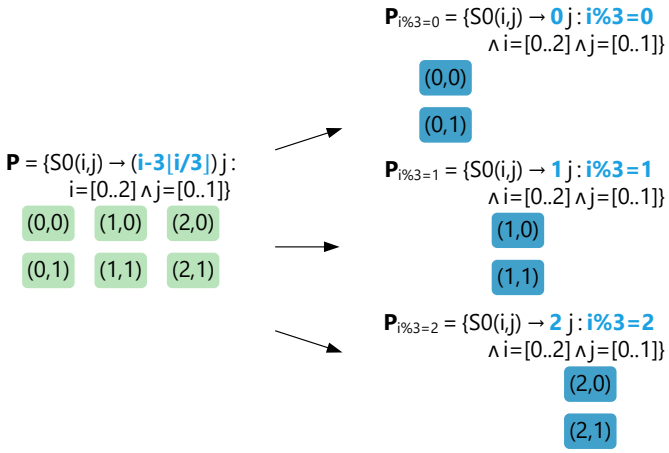


FIGURE 4.8: *Rasterization* replaces the non-affine piece $\mathbf{P}$ with the affine pieces $\mathbf{P}_{i\%3=0}$, $\mathbf{P}_{i\%3=1}$, and $\mathbf{P}_{i\%3=2}$ to model a stack distance that varies at every cache line offset.

contains the floor expressions $\lfloor (1+i)/3 \rfloor$ and $\lfloor (i)/3 \rfloor$. The two floor expressions are equal except if i modulo three is equal to two. Then the second floor expression is larger by one. The difference of the two floor expressions thus evaluates to zero for the first two elements and to one for the last element of every cache line. Figure 4.7 shows how to introduce simplified polynomials for the first two and the last element of every cache line. This *equalization* technique splits the cache line in multiple regions that typically contain more than one element.

The polynomials may also contain terms with the plain variable and other terms which compute the floor of the variable. For example, the piece

$$\mathbf{P} = \{S0(i,j) \to (i - 3\lfloor i/3 \rfloor)j : 0 \leq i < 3 \wedge 0 \leq j < 2\}$$

contains the floor expression $3\lfloor i/3 \rfloor$ which is equal to i except for a constant that depends on the cache line offset. Figure 4.8 shows how to replace the polynomial with one simplified polynomial per cache line offset. This *rasterization* technique enumerates all cache line offsets.

We apply the two floor elimination techniques in the order of presentation and only keep the results if the degree of at least one simplified polynomial is lower than the degree of the original polynomial.

4.2.4 *Counting the Compulsory Misses*

All memory accesses that touch a cache line for the first time are compulsory misses.

As the array `M` of our example program is initialized by the statement `S0`, the first map

$$\mathbf{F} = \{\mathtt{M}(i) \to \mathtt{S0}(i) : 0 \leq i < 4\}$$

relates every array element to the statement instance that accesses the element first which means the cardinality $|\mathbf{F}_{dom}|$ of the first map domain counts the compulsory misses.

The compulsory misses are the memory accesses with lexicographically minimal schedule value. The first map

$$\mathbf{F} = \mathbf{S}^{-1} \circ \text{lexmin}(\mathbf{S} \circ \mathbf{A}^{-1})$$

thus selects for every memory access the lexicographically minimal relation of the composition $\mathbf{S} \circ \mathbf{A}^{-1}$ that relates memory accesses to schedule values and composes the result with the inverse schedule $\mathbf{S}^{-1}$ to obtain the related statement instances. The composition with the inverse schedule allows us

to intersect the range of the first map with the iteration domain of the individual statements to count the compulsory misses per statement. For our example program, the composition

$$\mathbf{S} \circ \mathbf{A}^{-1} = \{\mathtt{M}(i) \to (0, i) : 0 \leq i < 4; \\ \mathtt{M}(j) \to (1, 3 - j) : 0 \leq j < 4\}$$

contains two accesses for every array element. The lexmin operator removes the second access due to the lexicographically larger schedule value. After the composition with the inverse schedule $\mathbf{S}^{-1}$, we use the Barvinok algorithm to count the compulsory misses $|\mathbf{F}_{dom}|$.

4.2.5 *Computational Complexity*

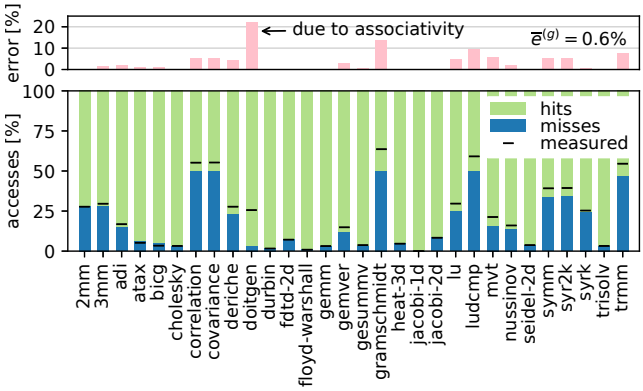
All compute-heavy parts of our cache model perform Presburger arithmetic that in general is known to have very high computational complexity [92, 93]. The established complexity bounds range from polynomial time decidable [94] for expressions with fixed dimensionality and only existential quantification to double exponential [95] for arbitrary expressions. Haase [92] presents further results that show a complexity increase with the dimensionality and the number of quantifier alternations of the Presburger expression.

The Presburger relations computed by our cache model have only existential quantification and the dimensionality is limited by the loop depth suggesting polynomial complexity. Yet, the cache model may introduce further variables to model divisions or modulo operations making the complexity exponential in the number of dimensions.

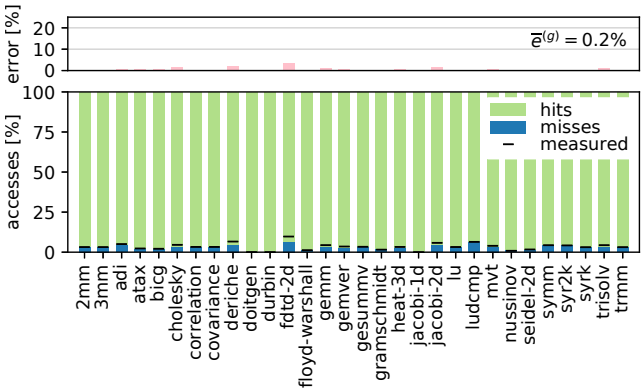
Although the cache model has exponential worst-case complexity, the empirical performance evaluation presented in Section 4.3.3 shows that our cache model performs well for typical input programs. The dimensionality of the observed Presburger relations remains limited since most real-world programs do not make extensive use of branch conditions and index expressions that result in integer divisions or modulo operations.

4.3 EVALUATION

We next evaluate the performance of HAYSTACK and compare its accuracy to simulated and measured results.



(a) L1 cache



(b) L2 cache

FIGURE 4.9: Cache misses and hits predicted by HAYSTACK compared to the measured cache misses (median of 10 measurements) for the PolyBench kernels with the prediction error relative to the number of memory accesses on top.

4.3.1 *Setup and Methodology*

We evaluate on a test system with two 18-core Intel Xeon Gold 6150 processors. Every core has a 32KiB L1 cache (8-way set associative) and an inclusive 1MiB L2 cache (16-way set associative). The non-inclusive 18x1.375MiB L3 cache (11-way set associative) is shared among all cores. A non-inclusive cache may and an inclusive cache has to duplicate all cache lines stored by the lower-level caches. All caches load the cache line before writing (write-allocate) and forward the write only if the cache line is evicted (write-back).

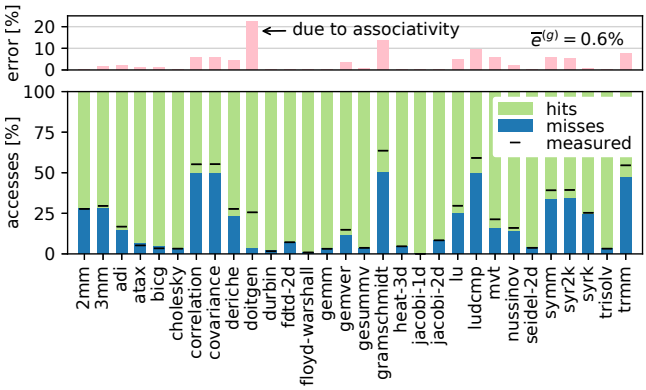
We compile with GCC 6.3 and use the Dinero IV cache simulator [86] to compute and the PAPI-C library [96] to measure the number of cache misses. We evaluate the model for a number of different kernels. PolyBench 4.2.1-beta [57] is a collection of static control programs that implement algorithmic motifs from scientific computing. If not stated otherwise the PolyBench experiments use the default configuration (large) and the model emulates fully associative L1 and L2 caches with the capacities of the test system.

All performance measurements run single-threaded using only one core of the test system. To quantify measurement noise, the execution times show the median and the non-parametric 95% confidence intervals [56] of 10 measurements.

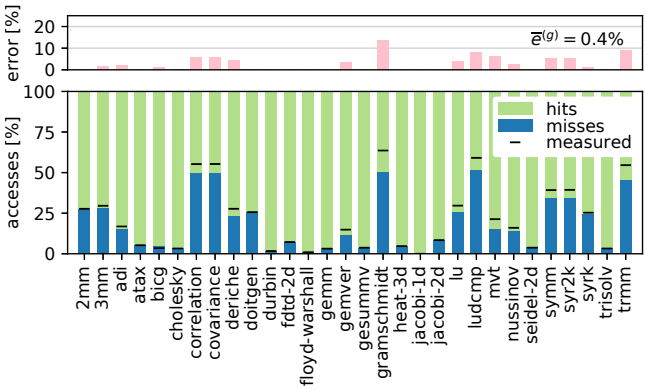
4.3.2 *Accuracy Overview*

All mathematical models are a trade-off between accuracy and complexity. A static cache model cannot predict dynamic measurement noise for example due to concurrent code execution. We aim at an accurate prediction of the cache misses without modeling too many implementation details.

A comparison to measurements on a real system is the main benchmark for every cache model. To measure the cache misses, we compile the PolyBench [57] kernels with PAPI [96] support using GCC optimization level O2. PolyBench [57] flushes the caches before every kernel execution which allows us to measure compulsory and capacity misses. We collect the counters `PAPI_L1_DCM` and `PAPI_L2_DCM` that sum the data cache misses for the L1 and L2 caches, respectively. Figure 4.9 compares the sum of the compulsory and capacity misses predicted by HAYSTACK to the measured cache misses shown by black lines. Most kernels cause more cache misses than predicted which is expected since we model idealized fully associative



(a) L1 cache (fully associative)



(b) L1 cache (8-way associative)

FIGURE 4.10: Cache misses and hits simulated by Dinero IV compared to the measured cache misses (median of 10 measurements) for the PolyBench kernels with the prediction error relative to the number of memory accesses on top.

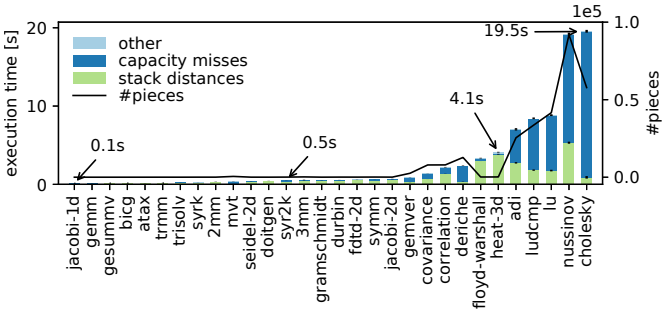


FIGURE 4.11: Execution times for the main components of HAYSTACK compared to the number of separately counted pieces for the PolyBench kernels sorted by execution time.

caches with LRU instead of pseudo-LRU replacement policy. We also do not consider possible overfetch due to the hardware prefetchers. To quantify the error, Figure 4.9 shows for every kernel the prediction error relative to the total number of memory accesses computed by the model. Most kernels have low single digit prediction errors with a geometric mean error of 0.6% and 0.2% for the L1 cache and the L2 cache, respectively. Only doitgen and gramschmidt have prediction errors above 10%.

We also execute the PolyBench kernels with Dinero IV [86] to simulate the number of cache misses with full associativity and with the associativity of our test system. Figure 4.10 compares the sum of the simulated compulsory, capacity, and conflict misses to the measured cache misses shown by black lines. We observe that the simulation results for the fully associative L1 cache qualitatively agree with the model. All simulation results are within 0.1% of the model for the L1 cache and within 3% of the model for the L2 cache (relative to the total number of memory accesses). We conclude that our design decisions of padding the innermost dimension of multi-dimensional arrays, discussed in Section 4.2.1, and modeling only array accesses and not scalar accesses, discussed in Section 4.1.2, have no significant impact on the accuracy of the model. The simulation results with test system associativity eliminate the error for the doitgen kernel. We conclude that modeling set associativity is only relevant for one of the PolyBench kernels. The error of the remaining kernels is dominated by other error sources such as the difference between LRU and pseudo-LRU replacement policy that are neither considered by the simulator nor by the model.

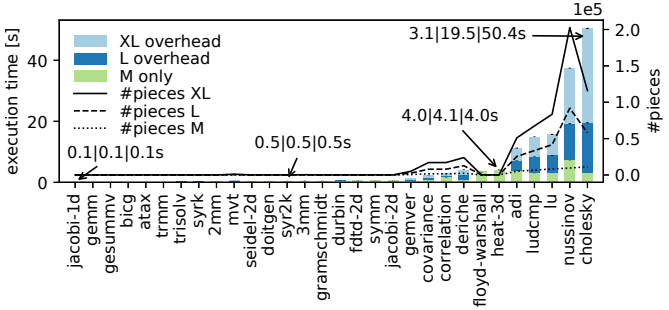


FIGURE 4.12: Execution times for the extra large (XL), large (L), and medium (M) problem sizes of PolyBench compared to the number of counted pieces.

HAYSTACK reproduces the simulation results for full associativity and the associativity mismatch compared to the test system does not dominate the modeling error.

4.3.3 *Performance Overview*

We next analyze the performance of HAYSTACK and its sensitivity to model parameters such as the problem size or the number of cache hierarchy levels.

Two components dominate the model execution time: 1) the stack distance computation discussed in Section 4.2.1 and 2) the capacity miss counting discussed in Section 4.2.2. Figure 4.11 shows the cost of the two components compared to the total model execution times for the PolyBench kernels. The analysis of most kernels terminates within 5 seconds (jacobi-1d to heat-3d) while the more expensive kernels take up to 20 seconds (adi to cholesky). The capacity miss counting dominates the cost of the expensive kernels. When counting the capacity misses, the *partial enumeration* and to a lesser extend the *equalization* and *rasterization*, discussed in Section 4.2.3, split the iteration domain into pieces with affine stack distance polynomials that support symbolic counting. The solid line in Figure 4.11 shows the number of counted pieces. We observe that the expensive kernels require more splits due to non-affine stack distance polynomials and that the counting costs correlate with the number of pieces.

Other than for a cache simulator, the model execution time is not proportional to the number of memory accesses. Figure 4.12 shows the model

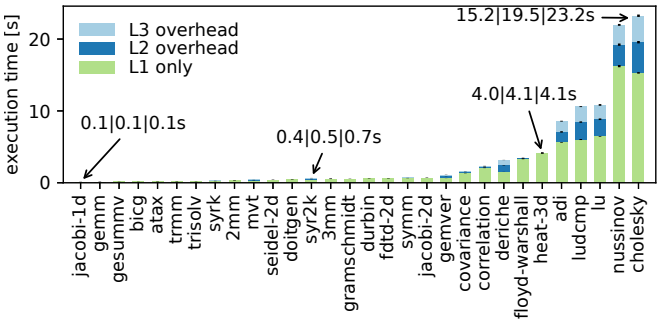


FIGURE 4.13: Comparison of the execution times when modeling one, two, or three cache hierarchy levels.

execution times for the three largest PolyBench problem sizes. The large (L) and the extra large (XL) problem size perform roughly 100 and 1000 times more memory access than the medium (M) problem size, respectively. Yet, the execution times remain constant for a majority of the kernels. Only the execution times of the expensive kernels increase since the *partial enumeration* requires more splits. The number of counted pieces, shown by the solid, dashed, and dotted lines in Figure 4.12, correlate with the cost increase for the larger problem sizes. Even for the expensive kernels, the increase of the execution time is not proportional to the number of memory accesses since we enumerate only the non-affine dimensions of the stack distance polynomials.

When counting the cache misses for multiple cache hierarchy levels, we reuse the stack distance polynomials and enumerate the non-affine dimensions only once. The counting of the individual pieces is the only step repeated for every cache size. As the Barvinok algorithm [49] supports parametric counting, we can count the capacity misses parametric in the cache size which avoids any additional overhead when modeling additional cache hierarchy levels. We benchmark the non-parametric version of the code as it runs faster even when modeling three cache hierarchy levels. Figure 4.13 shows minor increases of the total execution time for two and three cache hierarchy levels.

The *partial enumeration*, discussed in Section 4.2.2, combines enumeration of the non-affine dimensions with symbolic counting of the affine dimensions. Figure 4.14 compares *partial enumeration* to the explicit enumeration of all points. When considering only kernels with non-affine stack distance polynomials, we measure a geometric mean speedup of 12.4x with pieces

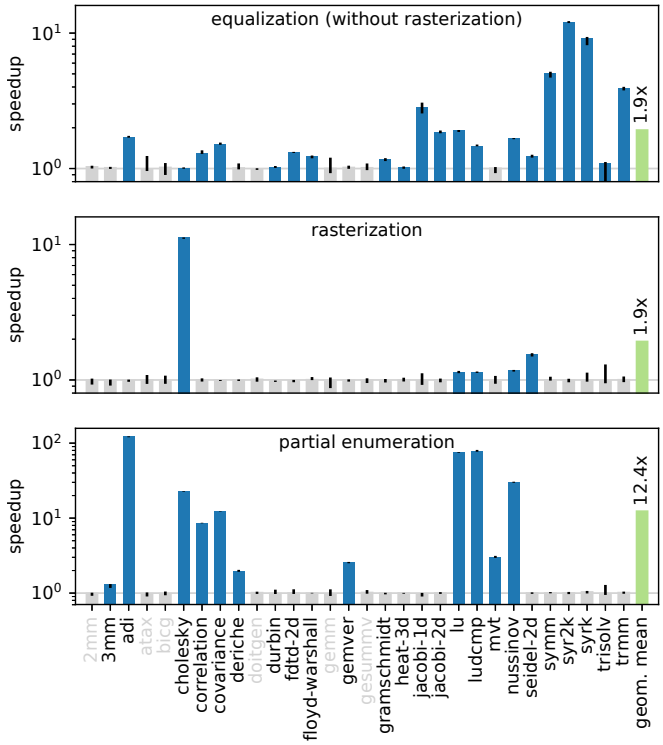


FIGURE 4.14: Speedup due to *equalization*, *rasterization*, and *partial enumeration*. All kernels without speedup (gray bars) are not included in the geometric mean. Only few kernels run fast without any optimization (gray labels).

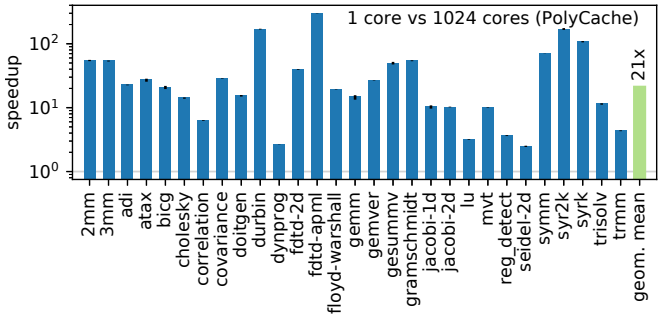
	3mm	adi	cholesky	correlation	covariance	deriche	gemver	lu	ludcmp	mvt	nussinov
0d-affine			3					4	3		7
1d-affine		7	11	3	3	6	4	48	52	2	18
2d-affine	1	59	85	1	1			27	20		48

TABLE 4.1: Number of non-affine polynomials with zero, one, or two affine dimensions.

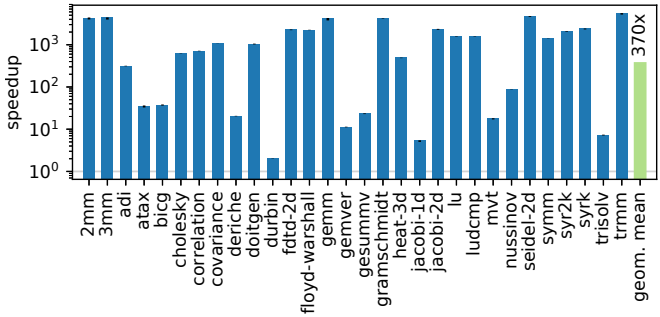
that contain 4,400 points on average. The more points per piece the bigger the efficiency gain due to our hybrid counting approach. We still require explicit enumeration for all non-affine polynomials without affine dimension. Table 4.1 shows that most non-affine polynomials have at least one affine dimension. For these polynomials, *partial enumeration* reduces the asymptotic complexity of the capacity miss counting.

As discussed by Section 4.2.3, the floor elimination techniques simplify non-affine stack distance polynomials with less splits than *partial enumeration* but are less generic and do not apply to all polynomials. Figure 4.14 shows the speedups for *equalization* compared to a baseline without *equalization* and *rasterization*. We disable both techniques since otherwise *rasterization* optimizes the polynomials normally handled by *equalization*. We observe a geometric mean speedup of 1.9x for the kernels that benefit. Figure 4.14 also compares the speedups for *rasterization* to a baseline without *rasterization*. We measure a geometric mean speedup of 1.9x for cholesky, lu, ludcmp, nussinov, and seidel-2d. Overall the floor elimination techniques reduce the number of counted pieces by more than 80% which results in bigger pieces with better counting performance.

A majority of the kernels perform well independent of problem size and number of cache hierarchy levels. Yet, the model execution times for kernels with non-affine polynomials are higher and problem size dependent. We mitigate this with efficient enumeration and floor elimination techniques.



(a) PolyCache



(b) Dinero IV

FIGURE 4.15: Speedup of HAYSTACK compared to PolyCache and Dinero for the PolyBench 3.2 and 4.2.1 kernels, respectively.

4.3.4 *Comparison to PolyCache and Dinero*

The polyhedral cache model PolyCache [97] and the cache simulator Dinero IV [86] are alternative cache modeling tools. We compare their performance to HAYSTACK .

PolyCache models set associative caches with an LRU replacement policy. We compare to the published results that show the performance for the default problem size of PolyBench 3.2 and adapt the configuration of our model to match the cache sizes of the published experiments (32KiB of L1 cache and 256KiB of L2 cache). The only difference is that we model fully associative caches instead of 4-way associative caches. Figure 4.15a shows an average speedup of 21x (geometric mean) of HAYSTACK compared to PolyCache even though PolyCache computes the cache misses for all 1024 cache sets in parallel.

Dinero IV is a trace driven cache simulator which means the expected simulation cost are proportional to the number of memory accesses (Figure 4.1). Figure 4.15b shows the speedup of HAYSTACK compared to the Dinero IV simulation times that include the trace generation with QEMU [98]. Dinero IV simulates the associativity of our test system while we model fully associative caches. As simulation and model run single core, the execution times are comparable. We measure an average speedup of 370x (geometric mean) for the large problem size that would be even bigger for the extra large problem size. Simulating full associativity further increases the average simulation time by factor 2.2x (geometric mean).

PolyCache models cache behavior in-depth, which allows developers to analyze the effects of set associativity and different write policies, but its high accuracy can make it costly to compute. Dinero IV works for small problem sizes but the cost increase for realistic problem sizes is dramatic.

4.3.5 *Performance for Tiled Codes*

A tiled code decomposes the iteration domain into tiles and executes tile-by-tile to improve the spacial locality. Tiling can double the loop nest depth which allows us to evaluate our approach for more complex codes. At the same time, estimating the benefits of tiling or even selecting optimal tile sizes is an important application for a cache model.

We employ the PPCG [15] source-to-source compiler to tile all PolyBench kernels with tile size 16. We limit the sum of all scheduling coefficients to one and disable loop fusion to obtain a rectangular tiling without loop

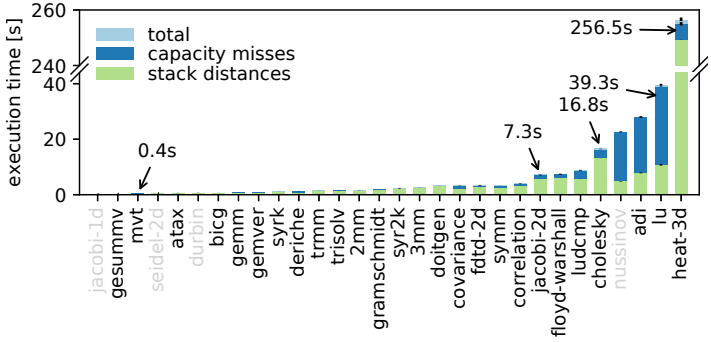


FIGURE 4.16: Execution times for the main components of HAYSTACK for tiled versions of the PolyBench kernels. A few kernels (gray labels) have no rectangular tiling.

skewing (time-tiling). All kernels except for jacobi-1d, durbin, seidel-2d, and nussinov have a rectangular tiling. Figure 4.16 shows the model execution times for the tiled kernels. Tiling makes the cache miss computation more expensive. Especially the stack distance computation of the head-3d kernel runs long. We attribute the cost increase to the more complex iteration domains and memory access patterns.

Tiling increases the model execution times but for a majority of the kernels the cache miss computation still takes only a few seconds.

4.4 RELATED WORK

Cache behavior analysis is a prerequisite when tuning for the memory hierarchy. We distinguish three main approaches: 1) simulation, 2) profiling, and 3) analytical modeling.

SIMULATORS Dinero [86] and CASPER [87] are examples of trace-based cache simulators that compute the cache misses for the full memory hierarchy. Sniper [88] and gem5 [89] have a broader scope and simulate the full system including the caches. All simulators execute the program to count the cache misses which means the simulation costs are proportional to the number of executed memory accesses.

PROFILING Multiple works discuss the analysis of memory access traces to extract locality metrics. Mattson et al. [82] compute the stack distance

using a linked list and derive the cache hit rate for different cache sizes. Tree based implementations [99–101] reduce the cost of the stack distance computation. Kim et al. [102] apply hashing and approximation to increase the efficiency. Ding et al. [84] discuss tree based approximate algorithms that reduce the time and space complexity of the stack distance computation and predict the stack distance histogram for arbitrary problem sizes given training inputs for few different problem sizes. Eklov et al. [103] sample the reuse distance for a few memory accesses and employ statistics to estimate stack distances and cache miss ratio. Xiang et al. [85] discuss five different locality metrics and show how to derive miss rate and reuse distance given the a single measure called average footprint which they compute with an efficient linear time algorithm [104]. A disadvantage of the profiling approaches is the acquisition and the handling of the large program traces. Chen et al. [105] sample the reuse time during compilation which allows them to estimate the cache miss ratio of complex loop nests.

ANALYTICAL MODELS Agarwal et al. [106] develop an analytical model that uses parameters extracted from the program trace. Harper et al. [107] model set associative caches for regular loop nests. Cost models [14, 36, 108] allow compilers to decide if data-locality transformations are beneficial. All of these models only approximate the number of cache misses.

Ferdinand et al. [109] use abstract interpretation to model set associative LRU caches. Model-checking [110, 111] increases the accuracy of this analysis that distinguishes always hit, always miss, and not classified. Touzeau et al. [112] show how to attain high accuracy without costly model-checking. The abstract interpretation approaches are complementary to our cache model since they support dynamic control flow but approximate the cache misses of loop nests by classifying all instances of a memory access at once.

Ghosh et al. [113] derive cache miss equations to count the cache misses for perfect loop nests with data dependencies represented by reuse vectors [27]. Assuming an LRU replacement policy, a cache miss occurs if the number of solutions to a cache miss equality exceeds the cache associativity. Counting the solutions for every point of the iteration domain is expensive. Vera and Xue et al. [114, 115] thus sample the iteration domain to speedup the cache miss computation which allows them to perform approximate whole-program analysis. Cascaval et al. [116] compute the stack distance histogram symbolically for perfect loop nests with uniform data dependencies. They model fully associative caches with an LRU replacement policy and use statistics to model set associative caches. Chatterjee et al. [117] use

Presburger formulas to express the set of compulsory and capacity misses of imperfect loop nests for associative caches. At the time, their approach was limited to small problem sizes and low associativity since the computation of analytical results for realistic hardware and even small benchmarks kernels was prohibitively complex. While Beyles et al. [90] did not address the cache miss problem, they use analytically computed stack distance to generate cache hints at runtime. Their stack distance computation, extended by our cache miss counting technique for non-affine polynomials, is the foundation of our cache model. PolyCache [97] presented the first analytical approach fast enough to compute the cache behavior of static control programs for interesting benchmark kernels and realistic hardware parameters. Its analytical model relates for every cache set successive accesses of distinct cache lines and repeatedly removes the shortest relations to model set associativity with LRU replacement policy. While PolyCache also uses symbolic counting techniques to avoid a complete enumeration of the computation, its complexity increases with high associativity. Our work provides a fast analytical model for fully associative caches and shows that fully associative models introduce only small errors compared to measurements on actual hardware.

4.5 SUMMARY OF THE APPROACH

As memory behavior depends on the cache state, understanding the cost of memory accesses is much more difficult than understanding the cost of arithmetic instructions. With HAYSTACK , we close this gap by providing developers with accurate information about the interaction of memory accesses with the large and deep cache hierarchy of modern processors. HAYSTACK allows the programmer to predict memory access costs accurately and to develop programs well optimized for the memory hierarchy. When striving for ultimate performance, both a good baseline and an accurate surrogate model accelerates empirical tuning. As a result, cache-aware program optimization becomes accessible.

Responsiveness is key for the adoption of any cache model. We demonstrate excellent often problem size independent response times that for the first time make analytical cache modeling practical. In addition, the cache size independent costs allow our model to easily scale to future hardware. We show the practicality of our deliberate decision against high fidelity and in favor of a generic fully associative cache model. The proposed model is robust to memory layout choices and hardware implementation details

and yet reaches very high accuracy on real hardware across a wide range of computations.

5

A COMMUNICATION-HIDING PROGRAMMING MODEL

Today we typically target GPU clusters using two programming models that separately deal with inter-node and single-node parallelization. For example, we may use MPI [118] to move data between nodes and CUDA [119] to implement the on-node computation. MPI provides point-to-point communication and collectives that allow synchronizing concurrent processes executing on different cluster nodes. Using a fork-join model, CUDA allows offloading compute kernels from the host to the massively parallel device. MPI-CUDA programs usually combine the two programming models by alternating between on-node kernel invocations and inter-node communication. While being functional, this approach also entails serious disadvantages.

The main disadvantage is that application developers need to know the concepts of both programming models and understand several intricacies to work around their inconsistencies. For example, the MPI software stack in meantime has been adapted to support direct device-to-device [120] data transfers. However, the control path remains on the host which causes frequent host-device synchronizations and redundant data structures on host and device.

On the other hand, the sequential execution of on-node computation and inter-node communication inhibits efficient utilization of the costly compute and network hardware. To mitigate this problem, application developers can implement manual overlap of computation and communication [10, 39]. In particular, there exist various approaches [121, 122] to overlap the communication with the computation on an inner domain that has no inter-node data dependencies. However, these code transformations significantly increase code complexity which results in reduced real-world applicability.

High-performance system design often involves trading off sequential performance against parallel throughput. The architectural difference between host and device processors perfectly showcases the two extremes of this design space. Both architectures have to deal with the latency of hardware components such as memories or floating point units. While the host processor employs latency minimization techniques such as prefetching and out-of-order execution, the device processor employs latency hiding techniques such as over-subscription and hardware threads.

To avoid the complexity of handling two programming models and to apply latency hiding at cluster scale, we introduce the DCUDA (distributed CUDA) programming model. We obtain a single coherent software stack by combining the CUDA programming model with a significant subset of the remote memory access capabilities of MPI [123]. More precisely, a global address space and device-side put and get operations enable transparent remote memory access using the high-speed network of the cluster. We thereby make use of over-decomposition to over-subscribe the hardware with spare parallelism that enables automatic overlap of remote memory accesses with concurrent computation. To synchronize the program execution, we additionally provide notified remote memory access operations [124] that after completion notify the target via notification queue.

We evaluate the DCUDA programming model using a stencil code, a particle simulation, and an implementation of sparse matrix-vector multiplication. To compare performance and usability, we implement DCUDA and MPI-CUDA versions of these mini-applications. Two out of three mini-applications show excellent automatic overlap of communication and computation. Hence, application developers not only benefit from the convenience of device-side remote memory access. More importantly, DCUDA enables automatic overlap of computation and communication without costly, manual code transformations. As DCUDA programs are less network latency sensitive, our development might even motivate more throughput oriented network designs. In brief, we make the following contributions:

- We implement the first device-side communication library that provides MPI like remote memory access operations and target notification for GPU clusters.
- We design the first GPU cluster programming model that makes use of over-subscription and hardware threads to automatically overlap inter-node communication with on-node computation.

5.1 PROGRAMMING MODEL

The CUDA programming model and the underlying hardware architecture have proven excellent efficiency for parallel compute tasks. To achieve high performance, CUDA programs offload the computation to an accelerator device with many throughput optimized compute cores that are over-subscribed with many more threads than they have execution units. To

overlap instruction pipeline latencies, in every clock cycle the compute cores try to select among all threads in flight some that are ready for execution. To implement context switches on a clock-by-clock basis, the compute cores split the register file and the scratchpad memory among the threads in flight. Hence, the register and scratchpad utilization of a code effectively limit the number of threads in flight. However, having enough parallel work is of key importance to fully overlap the instruction pipeline latencies. Little's law [125] states that this minimum required amount of parallel work corresponds to the product of bandwidth and latency. For example, we need 200kB of data on-the-fly to fully utilize a device memory with 200GB/s bandwidth and 1µs latency. Thereby, 200kB translate to roughly 12,000 threads in flight each of them accessing two double precision floating point values at once. We show in Section 5.3 that the network of our test system has 6GB/s bandwidth and 19µs latency. We therefore need 114kB of data or roughly 7,000 threads in flight to fully utilize the network. Based on the observation that typical CUDA programs make efficient use of the memory bandwidth, we conclude there should be enough parallelism to overlap network operations. Consequently, we suggest to use hardware supported overlap of computation and communication to program distributed memory systems.

5.1.1 *Distributed Memory*

One main challenge of distributed memory programming is the decomposition and distribution of program data to the different memories of the machine. Currently, most distributed memory programming models rely on manual domain decomposition and data synchronization since handling distributed memory automatically is hard.

Today, MPI is the most widely used distributed memory programming model in high-performance computing. Many codes thereby rely on two sided communication that simultaneously involves sender and receiver. This combination of data movement and synchronization is a bad fit for extending the CUDA programming model. On the one hand, CUDA programs typically perform many data movements in the form of device memory accesses before synchronizing the execution using global barriers. On the other hand, CUDA programs over-subscribe the device by running many more threads than there are hardware execution units. As sender and receiver might not be active at the same time, two sided communication is hardly practical. To avoid active target synchronization, MPI alternatively

provides one sided put and get operations that implement remote memory access [123] using a mapping of the distributed memory to a global address space. We believe that remote memory access programming is a natural extension of the CUDA programming model.

Finally, programming large distributed memory machines requires more sophisticated synchronization mechanisms than the barriers implemented by CUDA. We propose a notification based synchronization infrastructure [124] that after completion of the put and get operations enqueues notifications on the target. To synchronize, the target waits for incoming notifications enqueued by concurrent remote memory accesses. This queue based system enables to build linearizable semantics.

5.1.2 *Combining MPI & CUDA*

CUDA programs structure the computation using kernels, which embody phases of concurrent execution followed by result communication and synchronization. More precisely, kernels write to memory with relaxed consistency and only after an implicit barrier synchronization at the end of the kernel execution the results become visible to the entire device. To expose parallelism, kernels use a set of independent thread groups called blocks that provide memory consistency and synchronization among the threads of the block. The blocks are scheduled to the different compute cores of the device. Once a block is running its execution cannot be interrupted as todays devices do not support preemption [126]. While each compute core keeps as many blocks in flight as possible, the number of concurrent blocks is constraint by hardware limits such as register file and scratchpad memory capacity. Consequently, the block execution may be partly sequential and synchronizing two blocks might be impossible as one runs after the other.

MPI programs expose parallelism using multiple processes called ranks that may execute on the different nodes of a cluster. Similar to CUDA, we structure the computation in phases of concurrent execution followed by result communication and synchronization. In contrast to CUDA, we write the results to a distributed memory and we use either point-to-point communication or remote memory accesses to move data between the ranks.

Todays MPI-CUDA programs typically assign one rank to every device and whenever necessary insert communication in between kernel invocations. However, stacking the communication and synchronization mechanisms of two programming models makes the code unnecessarily

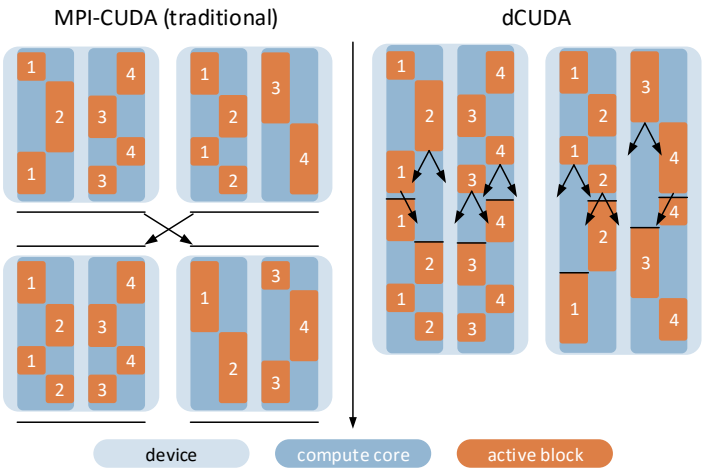


FIGURE 5.1: Block scheduling for MPI-CUDA and DCUDA.

complex. Therefore, we suggest to combine the two programming models into a single coherent software stack.

DCUDA programs implement the application logic using a single CUDA kernel that performs explicit data exchange during execution. To enable synchronization, we limit the over-subscription to the maximal number of concurrent hardware threads supported by the device. To move data between blocks no matter if they run on the same or on remote devices, we use device-side remote memory access operations. We identify each block with a unique rank identifier that allows to address data on the entire cluster. We map MPI ranks to CUDA blocks, as they represent the most coarse-grained execution unit that benefits from automatic latency overlap due to hardware threading. Hereafter, we use the terms rank and block interchangeably. To synchronize the rank execution, we implement remote memory access operations with target notification. An additional wait method finally allows to synchronize the target execution with incoming notifications.

Figure 5.1 compares the execution of an MPI-CUDA program to its DCUDA counterpart. We illustrate the program execution on two dual-core devices each of them over-subscribed with two blocks per core. We indicate communication using black arrows and synchronization using black lines. Both programs implement sequential compute and communication phases. While the DCUDA program uses over-subscription to automatically overlap

```
__shared__ dcuda_context ctx;
dcuda_init(param, ctx);
dcuda_comm_size(ctx, DCUDA_COMM_WORLD, &size);
dcuda_comm_rank(ctx, DCUDA_COMM_WORLD, &rank);

dcuda_win win, wout;
dcuda_win_create(ctx, DCUDA_COMM_WORLD,
    &in[0], len + 2 * jstride, &win);
dcuda_win_create(ctx, DCUDA_COMM_WORLD,
    &out[0], len + 2 * jstride, &wout);

bool lsend = rank - 1 >= 0;
bool rsend = rank + 1 < size;

int from = threadIdx.x + jstride;
int to = from + len;

for (int i = 0; i < steps; ++i) {
  for (int idx = from; idx < to; idx += jstride)
    out[idx] = -4.0 * in[idx] +
      in[idx + 1] + in[idx - 1] +
      in[idx + jstride] + in[idx - jstride];

  if (lsend)
    dcuda_put_notify(ctx, wout, rank - 1,
      len + jstride, jstride, &out[jstride], tag);
  if (rsend)
    dcuda_put_notify(ctx, wout, rank + 1,
      0, jstride, &out[len], tag);

  dcuda_wait_notifications(ctx, wout,
    DCUDA_ANY_SOURCE, tag, lsend + rsend);

  swap(in, out); swap(win, wout);
}

dcuda_win_free(ctx, win);
dcuda_win_free(ctx, wout);
dcuda_finish(ctx);
```

FIGURE 5.2: Stencil program with halo exchange communication.

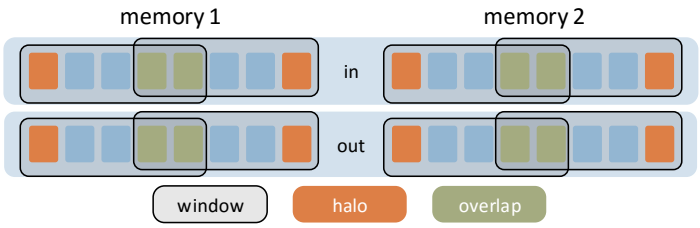


FIGURE 5.3: Overlapping windows of four ranks in two memories.

the communication and compute phases of competing blocks, the MPI-CUDA program leaves this optimization potential unused.

5.1.3 *Example*

Figure 5.2 shows an example program that uses DCUDA to implement a two-dimensional stencil computation. Using pointers adjusted to rank local memory, the program reads from an "in" array and writes to an "out" array. To distribute the work, the program performs a one-dimensional domain decomposition in the j-dimension. To satisfy all data dependencies, in every iteration the program exchanges one halo line with the left and right neighbor rank. For illustration purposes, the program listing highlights all methods and types implemented by the DCUDA framework. The calling conventions require that all threads of a block call the framework methods collectively with the same parameter values. To convert the example into a working program, we need additional boilerplate initialization logic that, among other things, performs the input/output and the domain decomposition.

On line 2, we initialize the context object using the "param" kernel parameter that contains framework information such as the notification queue address. The context object stores the shared state used by the framework methods.

On lines 3–4, we get size and identifier of the rank with respect to the world communicator. A communicator corresponds to a set of ranks that participate in the computation. Each rank has a unique identifier with respect to this communicator. We currently provide two predefined communicators that either represent all ranks of the cluster called "world communicator" or all ranks of the device called "device communicator".

On lines 6–10, we create two windows that provide remote memory access to the "in" and "out" arrays. When creating a window all participating ranks register their own local memory range with the window. The individual window sizes may differ and windows of shared memory ranks might overlap. We use windows to define a global address space where rank, window, offset tuples denote global distributed memory addresses. Figure 5.3 illustrates the overlapping windows of the example program. Each cell represents the memory of one j-position that stores "jstride" values in the i-dimension. Colors mark cells that belong to the domain boundaries of the rank. More precisely, the windows of shared memory ranks overlap and the windows of distributed memory ranks allocate additional halo cells that duplicate the domain boundaries.

On lines 24–30, we move the domain boundaries of the "out" array to the windows of the neighbor ranks. We address the remote memory using the window, rank, and offset parameters. Once the data transfer completes, the put operation additionally places a notification in the notification queue of the target rank. We can mark the notification with a tag that, in case of more complex communication patterns, allows disentangling different notification sources.

On lines 31–32, we wait until the notification of the neighboring ranks arrive in the notification queue. Our program waits for zero, one, or two notifications depending on the values of the lsend and rsend flags. We consider only notifications with specific window, rank, and tag values. To match multiple notifications at once, we can optionally use wildcard values that allow us to match notifications with any window, rank, or tag value.

On lines 37–39, we free the window objects to cleanup after the program execution. Overall, our implementation closely follows the MPI remote memory access specification [123]. On top of the functionality demonstrated by the example, we implement the window flush operation that allows to wait until all pending window operations are done. Furthermore, we cannot only put data to remote windows but also get data from remote windows. Finally, the barrier collective allows to globally synchronize the rank execution.

5.1.4 *Discussion*

Compared to MPI-CUDA, DCUDA slightly simplifies the code by moving the communication control to the device. For example, we have direct access to the size information of dynamic data structures and there is less need for

separate pack kernels that bundle data for the communication phase. While the distributed memory handling causes most of the code complexity for both programming models, with DCUDA we remove one synchronization layer and implement everything with a distributed memory view. We may thereby generate redundant put and get operations in shared memory, but our runtime can optimize them out.

5.2 IMPLEMENTATION

Moving the MPI functionality to the device-side raises multiple challenging implementation questions. To our knowledge so far there is no device-side MPI library, which might be partly attributed to the fact that calling MPI from the kernel conflicts with multiple CUDA mantras. On the one hand, the weak consistency memory model prevents shared memory communication during kernel execution. On the other hand, the missing block scheduling guarantees complicate the block synchronization.

5.2.1 *Architecture Overview*

Our research prototype consists of a device-side library that implements the actual programming interface and a host-side runtime system that controls the communication. More precisely, we run one library instance per rank and one runtime system instance per device. Connected via MPI, the runtime system instances control data movement and synchronization of any two ranks in the system. However, the data movement itself either takes place locally on the device or using direct device-to-device communication [120].

While a design without host involvement seems desirable, existing attempts to control the network interface card directly from the device are not promising [127] in terms of performance and system complexity. Furthermore, the host is a good fit for the synchronization required to order incoming notifications from different source ranks. To avoid device-side synchronization, we go even one step further and loop device local notifications through the host as well.

Moving data between ranks running on the same device requires memory consistency. In CUDA atomics are the only coherent memory operations at device-level. However, we did not encounter memory inconsistencies on Kepler devices with disabled L1 cache (which is the default setting). When

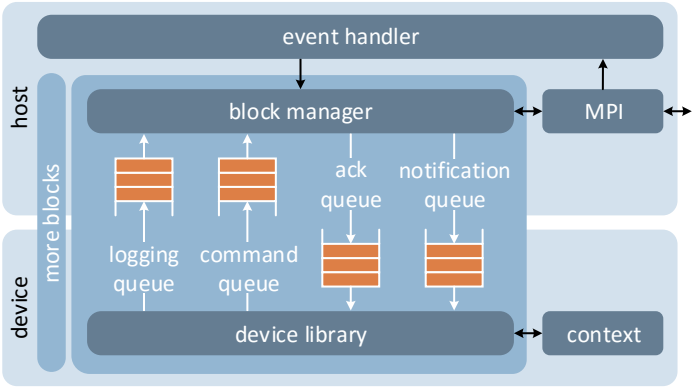


FIGURE 5.4: Architecture overview of the DCUDA runtime system.

polling device memory, we additionally use the volatile keyword to make sure the compiler issues a load instruction for every variable access.

To implement collectives such as barrier synchronization [128], all participating ranks have to be scheduled actively. Otherwise, the collective might deadlock. As discussed in Section 5.1.2, hardware constraints, such as the register file size and the lack of preemption, result in sequential block execution once we exceed the maximal number of concurrent hardware threads. Our implementation thus limits the number of blocks to the maximum the device can have in flight at once. However, we might still encounter starvation as there are no guarantees regarding the hardware thread scheduling implemented by the compute cores. For example, the compute cores might only run the threads that are waiting for notifications and pause the threads that send notifications.

Figure 5.4 illustrates the software architecture of the DCUDA runtime system. A host-side event handler starts and controls the execution of the actual compute kernel. To communicate with the blocks of the running kernel, we create separate block manager instances that interact with the device-side library components using queues implemented as circular buffers. The event handler dispatches incoming remote memory access requests to the matching target block manager and continuously invokes the block manager instances to process incoming commands and pending MPI requests. More precisely, using the command queue the device-side library triggers block manager actions such as window creation, notified remote memory access, and barrier synchronization. To guarantee progress using a single worker thread, the block manager implements these actions

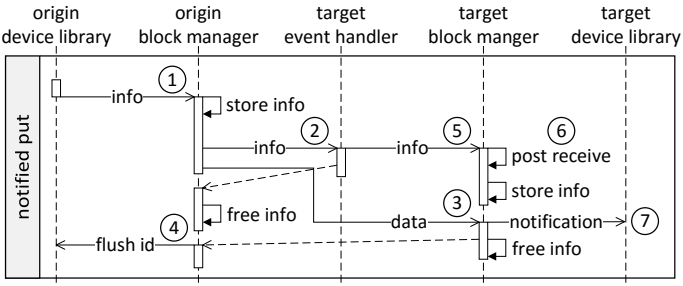


FIGURE 5.5: Sequence diagram of a notified distributed memory put.

using non-blocking MPI operations. Once the pending MPI request signals completion, the block manager notifies the device-side library using separate queues to acknowledge commands and to post notifications. An additional logging queue allows to print debug information during kernel execution. The device-side library uses a context object to store shared state such as queue or window information. Most of the times, the device-side library initiates actions on the host and waits for their completion. However, all remote memory accesses to shared memory ranks are directly executed on the device. We thereby perform no copy if source and target address of the remote memory access are identical, which commonly happens for overlapping shared memory windows. Furthermore, the device-side library implements the notification matching.

5.2.2 *Communication Control*

Figure 5.5 shows the end-to-end control flow for a distributed memory put operation with target notification. Initially, the origin device-library assembles a tuple containing data pointer, size, target rank, target window and offset, tag, and flush identifier of the transfer and 1) sends this meta information to the associated block manager. Using two non-blocking sends, 2) the origin block manager forwards the meta information to the target event handler and 3) copies the actual data directly from the origin device memory to the target device memory. Once the MPI requests signal that the send buffers are not in use anymore, 4) the origin block manager frees the meta information and updates the flush counter on the device. Using pre-posted receives, the target event handler waits for meta information arriving from an arbitrary origin rank and 5) immediately forwards the

incoming meta information to the associated target block manager. Finally, 6) the target block manager posts a non-blocking receive for the actual data transfer and 7) after completion notifies the target device-side library and frees the meta information.

The control flow for shared memory is simpler. Initially, the origin device-side library performs the actual data transfer. We thereby perform no copy in case source and target pointers are identical. Finally, we notify the target device-side library via the origin block manager.

The device-side library uses a counter to generate unique window identifiers. These counters get out of sync whenever only a subset of the ranks participate in the window creation. The block manager thus uses a hash map to translate the device-side identifiers to globally valid identifiers. Similarly, we use a counter to generate unique flush identifiers for remote memory access operations. The block manager keeps a history of the processed remote memory access operations and updates the device about the progress using a single variable set to the flush identifier of the last processed remote memory access operation whose predecessors are done as well.

5.2.3 *Performance Optimization*

As an efficient host-device communication is of key importance for the performance of our runtime system, we spend considerable effort in optimizing queue design and notification matching. Due to the Little's law assumption, we rather focus on throughput than on latency optimizations.

MEMORY MAPPING To move large amounts of data between host and device, the copy methods provided by the CUDA runtime are the method of choice. However, the DMA engine setup causes a considerable startup latency. Alternatively, we can directly access the device memory mapped in the address space of the host memory and vice versa. This approach is a much better fit for the small data transfers prevalent in queue operations. While CUDA out of the box provides support to map host memory in the device address space, we can map device memory in the host address space using an additional kernel module [129].

QUEUE DESIGN On todays machines the PCI-Express link between host and device is a major communication bottleneck. We therefore employ a circular buffer based queue design that provides an enqueue operation with

an amortized cost of a single PCI-Express transaction. To facilitate efficient polling, we place the circular buffer including its tail pointer in receiver memory. For example, Figure 5.4 shows that we allocate the notification queue in device memory and the command queue in host memory. To implement the enqueue operation using a single PCI-Express transaction, we embed an additional sequence number with every queue entry. The receiver then determines valid queue entries using the sequence number instead of the head pointer. Furthermore, we use a credit-based system to keep track of the available space. The sender starts with a free counter that is set to the queue size. With every enqueue operation, we decrement the free counter until it is zero. To recompute the available space, we then load the tail pointer from the receiver memory. The number of free counter updates depends on queue size and queue utilization. Overall, every enqueue operation requires one PCI-Express transaction to write the queue entry including its sequence number and an occasional PCI-Express transaction to update the free counter. We thereby assume queue entry accesses using a single vector instruction are atomic. On our test system, we never encountered inconsistencies when limiting the queue entry size to the vector instruction width.

NOTIFICATION MATCHING The notification matching is the most complex device-side library component. Two methods allow us to wait or test for a given number of incoming notifications. We can thereby filter the notifications depending on window identifier, source rank, and tag [124]. The matching happens in the order of arrival and after completion we remove the matched notifications. To fill potential gaps, we additionally compress the notification queue starting from the tail. Our implementation performs the matching using eight threads that work on separate four byte notification chunks. We read incoming notifications using coalesced reads and once the sequence number matches each thread compares the assigned notification chunk to a thread private query value. We initialize the query value depending on the thread index position with the window identifier, the source rank, the tag, or with a wild card value. To determine if the matching was successful, we reduce the comparison result using shuffle instructions. In case of a mismatch each thread buffers his notification chunk in a stack-allocated array. Otherwise, we increment a counter that keeps track of the successful matches. Finally, we remove the processed notification from the queue and repeat the procedure until we have enough successful matches. Once this is done, we copy the mismatched notifications

back from the stack-allocated array to the queue. We thereby assume the number of mismatched notifications is low enough for the stack-allocated array to fit in the L1 cache.

5.2.4 *Discussion*

To make our programming model production ready, additional modifications may be necessary. For example, we partly rely on undocumented hardware behavior and we could further optimize the performance. To develop a more reliable and efficient implementation, we suggest the following improvements to the CUDA environment.

SCHEDULING OF COMPUTATION & COMMUNICATION Our programming model packs the entire application logic in a single kernel. As discussed in Section 5.2.1, this approach conflicts with the scheduling guarantees and the weak memory consistency model of CUDA. For example, we might encounter starvation because the scheduler does not consider the ranks that are about to send notifications, or we might work with outdated data since there is no clean way to guarantee device-level memory consistency during kernel execution. We suggest an execution model with one master thread per rank that handles the communication using remote memory access and notifications. Similar to the dynamic parallelism feature of CUDA, the master thread additionally launches parallel compute phases in between the communication phases. To guarantee memory consistency, our execution model clears the cache before every compute phase. To prevent starvation, we suggest a yield call that guarantees execution time for all other ranks running on the same processing element. Hence, the master thread can yield the other ranks while waiting for incoming notifications. Currently, the compute phase with maximal register usage limits the available parallelism for the entire application. With the proposed execution model, we can adapt the number of threads for every compute phase and increase the overall resource usage.

NOTIFICATION SYSTEM An effective and low overhead notification system is crucial for the functioning of our programming model. Despite our optimization efforts, the current notification matching discussed in Section 5.2.3 increases register pressure and code complexity and consequently may impair the application performance. We suggest to at least partly integrate the notification infrastructure with the hardware. On the one hand, the

network may send data and notifications using a single transmission. Low level interfaces, such as uGNI [130] or InfiniBand Verbs, already provide the necessary support. On the other hand, the device may provide additional storage and logic for the notification matching or hardware support for on-chip notifications.

COMMUNICATION CONTROL While we move data directly from device-to-device, we still rely on the host to control the communication. We expect that moving this functionality to the device improves the overall performance of the system. Mellanox and NVIDIA recently announced a technology called GPUDirect Sync [131] that will enable device-side communication control.

5.3 EVALUATION

To analyze the performance of our programming model, we implement a set of microbenchmarks that measure latency, bandwidth, and the overlap of computation and communication for compute and memory bound tasks. We additionally compare the performance of mini-applications implemented using both DCUDA and MPI-CUDA.

5.3.1 *Experimental Setup & Methodology*

We perform all our experiments on the Greina compute cluster at the Swiss National Supercomputing Center CSCS. In total, Greina provides ten Haswell nodes equipped with one Tesla K80 GPU per node and connected via 4x EDR Infiniband. Furthermore, we use version 7.0 of the CUDA toolkit, the CUDA-aware version 1.10.0 of OpenMPI, and the gdrcopy kernel module [129]. We run all experiments using a single GPU per node with default device configuration. In particular, auto boost remains active which makes device-side time measurements unreliable.

To measure the performance of DCUDA and MPI-CUDA codes, we time the kernel invocations on the host-side and collect the maximum execution time found on the different nodes. In contrast, DCUDA programs pack the application code in a single kernel invocation that also contains a fair amount of setup code such as the window creation. To get a fair comparison, we therefore time multiple iterations and subtract the setup time estimated by running zero iterations. Furthermore, we repeat each time measurement multiple times and compute the median and the nonparametric confidence

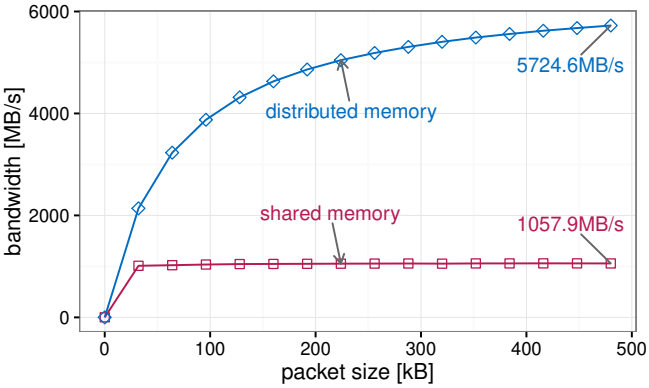


FIGURE 5.6: Put-bandwidth of shared and distributed memory ranks.

interval. More precisely, we perform 20 independent measurements of 100 and 5,000 iterations for the mini-applications and the microbenchmarks, respectively. Our plots visualize the 95% confidence interval using a gray band.

The DCUDA programming model focuses on multi-node performance. To eliminate measurement noise caused by single-node performance variations, we use the same launch configuration for all kernels (208 blocks per device and 128 threads per block), and we limit the register usage to 63 registers per thread which guarantees that all 208 blocks are in flight at once.

5.3.2 *Microbenchmarks*

To evaluate latency and bandwidth of our implementation, we run a ping-pong benchmark that in every iteration moves a data packet forth and back between two ranks using notified put operations. We either place the two ranks on the same device and communicate via shared memory, or we place the ranks on different devices and communicate via the network. We then derive the latency as half the execution time of a single ping-pong iteration and divide the packet size by the latency to compute the bandwidth. Figure 5.6 plots the put-bandwidth for shared and distributed memory ranks as function of the packet size. The low put-bandwidth of shared memory ranks can be explained by the fact that a single block cannot saturate the memory interface of the device. However, in real-world scenarios hundreds of blocks are active concurrently resulting in high

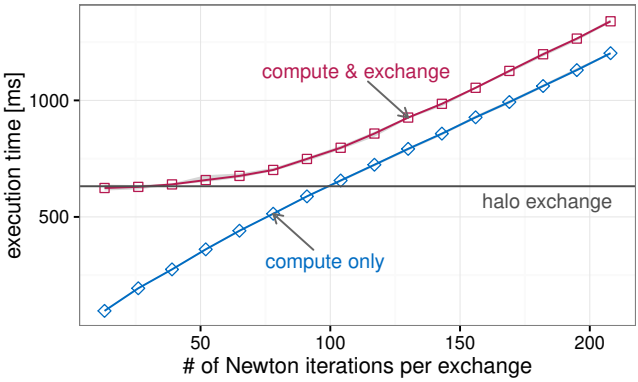


FIGURE 5.7: Overlap for square root calculation (Newton-Raphson).

aggregate bandwidth. For empty data packets, we measure a latency of 7.8µs and 19.3µs for shared and distributed memory, respectively. Hence, the latency of a notified put tops the device memory access latency [132] by one order of magnitude. We are aware that these numbers motivate further tuning. However, in the following we demonstrate that the DCUDA programming model is extremely latency agnostic.

The DCUDA programming model promises automatic overlap of computation and communication. To measure this effect, we design a benchmark that iteratively executes a compute phase followed by a halo exchange phase. To determine the overlap, we implement runtime switches that allow us to separately disable the compute and halo exchange phases. We use runtime switches to avoid code generation effects that might influence the overall performance of the benchmark. We expect that the execution time of the full benchmark varies between the maximum of compute and halo exchange time for perfect overlap and the sum of compute and halo exchange time for no overlap. To investigate the effect of different workloads, we additionally implement square root calculation (Newton-Raphson) and memory-to-memory copy as examples for compute-bound and memory bandwidth-bound computations. To demonstrate the overlap of computation and communication, Figure 5.7 and Figure 5.8 compare the execution time with and without halo exchange for increasing amounts of computation. An additional horizontal line marks the halo exchange only time. We run all experiments on eight nodes of our cluster. Each halo exchange moves 1kB packets, each copy iteration moves 1kB of data, and each square

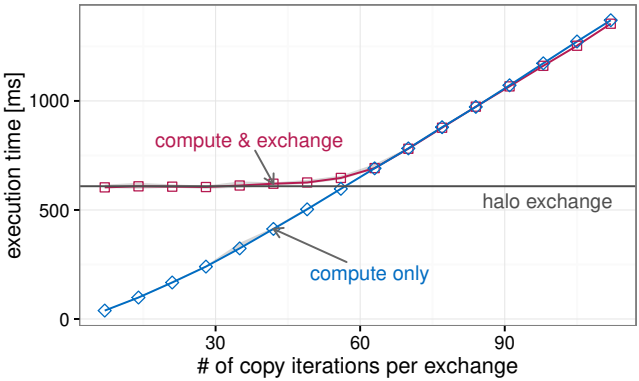


FIGURE 5.8: Overlap for memory-to-memory copy.

root iteration performs 128 divisions per rank. We measure perfect overlap for memory bandwidth-bound workloads and good overlap for compute-bound workloads. We explain the slightly lower overlap for compute-bound workloads by the fact that the notification matching itself is relatively compute heavy.

5.3.3 *Mini-applications*

To evaluate the absolute performance of our programming model, we compare MPI-CUDA and dCUDA variants of mini-applications that implement a particle simulation, a stencil program, and sparse matrix-vector multiplication. Three algorithmic motifs that are prevalent in high-performance computing. The main loops of the MPI-CUDA variants run on the host, invoke kernels, and communicate using two-sided MPI, while the main loops of the dCUDA variants run on the device and communicate using notified remote memory access. Otherwise, the implementation variants share the entire application logic and the overall structure. None of them implements manual overlap of computation and communication.

PARTICLE SIMULATION Our first mini-application simulates particles in a two-dimensional space that interact via short-range repulsive forces. We integrate the particle positions using simplified Verlet integration considering only forces between particles that are within a parameterizable cutoff distance. Just like the particle-in-cell method used for plasma simu-

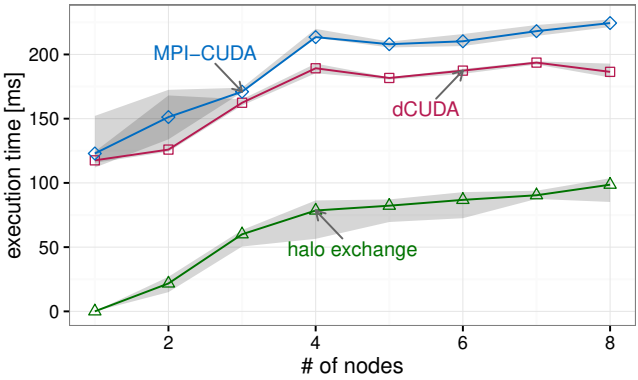


FIGURE 5.9: Weak scaling for the particle simulation example.

lations [133], we decompose our wide rectangular domain into cells that are aligned along the wide edge of the domain. Furthermore, we chose the cell width to be lower or equal to the cutoff distance and consequently only compute forces between particles that are either in the same cell or in neighboring cells. After each integration step we update the particle positions and move them to neighboring cells if necessary.

We organize the data using a structure of arrays that hold position, velocity, and acceleration of the particles. We thereby assign the cells to fixed-size, non-overlapping index ranges and use additional counters to keep track of the number of particles per cell. To deal with non uniform particle distributions among the cells, we allocate four times more storage than necessary to fit all particles. To support distributed memory, we decompose the arrays and allocate an additional halo cell at each sub-domain boundary.

The main loop of the particle simulation performs the following steps: 1) we perform a halo cell exchange between neighboring ranks, 2) we compute the forces and update the particle positions, 3) we sort out the particles that move to a neighbor cell, 4) we communicate the particles that move to a neighbor rank, and 5) we integrate the particles that arrived from a neighbor cell. To copy the minimal amount of data, the MPI-CUDA variant continuously fetches the book keeping counters to the host memory. In contrast, the main loop of the dCUDA variant runs on the device and has direct access to all data. Each rank registers one window per array that spans the cells assigned to the rank plus two halo cells. The windows of neighboring shared memory ranks physically overlap, which means,

as in case of the MPI-CUDA variant, actual data movement only takes place for distributed memory ranks. However, in contrast to MPI-CUDA the synchronization is much more fine-grained enabling overlap of computation and communication.

Figure 5.9 shows weak scaling for both implementation variants as well as the halo exchange time measured by the MPI-CUDA variant. We thereby use a constant workload of 416 cells and 41,600 particles per node. Typically, the simulation would be compute-bound, but as we are interested in communication we reduced the cutoff distance so that there are very few particle interactions. Consequently, the simulation becomes more memory bandwidth-bound. We perform two memory accesses in the innermost loop that computes the particles distances. Aggregated over 100 iterations and assuming a total execution time of 200ms, we get an estimated bandwidth requirement of roughly 100GB/s compared to 240GB/s peak bandwidth. This estimate shows that our implementation utilizes the available bandwidth well, especially considering the code also performs various other steps. While the two implementation variants perform similarly up to three nodes, the dCUDA variant clearly outperforms the MPI-CUDA variant for higher node counts. The scaling costs of the MPI-CUDA variant roughly correspond to the halo exchange time, while the dCUDA variant can partly overlap the halo exchange costs. However, the particle simulation is dynamic and during execution load imbalances evolve. For example, the minimal and maximal halo exchange times measured on eight nodes differ by a factor of two. We therefore do not expect an entirely flat scaling.

STENCIL PROGRAM Our second mini-application iteratively executes a simplified version [1] of the horizontal diffusion kernel derived from the COSMO atmospheric model [16]. The kernel consists of four dependent stencils that are applied to a three-dimensional regular grid with a limited number of vertical levels. The stencils themselves are small and consume between two and four neighboring points in the horizontal ij-plane.

Our implementation organizes the data using five three-dimensional arrays that are stored in column-major order. We perform a one-dimensional domain decomposition along the j-dimension and extend the sub-domains with a one-point halo in both j-directions. Consequently, the halos consist of one continuous storage segment per vertical k-level.

The main loop of the stencil program contains three compute phases each of them followed by a halo exchange. In total, we execute four stencils and communicate four one-point halos per loop iteration. To apply the stencils,

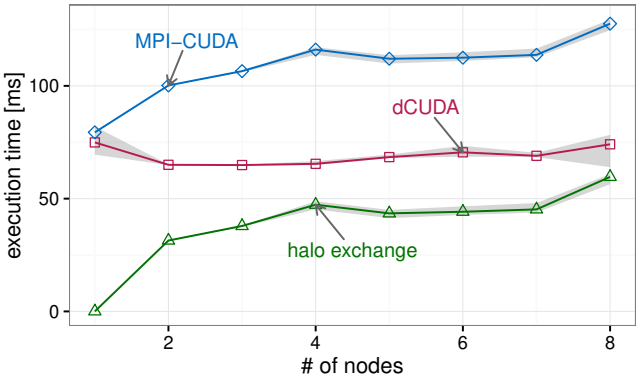


FIGURE 5.10: Weak scaling of the stencil program example.

we assign each block to an ij-patch that covers the full i-dimension. For each array, the dCUDA variant registers a window that spans the ij-patch assigned to the rank plus one halo line in each j-direction. The windows of neighboring shared memory ranks overlap and data movements only take place between distributed memory ranks. To improve the performance, the MPI-CUDA variant additionally copies the data to a continuous communication buffer that allows to wrap the entire halo exchange in a single message.

Figure 5.10 shows weak scaling for both implementation variants as well as the halo exchange time measured by the MPI-CUDA variant. We chose a domain size of $128 \times 320 \times 26$ grid points per device. The stencil program accesses eight different arrays per iteration. Aggregated over 100 iterations and assuming a total execution time of 70ms, we compute an approximate bandwidth requirement of 100GB/s compared to 240GB/s peak bandwidth. Hence, the overall performance of our implementation is reasonable. While both implementation variants have similar single-node performance, the dCUDA variant excels in multi-node setups. The scaling costs of the MPI-CUDA variant roughly correspond to the halo exchange time, while the dCUDA variant can completely overlap the significant halo exchange costs. This is possible since the stencil program is perfectly load balanced and the halo exchange costs are the only contribution the scaling costs. To achieve better bandwidth, OpenMPI by default stages messages larger than 30kB through the host. The MPI-CUDA variant sends one 26kB message per halo, while the dCUDA variant sends 26 separate 1kB messages (one per vertical

layer). Hence, with the given configuration both implementation variants perform direct device-to-device communication. However, introducing additional vertical layers improves the relative performance of the MPI-CUDA variant as it benefits from the higher bandwidth of host staged transfers.

SPARSE MATRIX-VECTOR MULTIPLICATION Our last mini-application implements sparse matrix-vector multiplication followed by a barrier synchronization. The barrier synchronization emulates possible follow up steps that synchronize the execution, the worst-case for DCUDA's overlap philosophy. For example, the normalization of the output vector performed by the power method.

We store the sparse matrix using the compressed row storage (CSR) format and distribute the data using a two-dimensional domain decomposition that splits the matrix into square sub-domains. Furthermore, we store the sparse input and output vectors along the first row and the first column of the domain decomposition, respectively. To process the matrix sub-domains, we assign each row of the matrix patch to exactly one block.

The main loop of the application performs the following steps: 1) we broadcast the input vector along the columns of the domain decomposition, 2) each rank locally computes the matrix-vector product, 3) we aggregate the result vectors along the rows of the domain decomposition, and 4) we synchronize the execution of all ranks. We manually implement the broadcast and reduction collectives using a binary tree communication pattern. The dCUDA variant over-decomposes the problem along the columns of the domain decomposition. Hence, the depth of the broadcast tree is higher, while the message size corresponds to the MPI-CUDA variant. In contrast, along the rows of the domain decomposition the reduction tree has the same depth while the dCUDA variant sends more but smaller messages.

Figure 5.11 shows the weak scaling for both implementation variants as well as the halo exchange time measured by the MPI-CUDA variant. We run our experiments using a 10,816×10,816 element matrix per device and randomly populate 0.3% of the elements. Our matrix-vector multiplication performs roughly a factor two slower than the cuSPARSE vendor library. While the MPI-CUDA variant performs slightly better for small node counts, the dCUDA variant seems to catch up for larger node counts. However, due to the tight synchronization, we do not observe relevant overlap of computation and communication. The scaling cost for both implementation variants corresponds roughly to the communication time. We therefore conjecture that the short and tightly synchronized compute phases provide

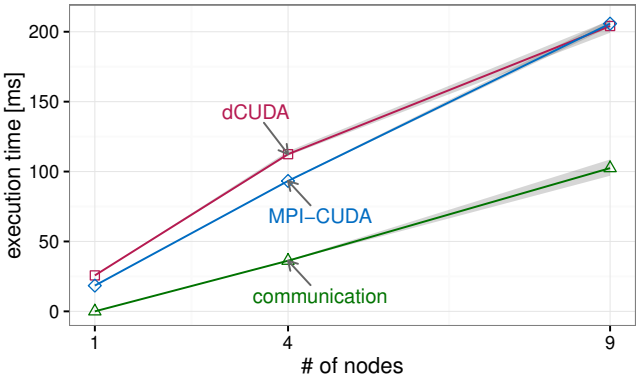


FIGURE 5.11: Weak scaling of the sparse matrix-vector example.

not enough room for overlap of computation and communication. Furthermore, the MPI-CUDA variant stages the reduction messages through the host, while the dCUDA variant due to the higher message rate uses direct device-to-device communication. Therefore, the dCUDA variant suffers from lower network bandwidth which might overcompensate potential latency hiding effects. We show this example to demonstrate that, even in the worst-case of very limited overlap, DCUDA performs comparable to MPI-CUDA. Advanced algorithmic methods could be used to enable automatic overlap even in Krylov subspace solvers [134].

5.4 DISCUSSION

To further improve the expressiveness and the performance of the DCUDA programming model, we briefly discuss possible enhancements.

COLLECTIVES Over-decomposition makes collectives more expensive as the their cost typically increase with the number of participating ranks. We suggest to implement highly-efficient collectives that leverage shared memory [135, 136]. Furthermore, one can imagine non-blocking collectives that run asynchronously in the background and notify the participating ranks after completion.

MULTI-DIMENSIONAL STORAGE Our implementation currently only supports one-dimensional storage similar to dynamically allocated memory

in C programs. We suggest to add support for multi-dimensional storage as it commonly appears in scientific applications. For example, we could provide a variant of the put method that copies a rectangular region of a two-dimensional array.

SHARED MEMORY With our programming model hundreds of ranks work on the same shared memory domain. We suggest to add functionality that makes better use of shared memory. For example, we could provide a variant of the put method that transfers data only once and then notifies all ranks associated to the target memory.

HOST RANKS To fully utilize the compute power of host and device, we suggest to extend our programming model with host ranks that like the device ranks communicate using notified remote memory access.

5.5 RELATED WORK

Over the past years, various GPU cluster programming models and approaches have been introduced. For example, the rCUDA [137] virtualization framework makes all devices of the cluster available to a single node. The framework intercepts calls to the CUDA runtime and transparently forwards them to the node that hosts the corresponding device. Consequently, CUDA programs that support multiple devices require no code changes to make use of the full cluster. Reaño et al. [138] provide an extensive list of the different virtualization frameworks currently available.

Multiple programming models provide some sort of device-side communication infrastructure. The FLAT [139] compiler transforms code with device-side MPI calls into traditional MPI-CUDA code with host-side MPI calls. Consequently, the approach provides the convenience of device-side MPI calls without actually implementing them. GPUNet [140] implements device-side sockets that enable MapReduce-style applications with the device acting as server for incoming requests. Key design choices of DCUDA, such as the circular buffer based host-device communication and the mapping of ranks to blocks, are inspired by GPUNet. DCGN [141] supports device-side as well as host-side compute kernels that communicate using message passing. To avoid device-side locking, the framework introduces the concept of slots that limit the maximum number of simultaneous communication requests. In a follow-up paper [142] the authors additionally discuss different rank to accelerator mapping options. GGAS [143] implements

device-side remote memory access using custom-built network adapters that enable device-to-device communication without host interaction. However, the programming model synchronizes the device execution before performing remote memory accesses and therefore prevents any hardware supported overlap of computation and communication. GPUrdma [144] was developed in parallel with DCUDA and implements device-side remote memory access over InfiniBand using firmware and driver modifications that enable device-to-device communication without host interaction.

Multiple works discuss technology aspects that are relevant for the programming model design. Oden et al. [127] control InfiniBand network adapters directly from the device without any host interaction. Their implementation relies on driver manipulations and system call interception. However, the host controlled communication nevertheless excels in terms of performance. Furthermore, Xiao and Feng [128] introduce device-side barrier synchronization and Tanasic et al. [126] discuss two different hardware preemption techniques.

5.6 SUMMARY OF THE APPROACH

With DCUDA we introduce a unified GPU cluster programming model that follows a latency hiding philosophy. We enhance the CUDA programming model with device-side remote memory access functionality. To hide memory and instruction pipeline latencies, CUDA programs over-decompose the problem and run many more threads than there are hardware execution units. In case there is enough spare parallelism, this technique enables efficient resource utilization. Using the same technique, DCUDA additionally hides the latency of remote memory access operations. Our experiments demonstrate the usefulness of the approach for mini-applications that implement different algorithmic motifs. We expect that real-word applications will draw significant benefit from automatic cluster-wide latency hiding and overlap of computation and communication. Especially since implementing manual overlap results in seriously increased code complexity. Overall, DCUDA stands for a paradigm shift away from coarse-grained sequential communication phases towards more fine-grained overlapped communication. Although the high message rate of fine-grained communication is clearly challenging for todays networks, our experiments show that the potential of the approach outweighs this drawback.

6

CONCLUSIONS AND FUTURE WORK

In this thesis, we present compiler framework, performance tool, and programming model advances that enable data movement optimizations. Our main contribution is the automatic end-to-end optimization of stencil programs. We automatically learn a performance model to perform fusion and tile size choices that minimize the predicted execution times. The presented techniques lay the foundation for domain-specific tools that automatically select good high-level code transformations while providing a smooth user experience similar to existing compilers. But automatic compilation does not work in all cases. We thus also discuss tool and programming model support to enable manual data movement optimizations.

Developing performance portable high-performance computing applications is a long-standing open problem. Data movement optimizations are an important aspect of performance portability. We contribute three projects to tackle the data movement challenge: 1) ABSINTHE is a stencil program optimizer that learns performance characteristics of the target system, 2) HAYSTACK is an analytical cache model that helps programmers to understand the cost of data movement, and 3) DCUDA is a communication-hiding programming model that enables the automatic overlap of computation and inter-node communication. All of our tools support the development of performance portable applications. But none of them solves the full problem since our tools are either domain-specific or solve only a subproblem. Solving the general problem independent of the application domain remains challenging. We believe domain-specific approaches are the method of choice to perform target system-specific code transformations for the time being.

6.1 FUTURE WORK

Our work motivates further research mainly to extend the scope of our tools to a broader class of programs. A prominent research direction is the development of compilers that learn a performance model to guide the selection of data-locality transformations.

6.1.1 *Automatic Performance Model Design*

ABSINTHE learns parameters that model latency and throughput of the innermost loop executions. The approach works well for the tested architectures. But some architectures may require performance model adaptations that go beyond choosing different parameters. For example, modeling the available parallelism is essential for architectures that hide the memory access latency using over-subscription and hardware threading. In these cases, learning the entire performance model function and not only parameters or weights is desirable – especially considering the increasingly diverse space of hardware architectures.

The manual performance model design is challenging. We typically perform three main steps: 1) identify the program features that determine the execution time, 2) identify the hardware components that limit the execution time, and 3) derive a function that estimates the execution time given the program features and the performance characteristics of the relevant hardware components. The learning of the performance characteristics may also require the design of benchmarks that stress the individual hardware components. The key to a successful performance model design is then to find the right balance of model accuracy and model complexity. Repeating this process for many different hardware architectures is very time-consuming. We thus believe an automatic performance model design will enhance the model availability and improve the model quality.

The rise of machine learning over the last decade further motivates the automatic performance model learning. Modern machine learning algorithms – provided sufficient amounts of data – demonstrate excellent results for many challenging tasks. The ability of automatic optimization frameworks to enumerate a lot of implementation variants will help us to collect enough data to enable the automatic performance model learning. Adams et al. [25] sample random programs to learn an artificial neuronal network that derives weight coefficients for twenty-seven hand-designed performance model terms. Their approach is an important step towards learning the entire performance model function but still relies on the manual design of the individual performance model terms. Once efficient learning techniques are in place, they will quickly supersede the manual performance model design.

6.1.2 *Scaling to Real-World Applications*

We evaluated our tools on benchmark kernels that perform significant amounts of work but that are small compared to real-world applications. Scaling our stencil optimizer ABSINTHE or the analytical cache model HAYSTACK to full applications is challenging due to their computational complexity. Scaling our tools to large programs is thus an interesting research direction.

Splitting the applications into smaller kernels and processing the individual kernels is one way of dealing with large programs. The approach sacrifices optimality, but if the kernels are well-chosen, the results should remain relevant. Often the codes anyways contain natural optimization barriers such as calls to external libraries to perform I/O or inter-node communication. If this is not the case, then methods that split programs into smaller pieces that can be optimized well are of interest. We could, for example, consider the data flow graph between the loop nests and search splits with minimal data-flow.

DYNAMIC PROGRAMMING The space of possible data-locality transformations is exponential in the number of stencils. Instead of optimizing the whole program, we may compute optimal fusion and tile size choices for kernels up to a maximal size and combine the results using dynamic programming. The approach limits fusion to the maximal kernel size but otherwise produces optimal results.

SURROGATE MODEL Presburger arithmetic is known to be computationally expensive. Instead of running the cache model directly on large programs, we may thus learn a surrogate model. The cache model allows us to generate a lot of training data for different programs and problem sizes. We could use this data to train a neuronal network that approximates the cache misses given input program and problem size. The approach sacrifices accuracy to speedup the cache miss computation.

6.1.3 *Cache Optimal Programs*

Already the current version of HAYSTACK can be used to derive cache optimal programs. But there are two areas of possible improvement: 1) the model execution times hinder the exhaustive search space exploration and 2) the model does not consider the parallel program execution.

PARAMETRIC STACK DISTANCE An automatic optimization framework does not necessarily need to evaluate the entire cache model. Instead, we can compute the stack distance for all data dependencies local to a tile and its neighbor tiles. We then introduce optimization constraints that ensure the stack distances are lower than or equal to the cache size. The solution guarantees all data dependencies local to the tile and its neighbor tiles are in cache. The advantage of the approach is that the stack distance can be computed parametric in the tile size. The parametric formulation has the potential to accelerate the tile size optimization significantly. We thus believe the stack distance is an interesting measure for future optimization frameworks.

PARALLEL CACHE MODEL Modeling the cache behavior of parallel programs is a long-standing open problem. We may use a sequential cache model to analyze the memory accesses of individual tiles that execute on a single core. But the sequential analysis fails to detect the data movement between tiles that execute in parallel on different cores. Unwanted effects such as false sharing – two cores access seemingly different data stored by the same cache line – may result in cache line ping-pong and other effects that hider the parallel execution. A parallel cache model can capture such effects but typically faces the challenge that the parallel schedule is unknown. Assuming we know the schedule, the model can compute the stack distance to detect the cache misses and on top identify the data movement between different cores. We can then repeat the analysis for several random schedules or determine a worst-case (fully parallel schedule) and a best-case (sequential schedule) scenario to gather statistics for the extreme cases. If the inter-core data movement heavily depends on the schedule, the program likely suffers from unwanted parallelization effects such as cache line ping-pong. A tool that helps programmers to detects such problems would be an important contribution to tackle the data movement challenge.

BIBLIOGRAPHY

1. Gysi, T., Grosser, T. & Hoefler, T. *MODESTO: Data-centric Analytic Optimization of Complex Stencil Programs on Heterogeneous Architectures* in *Proceedings of the 29th ACM on International Conference on Supercomputing* (ACM, Newport Beach, California, USA, 2015), 177. http://doi.acm.org/10.1145/2751205.2751223 (cit. on pp. ix, 37, 63, 116).
2. Gysi, T., Grosser, T. & Hoefler, T. *Absinthe: Learning an Analytical Performance Model to Fuse and Tile Stencil Codes in One Shot* Accepted at the 28th International Conference on Parallel Architectures and Compilation Techniques. 2019 (cit. on p. ix).
3. Gysi, T., Grosser, T., Brandner, L. & Hoefler, T. *A Fast Analytical Model of Fully Associative Caches* in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (ACM, Phoenix, AZ, USA, 2019), 816. http://doi.acm.org/10.1145/3314221.3314606 (cit. on p. ix).
4. Gysi, T., Bär, J. & Hoefler, T. *dCUDA: Hardware Supported Overlap of Computation and Communication* in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (IEEE Press, Salt Lake City, Utah, 2016), 52:1. http://dl.acm.org/citation.cfm?id=3014904.3014974 (cit. on p. ix).
5. Gysi, T., Osuna, C., Fuhrer, O., Bianco, M. & Schulthess, T. C. *STELLA: A Domain-specific Tool for Structured Grid Methods in Weather and Climate Models* in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (ACM, Austin, Texas, 2015), 41:1. http://doi.acm.org/10.1145/2807591.2807627 (cit. on pp. ix, 1, 63).
6. Fuhrer, O., Osuna, C., Lapillonne, X., Gysi, T., Cumming, B., Bianco, M., Arteaga, A. & Schulthess, T. Towards a Performance Portable, Architecture Agnostic Implementation Strategy for Weather and Climate Models. *Supercomput. Front. Innov.: Int. J.* **1**, 45. http://dx.doi.org/10.14529/jsfi140103 (2014) (cit. on pp. ix, 1, 10, 28).

7. Unat, D., Dubey, A., Hoefler, T., Shalf, J., Abraham, M., Bianco, M., Chamberlain, B. L., Cledat, R., Edwards, H. C., Finkel, H., Fuerlinger, K., Hannig, F., Jeannot, E., Kamil, A., Keasler, J., Kelly, P. H. J., Leung, V., Ltaief, H., Maruyama, N., Newburn, C. J. & Pericás, M. Trends in Data Locality Abstractions for HPC Systems. *IEEE Transactions on Parallel and Distributed Systems* **28**, 3007 (2017) (cit. on pp. 1, 37).

8. Zhou, X., Giacalone, J.-P., Garzarán, M. J., Kuhn, R. H., Ni, Y. & Padua, D. *Hierarchical Overlapped Tiling* in *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (ACM, San Jose, California, 2012), 207. http://doi.acm.org/10.1145/2259016.2259044 (cit. on pp. 1, 2, 14, 35, 40, 55).

9. Holewinski, J., Pouchet, L.-N. & Sadayappan, P. *High-performance Code Generation for Stencil Computations on GPU Architectures* in *Proceedings of the 26th ACM International Conference on Supercomputing* (ACM, San Servolo Island, Venice, Italy, 2012), 311. http://doi.acm.org/10.1145/2304576.2304619 (cit. on pp. 1, 14, 34).

10. Shimokawabe, T., Aoki, T., Muroi, C., Ishida, J., Kawano, K., Endo, T., Nukada, A., Maruyama, N. & Matsuoka, S. *An 80-Fold Speedup, 15.0 TFlops Full GPU Acceleration of Non-Hydrostatic Weather Model ASUCA Production Code* in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (IEEE Computer Society, Washington, DC, USA, 2010), 1. https://doi.org/10.1109/SC.2010.9 (cit. on pp. 1, 7, 97).

11. Tang, Y., Chowdhury, R. A., Kuszmaul, B. C., Luk, C.-K. & Leiserson, C. E. *The Pochoir Stencil Compiler* in *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures* (ACM, San Jose, California, USA, 2011), 117. http://doi.acm.org/10.1145/1989493.1989508 (cit. on pp. 1, 9, 63).

12. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F. & Amarasinghe, S. *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* (ACM, Seattle, Washington, USA, 2013), 519. http://doi.acm.org/10.1145/2491956.2462176 (cit. on pp. 1, 2, 9, 10, 14, 35, 37, 61, 63).

13. Mullapudi, R. T., Vasista, V. & Bondhugula, U. *PolyMage: Automatic Optimization for Image Processing Pipelines* in *Proceedings of the Twentieth International Conference on Architectural Support for Programming*

Languages and Operating Systems (ACM, Istanbul, Turkey, 2015), 429. http://doi.acm.org/10.1145/2694344.2694364 (cit. on pp. 1, 2, 35, 37, 61, 63).

14. Bondhugula, U., Hartono, A., Ramanujam, J. & Sadayappan, P. *A Practical Automatic Polyhedral Parallelizer and Locality Optimizer* in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (ACM, Tucson, AZ, USA, 2008), 101. http://doi.acm.org/10.1145/1375581.1375595 (cit. on pp. 1, 94).

15. Verdoolaege, S., Carlos Juega, J., Cohen, A., Ignacio Gómez, J., Tenllado, C. & Catthoor, F. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.* **9**, 54:1. http://doi.acm.org/10.1145/2400682.2400713 (2013) (cit. on pp. 1, 92).

16. Baldauf, M., Seifert, A., Förstner, J., Majewski, D., Raschendorfer, M. & Reinhardt, T. Operational Convective-Scale Numerical Weather Prediction with the COSMO Model: Description and Sensitivities. *Monthly Weather Review* **139**, 3887 (2011) (cit. on pp. 1, 37, 55, 58, 116).

17. *Consortium for Small-scale Modeling* http://www.cosmo-model.org/ (2018) (cit. on pp. 1, 37).

18. Fuhrer, O., Chadha, T., Hoefler, T., Kwasniewski, G., Lapillonne, X., Leutwyler, D., Lüthi, D., Osuna, C., Schär, C., Schulthess, T. C. & Vogt, H. Near-global climate simulation at 1 km resolution: establishing a performance baseline on 4888 GPUs with COSMO 5.0. *Geoscientific Model Development* **11**, 1665. https://www.geosci-model-dev.net/11/1665/2018/ (2018) (cit. on pp. 1, 37).

19. Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U.-M. & Amarasinghe, S. *OpenTuner: An Extensible Framework for Program Autotuning* in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation* (ACM, Edmonton, AB, Canada, 2014), 303. http://doi.acm.org/10.1145/2628071.2628092 (cit. on p. 2).

20. Ţapuş, C., Chung, I.-H. & Hollingsworth, J. K. *Active Harmony: Towards Automated Performance Tuning* in *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing* (IEEE Computer Society Press, Baltimore, Maryland, 2002), 1. http://dl.acm.org/citation.cfm?id=762761.762771 (cit. on p. 2).

21. Tiwari, A., Chen, C., Chame, J., Hall, M. & Hollingsworth, J. K. *A scalable auto-tuning framework for compiler optimization* in *2009 IEEE International Symposium on Parallel Distributed Processing* (2009), 1 (cit. on p. 2).

22. Christen, M., Schenk, O. & Burkhart, H. *PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures* in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium* (IEEE Computer Society, Washington, DC, USA, 2011), 676. https://doi.org/10.1109/IPDPS.2011.70 (cit. on pp. 2, 9, 34, 63).

23. Mullapudi, R. T., Adams, A., Sharlet, D., Ragan-Kelley, J. & Fatahalian, K. Automatically Scheduling Halide Image Processing Pipelines. *ACM Trans. Graph.* **35**, 83:1. http://doi.acm.org/10.1145/2897824.2925952 (2016) (cit. on pp. 2, 61, 63).

24. Low, T. M., Igual, F. D., Smith, T. M. & Quintana-Orti, E. S. Analytical Modeling Is Enough for High-Performance BLIS. *ACM Trans. Math. Softw.* **43**, 12:1. http://doi.acm.org/10.1145/2925987 (2016) (cit. on p. 2).

25. Adams, A., Ma, K., Anderson, L., Baghdadi, R., Li, T.-M., Gharbi, M., Steiner, B., Johnson, S., Fatahalian, K., Durand, F. & Ragan-Kelley, J. Learning to Optimize Halide with Tree Search and Random Programs. *ACM Trans. Graph.* **38**, 121:1. http://doi.acm.org/10.1145/3306346.3322967 (2019) (cit. on pp. 2, 124).

26. McKinley, K. S., Carr, S. & Tseng, C.-W. Improving Data Locality with Loop Transformations. *ACM Trans. Program. Lang. Syst.* **18**, 424. http://doi.acm.org/10.1145/233561.233564 (1996) (cit. on p. 2).

27. Wolf, M. E. & Lam, M. S. *A Data Locality Optimizing Algorithm* in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation* (ACM, Toronto, Ontario, Canada, 1991), 30. http://doi.acm.org/10.1145/113445.113449 (cit. on pp. 2, 94).

28. Chen, C., Chame, J. & Hall, M. *CHiLL: A framework for composing high-level loop transformations* tech. rep. (2008) (cit. on p. 2).

29. Bastoul, C., Cohen, A., Girbal, S., Sharma, S. & Temam, O. *Putting Polyhedral Loop Transformations to Work* in *LCPC* (2003) (cit. on p. 2).

30. Baghdadi, R., Ray, J., Romdhane, M. B., Del Sozzo, E., Akkas, A., Zhang, Y., Suriana, P., Kamil, S. & Amarasinghe, S. *Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code* in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (IEEE Press, Washington, DC, USA, 2019), 193. http://dl.acm.org/citation.cfm?id=3314872.3314896 (cit. on p. 2).

31. Steuwer, M., Fensch, C., Lindley, S. & Dubach, C. Generating Performance Portable Code Using Rewrite Rules: From High-level Functional Expressions to High-performance OpenCL Code. *SIGPLAN Not.* **50**, 205. http://doi.acm.org/10.1145/2858949.2784754 (2015) (cit. on p. 2).

32. Puschel, M., Moura, J. M. F., Johnson, J. R., Padua, D., Veloso, M. M., Singer, B. W., Jianxin Xiong, Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R. W. & Rizzolo, N. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE* **93**, 232 (2005) (cit. on p. 2).

33. Spampinato, D. G. & Püschel, M. *A Basic Linear Algebra Compiler* in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (ACM, Orlando, FL, USA, 2014), 23:23. http://doi.acm.org/10.1145/2581122.2544155 (cit. on p. 2).

34. Kennedy, K. Fast Greedy Weighted Fusion. *Int. J. Parallel Program.* **29**, 463. https://doi.org/10.1023/A:1012241830762 (2001) (cit. on p. 2).

35. Wahib, M. & Maruyama, N. *Scalable Kernel Fusion for Memory-bound GPU Applications* in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (IEEE Press, New Orleans, Louisana, 2014), 191. https://doi.org/10.1109/SC.2014.21 (cit. on pp. 2, 9, 35).

36. Kennedy, K. & McKinley, K. S. *Optimizing for Parallelism and Data Locality* in *Proceedings of the 6th International Conference on Supercomputing* (ACM, Washington, D. C., USA, 1992), 323. http://doi.acm.org/10.1145/143369.143427 (cit. on pp. 3, 94).

37. Bondhugula, U., Baskaran, M., Krishnamoorthy, S., Ramanujam, J., Rountev, A. & Sadayappan, P. *Automatic Transformations for Communication-minimized Parallelization and Locality Optimization in the Polyhedral Model* in *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction* (Springer-Verlag, Budapest, Hungary, 2008), 132.

http://dl.acm.org/citation.cfm?id=1788374.1788386 (cit. on p. 3).

38. IBM. *IBM ILOG CPLEX Optimization Studio CPLEX User's Manual* (2011) (cit. on pp. 4, 55).

39. Phillips, J. C., Stone, J. E. & Schulten, K. *Adapting a Message-driven Parallel Application to GPU-accelerated Clusters* in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (IEEE Press, Austin, Texas, 2008), 8:1. http://dl.acm.org/citation.cfm?id=1413370.1413379 (cit. on pp. 7, 97).

40. Hong, S. & Kim, H. An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. *SIGARCH Comput. Archit. News* **37**, 152. http://doi.acm.org/10.1145/1555815.1555775 (2009) (cit. on p. 7).

41. Doms, G. & Schättler, U. *The nonhydrostatic limited-area model LM (Lokal-Model) of the DWD. Part I: Scientific documentation* tech. rep. (German Weather Service (DWD), Germany, 1999). http://www.cosmo-model.org/ (cit. on pp. 9, 11, 13).

42. McMECHAN, G. A. Migration by extrapolation of time-dependent boundary values*. *Geophysical Prospecting* **31**, 413. http://dx.doi.org/10.1111/j.1365-2478.1983.tb01060.x (1983) (cit. on pp. 9, 37).

43. Taflove, A. Review of the formulation and applications of the finite-difference time-domain method for numerical modeling of electromagnetic wave interactions with arbitrary structures. *Wave Motion* **10**. Special Issue on Numerical Methods for Electromagnetic Wave Interactions, 547. http://www.sciencedirect.com/science/article/pii/0165212588900121 (1988) (cit. on pp. 9, 37).

44. Basu, P., Venkat, A., Hall, M., Williams, S., Van Straalen, B. & Oliker, L. *Compiler generation and autotuning of communication-avoiding operators for geometric multigrid* in *20th Annual International Conference on High Performance Computing* (2013), 452 (cit. on pp. 9, 35).

45. Frigo, M. & Strumpen, V. The Memory Behavior of Cache Oblivious Stencil Computations. *J. Supercomput.* **39**, 93. http://dx.doi.org/10.1007/s11227-007-0111-y (2007) (cit. on pp. 9, 34).

46. Olschanowsky, C., Strout, M. M., Guzik, S., Loffeld, J. & Hittinger, J. *A Study on Balancing Parallelism, Data Locality, and Recomputation in Existing PDE Solvers* in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (IEEE Press, New Orleans, Louisana, 2014), 793. https://doi.org/10.1109/SC.2014.70 (cit. on pp. 9, 35).

47. Williams, S., Waterman, A. & Patterson, D. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* **52**, 65. http://doi.acm.org/10.1145/1498765.1498785 (2009) (cit. on p. 17).

48. Verdoolaege, S. *isl: An Integer Set Library for the Polyhedral Model* in *Proceedings of the Third International Congress Conference on Mathematical Software* (Springer-Verlag, Kobe, Japan, 2010), 299. http://dl.acm.org/citation.cfm?id=1888390.1888455 (cit. on pp. 20, 68, 75).

49. Verdoolaege, S., Seghir, R., Beyls, K., Loechner, V. & Bruynooghe, M. Counting Integer Points in Parametric Polytopes Using Barvinok's Rational Functions. *Algorithmica* **48**, 37. http://dx.doi.org/10.1007/s00453-006-1231-0 (2007) (cit. on pp. 20, 65, 75, 88).

50. Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J. & Yelick, K. *Stencil Computation Optimization and Auto-tuning on State-of-the-art Multicore Architectures* in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (IEEE Press, Austin, Texas, 2008), 4:1. http://dl.acm.org/citation.cfm?id=1413370.1413375 (cit. on p. 34).

51. Zhang, Y. & Mueller, F. Autogeneration and Autotuning of 3D Stencil Codes on Homogeneous and Heterogeneous GPU Clusters. *IEEE Transactions on Parallel and Distributed Systems* **24**, 417 (2013) (cit. on p. 34).

52. Shirako, J., Sharma, K., Fauzia, N., Pouchet, L.-N., Ramanujam, J., Sadayappan, P. & Sarkar, V. *Analytical Bounds for Optimal Tile Size Selection* in *Proceedings of the 21st International Conference on Compiler Construction* (Springer-Verlag, Tallinn, Estonia, 2012), 101. http://dx.doi.org/10.1007/978-3-642-28652-0_6 (cit. on pp. 35, 62).

53. Renganarayana, L. & Rajopadhye, S. *Positivity, Posynomials and Tile Size Selection* in *ACM/IEEE Conf. on Supercomputing, Proc. of* (IEEE Press, Austin, Texas, 2008), 55:1. http://dl.acm.org/citation.cfm?id=1413370.1413426 (cit. on p. 35).

54. Cade, B. S. & Richards, J. D. Permutation Tests for Least Absolute Deviation Regression. *Biometrics* **52**, 886 (1996) (cit. on pp. 50, 58).

55. *ILP Part 6 – Faster multiplication* https://blog.adamfurmanek.pl/2015/09/26/ilp-part-6/. 2015 (cit. on p. 51).

56. Hoefler, T. & Belli, R. *Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses when Reporting Performance Results* in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (ACM, Austin, Texas, 2015), 73:1. http://doi.acm.org/10.1145/2807591.2807644 (cit. on pp. 55, 84).

57. Pouchet, L.-N. *Polybench: The polyhedral benchmark suite* https://sourceforge.net/projects/polybench/. 2012 (cit. on pp. 58, 84).

58. Jangda, A. & Bondhugula, U. *An Effective Fusion and Tile Size Model for Optimizing Image Processing Pipelines* in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (ACM, Vienna, Austria, 2018), 261. http://doi.acm.org/10.1145/3178487.3178507 (cit. on pp. 61, 63).

59. Lam, M. D., Rothberg, E. E. & Wolf, M. E. *The Cache Performance and Optimizations of Blocked Algorithms* in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (ACM, Santa Clara, California, USA, 1991), 63. http://doi.acm.org/10.1145/106972.106981 (cit. on p. 62).

60. Dongarra, J. & Schreiber, R. *Automatic Blocking of Nested Loops* tech. rep. (Knoxville, TN, USA, 1990) (cit. on p. 62).

61. Coleman, S. & McKinley, K. S. *Tile Size Selection Using Cache Organization and Data Layout* in *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation* (ACM, La Jolla, California, USA, 1995), 279. http://doi.acm.org/10.1145/207110.207162 (cit. on p. 62).

62. Hsu, C.-h. & Kremer, U. A Quantitative Analysis of Tile Size Selection Algorithms. *J. Supercomput.* **27**, 279. https://doi.org/10.1023/B:SUPE.0000011388.54204.8e (2004) (cit. on p. 62).

63. Sarkar, V. & Megiddo, N. *An Analytical Model for Loop Tiling and Its Solution* in *Proceedings of the 2000 IEEE International Symposium on Performance Analysis of Systems and Software* (IEEE Computer Society, Washington, DC, USA, 2000), 146. http://dl.acm.org/citation.cfm?id=1153923.1154542 (cit. on p. 62).

64. Mitchell, N., Högstedt, K., Carter, L. & Ferrante, J. Quantifying the multi-level nature of tiling interactions. *International Journal of Parallel Programming* **26**, 641 (1998) (cit. on p. 62).

65. Yotov, K., Xiaoming Li, Gang Ren, Garzaran, M. J. S., Padua, D., Pingali, K. & Stodghill, P. Is Search Really Necessary to Generate High-Performance BLAS? *Proceedings of the IEEE* **93**, 358 (2005) (cit. on p. 62).

66. Esseghir, K. *Improving data locality for caches* PhD thesis (Rice University, 1993) (cit. on p. 62).

67. Rivera, G. & Tseng, C.-W. *A Comparison of Compiler Tiling Algorithms* in *Proceedings of the 8th International Conference on Compiler Construction, Held As Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99* (Springer-Verlag, Berlin, Heidelberg, 1999), 168. http://dl.acm.org/citation.cfm?id=647475.727610 (cit. on p. 62).

68. Mehta, S., Beeraka, G. & Yew, P.-C. Tile Size Selection Revisited. *ACM Trans. Archit. Code Optim.* **10**, 35:1. http://doi.acm.org/10.1145/2541228.2555292 (2013) (cit. on p. 62).

69. Whaley, R. C. & Dongarra, J. J. *Automatically Tuned Linear Algebra Software* in *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing* (1998), 38 (cit. on p. 62).

70. Knijnenburg, P. M. W., Kisuki, T., Gallivan, K. & O'Boyle, M. F. P. The Effect of Cache Models on Iterative Compilation for Combined Tiling and Unrolling: Research Articles. *Concurr. Comput. : Pract. Exper.* **16**, 247. http://dx.doi.org/10.1002/cpe.v16:2/3 (2004) (cit. on p. 62).

71. Fraguela, B. B., Carmueja, M. G. & Andrade, D. Optimal tile size selection guided by analytical models. *parameters* **10**, 14 (2005) (cit. on p. 62).

72. Yuki, T., Renganarayanan, L., Rajopadhye, S., Anderson, C., Eichenberger, A. E. & O'Brien, K. *Automatic Creation of Tile Size Selection Models* in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (ACM, Toronto, Ontario, Canada, 2010), 190. http://doi.acm.org/10.1145/1772954.1772982 (cit. on p. 62).

73. Mendis, C., Renda, A., Amarasinghe, S. & Carbin, M. Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks (2018) (cit. on p. 62).

74. Rahman, M., Pouchet, L.-N. & Sadayappan, P. *Neural Network Assisted Tile Size Selection* in (2010) (cit. on p. 62).

75. Cociorva, D., Wilkins, J. W., Lam, C., Baumgartner, G., Ramanujam, J. & Sadayappan, P. *Loop Optimization for a Class of Memory-constrained Computations* in *Proceedings of the 15th International Conference on Supercomputing* (ACM, Sorrento, Italy, 2001), 103. http://doi.acm.org/10.1145/377792.377814 (cit. on p. 62).

76. Qasem, A. & Kennedy, K. *Profitable Loop Fusion and Tiling Using Model-driven Empirical Search* in *Proceedings of the 20th Annual International Conference on Supercomputing* (ACM, Cairns, Queensland, Australia, 2006), 249. http://doi.acm.org/10.1145/1183401.1183437 (cit. on p. 62).

77. Beaugnon, U., Pouille, A., Pouzet, M., Pienaar, J. & Cohen, A. *Optimization Space Pruning Without Regrets* in *Proceedings of the 26th International Conference on Compiler Construction* (ACM, Austin, TX, USA, 2017), 34. http://doi.acm.org/10.1145/3033019.3033023 (cit. on p. 62).

78. Henretty, T., Veras, R., Franchetti, F., Pouchet, L.-N., Ramanujam, J. & Sadayappan, P. *A Stencil Compiler for Short-vector SIMD Architectures* in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing* (ACM, Eugene, Oregon, USA, 2013), 13. http://doi.acm.org/10.1145/2464996.2467268 (cit. on p. 63).

79. Henretty, T., Stock, K., Pouchet, L.-N., Franchetti, F., Ramanujam, J. & Sadayappan, P. *Data Layout Transformation for Stencil Computations on Short-vector SIMD Architectures* in *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software* (Springer-Verlag, Saarbrücken, Germany, 2011), 225. http://dl.acm.org/citation.cfm?id=1987237.1987255 (cit. on p. 63).

80. Prajapati, N., Ranasinghe, W., Rajopadhye, S., Andonov, R., Djidjev, H. & Grosser, T. *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* (ACM, Austin, Texas, USA, 2017), 163. http://doi.acm.org/10.1145/3018743.3018744 (cit. on p. 63).

81. Liao, S. W., Tsai, S. J., Yang, C. H. & Lo, C. K. *Locality-Aware Scheduling for Stencil Code in Halide* in *2016 45th International Conference on Parallel Processing Workshops (ICPPW)* (2016), 72 (cit. on p. 63).

82. Mattson, R. L., Gecsei, J., Slutz, D. R. & Traiger, I. L. Evaluation Techniques for Storage Hierarchies. *IBM Syst. J.* **9**, 78. http://dx.doi.org/10.1147/sj.92.0078 (1970) (cit. on pp. 65, 93).

83. Beyls, K. & D'Hollander, E. H. *Reuse Distance as a Metric for Cache Behavior* in *In Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems* (2001), 617 (cit. on p. 65).

84. Ding, C. & Zhong, Y. *Predicting Whole-program Locality Through Reuse Distance Analysis* in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (ACM, San Diego, California, USA, 2003), 245. http://doi.acm.org/10.1145/781131.781159 (cit. on pp. 65, 94).

85. Xiang, X., Ding, C., Luo, H. & Bao, B. *HOTL: A Higher Order Theory of Locality* in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (ACM, Houston, Texas, USA, 2013), 343. http://doi.acm.org/10.1145/2451116.2451153 (cit. on pp. 65, 94).

86. Elder, J. & Hill, M. D. *Dinero IV Trace-Driven Uniprocessor Cache Simulator* http://www.cs.wisc.edu/~{}markhill/DineroIV/. 2003 (cit. on pp. 65, 84, 86, 92, 93).

87. Iyer, R. *On modeling and analyzing cache hierarchies using CASPER* in *11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003.* (2003), 182 (cit. on pp. 65, 93).

88. Carlson, T. E., Heirman, W. & Eeckhout, L. *Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-core Simulation* in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (ACM, Seattle, Washington, 2011), 52:1. http://doi.acm.org/10.1145/2063384.2063454 (cit. on pp. 65, 93).

89. Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D. & Wood, D. A. The Gem5 Simulator. *SIGARCH Comput. Archit. News* **39**, 1. http://doi.acm.org/10.1145/2024716.2024718 (2011) (cit. on pp. 65, 93).

90. Beyls, K. & D'Hollander, E. H. Generating Cache Hints for Improved Program Efficiency. *J. Syst. Archit.* **51**, 223. http://dx.doi.org/10.1016/j.sysarc.2004.09.004 (2005) (cit. on pp. 66, 95).

91. Hill, M. D. *Aspects of Cache Memory and Instruction Buffer Performance* AAI8813907. PhD thesis (1987) (cit. on p. 67).

92. Haase, C. A Survival Guide to Presburger Arithmetic. *ACM SIGLOG News* **5**, 67. http://doi.acm.org/10.1145/3242953.3242964 (2018) (cit. on pp. 68, 82).

93. Nguyen Luu, D. *The Computational Complexity of Presburger Arithmetic* PhD thesis (UCLA, 2018) (cit. on p. 82).

94. JR. H.W., L. Integer Programming with a Fixed Number of Variables. *Report 81-03, Mathematisch Instituut Amsterdam (1981)* **8** (1983) (cit. on p. 82).

95. Fischer, M. J. & Rabin, M. O. *SUPER-EXPONENTIAL COMPLEXITY OF PRESBURGER ARITHMETIC* tech. rep. (Cambridge, MA, USA, 1974) (cit. on p. 82).

96. Terpstra, D., Jagode, H., You, H. & Dongarra, J. *Collecting Performance Data with PAPI-C* in *Tools for High Performance Computing 2009* (eds Müller, M. S., Resch, M. M., Schulz, A. & Nagel, W. E.) (Springer Berlin Heidelberg, Berlin, Heidelberg, 2010), 157 (cit. on p. 84).

97. Bao, W., Krishnamoorthy, S., Pouchet, L.-N. & Sadayappan, P. Analytical Modeling of Cache Behavior for Affine Programs. *Proc. ACM Program. Lang.* **2**, 32:1. http://doi.acm.org/10.1145/3158120 (2017) (cit. on pp. 92, 95).

98. Bellard, F. *QEMU, a Fast and Portable Dynamic Translator* in *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (USENIX Association, Anaheim, CA, 2005), 41. http://dl.acm.org/citation.cfm?id=1247360.1247401 (cit. on p. 92).

99. Bennett, B. T. & Kruskal, V. J. LRU Stack Processing. *IBM J. Res. Dev.* **19**, 353. http://dx.doi.org/10.1147/rd.194.0353 (1975) (cit. on p. 94).

100. Olken, F. Efficient methods for calculating the success function of fixed space replacement policies (1981) (cit. on p. 94).

101. Sugumar, R. A. & Abraham, S. G. *Efficient Simulation of Caches Under Optimal Replacement with Applications to Miss Characterization* in *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (ACM, Santa Clara, California, USA, 1993), 24. http://doi.acm.org/10.1145/166955.166974 (cit. on p. 94).

102. Kim, Y. H., Hill, M. D. & Wood, D. A. *Implementing Stack Simulation for Highly-associative Memories* in *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (ACM, San Diego, California, USA, 1991), 212. http://doi.acm.org/10.1145/107971.107995 (cit. on p. 94).

103. Eklov, D. & Hagersten, E. *StatStack: Efficient modeling of LRU caches* in *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)* (2010), 55 (cit. on p. 94).

104. Xiang, X., Bao, B., Ding, C. & Gao, Y. *Linear-time Modeling of Program Working Set in Shared Cache* in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques* (IEEE Computer Society, Washington, DC, USA, 2011), 350. http://dx.doi.org/10.1109/PACT.2011.66 (cit. on p. 94).

105. Chen, D., Liu, F., Ding, C. & Pai, S. *Locality Analysis Through Static Parallel Sampling* in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (ACM, Philadelphia, PA, USA, 2018), 557. http://doi.acm.org/10.1145/3192366.3192402 (cit. on p. 94).

106. Agarwal, A., Hennessy, J. & Horowitz, M. An Analytical Cache Model. *ACM Trans. Comput. Syst.* **7**, 184. http://doi.acm.org/10.1145/63404.63407 (1989) (cit. on p. 94).

107. Harper, J. S., Kerbyson, D. J. & Nudd, G. R. Analytical Modeling of Set-Associative Cache Behavior. *IEEE Trans. Comput.* **48**, 1009. https://doi.org/10.1109/12.805152 (1999) (cit. on p. 94).

108. Carr, S., McKinley, K. S. & Tseng, C.-W. *Compiler Optimizations for Improving Data Locality* in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (ACM, San Jose, California, USA, 1994), 252. http://doi.acm.org/10.1145/195473.195557 (cit. on p. 94).

109. Ferdinand, C., Martin, F., Wilhelm, R. & Alt, M. Cache Behavior Prediction by Abstract Interpretation. *Sci. Comput. Program.* **35**, 163. http://dx.doi.org/10.1016/S0167-6423(99)00010-6 (1999) (cit. on p. 94).

110. Chattopadhyay, S. & Roychoudhury, A. Scalable and precise refinement of cache timing analysis via path-sensitive verification. *Real-Time Systems* **49**, 517. https://doi.org/10.1007/s11241-013-9178-0 (2013) (cit. on p. 94).

111. Touzeau, V., Maïza, C., Monniaux, D. & Reineke, J. *Ascertaining Uncertainty for Efficient Exact Cache Analysis* in (2017), 22 (cit. on p. 94).

112. Touzeau, V., Maïza, C., Monniaux, D. & Reineke, J. Fast and Exact Analysis for LRU Caches. *Proc. ACM Program. Lang.* **3**, 54:1. http://doi.acm.org/10.1145/3290367 (2019) (cit. on p. 94).

113. Ghosh, S., Martonosi, M. & Malik, S. Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior. *ACM Transactions on Programming Languages and Systems* **21** (2000) (cit. on p. 94).

114. Vera, X. & Xue, J. *Let's Study Whole-Program Cache Behaviour Analytically* in *Proceedings of the 8th International Symposium on High-Performance Computer Architecture* (IEEE Computer Society, Washington, DC, USA, 2002), 175. http://dl.acm.org/citation.cfm?id=874076.876456 (cit. on p. 94).

115. Xue, J. & Vera, X. Efficient and Accurate Analytical Modeling of Whole-Program Data Cache Behavior. *IEEE Trans. Comput.* **53**, 547. http://dx.doi.org/10.1109/TC.2004.1275296 (2004) (cit. on p. 94).

116. CaBcaval, C. & Padua, D. A. *Estimating Cache Misses and Locality Using Stack Distances* in *Proceedings of the 17th Annual International Conference on Supercomputing* (ACM, San Francisco, CA, USA, 2003), 150. http://doi.acm.org/10.1145/782814.782836 (cit. on p. 94).

117. Chatterjee, S., Parker, E., Hanlon, P. J. & Lebeck, A. R. *Exact Analysis of the Cache Behavior of Nested Loops* in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (ACM, Snowbird, Utah, USA, 2001), 286. http://doi.acm.org/10.1145/378795.378859 (cit. on p. 94).

118. Forum, M. *MPI: A Message-Passing Interface Standard. Version 3.1* http://www.mpi-forum.org. 2015 (cit. on p. 97).

119. Nickolls, J., Buck, I., Garland, M. & Skadron, K. Scalable Parallel Programming with CUDA. *Queue* **6**, 40. http://doi.acm.org/10.1145/1365490.1365500 (2008) (cit. on p. 97).

120. Potluri, S., Hamidouche, K., Venkatesh, A., Bureddy, D. & Panda, D. K. *Efficient Inter-node MPI Communication Using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs* in *Proceedings of the 2013 42Nd International Conference on Parallel Processing* (IEEE Computer Society, Washington, DC, USA, 2013), 80. http://dx.doi.org/10.1109/ICPP.2013.17 (cit. on pp. 97, 105).

121. White III, J. B. & Dongarra, J. J. *Overlapping Computation and Communication for Advection on Hybrid Parallel Computers* in *2011 IEEE International Parallel Distributed Processing Symposium* (2011), 59 (cit. on p. 97).

122. Phillips, E. H. & Fatica, M. *Implementing the Himeno benchmark with CUDA on GPU clusters* in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)* (2010), 1 (cit. on p. 97).

123. Hoefler, T., Dinan, J., Thakur, R., Barrett, B., Balaji, P., Gropp, W. & Underwood, K. Remote Memory Access Programming in MPI-3. *ACM Trans. Parallel Comput.* **2**, 9:1. http://doi.acm.org/10.1145/2780584 (2015) (cit. on pp. 98, 100, 104).

124. Belli, R. & Hoefler, T. *Notified Access: Extending Remote Memory Access Programming Models for Producer-Consumer Synchronization* in *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium* (IEEE Computer Society, Washington, DC, USA, 2015), 871. http://dx.doi.org/10.1109/IPDPS.2015.30 (cit. on pp. 98, 100, 109).

125. Little, J. D. C. A Proof for the Queuing Formula: L = λW. *Oper. Res.* **9**, 383. http://dx.doi.org/10.1287/opre.9.3.383 (1961) (cit. on p. 99).

126. Tanasic, I., Gelado, I., Cabezas, J., Ramirez, A., Navarro, N. & Valero, M. *Enabling Preemptive Multiprogramming on GPUs* in *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (IEEE Press, Minneapolis, Minnesota, USA, 2014), 193. http://dl.acm.org/citation.cfm?id=2665671.2665702 (cit. on pp. 100, 121).

127. Oden, L., Fröning, H. & Pfreundt, F. *Infiniband-Verbs on GPU: A Case Study of Controlling an Infiniband Network Device from the GPU* in *2014 IEEE International Parallel Distributed Processing Symposium Workshops* (2014), 976 (cit. on pp. 105, 121).

128. Xiao, S. & Feng, W. *Inter-block GPU communication via fast barrier synchronization* in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)* (2010), 1 (cit. on pp. 106, 121).

129. Rosetti, D. *A fast GPU memory copy library based on NVIDIA GPUDirect RDMA technology* https://github.com/NVIDIA/gdrcopy. 2015 (cit. on pp. 108, 111).

130. Inc., C. *Using the GNI and MAPP APIs. Ver. S-2446-52.* http://docs.cray.com/. 2014 (cit. on p. 111).

131. Goldenberg, D. *Co-Desing Architecture* http://slideshare.net/insideHPC/co-design-architecture-for-exascale. 2016 (cit. on p. 111).

132. Micikevicius, P. *GPU Performance Analysis and Optimization* http://on-demand.gputechconf.com/gtc/2012/presentations/S0514-GTC2012-GPU-Performance-Analysis.pdf. 2012 (cit. on p. 113).

133. Dawson, J. M. Particle simulation of plasmas. *Rev. Mod. Phys.* **55**, 403. http://link.aps.org/doi/10.1103/RevModPhys.55.403 (2 1983) (cit. on p. 115).

134. Ghysels, P. & Vanroose, W. Hiding Global Synchronization Latency in the Preconditioned Conjugate Gradient Algorithm. *Parallel Comput.* **40**, 224. http://dx.doi.org/10.1016/j.parco.2013.06.001 (2014) (cit. on p. 119).

135. Graham, R. L. & Shipman, G. *MPI Support for Multi-core Architectures: Optimized Shared Memory Collectives* in *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface* (Springer-Verlag, Dublin, Ireland, 2008), 130. http://dx.doi.org/10.1007/978-3-540-87475-1_21 (cit. on p. 119).

136. Li, S., Hoefler, T. & Snir, M. *NUMA-aware Shared-memory Collective Communication for MPI* in *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing* (ACM, New York, New York, USA, 2013), 85. http://doi.acm.org/10.1145/2493123.2462903 (cit. on p. 119).

137. Duato, J., Igual, F. D., Mayo, R., Peña, A. J., Quintana-Ortí, E. S. & Silla, F. *An Efficient Implementation of GPU Virtualization in High Performance Clusters* in *Proceedings of the 2009 International Conference on Parallel Processing* (Springer-Verlag, Delft, The Netherlands, 2010), 385. http://dl.acm.org/citation.cfm?id=1884795.1884840 (cit. on p. 120).

138. Reaño, C., Mayo, R., Quintana-Ortí, E. S., Silla, F., Duato, J. & Peña, A. J. *Influence of InfiniBand FDR on the performance of remote GPU virtualization* in *2013 IEEE International Conference on Cluster Computing (CLUSTER)* (2013), 1 (cit. on p. 120).

139. Miyoshi, T., Irie, H., Shima, K., Honda, H., Kondo, M. & Yoshinaga, T. *FLAT: A GPU Programming Framework to Provide Embedded MPI* in *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units* (ACM, London, United Kingdom, 2012), 20. http://doi.acm.org/10.1145/2159430.2159433 (cit. on p. 120).

140. Kim, S., Huh, S., Zhang, X., Hu, Y., Wated, A., Witchel, E. & Silberstein, M. *GPUnet: Networking Abstractions for GPU Programs* in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (USENIX Association, Broomfield, CO, 2014), 201. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kim (cit. on p. 120).

141. Stuart, J. A. & Owens, J. D. *Message Passing on Data-parallel Architectures* in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing* (IEEE Computer Society, Washington, DC, USA, 2009), 1. https://doi.org/10.1109/IPDPS.2009.5161065 (cit. on p. 120).

142. Stuart, J. A., Balaji, P. & Owens, J. D. *Extending MPI to Accelerators* in *Proceedings of the 1st Workshop on Architectures and Systems for Big Data* (ACM, Galveston Island, Texas, USA, 2011), 19. http://doi.acm.org/10.1145/2377978.2377981 (cit. on p. 120).

143. Oden, L. & Fröning, H. *GGAS: Global GPU address spaces for efficient communication in heterogeneous clusters* in *2013 IEEE International Conference on Cluster Computing (CLUSTER)* (2013), 1 (cit. on p. 120).

144. Daoud, F., Watad, A. & Silberstein, M. *GPUrdma: GPU-side Library for High Performance Networking from GPU Kernels* in *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers* (ACM, Kyoto, Japan, 2016), 6:1. http://doi.acm.org/10.1145/2931088.2931091 (cit. on p. 121).