INFORMATIK

HIGH-PERFORMANCE DOMAIN-SPECIFIC COMPILATION WITHOUT DOMAIN-SPECIFIC COMPILERS

Inaugural-Dissertation
zur Erlangung des Doktorgrades
Dr. rer. nat.
der Naturwissenschaften im Fachbereich
Mathematik und Informatik
der Mathematisch-Naturwissenschaftlichen Fakultät
der Westfälischen Wilhelms-Universität Münster

vorgelegt von

BASTIAN HAGEDORN

aus Ibbenbüren

– 2020 –

High-Performance Domain-Specific Compilation without Domain-Specific Compilers © Bastian Hagedorn 2020

DEKAN:	Prof. Dr. Xiaoyi Jiang
ERSTER GUTACHTER:	Prof. Dr. Sergei Gorlatch
ZWEITER GUTACHTER:	Prof. Dr. Herbert Kuchen
DRITTER GUTACHTER:	Lecturer Dr. Michel Steuwer
TAG DER MÜNDLICHEN PRÜFUNG:	....
TAG DER PROMOTION:	....

Dedicated to my wife and my son.

ABSTRACT

Developing efficient software for scientific applications with high computing power demands, like physics simulations or deep learning, becomes increasingly difficult due to the ever-changing hardware landscape consisting of multi-core CPUs, many-core GPUs, and domain-specific accelerators. Domain-specific compilers pose a viable solution to this problem. They provide convenient high-level programming abstractions in the form of Domain-Specific Languages (DSLs) while automatically performing the heavy lifting of generating high-performance code for the target architecture. However, this places the burden of high-performance code generation solely on the developers of the domain-specific compiler. Typically, there is little to no reuse between domain-specific compilers: A domain-specific compiler is specific to one domain and is generally not reusable for generating code for another domain. Therefore, developing a domain-specific compiler, especially its Intermediate Representation (IR) and optimization passes, is difficult because of the need to start from scratch for every new compiler.

This thesis presents a novel approach to achieving the benefits of high-performance domain-specific compilation without the need to develop domain-specific compilers. The key idea of our approach is decomposing both domain-specific computations and their optimizations into a small set of generic building blocks. Using only those building blocks, we develop a domain-agnostic compilation approach to generating high-performance code. This approach allows us to achieve the benefits of domain-specific compilation by simply expressing both domain-specific computations and their optimizations as compositions of generic building blocks.

The thesis starts with a case study highlighting the benefits and potential of domain-specific compilation and demonstrates the drawbacks of developing a domain-specific compiler from scratch. Specifically, we show why domain-specific compilation is far superior to using high-performance libraries or manually developing high-performance implementations. At the same time, we highlight the limitations of representing computations in a domain-specific compiler's IR that is optimized using domain-specific optimization strategies. We then show how to decompose both computations and optimizations into generic building blocks, and discuss how to re-compose those to express domain-specific computations and optimizations. Finally, we show that our approach achieves competitive performance to state-of-the-art domain-specific compilers without having their drawbacks.

PUBLICATIONS

This thesis is based on ideas and results that have been presented in the following publications[1]:

[1] Using rankings according to CORE: *https://www.core.edu.au/* latest rankings: Journals: CORE2020 Conferences: CORE2018

[1] Larisa Stoltzfus, **Bastian Hagedorn**, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. "Tiling Optimizations for Stencil Computations Using Rewrite Rules in Lift." In: *ACM Transactions on Architecture and Code Optimization (TACO)* 16.4 (2020), 52:1–52:25, Rank B.

[2] **Bastian Hagedorn**, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. "Fireiron: A Data-Movement-Aware Scheduling Language for GPUs." In: *29th International Conference on Parallel Architectures and Compilation Techniques, PACT 2020, Online, October 3-7, 2020*. (accepted for publication). IEEE, 2020, Rank A.

[3] **Bastian Hagedorn**, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. "Fireiron: A Scheduling Language for High-Performance Linear Algebra on GPUs." In: *arXiv preprint: 2003.06324 [cs.PL]* (2020).

[4] **Bastian Hagedorn**, Johannes Lenfers, Thomas Koehler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. "Achieving High-Performance the Functional Way - A Functional Pearl on Expressing High-Performance Optimizations as Rewrite Strategies." In: *25th ACM SIGPLAN International Conference on Functional Programming (ICFP), Online, August 24-26, 2020*. (accepted for publication). ACM, 2020, Rank A*.

[5] **Bastian Hagedorn**, Michel Steuwer, and Sergei Gorlatch. "A Transformation-Based Approach to Developing High-Performance GPU Programs." In: *Perspectives of System Informatics - 11th International Andrei P. Ershov Informatics Conference, PSI 2017, Moscow, Russia, June 27-29, 2017, Revised Selected Papers*. Ed. by Alexander K. Petrenko and Andrei Voronkov. Vol. 10742. Lecture Notes in Computer Science. Springer, 2017, 179–195, Rank B.

[6] **Bastian Hagedorn**, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. "High Performance Stencil Code Generation with Lift." In: *Proceedings of the 2018 ACM/IEEE International Symposium on Code Generation and Optimization, CGO 2018, Vösendorf / Vienna, Austria, February 24-28, 2018*. Ed. by Jens Knoop, Markus Schordan, Teresa

Johnson, and Michael F. P. O'Boyle. **Best Paper Award**. ACM, 2018, 100–112, Rank A.

[7] **Hagedorn, Bastian**, Johannes Lenfers, Thomas Koehler, Sergei Gorlatch, and Michel Steuwer. "A Language for Describing Optimization Strategies." In: *arXiv preprint: 2002.02268 [cs.PL]* (2020).

During the work on this thesis, I also co-authored the following publications:

[1] Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, **Bastian Hagedorn**, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodík. "Swizzle Inventor: Data Movement Synthesis for GPU Kernels." In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*. Ed. by Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck. ACM, 2019, 65–78, Rank A*.

[2] Toomas Remmelg, **Bastian Hagedorn**, Lu Li, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. "High-level hardware feature extraction for GPU performance prediction of stencils." In: *GPGPU@PPoPP '20: 13th Annual Workshop on General Purpose Processing using Graphics Processing Unit colocated with 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 23, 2020*. Ed. by Adwait Jog, Onur Kayiran, and Ashutosh Pattnaik. ACM, 2020, pp. 21–30.

ACKNOWLEDGMENTS

This thesis marks the end of an incredible four-year-long journey. I could have never anticipated the experiences and friendships I was able to make during this time.

First of all, I would like to thank my supervisor Prof. Dr. Sergei Gorlatch, without whose support none of this would have been possible. I was fortunate to work with great colleagues in Münster and especially thank Ari Rasch, Tim Humernbrum, Michael Haidl, Bastian Köpcke, Johannes Lenfers, Huihui Sun, Jens Gutsfeld, and Richard Schulze for plenty of coffee breaks and for sharing the day-to-day struggles of being a PhD student. Special thanks to Julia Kaiser-Mariani for always finding a way to realize my sometimes rather demanding projects, which often involved a lot of paperwork and fighting university bureaucracy.

I was fortunate to make several long-term visits to the Universities of Edinburgh and Glasgow, where I met a lot of inspiring people, too many to mention here. Thanks to HiPEAC, Eurolab-4-HPC, and HPC-Europa3 for financially supporting these visits. Thanks to Christophe Dubach for hosting me multiple times at the University of Edinburgh. Thanks to the entire Lift, RI**SE**, and ELEVATE team. It was a great pleasure to collaborate with every one of you. Special thanks to Larisa Stoltzfus and Federico Pizutti, you made Edinburgh my home away from home.

Thanks to NVIDIA for believing in my research vision and for giving me the time to pursue it by financially supporting me with a PhD Fellowship. It was a great pleasure to work with Ras Bodik, Vinod Grover, and all other colleagues at NVIDIA. I'm looking forward to starting a new journey there.

Thanks to Michel Steuwer. Honestly, I'm struggling to find words that do justice to the support you've provided. Thanks for hosting me multiple times, for believing in me and my research, for the countless hours of working together, and generally, for the great times we've had.

Thanks to my parents, my sister, my family, and friends for their endless support and interest even though I'm still struggling to clearly explain what I've been working on for the past four years. Finally, I thank my wife, Lena, for being the partner on my side and for sharing all ups and downs throughout this journey. Thanks for your never-ending love, encouragement, and support.

Bastian Hagedorn
July 2020, Ibbenbüren

CONTENTS

1 INTRODUCTION

Computer architectures are evolving rapidly to satisfy the growing demand for high performance required by today's scientific application domains, including physics simulations or machine learning. The current trend goes towards adding specialized hardware units to classical multi-core CPU and many-core GPU architectures, providing the most common computational building blocks as built-in hardware instructions. For example, Tensor Cores [110] on NVIDIA GPUs or Google's Tensor Processing Unit (TPU) [84] provide specialized instructions computing small low-precision matrix-matrix multiplications to accelerate the training and inference of deep neural networks significantly.

Developing efficient programs for the ever-changing hardware landscape becomes increasingly difficult. Due to the end of Dennard scaling[1] [43] and Moore's Law[2] [106], all of today's high-performance architectures are highly parallel and contain thousands of cores. For example, NVIDIA's RTX 6000 contains 4608 CUDA cores, 576 Tensor Cores, and 72 RT (Ray Tracing) Cores that achieve a peak of 16.3 TFLOPS single-precision performance.

[1] *Dennard Scaling: As transistors get smaller, their power density stays constant. [43]*

[2] *Moore's Law: The number of transistors in an integrated circuit doubles about every two years. [106]*

ACHIEVING HIGH PERFORMANCE ON PARALLEL HARDWARE

Efficient software achieving practical peak performance on parallel architectures needs to manage a complex multi-layered compute and memory hierarchy. Programmers have to efficiently partition the computational task among thousands of threads running at the same time in parallel. Additionally, the processed data must be carefully transferred from the slow off-chip memory to fast registers via a deep hierarchy of memory levels to provide enough work to each thread. On multi-core CPUs, this memory hierarchy consists of multiple caches whose different replacement policies affect how programmers need to optimize memory access patterns. On many-core GPUs, this hierarchy consists of both caches and self-managed shared memory, which requires carefully synchronizing groups of threads sharing read and write access.

Typically, high-performance programs are written in low-level programming languages like OpenCL or CUDA and include inline assembly to access the functionality of specialized instructions. Developing these programs requires intimate knowledge of the hardware and is hard even for performance engineers and, therefore, not manageable for domain-scientists like machine learning experts.

To make the high performance accessible for domain-scientists, hardware vendors provide manually tuned domain-specific libraries like NVIDIA's cuBLAS and cuDNN or Intel's MKL. These libraries contain high-performance implementations, carefully tuned by performance engineers, of the most common computational building blocks occurring in particular application domains such as matrix-matrix multiplication or convolution.

DEVELOPING HIGH-PERFORMANCE LIBRARIES The task of the library programmer is very challenging: For every building block of the library, say Generalized Matrix Multiplication (GEMM), multiple high-performance implementations have to be provided for all commonly used architectures, input sizes, varying precision, and storage formats.

For example, in the domain of dense linear algebra, cuBLAS is tuned differently for multiple NVIDIA GPU architectures ranging from *Kepler* (introduced in April 2012) to *Ampere* (introduced in May 2020). Every architecture comes with multiple chips with different characteristics to consider in library implementations, ranging from embedded *Jetson* boards used in autonomous vehicles up to server class *Tesla* GPUs used in data-centers and supercomputers. Additionally, depending on the problem size, different algorithmic implementations are required to achieve the highest performance possible: For GEMM, in case of extreme input sizes (e.g., where the M and N dimension of the input matrices are small like in implicit GEMM kernels ($M = N = 64, K = 401408$)), the only way to assign enough work to all threads of the GPU is to parallelize the K-dimension as well. This additional parallelization is not necessary for more regular input sizes like large square matrices. Various implementations for different precision need to be provided, including half-, single- and double-precision. The storage formats (row-major, column-major) of the input matrices must also be considered because they have a significant impact on performance due to different memory access patterns. To make matters worse, high-performance library implementations usually need to be provided as, or at least partially use, optimized low-level assembly code because important performance-critical architectural features are only exposed at this API level.

Clearly, providing and maintaining high-performance libraries is only feasible for the most common and most essential building blocks of application domains and requires significant effort from multiple performance engineers.

1.1 A SOLVED PROBLEM: DOMAIN-SPECIFIC COMPILATION

Domain-specific compilation is a viable solution to the problem of providing high-level abstractions to domain-scientists while achieving high-performance on modern hardware. Figure 1.2 shows an overview of the process of domain-specific compilation. The key idea is the separation of concerns: The application developer, usually a domain-scientist, solely focuses on expressing computations in a domain-specific program using high-level programming abstractions provided by a domain-specific language (DSL), without optimizing the program for performance. Instead of the necessity to have human experts who optimize and fine-tune high-performance programs, a domain-specific compiler generates these automatically from the high-level DSL program. As research has shown, this approach works well across multiple domains such as databases (SQL [25]) or machine learning (e.g., TensorFlow [2, 3], TVM [29]).

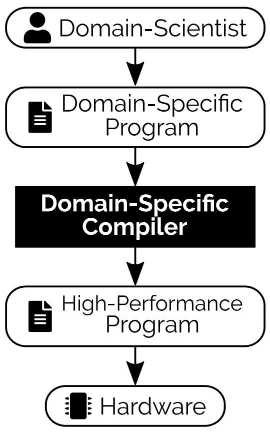
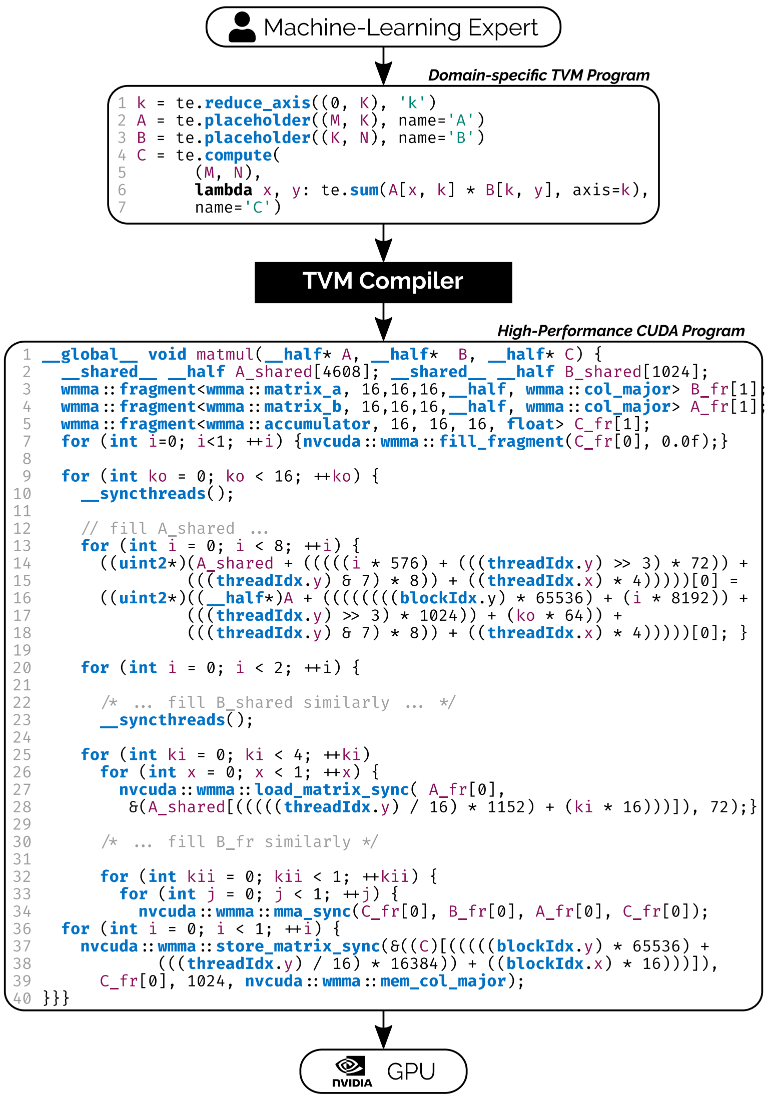


Figure 1.2: Overview of domain-specific compilation.

A MOTIVATING EXAMPLE On the high abstraction level, domain-scientists express computations using familiar abstractions. For example, in the domain of machine learning, the TVM DSL and compiler allows a machine learning expert to define layers of a neural network as simple mathematical expressions. Figure 1.3 (top) shows a program defining a matrix multiplication computation using TVM's high-level DSL. First, the two input matrices `A` and `B` are defined in lines 2 and 3. Then, the matrix multiplication is specified in lines 4-7 by computing the dot-product of the rows of matrix `A` and the columns of matrix `B`.

Figure 1.3 (bottom) shows CUDA code targeting NVIDIA GPUs generated by the TVM compiler using the high-level matrix multiplication specification. This low-level code expresses the same computation as the high-level domain-specific program (a matrix multiplication); however, it is significantly longer and more complicated because it is optimized to run efficiently on the target architecture. Specifically, the CUDA implementation performs the computation using multiple threads (identified by `threadIdx` and `blockIdx`) running in parallel. In order to achieve high-performance, the memory hierarchy of the GPU is used by allocating multiple temporary buffers (lines 2-5), and data is fetched into shared memory (lines 12-22) and registers (lines 25-30) before the result is computed in lines 32-39. This implementation explicitly targets modern NVIDIA GPUs and computes the results using the Tensor Cores by leveraging the Warp-level Matrix Multiply Accumulate (WMMA) API [114] for efficiently computing small matrix multiplications.

Generating and understanding this parallel program requires understanding the CUDA programming model and knowledge of GPU architectures. Compiling the high-level TVM program to parallel



```python
k = te.reduce_axis((0, K), 'k')
A = te.placeholder((M, K), name='A')
B = te.placeholder((K, N), name='B')
C = te.compute(
      (M, N),
      lambda x, y: te.sum(A[x, k] * B[k, y], axis=k),
      name='C')
```

```cuda
__global__ void matmul(__half* A, __half*  B, __half* C) {
  __shared__ __half A_shared[4608]; __shared__ __half B_shared[1024];
  wmma::fragment<wmma::matrix_a, 16,16,16,__half, wmma::col_major> B_fr[1];
  wmma::fragment<wmma::matrix_b, 16,16,16,__half, wmma::col_major> A_fr[1];
  wmma::fragment<wmma::accumulator, 16, 16, 16, float> C_fr[1];
  for (int i=0; i<1; ++i) {nvcuda::wmma::fill_fragment(C_fr[0], 0.0f);}

  for (int ko = 0; ko < 16; ++ko) {
    __syncthreads();

    // fill A_shared ...
    for (int i = 0; i < 8; ++i) {
      ((uint2*)(A_shared + (((((i * 576) + (((threadIdx.y) >> 3) * 72)) +
                (((threadIdx.y) & 7) * 8)) + ((threadIdx.x) * 4))))[0] =
      ((uint2*)((__half*)A + ((((((((blockIdx.y) * 65536) + (i * 8192)) +
                (((threadIdx.y) >> 3) * 1024)) + (ko * 64)) +
                (((threadIdx.y) & 7) * 8)) + ((threadIdx.x) * 4))))[0]; }

    for (int i = 0; i < 2; ++i) {

      /* ... fill B_shared similarly ... */
      __syncthreads();

    for (int ki = 0; ki < 4; ++ki)
      for (int x = 0; x < 1; ++x) {
        nvcuda::wmma::load_matrix_sync( A_fr[0],
         &(A_shared[((((threadIdx.y) / 16) * 1152) + (ki * 16)))]), 72);}

      /* ... fill B_fr similarly */

      for (int kii = 0; kii < 1; ++kii) {
        for (int j = 0; j < 1; ++j) {
          nvcuda::wmma::mma_sync(C_fr[0], B_fr[0], A_fr[0], C_fr[0]);
  for (int i = 0; i < 1; ++i) {
    nvcuda::wmma::store_matrix_sync(&((C)[(((((blockIdx.y) * 65536) +
            (((threadIdx.y) / 16) * 16384)) + ((blockIdx.x) * 16)))]),
      C_fr[0], 1024, nvcuda::wmma::mem_col_major);
}}}
```

Figure 1.3: Expressing matrix multiplication in a high-level Domain-Specific Language (top), such as the one provided by TVM, liberates domain scientists from writing low-level high-performance code (bottom), which is instead automatically generated by the TVM domain-specific compiler.

CUDA code, and thereby restricting its execution to NVIDIA GPUs only, is just one of many possible options. To target different hardware, the TVM compiler can generate OpenMP code targeting multi-core CPUs or OpenCL code targeting various kinds of hardware accelerators. Generally, domain-specific compilation liberates domain-scientists from learning these low-level programming models. Due to this separation of concerns, domain-specific compilation is a popular technique for generating high-performance programs.

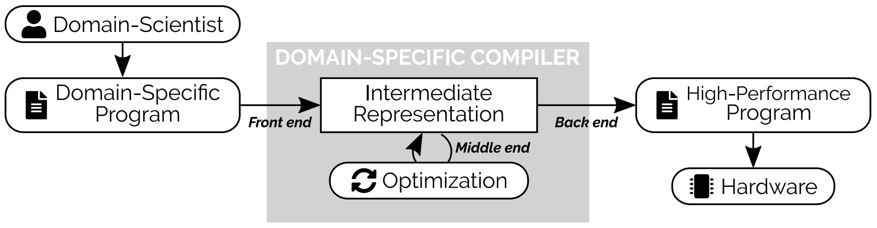


Figure 1.4: Overview of domain-specific compilation: A domain-scientist writes a program using a high-level domain-specific language. This program is compiled by a domain-specific compiler that generates a high-performance program that runs on the hardware.

1.2 THE NEW CHALLENGE: DOMAIN-SPECIFIC COMPILERS

While the use of DSLs and domain-specific compilers provides a helpful solution for the domain-scientist requiring high-performance implementations, this solution is costly in terms of compiler development. Typically, there is little to no reuse between domain-specific compilers almost by definition: A domain-specific compiler is specific to one domain and cannot be reused for generating code of another domain. This lack of reusability makes developing a domain-specific compiler difficult because of the need to start from scratch for every new compiler. This approach is not sustainable, given the number of application domains and the ever-growing hardware landscape.

Figure 1.4 shows the three steps of compiling a DSL source program into a high-performance program. The compiler front-end lowers a program written in a DSL into a compiler-internal intermediate representation (IR) more suitable for applying program transformations and code generation. In the middle-end, sometimes also called the optimizer, the compiler applies program transformations to the IR, aiming to optimize the performance of the program to be generated. Finally, the compiler back-end uses the optimized IR to generate a high-performance program.

In order to develop a domain-specific compiler, two questions need to be answered: (1) How to represent domain-specific computations, and (2) how to encode and apply optimizations on this representation? These questions drive the design of the two most critical components of a domain-specific compiler: its *intermediate representation (IR)* and the *optimizations* applied to this IR for generating high-performance code. The development of the front-, middle- and back-end depends critically on the design decisions made for these two components.

Every time a new domain-specific compiler is built, as of today, that is every time a new application domain is targeted, or a new hardware architecture needs to be supported, these two core components must be re-designed from scratch. While there are approaches like Delite [170] and AnyDSL [94] aiming to simplify the development of domain-specific compilers, these approaches often focus on supporting multiple application domains but only target one specific hardware architecture. However, due to rapidly changing hardware landscapes, e.g., NVIDIA alone has introduced four different new GPU architectures since the beginning of the work on this thesis, supporting new hardware architectures is as important as supporting new application domains. In this thesis, we argue that domain-specific compilation is achievable without re-building domain-specific compilers every time by solving two main challenges defined and discussed in the following.

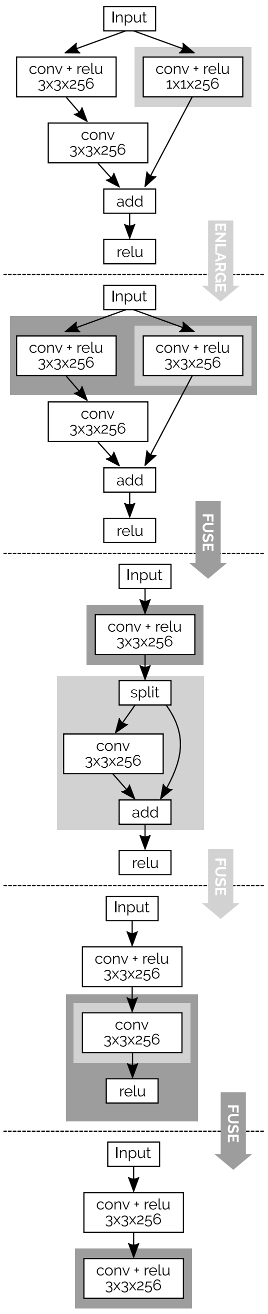


Figure 1.5: A high-level, graph-based IR representing machine learning computations. Progressively applying machine learning specific optimizations as graph-substitutions allows to reduce the overall number of layers.

1.2.1 *The Intermediate Representation Challenge*

The Intermediate Representation (IR) used in an optimizing domain-specific compiler needs to be developed with specific goals in mind, affecting the development of the compiler's front-, middle- and back-end. First of all, the IR must be able to represent all computations expressible in the high-level DSL. Ideally, the IR is extensible and flexible enough such that multiple front-ends can lower different DSLs into the same IR. Second, the IR must be suitable for the application of transformations aiming to optimize the performance when targeting various hardware devices. Finally, the IR must be suitable for code generation to eventually produce a high-performance program written in a lower-level programming language like CUDA or the assembly language of the targeted hardware device.

In the following, we introduce state-of-the-art IRs used in optimizing compilers today and identify problems with their design.

HIGH-LEVEL INTERMEDIATE REPRESENTATIONS Figure 1.5 shows an example of a high-level graph-based IR, as introduced in [83]. These are popular in machine learning compilers and model computations performed in a deep neural network as a graph of connected layers. Each node represents a computation to perform on its input tensors, and each edge in the IR represents a data dependency. Machine learning specific operations like `conv`, `add`, and `relu` are directly encoded as built-in nodes in the IR. For example, the monolithic `conv-relu` node represents a fused version of the `conv` and the `relu` node. Specifically, it represents applying the convolution computation to the input tensor followed by applying the element-wise Rectified Linear Unit (ReLU) activation function defined as $x = max(0, x)$.

Applying machine learning specific optimizations to this IR is conceptually straight-forward because known transformations can be encoded as graph transformations. Figure 1.5 shows the application of machine learning specific optimizations to the IR graph by progressively substituting parts of the graph with equivalent subgraphs. One example for exploiting the domain-specific knowledge in optimizing machine learning computations is that the filter of a convolution (shape 1×1 in Figure 1.5) can be enlarged by padding it with zeros (to, for example, shape 3×3). This transformation enables the subsequent fusion of two `conv` nodes and the fusion of `conv` with multiple other layers. Modifying and fusing nodes of the initial IR graph allows the decrease of the total number of required layers and thus computations to perform, leading to higher performance.

However, since domain-specific computations are built-in abstractions in the high-level IRs, an IR in the domain of machine learning is naturally highly specialized for that domain. Reusing the same high-level IR in other compilers targeting different application domains is inherently difficult.

Additionally, high-performance code generation directly from this IR is exceptionally challenging because of the large abstraction gap between the high-level IR and the low-level code that the domain-specific compiler eventually needs to generate. Therefore, domain-specific compilers using high-level IRs often rely on simply calling high-performance library routines (such as NVIDIA's cuDNN for computing the `conv` layer) whenever possible. This approach significantly complicates the extension and reuse of the IR even within the same domain, as noted in [11] by two of the original authors of TensorFlow, one of the most popular machine learning compilers that also uses a high-level graph-based IR.

LOW-LEVEL INTERMEDIATE REPRESENTATIONS As discussed in the previous paragraph, encoding domain-specific computations as built-in high-level abstractions in the IR prevents reuse and limits extensibility. Instead, another possibility is to represent domain-specific computations using a low-level IR.

The LLVM Compiler Infrastructure Project [92], for example, provides the low-level LLVM IR that is widely used in industry and academia, for example, in many general-purpose compilers including `clang` and the compiler for the Swift programming language. The LLVM IR is a Static Single Assignment (SSA) [158], text-based IR providing low-level operations close to machine-level assembly instructions. It is designed to provide the *capability of representing 'all' high-level languages cleanly* [135], which facilitates the reuse of LLVM IR for representing computations of all kinds of application domains. There exist multiple hundreds of optimization passes for transforming and optimizing LLVM IR. At the same time, code generation for different hardware devices is straight-forward due to its low-level assembly-like nature. In fact, LLVM already provides multiple back-ends, often officially supported by the hardware vendors, for various hardware devices.

```
 1 ; ... 39 lines left out
 2 28:                                              ; preds = %25
 3   %29 = add nsw i64 %27, %24
 4   %30 = getelementptr inbounds float, float* %0, i64 %29
 5   %31 = load float, float* %30, align 4, !tbaa !4
 6   %32 = mul nsw i64 %27, %12
 7   %33 = getelementptr inbounds float, float* %1, i64 %32
 8   %34 = load float, float* %33, align 4, !tbaa !4
 9   %35 = fmul float %31, %34
10   %36 = fadd float %26, %35
11   store float %36, float* %23, align 4, !tbaa !4
12   br label %37
13 ; ... 58 lines left out
```

Listing 1.1: LLVM IR code representing the `conv` layer of a deep neural network.

Listing 1.1 shows a snippet of LLVM IR representing the `conv` layer of a deep neural network. Representing a complete neural network may require thousands of LLVM IR code lines, due to being more explicit compared to the high-level graph-based IR. In the previous example showing the high-level IR, the `conv` layer was represented by a single node in the computation graph. In LLVM IR, however, representing the same computation requires 108 lines of code. For example, lines 9-10 explicitly compute parts of the convolution computation, multiplying the input tensors elements (`%31` - line 5) with the weights (`%34` - line 8) followed by an accumulation (`%36` - line 10). The information that a convolution computes a weighted sum of local neighborhoods in an input tensor is implicitly encoded by merely having a single `conv` node in the high-level IR. Lowering a DSL program into a high-level graph-based IR is usually straightforward, e.g., both might contain built-in abstractions for `conv` layers. However, lowering a DSL into the low-level LLVM IR is significantly more challenging due to a notable gap in abstraction.

Additionally, even though LLVM IR is capable of representing computations occurring in multiple domains, using it for domain-specific compilation is complicated and impractical: Applying domain-specific optimizations requires to re-discover domain knowledge, which is lost during the process of lowering a high-level DSL to low-level LLVM IR. For example, applying the domain-specific filter enlargement transformation mentioned above, requires re-discovering that the hundreds of lines of LLVM IR represent the `conv` layer.

HIERARCHICAL INTERMEDIATE REPRESENTATIONS Using only one level of abstraction in the IR of a domain-specific compiler quickly leads to problems or severe limitations for domain-specific compilation. As a last example, we consider using a hierarchy of IRs for domain-specific compilation, each providing a different abstraction level, mitigating the drawbacks discussed in the previous examples.

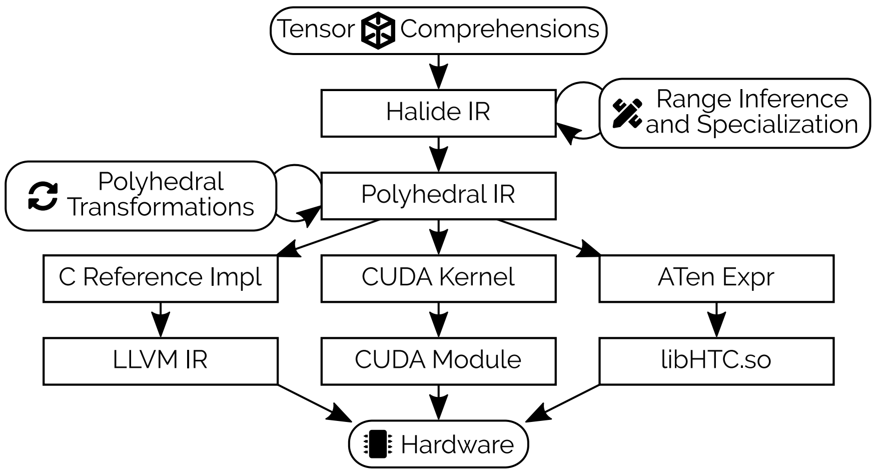


Figure 1.7: Overview of the Tensor Comprehensions domain-specific compiler for high-performance machine learning code generation. (Fig. inspired by [153].)

Figure 1.7 shows the internal configuration of the Tensor Comprehensions (TC) [175, 176] domain-specific compiler targeting machine learning computations, introduced by Facebook in 2018. On the highest abstraction level, TC uses the same graph-based IR as the domain-specific Halide compiler [141] initially designed for image processing pipelines. This IR is suitable for expressing machine learning and linear algebra computations, and we will discuss Halide's domain-specific compilation approach in more detail in the following chapters. The IR is then lowered into a loop-based polyhedral IR. This IR operates on affine loop nests by transforming them into a mathematical polyhedron representation [39] suitable for various loop optimizations like tiling to improve locality in memory accesses. From this abstraction level, there are multiple paths depending on what hardware to target. If TC is used for targeting multi-core CPUs, for example, then the polyhedral IR is lowered into LLVM IR. This way, using a hierarchical IR composition for domain-specific compilation, the high-level DSL code (in this case, a machine learning expression) is progressively lowered until eventually machine-executable code is generated.

Constructing domain-specific compilers with hierarchical IRs allows to reuse parts of existing domain-specific compilers, but it requires significant engineering effort. Mitigating the drawbacks of having only a single abstraction level comes at the cost of managing multiple layers of different IRs. Instead of merely writing one middle- and back-end to optimize the IR and to generate a high-performance program, many middle- and back-ends have to be developed and maintained to connect the IR layers. Furthermore, now the exact composition of the IR hierarchy is domain-specific and needs to be adjusted when constructing a new compiler targeting a different application domain.

Nevertheless, domain-specific compilation significantly benefits from a hierarchical IR structure offering multiple levels of abstractions. Google recently introduced the Multi-Layer Intermediate Representation (MLIR) [93], which is now part of the LLVM Compiler Infrastructure Project, aiming to simplify the development of hierarchical IRs and the management of lowering progressively between IRs. Specifically, MLIR allows compiler developers to define IRs as so-called *Dialects* and already provides an LLVM IR and a polyhedral dialect. Compiler developers can define their own dialects and transformations for implementing the lowering of dialects. With MLIR, managing multiple IRs (now dialects) in the same framework becomes significantly easier. However, to achieve domain-specific compilation, the concrete stack of IRs remains domain-specific and, therefore, has to be adjusted when targeting different domains.

SUMMARY: THE IR CHALLENGE In the previous paragraphs, we discussed different approaches to designing IRs for optimizing compilers and their limitations for domain-specific compilation. High-level domain-specific IRs prevent reuse across application domains. At the same time, low-level IRs complicate the lowering of DSLs, due to a large gap in abstraction, and they complicate developing optimizations, because domain knowledge has to be rediscovered. Constructing hierarchical IRs allows the reuse of single IR layers, but requires managing the progressive lowering between abstraction levels and requires the construction of new hierarchies for different application domains. Due to the inherent domain-specific nature of the IRs or the domain-specific composition of multiple IR layers used in state-of-the-art domain-specific compilers, extending and reusing IRs is still difficult.

Generally, the Intermediate Representation (IR) challenge is summarized as the following problem:

How to define an IR for high-performance domain-specific compilation that can be reused across application domains and hardware architectures while providing multiple levels of abstraction?

In Chapter 3, we will analyze the advantages of defining a domain-specific IR. We show the potential of high-performance domain-specific compilation by targeting modern NVIDIA GPUs and generating code that is competitive and even outperforms code tuned by human experts. In Chapter 4 of this thesis, we discuss and evaluate a novel approach towards using and extending a *domain-agnostic* IR for high-performance domain-specific compilation addressing the IR challenge.

1.2.2 *The Optimization Challenge*

Optimizing programs is essential for many application domains. Programs can be optimized with respect to different metrics, including energy efficiency or memory usage. In this thesis, we focus on optimizations aiming to improve a program's execution time, i. e., lowering the time it takes to execute a program on the targeted hardware device. The difference in execution time between an unoptimized program and its optimized version is often multiple orders of magnitude. For example, in the domain of machine learning, it is required to train deep neural networks before they can be used in the inference phase. The training phase is by far more time-intensive and requires to repetitively perform deep pipelines of computations like matrix multiplications on vast quantities of data. Optimizing deep learning computations allows us to decrease the required training time from months to hours or minutes. In order to achieve this improvement in performance, many optimizations must be applied to the program.

EXAMPLE: APPLYING HIGH-PERFORMANCE GPU OPTIMIZATIONS Optimizing programs for high-performance is a challenging task. Gradually improving the performance of an initial—correct but simple—implementation quickly leads to an explosion in both code size and complexity. For example, the following listing shows a simple matrix multiplication program written in CUDA.

```
__global__ void matmul(float *A, float *B, float *C, int K, int M, int N) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    float acc = 0.0;
    for (int k = 0; k < K; k++) {
        acc += A[y * M + k] * B[k * N + x];
    }
    C[y * N + x] = acc;
}
```

Here, every thread computes a single element of the output matrix `C` by computing the dot-product of a row of the `A` matrix and a column of the `B` matrix. This code is arguably easy to understand, given some basic knowledge about the CUDA programming model and GPU architectures. We will introduce both in Chapter 2. The shown code computes the correct result if the number of threads launched matches the size of the output to compute.

However, the performance achieved by this implementation when executed on a modern GPU is far from optimal because it has not been particularly optimized. For achieving high performance close to the practically achievable peak-performance of modern GPUs, the source code has to be significantly modified.

```
__global__ optimized_matmul(const __half *A, const __half *B, __half *C,
                            int K, int M, int N) {
  // ... 164 lines skipped
  #pragma unroll
  for (int mma_k = 1; mma_k < 4; mma_k++) {

    // load A from shared memory to register file
    #pragma unroll
    for (int mma_m = 0; mma_m < 4; mma_m++) {
      int swizzle1 = swapBits(laneLinearIdx, 3, 4);
      laneIdx = make_uint3(
        ((swizzle1 % 32) % 16), (((swizzle1 % 32)/16) % 2), (swizzle1/32));
      if (laneIdx.x < 16) { if (laneIdx.y < 2) {
        const int4 * a_sh_ptr = (const int4 *) &A_sh[((((warpIdx.y*64) +
        (mma_m*16)+laneIdx.x))*32)+((((((laneLinearIdx>>1))&3)^mma_k)*8))];
        int4 * a_rf_ptr = (int4 *) &A_rf[(mma_k & 1)][mma_m][0][0];
        *a_rf_ptr = *a_sh_ptr; }}}

    // load B from shared memory to register file
    #pragma unroll
    for (int mma_n = 0; mma_n < 4; mma_n++) {
      int swizzle2 = swapBits((swapBits(laneLinearIdx, 2, 3)), 3, 4);
      laneIdx = make_uint3(
        ((swizzle2%32)%16), (((swizzle2%32)/16)%2), (swizzle2/32));
      if (laneIdx.y < 2) { if (laneIdx.x < 16) {
        const int4 * b_sh_ptr = (const int4 *) &B_sh[
          ((((warpIdx.x*64) + (mma_n*16) + laneIdx.x)) * 32) +
          ((((((swapBits(laneLinearIdx,2,3))>>1))&3)^mma_k)*8))];
        int4 * b_rf_ptr = (int4 *) &B_rf[(mma_k & 1)][0][mma_n][0];
        *b_rf_ptr = *b_sh_ptr; }}}

    // compute matrix multiplication using tensor cores
    #pragma unroll
    for (int mma_m = 0; mma_m < 4; mma_m++) {
      #pragma unroll
      for (int mma_n = 0; mma_n < 4; mma_n++) {
        int * a = (int *) &A_rf[((mma_k - 1) & 1)][mma_m][0][0];
        int * b = (int *) &B_rf[((mma_k - 1) & 1)][0][mma_n][0];
        float * c = (float *) &C_rf[mma_m][mma_n][0];

        asm volatile( \
          "mma.sync.aligned.m8n8k4.row.col.f32.f16.f16.f32\n" \
          "    {%0, %1, %2, %3, %4, %5, %6, %7}, \n" \
          "    {%8, %9}, \n" \
          "    {%10, %11}, \n" \
          "    {%0, %1, %2, %3, %4, %5, %6, %7}; \n" \
               : "+f"(c[0]), "+f"(c[2]), "+f"(c[1]), "+f"(c[3])
               , "+f"(c[4]), "+f"(c[6]), "+f"(c[5]), "+f"(c[7])
               :  "r"(a[0]),  "r"(a[1])
               ,  "r"(b[0]),  "r"(b[1]));
        asm volatile( \
          "mma.sync.aligned.m8n8k4.row.col.f32.f16.f16.f32\n" \
          "    {%0, %1, %2, %3, %4, %5, %6, %7}, \n" \
          "    {%8, %9}, \n" \
          "    {%10, %11}, \n" \
          "    {%0, %1, %2, %3, %4, %5, %6, %7}; \n" \
               : "+f"(c[0]), "+f"(c[2]), "+f"(c[1]), "+f"(c[3])
               , "+f"(c[4]), "+f"(c[6]), "+f"(c[5]), "+f"(c[7])
               :  "r"(a[2]),  "r"(a[3])
               ,  "r"(b[2]),  "r"(b[3])); }}}
  // ... 95 lines skipped
}
```

Listing 1.2: CUDA code showing an optimized matrix multiplication implementation targeting modern NVIDIA GPUs. (321 lines of code in total.)

Listing 1.2 shows a snippet of an optimized matrix multiplication implementation targeting NVIDIA GPUs with Tensor Cores. This code achieves performance competitive with hand-tuned assembly implementations provided in libraries like cuBLAS. This high performance is only achievable by carefully applying algorithmic optimizations (e.g., computing small matrix multiplications instead of dot-products), as well as optimizations affecting the use of the multi-layer memory and compute hierarchy. For example, this implementation performs vectorized loads and stores to move data as efficiently as possible from the GPU's shared memory to the fast onchip register file (lines 7-30). Once the matrices are arranged using a specific memory layout in the GPU's register file, inline assembly instructions are used (lines 42 and 52) to compute small matrix multiply operations by calling specialized Tensor Core instructions.

Only the perfect orchestration of where, when, and how to apply these and more optimizations leads to the desired performance improvement. Unfortunately, this improvement comes at the cost of an increase in code size of about 30× and a significant increase in code complexity. Experimenting with, and applying different optimizations to adjust the code for targeting other hardware architectures is a challenging and immensely time-intensive process, even for experts. Due to these reasons, optimizing compilers aim to automate this process. Two key aspects are essential when designing optimizations for optimizing compilers targeting high-performance code generation: How to encode and how to apply optimizations as transformations on the IR? In the following, we briefly analyze how traditional and domain-specific optimizing compilers encode and apply optimizations and then identify problems preventing the current approaches from achieving high-performance domain-specific compilation.

GENERAL-PURPOSE OPTIMIZING COMPILERS General-purpose compilers like `clang` [133] typically encode optimizations as so-called *passes*. Each pass either encodes a specific optimization or performs an analysis creating meta-data to enable optimizations in subsequent passes. Passes are applied in a specific order, predetermined by the compiler developer, and the compilers expose different pass arrangements to the user as so-called *code generation options*.

Listing 1.3 shows the pass arrangement for `clang`'s most common option for generating high-performance code: `-O3`. Each of the 55 unique names in the listing stands for one pass to be applied, and the order of appearance reflects the order of pass application. Note that many passes, for example `-basiccg`, are applied multiple times, and in total, 251 passes are applied in this sequence. The `clang` documentation describes the `-O3` code generation option as follows: *-O3: Like -O2, except that it enables optimizations that take longer to per-*

```
Pass Arguments:  -targetlibinfo -tti -tbaa -scoped-noalias -assumption-cache-tracker -profile-summary-
      info -forceattrs -inferattrs -domtree -callsite-splitting -ipsccp -called-value-propagation -
      attributor -globalopt -domtree -mem2reg -deadargelim -domtree -basicaa -aa -loops -lazy-branch-
      prob -lazy-block-freq -opt-remark-emitter -instcombine -simplifycfg -basiccg -globals-aa -prune
      -eh -inline -functionattrs -argpromotion -domtree -sroa -basicaa -aa -memoryssa -early-cse-
      memssa -speculative-execution -basicaa -aa -lazy-value-info -jump-threading -correlated-
      propagation -simplifycfg -domtree -aggressive-instcombine -basicaa -aa -loops -lazy-branch-prob
       -lazy-block-freq -opt-remark-emitter -instcombine -libcalls-shrinkwrap -loops -branch-prob -
      block-freq -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -pgo-memop-opt -basicaa -aa -
      loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -tailcallelim -simplifycfg -
      reassociate -domtree -loops -loop-simplify -lcssa-verification -lcssa -basicaa -aa -scalar-
      evolution -loop-rotate -licm -loop-unswitch -simplifycfg -domtree -basicaa -aa -loops -lazy-
      branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -loop-simplify -lcssa-
      verification -lcssa -scalar-evolution -indvars -loop-idiom -loop-deletion -loop-unroll -mldst-
      motion -phi-values -basicaa -aa -memdep -lazy-branch-prob -lazy-block-freq -opt-remark-emitter
      -gvn -phi-values -basicaa -aa -memdep -memcpyopt -sccp -demanded-bits -bdce -basicaa -aa -loops
       -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -lazy-value-info -jump-
      threading -correlated-propagation -basicaa -aa -phi-values -memdep -dse -loops -loop-simplify -
      lcssa-verification -lcssa -basicaa -aa -scalar-evolution -licm -postdomtree -adce -simplifycfg
      -domtree -basicaa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -
      instcombine -barrier -elim-avail-extern -basiccg -rpo-functionattrs -globalopt -globaldce -
      basiccg -globals-aa -float2int -domtree -loops -loop-simplify -lcssa-verification -lcssa -
      basicaa -aa -scalar-evolution -loop-rotate -loop-accesses -lazy-branch-prob -lazy-block-freq -
      opt-remark-emitter -loop-distribute -branch-prob -block-freq -scalar-evolution -basicaa -aa -
      loop-accesses -demanded-bits -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -loop-
      vectorize -loop-simplify -scalar-evolution -aa -loop-accesses -lazy-branch-prob -lazy-block-
      freq -loop-load-elim -basicaa -aa -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -
      instcombine -simplifycfg -domtree -loops -scalar-evolution -basicaa -aa -demanded-bits -lazy-
      branch-prob -lazy-block-freq -opt-remark-emitter -slp-vectorizer -opt-remark-emitter -
      instcombine -loop-simplify -lcssa-verification -lcssa -scalar-evolution -loop-unroll -lazy-
      branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -loop-simplify -lcssa-
      verification -lcssa -scalar-evolution -licm -lazy-branch-prob -lazy-block-freq -opt-remark-
      emitter -transform-warning -alignment-from-assumptions -strip-dead-prototypes -globaldce -
      constmerge -domtree -loops -branch-prob -block-freq -loop-simplify -lcssa-verification -lcssa -
      basicaa -aa -scalar-evolution -block-freq -loop-sink -lazy-branch-prob -lazy-block-freq -opt-
      remark-emitter -instsimplify -div-rem-pairs -simplifycfg -verify
```

Listing 1.3: Passes applied in the `clang` compiler using `-O3`.

form or that may generate larger code (in an attempt to make the program run faster) [134]. Note how this description is intentionally vague because it is hard (if not impossible) to reason about the effect of applying `-O3` to an arbitrary program. To understand how an input program is transformed, one has to understand how every single pass is transforming the IR. Due to similarly imprecise or missing pass documentation, it is usually impossible to gain a precise understanding of the transformations without studying the low-level pass implementations themselves. For example, 31 out of the 55 passes used in `-O3` are not documented. The existing documentation is again vague; for instance the one for `adce`: *ADCE aggressively tries to eliminate code* [136].

The problems of selecting which pass to apply and in which order to apply them are known as the Phase Selection and Phase Ordering problems. Existing pass arrangements are based on best practices and heuristics and aim to improve the performance for the majority of input programs as a one-size-fits-all solution. For improving the performance of domain-specific computations, however, we desire precise control over where and how optimizations are applied. Modifying pass arrangements like `clangs` `-O3` code generation option is not sustainable for incorporating and applying domain-specific optimizations.

CONTROLLING OPTIMIZATIONS USING SCHEDULE LANGUAGES

Since general-purpose optimizing compilers often do not deliver the required performance for domain-specific computations, many high-performance implementations are still developed manually in low-level code. Intertwining the computation with optimizations complicates porting implementations to different hardware architectures. Halide [141], a domain-specific compiler developed for generating high-performance code for image processing pipelines, pioneered a new *schedule-based* approach to domain-specific compilation. A schedule-based compiler allows the separation of the program describing the computation (called *algorithm*) from the optimizations applied to it that are described in a separate program, the so-called *schedule*.

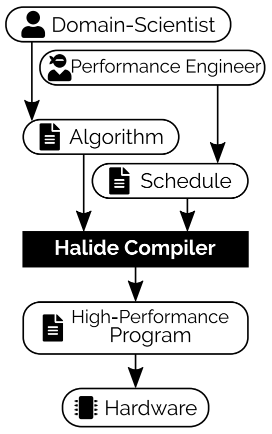


Figure 1.9: Overview of schedule-based compilation using Halide. A domain-scientist writes an algorithm describing the computation to perform. A performance engineer additionally writes a separate schedule describing the optimizations to apply to the algorithm. The algorithm and schedule are used by the Halide compiler to generate a high-performance program for the target hardware.

Figure 1.9 shows an overview of the schedule-based domain-specific compilation approach. The domain-scientist writes an algorithm in a high-level DSL. In addition, the compiler is provided with a schedule (in a separate scheduling language) describing the optimizations to apply. The domain-scientist can provide this schedule, or it can be refined or developed separately by a performance engineer for the targeted hardware architecture. In the case of Halide, the scheduling language contains primitives encoding common domain-specific optimizations that programmers typically apply to image processing pipelines. In addition to image processing, Halide is also capable of optimizing dense linear algebra computations such as matrix multiplications.

Listing 1.4 shows an algorithm expressing matrix multiplication in Halide as well as a schedule applying optimizations for targeting NVIDIA GPUs. Halide's implementation is embedded in C++, so the syntax used here is C++. Lines 2–4 define the matrix-matrix multiplication computation: `A` and `B` are multiplied by performing the dot product for each coordinate pair `(x,y)`. The dot product is expressed as pairwise multiplications followed by the reduction over domain `r` using the `+=` operator (line 3).

The other lines in Listing 1.4 define the schedule specifying the optimizations to be performed. The Halide compiler takes this C++ program and generates efficient GPU code coming close to manually optimized, low-level library code. By looking at the code in Listing 1.4, it is immediately apparent that writing Halide code is much simpler than optimizing programs manually. However, writing a schedule is still significantly more challenging than writing the algorithm describing matrix-matrix multiplication. Schedules are written using a sequence of API calls on the C++ objects representing the input (`A`, `B`) and output (`out`) data. The `prod` function represents the reduction operation in Halide's internal representation. While the algorithm and schedule are separated, they still share the same C++

```
 1 // the algorithm: functional description of matrix multiplication
 2 Var x("x"), y("y"); Func prod("prod"); RDom r(0, size);
 3 prod(x, y) += A(x, r) * B(r, y);
 4  out(x, y)  = prod(x, y);
 5
 6 // schedule for Nvidida GPUs
 7 const int warp_size = 32; const int vec_size = 2;
 8 const int x_tile    =  3; const int y_tile   = 4;
 9 const int y_unroll  =  8; const int r_unroll = 1;
10 Var xi,yi,xio,xii,yii,xo,yo,x_pair,xiio,ty; RVar rxo,rxi;
11 out.bound(x, 0, size).bound(y, 0, size)
12     .tile(x, y, xi, yi, x_tile * vec_size * warp_size,
13           y_tile * y_unroll)
14     .split(yi, ty, yi, y_unroll)
15     .vectorize(xi, vec_size)
16     .split(xi, xio, xii, warp_size)
17     .reorder(xio, yi, xii, ty, x, y)
18     .unroll(xio).unroll(yi)
19     .gpu_blocks(x, y).gpu_threads(ty).gpu_lanes(xii);
20 prod.store_in(MemoryType::Register).compute_at(out, x)
21     .split(x, xo, xi, warp_size * vec_size, RoundUp)
22     .split(y, ty, y, y_unroll)
23     .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
24     .unroll(xo).unroll(y).update()
25     .split(x, xo, xi, warp_size * vec_size, RoundUp)
26     .split(y, ty, y, y_unroll)
27     .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
28     .split(r.x, rxo, rxi, warp_size)
29     .unroll(rxi, r_unroll).reorder(xi, xo, y, rxi, ty, rxo)
30     .unroll(xo).unroll(y);
31 Var Bx = B.in().args()[0], By = B.in().args()[1];
32 Var Ax = A.in().args()[0], Ay = A.in().args()[1];
33 B.in().compute_at(prod, ty).split(Bx, xo, xi, warp_size)
34       .gpu_lanes(xi).unroll(xo).unroll(By);
35 A.in().compute_at(prod, rxo).vectorize(Ax, vec_size)
36       .split(Ax,xo,xi,warp_size).gpu_lanes(xi).unroll(xo)
37       .split(Ay,yo,yi,y_tile).gpu_threads(yi).unroll(yo);
38 A.in().in().compute_at(prod, rxi).vectorize(Ax, vec_size)
39       .split(Ax, xo, xi, warp_size).gpu_lanes(xi)
40       .unroll(xo).unroll(Ay);
```

Listing 1.4: Matrix-matrix multiplication in Halide. Lines 2–4 define the computation $A \times B$, the other lines define the schedule specifying the optimizations to be applied by the compiler. From: https://github.com/halide/Halide/blob/master/apps/cuda_mat_mul/mat_mul_generator.cpp.

identifiers and must, therefore, be written in the same C++ scope, limiting the reuse of schedules across algorithms.

The Halide schedule in Listing 1.4 uses 12 built-in optimization primitives (`bound`, `tile`, `split`, `vectorize`, `reorder`, `unroll`, `update`, `compute_at`, `store_in`, `gpu_blocks`, `gpu_threads`, `gpu_lanes`). Some of these optimizations are specific for the hardware (like `vectorize` or `gpu_threads`), others are generally useful algorithmic optimizations for many applications (like `tile` applying tiling to increase data locality), and others are low-level optimizations (like `unroll` and `reorder` that transform loop nests). Some primitives' behavior is not intuitive and the documentation provides only informal descriptions, e.g., for `update`: *"Get a handle on an update step for the purposes* *of scheduling it."* The lack of precise semantics or even informal descriptions of the optimization primitives makes reasoning about the schedule difficult. For example, it is unclear why lines 21–23 are repeated at lines 25–27 with calls to `unroll` and `update` in between.

The Halide framework is not easily extensible. Adding a new optimization primitive to the schedule API requires extending the Halide compiler. For example, a primitive like `tile` that can be implemented with `split` and `reorder` [73] is represented not as a composition but rather provided as a built-in abstraction.

Even though schedule-based, domain-specific compilation achieves the desired performance and exposes fine-grained control over where, when, and how optimizations are applied, the approach remains inherently domain-specific due to a fixed set of built-in domain-specific optimization primitives. More crucially, the so-called scheduling languages lack desirable properties of actual programming languages that would allow us to define custom user-defined abstractions for optimizations to make them reusable across multiple domains.

SUMMARY: THE OPTIMIZATION CHALLENGE In the previous paragraphs, we discussed the importance of optimizing domain-specific computations and the challenges for automating this process using optimizing compilers. Generally, significantly optimizing performance comes at the cost of increasing code size and complexity. Optimizing compilers aim to automate this process, but general-purpose compilers are not suitable for high-performance domain-specific compilation. They only provide a one-size-fits-all solution where precise control is needed. Schedule-based domain-specific compilers offer this control; however, their schedule languages are not extensible and inherently domain-specific, thus not portable to other application domains.

Generally, the Optimization Challenge is summarized by the following question:

How can we encode and apply domain-specific optimizations for high-performance code generation while providing precise control and the ability to define custom optimizations, thus achieving a reusable optimization approach across application domains and hardware architectures?

Chapter 3 will demonstrate the optimization potential of domain-specific compilation with an in-depth case study of the domain-specific compiler that generated the optimized CUDA code shown in Listing 1.2. In Chapter 5 of this thesis, we introduce, discuss, and evaluate a novel language for specifying optimization strategies as a possible solution to the optimization challenge.

1.3 CONTRIBUTIONS OF THIS THESIS

This thesis proposes a new approach to the construction of compilers. We aim to achieve the benefits of domain-specific compilation targeting different application domains and hardware architectures without the need to construct multiple domain-specific compilers.

This thesis makes the following three contributions:

For demonstrating the potential of domain-specific compilation:

A COMPILER FOR OPTIMIZING DATA MOVEMENTS ON GPUS
We introduce Fireiron, a novel domain-specific compiler targeting GPUs. We demonstrate the importance of efficient data movements and develop a domain-specific scheduling language for their optimization. In an in-depth case study of matrix multiplications, we evaluate the advantages of domain-specific compilation and compare the generated code to high-performance library implementations. We show that Fireiron generates code that achieves performance on par or even outperforms code tuned by human performance engineers. We also discuss the construction and drawbacks of developing a domain-specific compiler from scratch, leading to the two following contributions.

For addressing the Intermediate Representation Challenge:

EXTENDING THE LIFT IR FOR STENCIL COMPUTATIONS
We show how a purely functional intermediate representation can be made suitable for addressing the IR challenge. By defining and composing a small set of domain-agnostic computational building blocks, we provide high-level domain-specific abstractions. Specifically, we extend the LIFT intermediate representation, which so far has been used for expressing dense linear algebra computations, for adding support for expressing and optimizing stencil computations. We also show that the achieved performance is on par with existing domain-specific compilers and handwritten benchmarks.

For addressing the Optimization Challenge:

A LANGUAGE FOR DESCRIBING OPTIMIZATION STRATEGIES
We present ELEVATE, a novel language allowing to specify high-performance program optimizations as strategies based on rewrite rules. Similar to the LIFT IR, which provides generic computational building blocks, ELEVATE defines a small set of domain-agnostic building blocks for specifying program transformations. Composing those allows us to express domain-specific optimizations. We show how to implement modern scheduling languages from first principles; however, without having the drawbacks of existing schedule-based compilers. Finally, we evaluate our optimization approach by comparing against state-of-the-art domain-specific compilers.

1.4 OUTLINE OF THIS THESIS

The remainder of this thesis is structured as follows.

CHAPTER 2 introduces the necessary background required for understanding the contributions made in this thesis. We briefly introduce modern parallel hardware architectures and the parallel programming models we use throughout this thesis.

CHAPTER 3 demonstrates the necessity for and the advantages of domain-specific compilation. We introduce Fireiron, a domain-specific scheduling language, and compiler for generating high-performance matrix multiplication kernels targeting NVIDIA GPUs. We demonstrate how high-performance GPU implementations are concisely expressed using Fireiron's high-level abstractions. We show that our domain-specific compiler achieves performance competitive to manually tuned library implementations. Finally, we discuss the drawbacks of developing a domain-specific compiler from scratch.

CHAPTER 4 introduces our solution for the IR Challenge. We demonstrate how domain-specific computations are expressible using a domain-agnostic IR. Specifically, we show that we can reuse the existing LIFT IR for expressing domain-specific stencil computations by decomposing them into three generic building blocks. We extend the LIFT IR with computational primitives implementing these generic building blocks and show that re-composing those allows us to express complex multi-dimension stencil computations.

CHAPTER 5 presents our solution for the Optimization Challenge. We introduce ELEVATE, a language for expressing program optimization as rewrite strategies. Inspired by existing strategy languages for term-rewriting systems, we show how to define complex program optimizations as strategies that compose simple rewrite rules. As a case study, we implement TVM's state-of-the-art scheduling language for optimizing machine learning computations from first principles and show that our approach achieves the same performance without having the identified drawbacks.

CHAPTER 6 presents a holistic approach towards domain-specific compilation, which unifies the contributions explained in the previous technical chapters. After briefly summarizing each separate contribution, we discuss how the concepts introduced in this thesis can be further combined and extended.

CHAPTER 7 concludes with a comparison against related work.

2 BACKGROUND

In this chapter, we describe the technical background required to understand the contributions of this thesis. Background that is specific only to a particular chapter will be separately introduced later, and related work that is not necessarily required for understanding our contributions is discussed and compared to our work in Chapter 7.

In the following chapters, we introduce approaches to domain-specific compilation for achieving high performance on modern parallel processors. Therefore, we begin by describing the architecture of CPUs and GPUs, the two hardware architectures targeted by the techniques discussed in the following chapters. Instead of giving an in-depth overview of these architectures, we merely focus on introducing their core functionality and the requirements for achieving high performance.

Afterward, we introduce parallel programming models that allow to program CPUs and GPUs. Specifically, we introduce OpenMP for programming CPUs and CUDA and OpenCL for programming GPUs because these are the target languages generated by the compilers discussed in the following chapters.

2.1 MODERN PARALLEL PROCESSORS

Due to the end of Dennard scaling[1] [43] and Moore's Law[2] [106], parallel processors are ubiquitous today. Traditionally, the performance of a processor scaled linearly with its clock frequency. At about 2005, due to reaching physical limits, hardware manufacturers could not increase the clock frequency of their processors further. This limitation led to the introduction of *parallel processors*.

[1]*Dennard Scaling: As transistors get smaller, their power density stays constant. [43]*

[2]*Moore's Law: The number of transistors in an integrated circuit doubles about every two years. [106]*

A parallel processor generally contains two or more *cores* that can perform independent computations simultaneously. The advent of parallel processors allowed hardware manufactures to continue producing more powerful hardware, which came at the cost of programmers. Since then, programmers have to explicitly manage the available parallelism to achieve the targeted parallel hardware's full potential.

As discussed in Chapter 1, domain-specific compilers aim to simplify the development of parallel programs for achieving high performance on parallel processors. To understand the challenge for programmers and domain-specific compilers in achieving high performance on modern parallel processors, in the following, we briefly introduce the architectures of modern CPUs and GPUs, the two most common parallel hardware architectures.

Generally, CPUs are classified as latency-oriented architectures, whereas GPUs are throughput-oriented [57]. Latency-oriented hardware architectures aim to minimize the runtime of programs by minimizing the latency between the start and end of executing a single thread of instructions using various sophisticated techniques such as out-of-order execution and a hierarchy of caches. Throughput-oriented hardware architectures instead aim to minimize a program's runtime by maximizing the computational throughput using multiple simple cores for executing many tasks in parallel.

2.1.1 *Multi-Core CPUs*

Example: Intel i7 (Ice Lake Architecture)
8MiB L3 cache per CPU
512KiB L2 cache per core
32KB L1 instr. cache per core
48 KiB L1 data cache per core

A multi-core CPU consists of multiple independently operating cores that perform computations in parallel. A CPU also contains a hierarchy of caches, small but fast memory located close to the cores. This hierarchy is managed automatically by different so-called *replacement policies* that move data between the cache levels transparently for the programmer or compiler. The cache hierarchy typically consists of three levels named L1, L2, and L3. The first two levels are private to each core, and all cores share a common slowest but largest third cache level. The fastest and smallest first cache level closest to each core is typically divided into two parts: an instruction cache and a cache for data.

Each CPU core is capable of executing a separate thread of instructions. Being latency-oriented, CPUs aim to execute the instructions of a single serial thread as fast as possible. In the following, we briefly explain three kinds of parallelism that CPUs exploit to minimize this latency: *Instruction-Level Parallelism*, *Thread-Level Parallelism*, and *Data Parallelism*.

INSTRUCTION-LEVEL PARALLELISM Generally, Instruction-Level Parallelism (ILP) denotes the ability of a CPU core to execute multiple instructions in parallel. Different micro-architectural approaches exist to exploit ILP.

Instruction Pipelining describes a technique to split a single instruction into a series of sequential steps that are each executed by a different processing unit. Five subsequent so-called micro-operations describe a typical pipeline for executing a single instruction: instruction-fetch, instruction-decode, execute, memory access, and write-back. Splitting an instruction into multiple micro-operations allows for overlapping the execution of instructions and minimizes the overall latency of executing a program. For example, as soon as the first instruction is fetched, this instruction will be decoded while the next instruction can be fetched simultaneously.

A *superscalar* processor contains multiple execution units allowing to process several instructions in parallel by, for example, fetching two instructions simultaneously using separate execution units. A CPU contains several kinds of execution units, including arithmetic logic units (ALU), address generation units (AGU), and load-store units (LSU). For example, the *Sunny Cove* core architecture used in Intel's latest *Ice Lake* CPU architecture describes a superscalar core that can execute up to 10 instructions per cycle.

Other techniques to implement and improve the efficacy of ILP are out-of-order execution, register renaming, speculative execution, and branch prediction. However, all techniques for exploiting instruction-level parallelism lie outside the programmer's control and are performed transparently by the hardware. In contrast, the programmer or compiler must explicitly use the following two kinds of parallelism that we briefly describe.

THREAD-LEVEL PARALLELISM As mentioned earlier, the advent of parallel hardware architectures such as multi-core CPUs put a significant burden on the programmer. Thread-level parallelism refers to the ability to execute a separate thread of instructions independently per CPU core. To achieve high performance on multi-core CPUs, the programmer must develop parallel programs that saturate the available cores by explicitly managing multiple threads. In the next section, we briefly introduce programming models that enable the development of such parallel programs.

Executing a single thread per core is often not enough to use the available execution units on a core as efficiently as possible. *Simultaneous Multi-Threading* (SMT) is a technique for improving the efficiency of superscalar CPUs by enabling the execution of multiple threads on the same core. Sharing the resources of a core among multiple threads allows keeping execution units busy that would be idle without SMT during the execution of only a single thread. Modern CPUs typically support the execution of 2-4 threads per core. For example, Intel's implementation of SMT is called *Hyper-threading*, and the i7 Ice Lake processors support the execution of two threads per core. Typically, the scheduling, i.e., the mapping of which core executes which thread happens transparently from the programmer. However, SMT increases the number of parallel threads a programmer has to manage.

DATA PARALLELISM Most modern CPU architectures support a third kind of parallelism by providing so-called Single Instruction Multiple Data (SIMD) instruction extensions like Streaming SIMD Extensions (SSE) or Advanced Vector Extensions (AVX). SSE and AVX are examples for instruction set extensions that provide SIMD instructions to manipulate multiple data elements (vectors) using a single instruction. A CPU that supports extensions like SSE and AVX contains dedicated specialized execution units that perform operations on vector registers. For example, Intel's *Ice Lake* CPUs support the latest AVX-512 extension, which entails that they contain special 512-bit wide vector registers that can, for example, hold up to 16 single-precision floating-point values. AVX-512 instructions operate on these vector registers and exploit data parallelism by operating on several values (e.g., 16 floats) at the same time. For example, instead of sequentially adding floating-point values using 16 traditional add-instructions, a single AVX-512 add instruction allows the computation of the same results in one clock cycle by adding the contents of two vector registers.

Again, it is the programmer's or the compiler's task to exploit data parallelism in a program by explicitly using the operations contained in the various SIMD instruction set extensions.

In the following, we briefly introduce the architecture of GPUs and discuss the differences compared to the CPU architecture.

2.1.2 *Many-Core GPUs*

In contrast to CPUs, which contain only a few but complex cores to minimize the latency of executing separate threads, GPUs instead contain many simple cores to perform a large number of tasks in parallel. Here, simple refers to the lack of sophisticated techniques such as branch prediction or speculative execution. The GPU design is throughput-oriented, i.e., maximizing the overall work done in a given time frame, by focusing on performing many tasks in parallel instead of optimizing the execution of a single serial thread.

Example:
NVIDIA A100 [118]
(Ampere Architecture)
108 SMs per GPU
64 INT cores per SM
64 FP32 cores per SM
32 FP64 cores per SM
432 Tensor Cores per GPU

COMPUTE HIERARCHY The architecture of a GPU is generally designed to have hundreds of simple cores capable of executing thousands of threads in parallel to maximize throughput. NVIDIA's A100 GPU of the recently introduced *Ampere* architecture [118], for example, contains 108 so-called Streaming Multiprocessors (SM), which correspond to the CPU's cores. Each SM contains multiple execution units that are called *cores* in NVIDIA architectures. The whole A100 GPU, for example, contains a total of 6912 integer and single-precision floating-point cores, and 3456 double-precision floating-point cores to perform computations in parallel. Additionally, since the introduction of the *Volta* architecture [110] released in 2017, modern NVIDIA GPUs contain specialized cores for computing machine-learning-specific workloads. These are called *Tensor Cores*, and they are used for computing matrix multiply and accumulate operations and achieve a significant speedup compared to performing the same computation using traditional cores.

MEMORY HIERARCHY The memory hierarchy of a GPU consists of multiple levels. The slowest but largest memory region on a GPU is the off-chip DRAM that is shared among all SMs. Similar to the CPU, a GPU also contains a hierarchy of caches (L2, L1, and L0 on Ampere) that are managed by the hardware transparently to the programmer. For example, the A100 contains a 40MB L2 cache that is shared among all SMs of the GPU. However, in contrast to CPUs, a GPU additionally contains a so-called *shared memory*. Shared memory can be viewed as a software-managed cache entirely in control of the programmer and shared among all cores of an SM. On the A100, each SM has a combined 192 KB L1 data cache and shared memory. The shared memory size on the Ampere architecture is software-configurable to take up as much as 164 KB of the 192KB region; the rest is automatically dedicated to the L1 cache.

Example:
NVIDIA A100 [118] (Ampere Architecture):
40GB DRAM per GPU
40MB L2 per GPU
192KB (L1/sh.mem.) per SM
256KB Register File per SM

The fastest memory on a GPU is the register file. Due to the high number of cores, a GPU typically also contains a large register file to provide enough fast storage for operands and temporary results to the single cores. For example, all recent NVIDIA architectures (e.g., Pascal, Volta, or Ampere) contain 65536 32-bit registers per SM [117].

SIMT-ARCHITECTURE Single Instruction Multiple Threads (SIMT) describes the execution model implemented by NVIDIA GPUs [111]. As mentioned before, the Streaming Multiprocessor is designed to execute thousands of threads in parallel. On a GPU, threads are scheduled and executed in small groups of 32 threads called *warps* (or wavefronts on AMD GPUs).

A warp always executes the same instruction at a time, hence the name SIMT. Threads of the same warp might diverge in control-flow, for example, by taking different paths after a branch. In this case, a warp executes each branch taken sequentially, and threads that are not on the current path (i.e., so-called *passive threads*) are disabled – this is called *branch diversion*. For achieving high GPU performance, branch diversion must be avoided whenever possible to always have 32 active threads performing the same computation.

MEMORY ACCESS COALESCING AND BANK CONFLICTS Besides avoiding branch diversion, two other crucial optimizations for improving GPU performance are *memory access coalescing* and the avoidance of shared memory *bank conflicts*.

When a warp accesses data stored in the GPU's DRAM, the load operation's efficiency depends on whether the DRAM access is coalesced. DRAM is accessible using 32-, 64-, and 128-byte transactions. If all memory accesses of the threads of a warp fall within the same aligned memory region, all 32 loads *coalesce* into a single load transaction for the whole warp. A memory region is *aligned* if its start address is a multiple of 32, 64, or 128. For example, if all 32 threads of a warp access adjacent and aligned single-precision floating-point values (i.e., four bytes), the whole transaction is performed using a single 128-bit load instruction. Depending on the size of the word accessed by a thread, the warp's access pattern, and the targeted GPU architecture, multiple load instructions might be required if accesses cannot be coalesced.

For improving shared memory accesses, a programmer must consider how shared memory is implemented in hardware. On a GPU, shared memory is organized in so-called banks. Successive 32-bit words are organized into 32 banks (16 banks on pre-Fermi architectures) that are simultaneously accessible by the threads of a warp. As soon as multiple threads access the same bank, a so-called *bank conflict* occurs, and the accesses must be serialized, which reduces the shared memory throughput. To achieve high performance on GPUs, shared memory load and store conflicts must be avoided whenever possible by optimizing the warp's shared memory access pattern.

TENSOR CORES Finally, we briefly discuss NVIDIA's Tensor Cores in more detail as it is required to use them efficiently to achieve near peak performance. Tensor Cores were introduced with the Volta architecture and are designed to accelerate machine learning workloads specifically. Since low-precision matrix multiplication is at the core of the majority of machine learning computations, Tensor Cores provide new instructions that multiply and accumulate small matrices very efficiently. On Volta, for example, multiplying matrices using Tensor Cores provides up to 12× higher peak FLOPS [110] than performing the same computation using regular fused-multiply-add (FMA) instructions. The multiplication is typically performed using half-precision (16-bit floating-point), while the accumulation can be performed in up to single precision. Newer architectures like Turing and Ampere support more low-precision formats, including an 8-bit integer or a new so-called 32-bit *Tensor Float* [118].

NVIDIA's PTX assembly [109] exposes Tensor Cores in two ways: 1) The `wmma.mma` instruction performs warp-wide matrix multiplications (the supported operand shapes depend on the used data type). 2) The `mma` instruction performs more fine-grained matrix multiplications executed by groups of eight specific threads, which we will call *quad-pairs* in the following. Both instructions operate on small collections of registers, so-called register fragments. The operand matrix values must be distributed across the quad-pair's register fragments precisely according to complex and prescribed mappings defined in the ISA. Achieving high-performance using Tensor Cores is challenging due to the low-level and fine-grained control required to use Tensor Core instructions correctly.

1 Warp = 4 Quad-Pairs
1 QP = 8 Threads (T)
QP0: T0-3 & T16-19
QP1: T4-7 & T20-23
QP2: T8-11 & T24-27
QP3: T12-15 & T28-31

2.2 PROGRAMMING PARALLEL PROCESSORS

In this section, we describe programming models for targeting parallel hardware such as multi-core CPUs and many-core GPUs with Tensor Cores. We specifically only focus on the programming models we use in the subsequent chapters and defer the discussion of related approaches to Chapter 7.

To achieve the highest performance on CPUs and GPUs, parallel threads must be managed to saturate all available cores. There exists a wide variety of parallel programming models that allow us to develop such parallel programs. In the following, we begin with a brief introduction of OpenMP that we use for programming multi-core CPUs. Afterward, we describe the CUDA and OpenCL programming models that we use to target GPUs. Generally, the compilers we introduce in the following technical chapters generate parallel OpenMP, OpenCL, and CUDA programs for targeting different CPU and GPU architectures.

2.2.1 *OpenMP*

OpenMP [15] is an API for developing parallel C, C++, and Fortran programs. It has been introduced in 1997 and was initially designed for parallelizing sequential programs targeting the CPU. Since version 4.5, OpenMP can also be used to target accelerators like GPUs. Chapter 5 will introduce the RISE programming language whose compiler generates parallel OpenMP code for targeting multi-core CPUs. In the following, we only briefly explain the OpenMP features used by the RISE compiler.

OpenMP provides a set of so-called *pragmas* that a programmer or compiler uses to annotate a sequential program. An OpenMP pragma specifies how and which regions of a sequential program are executed in parallel. A compiler supporting OpenMP compiles the annotated regions into parallel code that, for example, uses multiple threads to exploit the available cores of the targeted parallel hardware.

Listing 2.1 shows two examples for the parallelization of sequential C loops using OpenMP. Annotating the first loop using `#pragma omp parallel for` (line 2) causes the body of the loop to be executed using multiple threads. The exact number of threads launched for executing this loop is determined at runtime following an algorithm defined in the OpenMP specification [15]. Generally, OpenMP uses the *fork-join* model of parallel execution. At the beginning of the execution of an OpenMP program, a single thread is used. As soon as this initial thread reaches a parallel region introduced by `#pragma omp parallel`, it forks multiple child threads for computing the annotated region in parallel. In our example, the for-loop body

```
1 void openmp_example(int n, float *a) {
2   #pragma omp parallel for
3   for(int i = 0; i < n; i++)
4     a[i] = a[i] * 2.0f; // parallel execution
5
6   #pragma omp simd
7   for(int i = 0; i < n; i++)
8     a[i] = a[i] * 2.0f; // vectorized execution
9 }
```

Listing 2.1: Two examples for parallelizing a sequential loop using OpenMP pragmas. The first loop is parallelized by using multiple threads to compute the output elements. The second loop is parallelized by instructing the compiler to generate SIMD vector instructions for computing the output elements.

is executed in parallel, and multiple threads compute independent elements of the output array. Reaching the end of the parallel region (line 5) causes the termination of all forked threads (join). The single initial thread continues executing the program until another parallel region is approached.

Using `#pragma omp simd` (line 6) is another way to exploit parallelism in sequential programs. The `simd` keyword instructs the compiler to generate SIMD vector instructions for computing the body of the loop.

The OpenMP API provides many more pragmas and so-called clauses that expose more fine-grained control about the parallelization. However, the `parallel for` and `simd` directives already suffice to achieve high competitive performance on modern CPUs, as we will discuss in Chapter 5.

2.2.2 *CUDA and OpenCL*

In this section, we briefly introduce CUDA and OpenCL, two parallel programming models we use for targeting GPUs. OpenCL is capable of targeting multiple other architectures, including CPUs; however, we restrict this OpenCL introduction for brevity and only discuss the programming of GPUs. In fact, we will mainly introduce CUDA and end this section with a brief comparison to OpenCL since both follow a similar approach when targeting GPUs.

CUDA is a parallel programming model introduced in 2006 by NVIDIA for enabling general-purpose computing on GPUs. As their name suggests, GPUs were initially designed and mostly used for processing graphics applications that are inherently data-parallel. CUDA enables programmers to use GPUs for application domains besides graphics-processing, e.g., dense linear algebra computations, which also significantly benefit from the parallel GPU architecture.

KERNELS AND COMPUTE HIERARCHY CUDA is a language and runtime extension for C/C++ and enables the definition of special functions called *kernels* that are computed on the GPU. In the CUDA programming model, we differentiate between a *host* (typically the CPU) and a *device*, the GPU. A kernel is defined using the `__global__` qualifier and launched on the device using a special syntax that allows to specify how many threads execute it. Generally, the same instance of a kernel is executed by all threads – following the Single Program Multiple Data (SPMD) idiom. Every thread is identified by a unique ID that is accessible within a kernel using the keyword `threadIdx`. Using the thread-ID, for example, to access an array in a kernel, allows threads to process different values and achieves data parallelism.

Threads are arrangeable in 1D, 2D, or 3D so-called *thread-blocks*, which themselves can be arranged in up to 3D so-called *grids*. This hierarchy mirrors the compute hierarchy we find in the GPU architecture: At runtime, blocks of a grid are scheduled to be executed on the GPU's SMs while threads of a block are executed on the cores of an SM. As mentioned before, on the hardware, threads are scheduled and executed as warps - groups of 32 threads. However, this happens transparently to the programmer, and it is not necessary, though beneficial in terms of performance, to define the block size to be a multiple of 32.

MEMORY HIERARCHY The CUDA memory hierarchy similarly mirrors the different memories existent on a GPU, and consists of so-called *global-*, *shared-*, and *local memory*. Global memory corresponds to the GPUs DRAM and is accessible by all blocks of a grid. Furthermore, it is persistent across kernels launches. The CUDA API provides functions for the data management between the host and the device, including data allocation and transferring data to and from the GPU. In the following chapters, we are mostly interested in generating high-performance kernels and, therefore, omit a further discussion about efficient host-to-device data transfers.

Shared memory is accessible by all threads of a block and is allocated using the `__shared__` qualifier. Local memory is private to each thread and is mapped into the GPUs DRAM. Whenever possible, the compiler uses the fast register file for thread-private variables. However, if a kernel requires more registers than available on the SM, the remaining variables are automatically *spilled* into the slow local memory.

Finally, CUDA provides additional read-only memory regions: *constant-*, *texture-*, and *surface memory*, which are also mapped to the DRAM and optimized for different memory usages.

COMPILATION AND INLINE ASSEMBLY At compile-time, the NVIDIA `nvcc` compiler separates all host code from device code, i.e., the kernel functions annotated with `__global__`. The host code is compiled with a standard C/C++ compiler, such as `clang`. The device code is compiled to PTX, NVIDIA's stable virtual machine ISA. PTX is not the assembly code eventually executed on the targeted GPU. Instead, it is an intermediate language that enables the execution of the same compiled program across different GPU architectures. At runtime, the PTX code is just-in-time compiled into the actual machine code of the targeted hardware. The intermediate step via PTX enables the execution of the same code on multiple architectures, including GPUs that will be introduced in the future whose machine code is not available yet.

Even though GPU kernels are typically developed in CUDA, it is sometimes beneficial to explicitly use specific PTX instructions. These typically offer more-fine grained control compared to what is expressible in pure CUDA. This is especially true for targeting Tensor Cores introduced with the Volta architecture. The CUDA API exposes Tensor Cores using the WMMA API [119], a library providing functions for executing warp-wide matrix multiplications. PTX instead, additionally exposes the quad-pair-level `mma` instructions, which offer more flexibility and, therefore, sometimes lead to higher performance, as we will show in the next chapter.

CUDA allows the use of PTX instructions via inline assembly [112], similar to how intrinsics can be inserted into C programs. The `asm` keyword enables the insertion of PTX instructions into CUDA code. For example, the following code is translated into the `add.s32` PTX instruction:

```
asm("add.s32 %0, %1, %2;" : "=r"(i) : "r"(j), "r"(k));
```

Here, `%0-2` represent arguments to the PTX `add.s32` instruction. During compilation, these arguments are replaced by the CUDA values `i`, `j`, and `k`. The letter `r` specifies that these values reside in unsigned 32-bit registers, and `=r` specifies that this register is written to. In the next chapter, we show more examples of how to use inline assembly for developing high-performance GPU kernels with CUDA.

COMPARISON TO OPENCL OpenCL is an open standard [61] for the parallel programming of heterogeneous platforms introduced in 2009 and developed and maintained by the Khronos group. OpenCL is supported by a wide range of devices from different hardware vendors, including NVIDIA, AMD, ARM, Intel, and runs on various architectures, including CPUs, GPUs, as well as other kinds of accelerators such as FPGAs.

In the following chapters, we use OpenCL to program GPUs only. For this purpose, OpenCL can be viewed as a programming model

CUDA	OpenCL
Streaming Multiprocessors (SMs)	Compute Units
Cores	Processing Elements
Thread Block	Work-Group
Thread	Work-Item
Global Memory	Global Memory
Constant Memory	Constant Memory
Shared Memory	Local Memory
Local Memory	Private Memory

Table 2.1: Comparing OpenCL and CUDA concepts.

similar to CUDA, which merely uses different names for the same concepts. For example, OpenCL also differentiates between a host and a device that is used for offloading computations. Since OpenCL runs on multiple architectures, a device is defined as an abstract processor consisting of *Compute Units* that themselves contain *Processing Elements*. If OpenCL is used to target GPUs, the compute units map to the GPUs SMs, and the processing elements map to the SM's cores.

Essentially, OpenCL supports the same compute and memory hierarchy as defined by CUDA, and Table 2.1 compares the names used in both programming models for the same concepts. In the next chapter, we use the CUDA terminology because we introduce a domain-specific compiler that generates high-performance CUDA kernels. In Chapter 4, we use the OpenCL terminology because we introduce a compiler IR that is translated into high-performance OpenCL kernels.

3 FIREIRON: DOMAIN-SPECIFIC COMPILATION FOR GPUS

In this chapter, we demonstrate the advantages of domain-specific compilation by introducing Fireiron: A framework that includes a data-movement-aware IR, scheduling language, and domain-specific compiler for generating high-performance GPU matrix multiplication implementations. Fireiron is aimed at performance experts. It provides high-level abstractions for expressing GPU optimizations that are unavailable in other compilers and which so far must be manually written in assembly instead.

This chapter is largely based on the publication: "Fireiron: A Data-Movement-Aware Scheduling Language for GPUs" [66] by Hagedorn, Elliott, Barthels, Bodik, and Grover published at PACT'20.

We start by motivating the need for a schedule-based domain-specific compiler targeting GPUs and matrix multiplications specifically. We identify the importance of expressing precise data movement optimizations and targeting low-level, hardware-specific instructions. After discussing the limitations of existing compilers, we introduce Fireiron's IR, the scheduling language for constructing it, and how to generate high-performance CUDA code.

We evaluate Fireiron on three GPU architectures against expert-written advanced matrix multiplications. First, we show that Fireiron schedules can express the optimizations applied in these implementations while requiring about 6× less lines of code. Second, we show that the code optimized by Fireiron schedules outperforms the fastest implementations (provided by the manually-optimized cuBLAS library) by more than 2×. We conclude this chapter by identifying the advantages of our approach and discussing the drawbacks of developing a domain-specific compiler from scratch.

3.1 INTRODUCTION

Developing high-performance GPU kernels is challenging because of their complex multi-layered compute and memory hierarchies. Only if a kernel makes optimal use of both hierarchies, implementations achieve performance close to the hardware's theoretical peak.

On modern GPUs, achieving optimal performance essentially boils down to carefully organized data movements and the precise use of specialized hardware units such as NVIDIA's Tensor Cores. Today, there remains a significant gap between optimizing compilers versus what human experts achieve by hand-tuning implementations using low-level assembly. Figure 3.1 (a) shows the performance of the best matrix multiplication implementations we found for Halide [141]

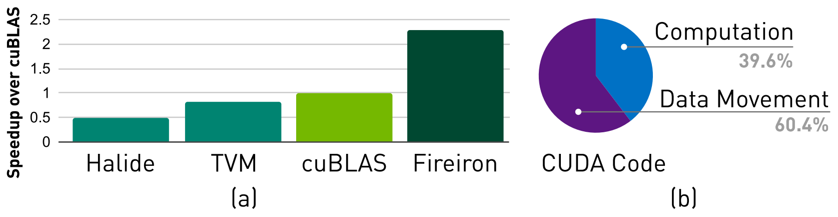


Figure 3.1: (a) Comparing Halide's [72] and TVM's [173] matrix multiplication performance against cuBLAS (higher is better) reveals a significant remaining gap in performance. Fireiron allows GPU experts to specify implementations that even outperform hand-tuned cuBLAS library code. (b) The Fireiron-generated CUDA code achieving this performance contains mostly data movements, which motivates a scheduling language where data movements are first-class constructs.

and TVM [29], two state-of-the-art compilers, compared to the performance achieved by NVIDIA's experts providing manually tuned implementations in the high-performance cuBLAS library. Manually developing efficient kernels is time-intensive and error-prone even for experts, and, more crucially, it complicates experimentation and thus hinders potentially unlocking even higher performance.

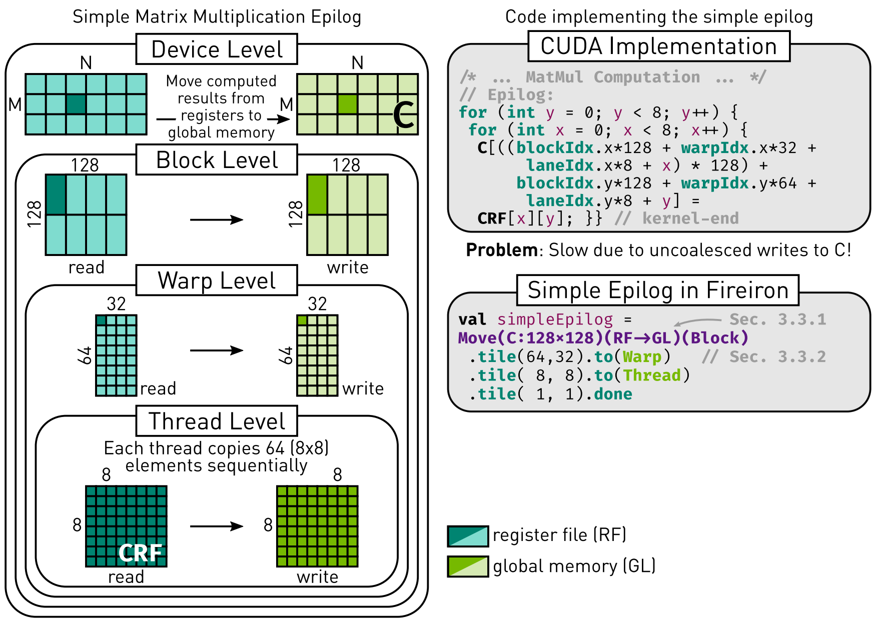
Schedule-based compilation [29, 141], which gained popularity with the introduction of Halide, is a huge step towards providing experts with a powerful tool for developing high-performance programs. However, the current approaches still prevent experts from closing the performance gap because they treat data movements as second-class citizens: In order to unlock the highest performance, it is crucial to define precise mappings of computations to parallel compute units but also how data movements are coordinated through the memory hierarchy.

In this chapter, we propose Fireiron, a scheduling language, IR, and compiler for performance experts. With Fireiron, programmers can define where computations *and* data movements take place. This control is required to unlock the potential of specialized hardware such as Tensor Cores. We make the following contributions:

1. We introduce a compiler IR in which both computations and *data movements* are first-class citizens, meaning that they can be scheduled with the same primitives.

2. Our scheduling language provides high-level abstractions for *gradually decomposing* computations and data movements until they match assembly instructions, accelerator primitives, or predefined microkernels, each represented using precise *specifications*.

3. We show that Fireiron schedules can express optimizations used in handwritten kernels while requiring 6× less code. With Fireiron, experts can develop high-performance GPU kernels computing matrix multiplications that even outperform cuBLAS hand-tuned library implementations by more than 2×.


Simple Matrix Multiplication Epilog
Device Level
N
M
Move computed results from registers to global memory
N
M
C
Block Level
128
128
read
128
128
write
Warp Level
32
64
read
32
64
write
Thread Level
Each thread copies 64 (8x8) elements sequentially
8
8
CRF
read
8
8
write
Code implementing the simple epilog
CUDA Implementation
Problem: Slow due to uncoalesced writes to C!
Simple Epilog in Fireiron
register file (RF)
global memory (GL)

```
/* ... MatMul Computation ... */
// Epilog:
for (int y = 0; y < 8; y++) {
 for (int x = 0; x < 8; x++) {
  C[((blockIdx.x*128 + warpIdx.x*32 +
       laneIdx.x*8 + x) * 128) +
      blockIdx.y*128 + warpIdx.y*64 +
       laneIdx.y*8 + y] =
  CRF[x][y]; }} // kernel-end
```

```
val simpleEpilog =            Sec. 3.3.1
Move(C:128×128)(RF→GL)(Block)
 .tile(64,32).to(Warp)   // Sec. 3.3.2
 .tile( 8, 8).to(Thread)
 .tile( 1, 1).done
```

Figure 3.2: Visualization of a simple Matrix Multiplication epilog and its implementation in both CUDA and Fireiron. Every thread directly copies its computed results from register file (CRF) to global memory (C). This implementation strategy is not efficient due to uncoalesced writes to global memory.

Epilog: The last part of a GPU kernel in which the computed results need to be stored back to global memory

3.2 HOW TO ACHIEVE HIGH GPU PERFORMANCE

Efficiently using the GPU's compute and memory hierarchy requires the coordinated application of multiple optimizations. When optimizing data movements, for example, depending on the instructions used, the ownership (that is the specific mapping of threads to data) is fixed, which complicates achieving efficient memory access patterns. With the introduction of Tensor Cores, data movements become even more challenging because their `mma.sync` instructions [113] impose complicated ownerships involving small groups of eight threads (Quad-Pairs). To alleviate ownership-related restrictions, data movements can be decomposed into two steps, as discussed in the following two examples, allowing threads to exchange data in between to achieve more efficient reads and writes.

Consider the *epilog* of a matrix multiplication kernel as an (often neglected) example. It is crucial to optimize the epilog because it is entirely memory bandwidth limited due to only moving data: After performing the computation, in the epilog, each thread-block must move its computed results back to global memory. Typically, these results are distributed across the registers of a block's threads. Figure 3.2 shows a simple epilog strategy in which every thread directly copies its computed results to global memory and CUDA code implementing this data movement. This implementation is not very efficient due to uncoalesced writes to global memory, i.e., `C` needs to be accessed by writing rows instead of 8×8 tiles.

The Quad-Pair tiles interleave because they are non-contiguous (here: four 4×4 distributed sub-tiles)

1 Warp = 4 Quad-Pairs
1 QP = 8 Threads (T)
QP0: T0-3 & T16-19
QP1: T4-7 & T20-23
QP2: T8-11 & T24-27
QP3: T12-15 & T28-31

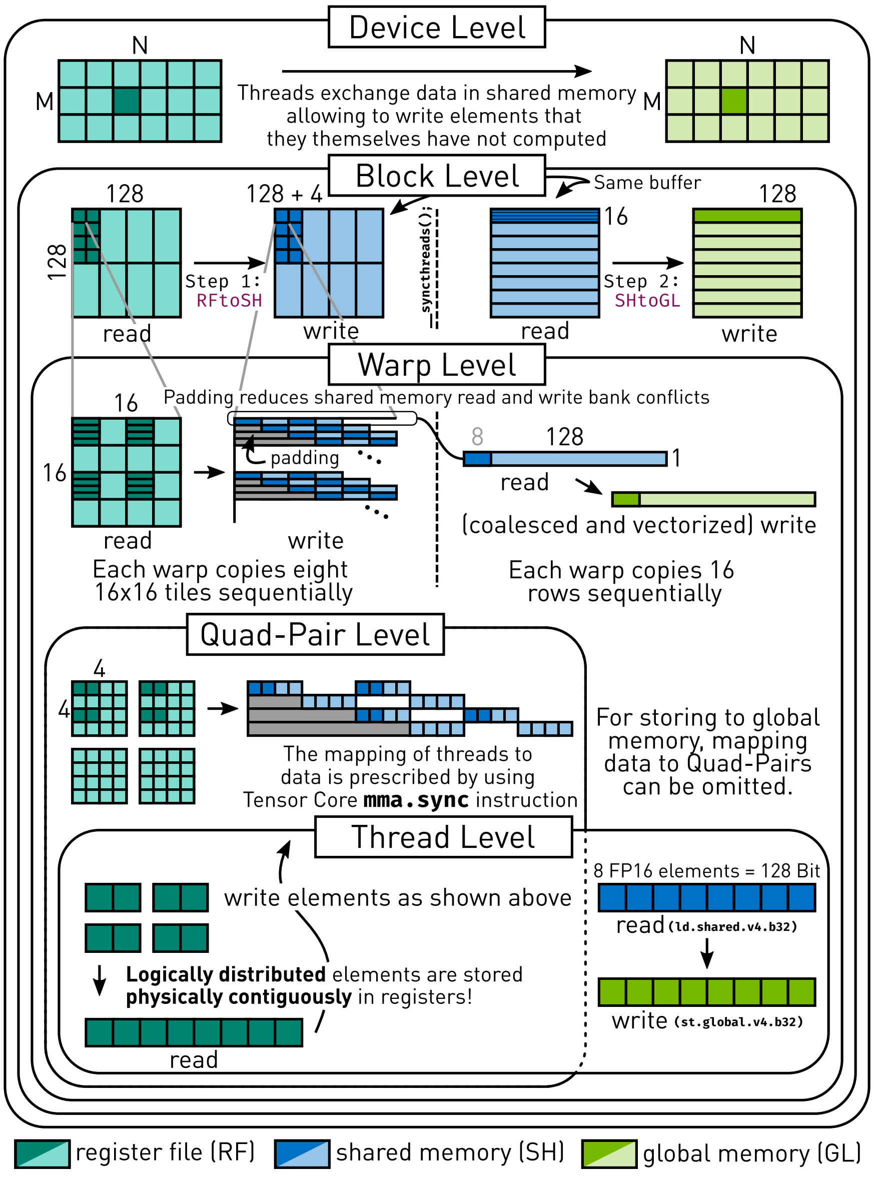

Device Level
N
M
Threads exchange data in shared memory allowing to write elements that they themselves have not computed
Block Level
Same buffer
128
128 + 4
Step 1: RFtoSH
__syncthreads();
16
Step 2: SHtoGL
read
write
Warp Level
Padding reduces shared memory read and write bank conflicts
padding
8
1
(coalesced and vectorized) write
Each warp copies eight 16x16 tiles sequentially
Each warp copies 16 rows sequentially
Quad-Pair Level
4
The mapping of threads to data is prescribed by using Tensor Core mma.sync instruction
For storing to global memory, mapping data to Quad-Pairs can be omitted.
Thread Level
write elements as shown above
Logically distributed elements are stored physically contiguously in registers!
8 FP16 elements = 128 Bit
read (ld.shared.v4.b32)
write (st.global.v4.b32)
register file (RF)
shared memory (SH)
global memory (GL)

Figure 3.3: An optimized matrix multiplication epilog: Synchronize via shared memory. This allows vectorized and coalesced stores to global memory while avoiding bank conflicts using padding in shared memory.

Figure 3.3 shows an optimized epilog, and here, we additionally assume that the results were computed using Tensor Cores. When Tensor Cores are used, at runtime, a warp is partitioned into four Quad-Pairs, groups of eight specific threads, which cooperatively execute a `mma.sync` instruction to compute an 8×8 tile of the output. Tensor Cores are programmable using a family of `mma.sync` variants for different operand storage layouts (row- or column-major) and accumulation precision (`FP16` or `FP32`). Each variant prescribes a different quad-pair-level ownership. In this version, the quad-pairs, and thus their threads, computed logically distributed tiles that are physically stored in contiguous registers. For achieving coalesced

```
val RFtoSH = Move(C:128x128)(RF->SH)(Block) // implement step 1
  .tile(64,32).to(Warp)          // assign 64x32 tiles to warps
  .tile(16,16)                   // process 16x16 tiles sequentially
  // The following mapping is prescribed by Tensor Cores ...
  .tile((4,8),(4,8)).to(QuadPair) // strided tiles for quad-pairs/threads:
  .tile((1,2),(2,4)).to(Thread)   // .tile((rowSize,rowOffset), ...)
  .tile(1,1)                     // copy elements sequentially to SH

val SHtoGL = Move(C:128x128)(SH->GL)(Block) // implement step 2
  .tile(16, 128).to(Warp)        // assign 16 rows to a single warp
  .tile(1,  128)                 // copy rows sequentially
  .tile(1,    8).to(Thread)      // each thread stores 128Bit (8*FP16)

val optimizedEpilog = Move(C:128x128)(RF->GL)(Block)
  .move(Move.src, SH, RFtoSH).pad(4)  // step 1: Move results from RF to SH
  .apply(SHtoGL).done                 // step 2: Move results from SH to GL
```

Listing 3.1: Expressing the optimized epilog (Figure 3.3) in Fireiron.

writes to global memory, threads have to exchange data in shared memory first to be able to store results which they themselves have not computed. Every block allocates a temporary buffer in shared memory for the coordinated data exchange (indicated by the different write and read patterns). The second step of this optimized epilog (`SHtoGL`) is rather standard: All threads of a warp move a complete row from shared- to global memory achieving coalesced and vectorized writes. The first step (`RFtoSH`), is more complicated due to the dictated Tensor Core ownership. Instead of simply materializing the ownership in shared memory in the first step, the data movement to and from shared memory can be further optimized: Padding the shared memory buffer with additional columns achieves a skewed read and write access pattern (see the warp-level write to shared memory in Figure 3.3), reducing the number of read and write bank conflicts.

Certain memory access patterns enable the GPU to coalesce groups of reads or writes into one operation. [111] (See Section 2.1.2)

Shared memory is organized in banks that can be accessed by one thread at a time.

A bank conflict occurs when multiple threads access the same bank. (See Section 2.1.2)

Implementing the optimized epilog requires about 7× more lines of CUDA code and is significantly more complex than the simple version. In the next section, we explain how to express both versions in a precise way, using Fireiron, as shown in Figure 3.2 and Listing 3.1.

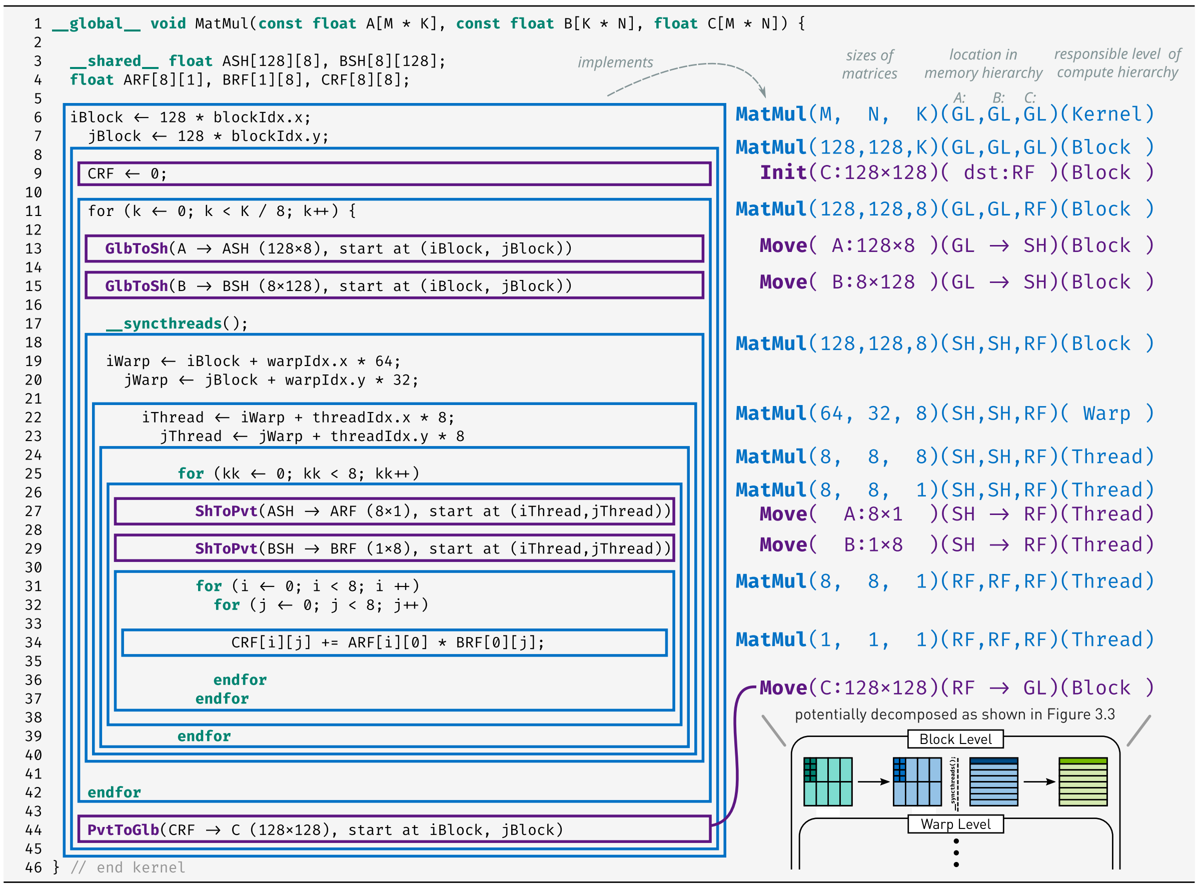
LIMITATIONS OF TODAY'S SCHEDULE-BASED APPROACHES In schedule-based compilers like TVM and Halide, data movements are generally treated as second-class citizens. Therefore, expressing optimizations, especially for data movements, is a stretch and blurs the line between algorithms and schedules. For example, optimizations like storage layout transformations require drastic changes to the algorithm instead of being expressible as a schedule. Listing 3.2 (top) shows a naive matrix multiplication algorithm in TVM [172]. Because existing scheduling languages can only decompose computations, a modified and optimized algorithm in which a new function `packedB`

```python
# Naive Algorithm
k = te.reduce_axis((0, K), 'k')
A = te.placeholder((M, K), name='A')
B = te.placeholder((K, N), name='B')
C = te.compute((M, N),
 lambda x,y: te.sum(A[x,k]*B[k,y], axis=k),name='C')

# Optimized Algorithm
packedB = te.compute((N/bn,K,bn),lambda x,y,z:B[y,x*bn+z],name='packedB')
C_opt = te.compute((M, N), lambda x, y: te.sum(A[x, k] *
  packedB[y // bn, k, tvm.tir.indexmod(y, bn)], axis=k), name = 'C_opt')
```

Listing 3.2: Schedule-based compilers like TVM aim to separate expressing computations as algorithms and optimizations as schedules. However, performing data movement-related optimizations such as storage layout transformations requires to drastically modify the algorithm instead of the schedule.

must be introduced to permute elements in memory (bottom). Additionally, the existing languages allow to allocate temporary buffers in specific locations (e.g., using Halide's `store_in` primitive), and by introducing identity computations as redundant compute stages, data movements are implicitly scheduled by scheduling the computation of the producer and consumer stage. The compiler then needs to infer the implications for the associated reads and writes of this data movement during code generation. However, inferring the coordination required for expressing advanced data movements such as the optimized epilog shown in Figure 3.3 is beyond the reach of automatic compiler analysis.

Additionally, using Tensor Cores efficiently on modern GPUs is crucial but challenging because it requires considering precise data-to-thread mappings: At runtime, a warp executes four `mma.sync` instructions simultaneously. Each instruction is collectively executed with eight specific threads (a quad-pair). For example, the first quad-pair consists of threads 0-3 and threads 16-19. Every thread of a quad-pair contains four values of both operands in registers, which they share to compute an 8×8 output tile collectively. NVIDIA provides a CUDA interface (WMMA [119]) that exposes a conceptually single, *warp-wide* macro-`mma` using a fixed data-to-quad-pair mapping, which is optimal in some cases but not all. Some situations require more sophisticated mappings such as the one shown in Figure 3.3, where a quad-pair, and thus its threads, operate on interleaved distributed tiles. In Fireiron, one can a) flexibly decompose warp-level computations to quad-pairs and b) implement the required data movements by treating moves as schedulable operations, as we will show in the following. Existing scheduling languages lack these mechanisms and currently only target WMMA, which potentially explains the remaining gap in performance shown in Figure 3.1.



```
__global__ void MatMul(const float A[M * K], const float B[K * N], float C[M * N]) {

  __shared__ float ASH[128][8], BSH[8][128];
  float ARF[8][1], BRF[1][8], CRF[8][8];

  iBlock ← 128 * blockIdx.x;
    jBlock ← 128 * blockIdx.y;

    CRF ← 0;

    for (k ← 0; k < K / 8; k++) {

      GlbToSh(A → ASH (128×8), start at (iBlock, jBlock))

      GlbToSh(B → BSH (8×128), start at (iBlock, jBlock))

      __syncthreads();

      iWarp ← iBlock + warpIdx.x * 64;
        jWarp ← jBlock + warpIdx.y * 32;

          iThread ← iWarp + threadIdx.x * 8;
            jThread ← jWarp + threadIdx.y * 8

              for (kk ← 0; kk < 8; kk++)

                ShToPvt(ASH → ARF (8×1), start at (iThread,jThread))

                ShToPvt(BSH → BRF (1×8), start at (iThread,jThread))

                for (i ← 0; i < 8; i ++)
                  for (j ← 0; j < 8; j++)

                    CRF[i][j] += ARF[i][0] * BRF[0][j];

                  endfor
                endfor

              endfor


    endfor

    PvtToGlb(CRF → C (128×128), start at iBlock, jBlock)

} // end kernel
```

```
MatMul(M,   N,   K)(GL,GL,GL)(Kernel)
MatMul(128,128,K)(GL,GL,GL)(Block )
  Init(C:128×128)( dst:RF )(Block )
MatMul(128,128,8)(GL,GL,RF)(Block )
  Move( A:128×8 )(GL → SH)(Block )
  Move( B:8×128 )(GL → SH)(Block )
MatMul(128,128,8)(SH,SH,RF)(Block )
MatMul(64, 32, 8)(SH,SH,RF)( Warp )
MatMul(8,  8,  8)(SH,SH,RF)(Thread)
MatMul(8,  8,  1)(SH,SH,RF)(Thread)
  Move(  A:8×1  )(SH → RF)(Thread)
  Move(  B:1×8  )(SH → RF)(Thread)
MatMul(8,  8,  1)(RF,RF,RF)(Thread)
MatMul(1,  1,  1)(RF,RF,RF)(Thread)
  Move(C:128×128)(RF → GL)(Block )
```

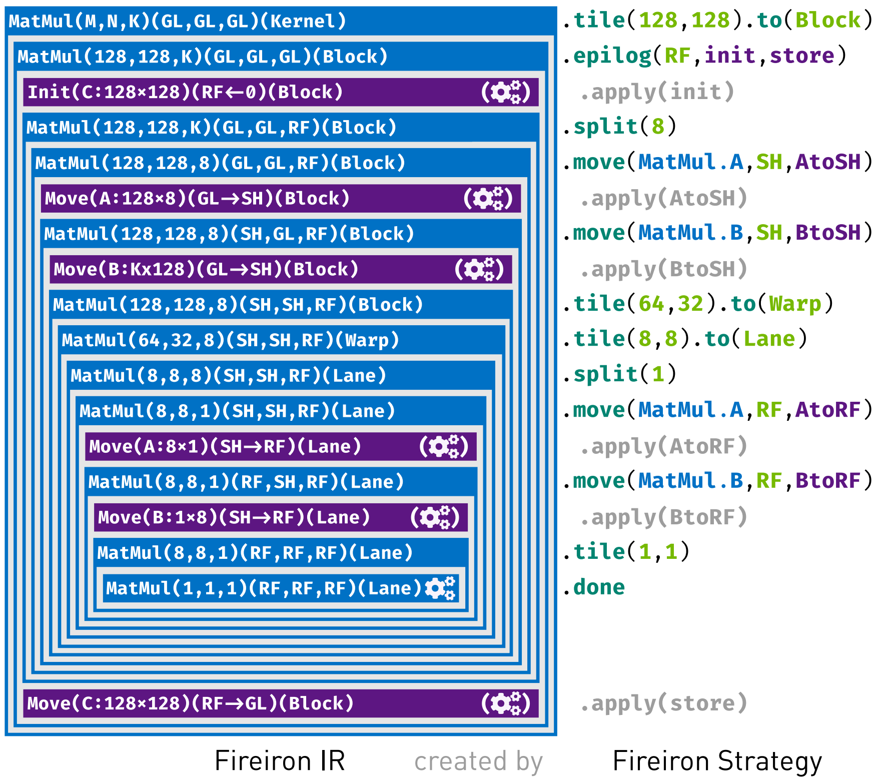
implements
sizes of matrices
A: B: C:
location in memory hierarchy
responsible level of compute hierarchy
potentially decomposed as shown in Figure 3.3
Block Level
__syncthreads();
Warp Level

Figure 3.4: Showing the hierarchical structure of GPU kernels using a matrix multiplication as an example. Within a typical GPU kernel, we gradually descend the compute and memory hierarchy while computing smaller instances of the original problem.

3.3 RETHINKING SCHEDULING LANGUAGES

With Fireiron, we aim to design an IR that reflects the hierarchical structure of high-performance code, and a scheduling language generating this IR, that allows experts to express optimizations for computations and data movements.

We start by highlighting the hierarchical structure of GPU kernels and the role of data movements in the code. In Figure 3.4, we show this structure using a simple matrix multiplication kernel as an example. The computation is hierarchically decomposed into sub-computations of the same kind, i.e., matrix multiplications operating on smaller shapes (blue boxes), until eventually, every thread computes a single FMA instruction in registers (innermost blue box) that can be viewed as a matrix multiplication of matrices containing only a single element. In between, data is moved to lower levels of the memory hierarchy (purple boxes), and for brevity, we show no data movement implementations, i.e., no purple box contains nested boxes. However, every data movement is similarly decomposed, as



```
MatMul(M,N,K)(GL,GL,GL)(Kernel)
 MatMul(128,128,K)(GL,GL,GL)(Block)
  Init(C:128×128)(RF←0)(Block)
  MatMul(128,128,K)(GL,GL,RF)(Block)
   MatMul(128,128,8)(GL,GL,RF)(Block)
    Move(A:128×8)(GL→SH)(Block)
    MatMul(128,128,8)(SH,GL,RF)(Block)
     Move(B:Kx128)(GL→SH)(Block)
     MatMul(128,128,8)(SH,SH,RF)(Block)
      MatMul(64,32,8)(SH,SH,RF)(Warp)
       MatMul(8,8,8)(SH,SH,RF)(Lane)
        MatMul(8,8,1)(SH,SH,RF)(Lane)
         Move(A:8×1)(SH→RF)(Lane)
         MatMul(8,8,1)(RF,SH,RF)(Lane)
          Move(B:1×8)(SH→RF)(Lane)
          MatMul(8,8,1)(RF,RF,RF)(Lane)
           MatMul(1,1,1)(RF,RF,RF)(Lane)
  Move(C:128×128)(RF→GL)(Block)
```

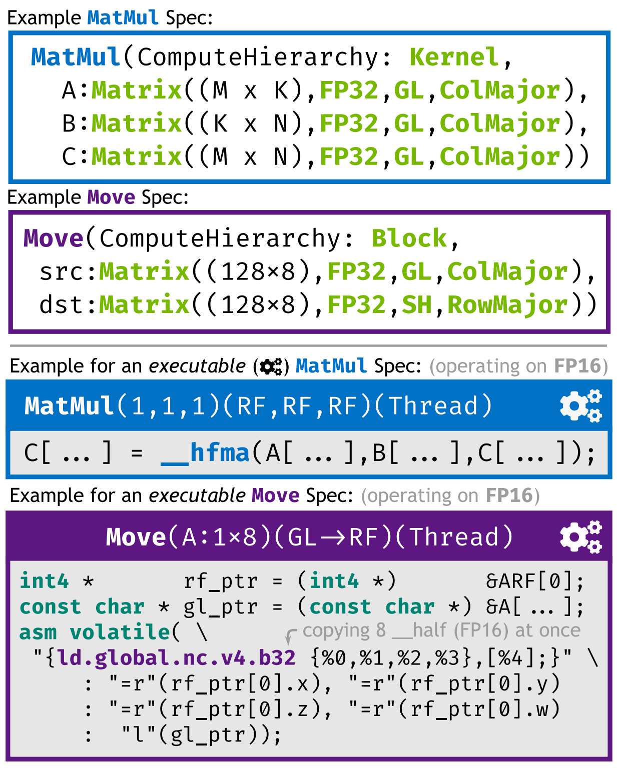
```
.tile(128,128).to(Block)
.epilog(RF,init,store)
  .apply(init)
.split(8)
.move(MatMul.A,SH,AtoSH)
  .apply(AtoSH)
.move(MatMul.B,SH,BtoSH)
  .apply(BtoSH)
.tile(64,32).to(Warp)
.tile(8,8).to(Lane)
.split(1)
.move(MatMul.A,RF,AtoRF)
  .apply(AtoRF)
.move(MatMul.B,RF,BtoRF)
  .apply(BtoRF)
.tile(1,1)
.done

  .apply(store)
```

Figure 3.5: Describing and representing the implementation shown in Figure 3.4 using Fireiron's decompositions to construct the IR of nested specifications.

indicated on the bottom where the epilog of this kernel (the last purple box) might be implemented as explained in Figure 3.3.

Figure 3.5 shows how the code in Figure 3.4 is expressed using our scheduling language (Fireiron Strategy) and the corresponding IR that is created. Again, we omit the strategies for data movements and instead indicate their decomposition (using `.apply`(*strategy*)). For example, **`store`** can be replaced by the `optimizedEpilog` strategy shown in Listing 3.1.

The rest of this section describes our IR consisting of nested *Specifications* (specs) and the scheduling primitives called *Decompositions* that create the IR and describe implementation strategies. The key idea behind Fireiron is that implementations are described by gradually decomposing specs until only specs remain for which we know how to generate code. In the following, a Fireiron program using decompositions is called a *strategy* because it is meant to capture the implementation strategy thought of by a human expert when developing low-level code. Currently, Fireiron is implemented as a domain-specific language embedded in Scala and generates CUDA kernels with inline PTX assembly.



Example MatMul Spec:

```
MatMul(ComputeHierarchy: Kernel,
  A:Matrix((M x K),FP32,GL,ColMajor),
  B:Matrix((K x N),FP32,GL,ColMajor),
  C:Matrix((M x N),FP32,GL,ColMajor))
```

Example Move Spec:

```
Move(ComputeHierarchy: Block,
 src:Matrix((128×8),FP32,GL,ColMajor),
 dst:Matrix((128×8),FP32,SH,RowMajor))
```

Example for an *executable* (⚙) MatMul Spec: (operating on FP16)

```
MatMul(1,1,1)(RF,RF,RF)(Thread)
C[ … ] = __hfma(A[ … ],B[ … ],C[ … ]);
```

Example for an *executable* Move Spec: (operating on FP16)

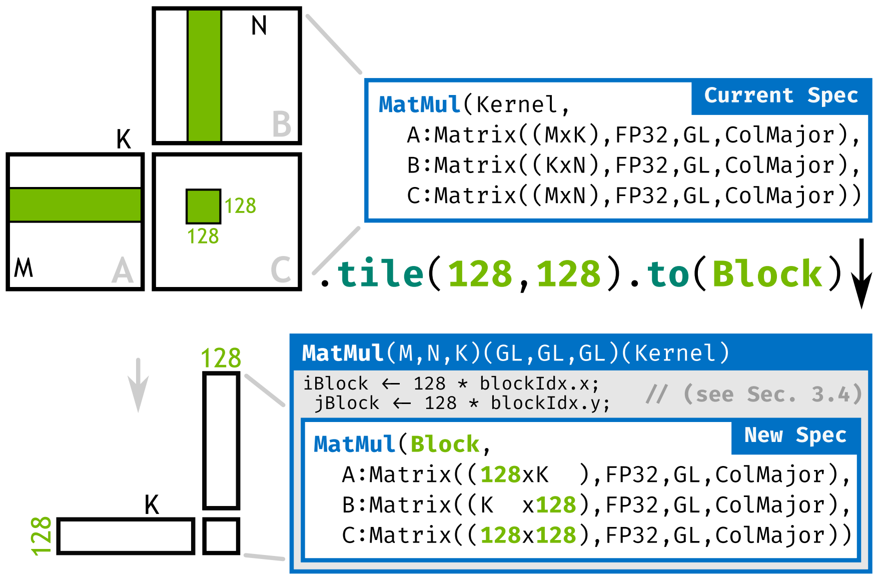
```
Move(A:1×8)(GL→RF)(Thread)
int4 *         rf_ptr = (int4 *)        &ARF[0];
const char * gl_ptr = (const char *) &A[ … ];
asm volatile( \          copying 8 __half (FP16) at once
 "{ld.global.nc.v4.b32 {%0,%1,%2,%3},[%4];}" \
    : "=r"(rf_ptr[0].x), "=r"(rf_ptr[0].y)
    : "=r"(rf_ptr[0].z), "=r"(rf_ptr[0].w)
    :  "l"(gl_ptr));
```

Figure 3.6: Examples of Fireiron specs: A kernel-level matrix multiplication computation and a specification for a data movement from global to shared memory. An *executable* spec directly corresponds to an instruction like `__hfma` in CUDA or `ld.global.nc.v4.b32` in PTX.

3.3.1 *Specifications*

Each box in Figure 3.4 can be labeled with an accurate description of the task performed inside it. We call this label the *Specification* (spec). Its implementation is observed by looking inside the box where the task is further decomposed. In Fireiron, a spec is a data structure describing the task to implement. A spec contains enough information such that a programmer would be able to manually provide an implementation. This especially entails that it contains the shapes, locations, and storage layouts of its operands, including the responsible level of the compute hierarchy performing this operation. Currently, Fireiron supports two main classes of specs: Matrix Multiplication (`MatMul`), and data movement (`Move`). Figure 3.6 (top) shows a kernel-level `MatMul` spec and a `Move` spec describing the movement of a matrix `src` from global to shared memory during which the storage layout is transformed from column- to row-major.

For matrices, Fireiron supports constant and symbolic shapes written as arithmetic expressions. We write `MatMul(M,N,K)(GL,GL,GL)(Kernel)` and `Move(src:128x8)(GL->SH)(Block)` as a short-form for the two upper specs shown in Figure 3.6.



```
MatMul(Kernel,
  A:Matrix((MxK),FP32,GL,ColMajor),
  B:Matrix((KxN),FP32,GL,ColMajor),
  C:Matrix((MxN),FP32,GL,ColMajor))
.tile(128,128).to(Block)
MatMul(M,N,K)(GL,GL,GL)(Kernel)
iBlock ← 128 * blockIdx.x;
 jBlock ← 128 * blockIdx.y;     // (see Sec. 3.4)
MatMul(Block,
  A:Matrix((128xK  ),FP32,GL,ColMajor),
  B:Matrix((K  x128),FP32,GL,ColMajor),
  C:Matrix((128x128),FP32,GL,ColMajor))
```

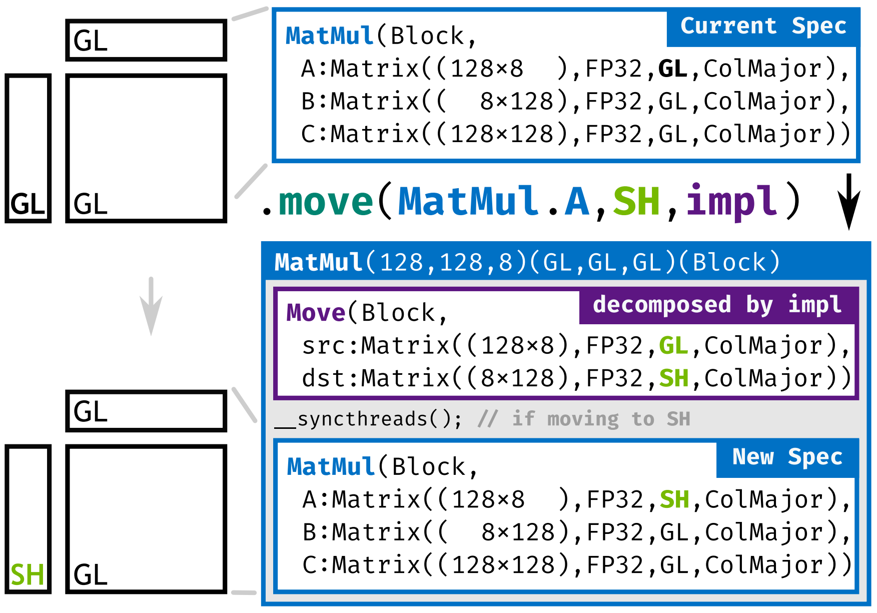
Figure 3.7: Tiling a `MatMul` spec results in a decomposed subspec with adjusted dimensions and optionally adjusted compute hierarchy to indicate parallel execution.

EXECUTABLE SPECIFICATIONS A specification is called *executable* when it describes the semantics of a built-in instruction. Fireiron provides a predefined set of executable specs matching different CUDA and PTX instructions. Figure 3.6 (bottom) shows examples for executable `MatMul` and `Move` specs and their associated code snippets. The idea is to gradually decompose specifications until only executable specs remain for which we know how to generate code. At any time during the decomposition, a user can additionally provide a handwritten micro-kernel instead which implements the current spec. Providing handwritten micro-kernels allows the user to break out of the DSL barriers and use custom implementations for which we cannot yet provide good abstractions.

3.3.2 *Decompositions*

A *Decomposition* describes how to implement a spec. Applying a decomposition to a spec generally creates one or more nested sub-specs representing the smaller sub-problems that must be further decomposed until they are executable. Fireiron provides two main decompositions: `tile` and `move`, for mapping tasks to the GPU's compute and memory hierarchy.

THE `.tile` DECOMPOSITION Figure 3.7 shows the application of the `tile` decomposition to a `MatMul` spec. Generally, `spec.tile(r,c)` creates $r \times c$ shaped tiles in the output, and the input matrices are tiled accordingly. Tiles can also be non-contiguous, as discussed for the optimized epilog example (shown in Figure 3.3), in which case



```
MatMul(Block,
 A:Matrix((128×8  ),FP32,GL,ColMajor),
 B:Matrix((  8×128),FP32,GL,ColMajor),
 C:Matrix((128×128),FP32,GL,ColMajor))
```

```
.move(MatMul.A,SH,impl)
```

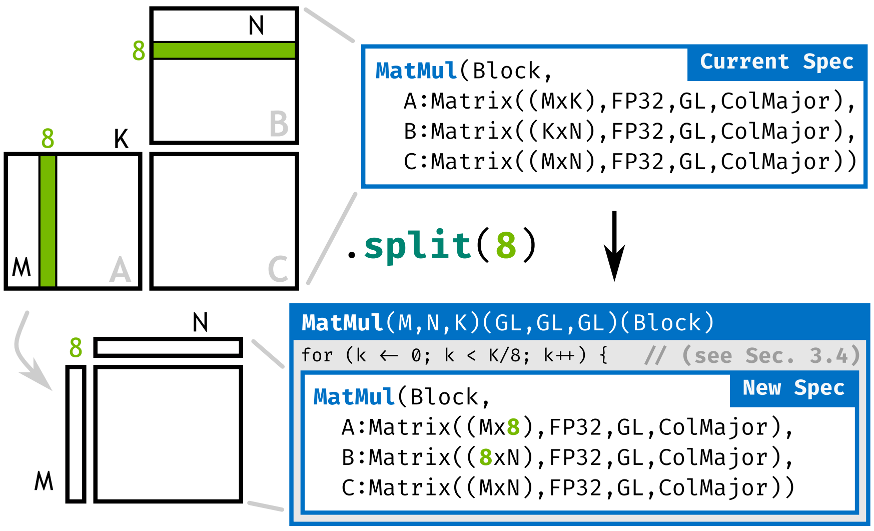
```
MatMul(128,128,8)(GL,GL,GL)(Block)
Move(Block,
 src:Matrix((128×8),FP32,GL,ColMajor),
 dst:Matrix((8×128),FP32,SH,ColMajor))
__syncthreads(); // if moving to SH
MatMul(Block,
 A:Matrix((128×8  ),FP32,SH,ColMajor),
 B:Matrix((  8×128),FP32,GL,ColMajor),
 C:Matrix((128×128),FP32,GL,ColMajor))
```

Figure 3.8: Applying `move` to a `MatMul` spec results in a new spec in which the memory location of the specified operand has changed. A `Move` spec is created representing the data movement, which is implemented as specified in the strategy *impl*.

`.tile` expects a width and offset for both dimensions. For assigning tiles to a compute hierarchy level, we can *refine* the tiling using `.to(level)` which changes the responsible compute hierarchy level for the resulting tiled spec. `tile` is also applicable to a `Move` spec in which case the input (`src`) and output (`dst`) matrices are tiled in the same way.

THE `.move` DECOMPOSITION The `.move` decomposition is the primary way of introducing `Move` specs, explicitly representing data movements, to the IR. Figure 3.8 shows the application of the `move` decomposition to a `MatMul` spec. Here, we move the `A` operand from global to shared memory. The `move` decomposition expects three arguments: the matrix to move, a destination in the memory hierarchy, and a strategy *impl* describing how to implement the movement. Applying `move` always creates *two* new nested specs: First, a `Move` representing the data movement to the new location whose implementation is described in the *impl* strategy. Second, an updated version of the input spec where the location of the moved operand has changed.

The `move` decomposition can also be applied to a `Move` spec, which allows us to specify data movements via an indirection as described and shown in Figure 3.3.



```
MatMul(Block,
  A:Matrix((MxK),FP32,GL,ColMajor),
  B:Matrix((KxN),FP32,GL,ColMajor),
  C:Matrix((MxN),FP32,GL,ColMajor))
```

```
.split(8)
```

```
MatMul(M,N,K)(GL,GL,GL)(Block)
for (k ← 0; k < K/8; k++) {   // (see Sec. 3.4)
  MatMul(Block,
    A:Matrix((Mx8),FP32,GL,ColMajor),
    B:Matrix((8xN),FP32,GL,ColMajor),
    C:Matrix((MxN),FP32,GL,ColMajor))
```

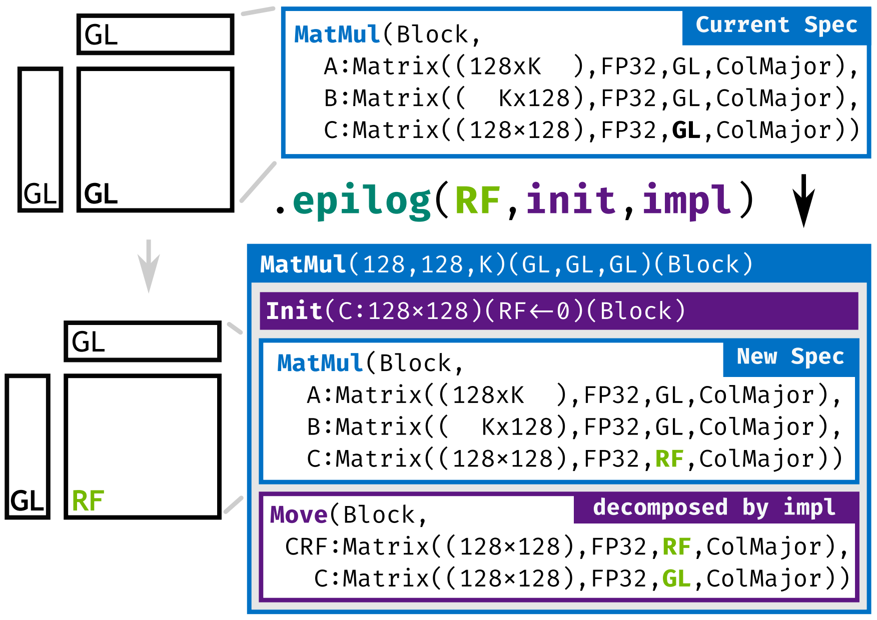
Figure 3.9: The `split` decomposition allows the creation of tiles in the K-dimension of the input operands of a `MatMul` spec.

3.3.3 *`MatMul`-specific Decompositions*

The `tile` and `move` decompositions already allow to specify a wide variety of implementations for both `MatMul` and `Move` specs. However, to describe implementations that achieve performance on par with hand-tuned libraries, we need two more ways to decompose a `MatMul` computation.

THE `.split` DECOMPOSITION The `tile` decomposition allows us to create tiles in the M and N dimension of the `MatMul` operands, however, we also need to be able to create tiles in the K-dimension. Figure 3.9 shows the application of the `split` decomposition, which enables this.

THE `.epilog` DECOMPOSITION Finally, we need to be able to specify that results shall be accumulated in lower levels of the memory hierarchy, typically in registers, and then moved back to global memory in the epilog of a matrix multiplication kernel. Figure 3.10 shows the application of the `epilog` decomposition, which expects three arguments: The location of the accumulation buffer, a strategy *init* describing its initialization, and a strategy *impl* specifying how to move the computed results back to global memory. Similar to `move`, `epilog` creates multiple sub-specs: First, an `Init` spec (a variant of `Move` without source operand) representing the initialization of the buffer. Its implementation is described in the *init* strategy. Second, the new `MatMul` spec with an updated location of the `C` matrix. Third, the `Move` representing the movement of the results to global memory.



```
MatMul(Block,
  A:Matrix((128xK  ),FP32,GL,ColMajor),
  B:Matrix((  Kx128),FP32,GL,ColMajor),
  C:Matrix((128×128),FP32,GL,ColMajor))
```

```
MatMul(128,128,K)(GL,GL,GL)(Block)
  Init(C:128×128)(RF←0)(Block)
  MatMul(Block,
    A:Matrix((128xK  ),FP32,GL,ColMajor),
    B:Matrix((  Kx128),FP32,GL,ColMajor),
    C:Matrix((128×128),FP32,RF,ColMajor))
  Move(Block,
   CRF:Matrix((128×128),FP32,RF,ColMajor),
     C:Matrix((128×128),FP32,GL,ColMajor))
```

Figure 3.10: The **epilog** decomposition allows the accumulation of the results of a matrix multiplication in the lower levels of the memory hierarchy and specifies the data movement of the results back to global memory.

Inspired by Chapel [24], Fireiron provides the illusion of a block-wide matrix residing in registers in the created sub-specs. This *distributed array* abstraction allows us to think of this matrix as a contiguous block-wide matrix, whereas actually, every thread contains only a small tile in its registers.

3.4 CODE GENERATION AND TARGETING TENSOR CORES

Fireiron's IR of nested specs naturally reflects how GPU kernels are structured. Therefore, code generation almost boils down to pretty-printing the IR, traversing it from top to bottom. Figure 3.5 shows a compressed view of the IR where specs are directly nested. Figures 3.7-3.10 additionally show the code snippets emitted for each used decomposition. For example, using **tile** generally emits two `for`-loops that sequentially iterate over the created tiles (e.g., Figure 3.4, lines 31-32). If tiles are assigned to the compute hierarchy using **.to**, instead of emitting sequential loops, the tiles are computed in parallel using the unique compute hierarchy indices for accessing the matrices. Figure 3.2 shows a Fireiron example using both sequential and parallel tiles and the corresponding CUDA code we generate.

Using the **split** decomposition emits one loop iterating over the tiles in the K-dimension. The **epilog** and **move** decompositions emit no code themselves (except for synchronization if the destination is shared memory). Instead, the created sub-specs will be further compiled to CUDA code. The **done** operator is called on an executable spec to trigger code generation where we inject the associated code snippet. Optionally, `done` accepts a `String`: a micro-kernel we inline during code generation that implements the current spec.

MEMORY ALLOCATION Memory allocation is equally straightforward. In order to allocate enough memory, we traverse the IR once and register all `Move` specs, which specify how much to allocate where - by definition. For example, when visiting the spec `Move(A:128x8)(GL->SH)(Block)`, we emit

```
float __shared__ ASH[128*8];
```

at the beginning of the kernel. Memory allocation for distributed arrays is more complicated. `Init(C:128x128)(GL->RF)(Block)`, for example, specifies the need to allocate memory in the register file. However, allocating 128 × 128 elements per thread is far too many because a single thread only owns a small piece of the whole matrix. To determine how many elements we need to allocate per thread, we need to traverse the decomposition until we find the tile size assigned to threads:

```
Init(C:128x128)(GL->RF)(Block)
.tile(64,32).to(Warp)
.tile( 8, 8).to(Thread)//<-allocate 8x8 floats per thread
.tile( 1, 1).done
```

In this case, we emit `float CRF[8*8];`.

Generally, every level of the compute hierarchy is associated with a level of the memory hierarchy (`Kernel`→`GL`, `Block`/`Warp`→`SH`, `Thread`→`RF`). If the responsible compute hierarchy of the `Move` or `Init` spec does not match the associated destination, we need to continue traversing the decomposition, as shown in the example.

INDEX COMPUTATION Index expressions for every matrix are computed transparently while decomposing specs. Every matrix has an associated row- and column index that is gradually updated. Every application of a decomposition returns a new spec: We either have a smaller version of the original matrix or we have the same matrix moved to a new memory location. For example, applying `MatMul.tile(rs,cs)` generates two nested sequential for-loops with indices `rowTile`, and `colTile`. The index expressions for the matrices in the resulting tiled `MatMul` spec are updated as follows:

```
C.rowIndex += rowTile * rs;
C.colIndex += colTile * cs;
A.rowIndex += rowTile * rs;
B.colIndex += colTile * cs;
```

If tiles are computed in parallel, for example, by using `.to(Block)`, we use `blockIdx.x` and `blockIdx.y` instead of `rowTile` and `colTile`.

Similarly to memory allocation for distributed arrays, we have to compare the current compute hierarchy with the memory location of the array to update: In the following example

```
MatMul(128,128,8)(GL,GL,RF)(Block).tile(64,32).to(Warp)
```

we do update the `A` and `B` index expressions but not the index expressions for `C`. This is because `C` resides in registers, and accessing it using the warp indices would be incorrect. Instead, we only start updating the index expression for `C` as soon as we pass the `Thread`-level. The CUDA code in Figure 3.2 shows exactly this effect where `C` (residing in global memory) is accessed using all compute hierarchy indices. `CRF` (residing in registers) is only accessed using the indices used below the `Thread`-level. The `split` decomposition updates the indices as expected. Every time we allocate a matrix in a new memory location (using `move` or `epilog`), we start with fresh index expressions in the nested specs.

To summarize, code generation, including memory allocation and index computations is straightforward due to our modular IR design.

3.4.1 *Supporting Tensor Cores with Fireiron*

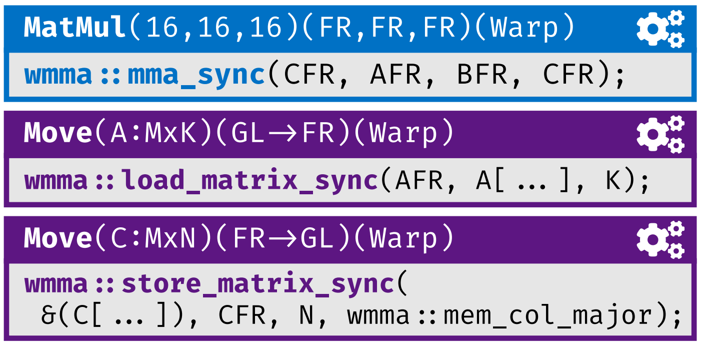
To support Tensor Cores, we extend the set of executable specs and target them using decompositions.

SUPPORTING WMMA IN CUDA The WMMA-API [119] in CUDA introduces warp-wide matrix multiply primitives operating on register collections called *fragments*. For generating kernels using WMMA primitives, we extend Fireiron in two ways: First, we extend the memory hierarchy and add a new level `Fragment<M,N,K>` (labeled `FR` if $M = N = K = 16$) in between shared memory and registers. Fragments are parameterized because, in the CUDA API, sizes are part of the fragment type, which must be specified at the allocation.

Second, we define new executable specs corresponding to the CUDA API calls. Figure 3.11 shows examples of new executable WMMA specs. Listing 3.3 shows how these additions allow us to write a strategy targeting the new executable WMMA specs. It computes the matrix multiplication as follows:

1. assign 64×64 elements to a block (line 2);
2. initialize 16 (4×4) accumulator fragments (line 4);
3. fill operand fragments (lines 8–9);
4. compute the result (line 10); and
5. store results from fragments to global memory (line 5).

Note that we only need to decompose specs to the level of warps because of the new warp-level executable specs.



```
MatMul(16,16,16)(FR,FR,FR)(Warp)
wmma::mma_sync(CFR, AFR, BFR, CFR);

Move(A:MxK)(GL→FR)(Warp)
wmma::load_matrix_sync(AFR, A[...], K);

Move(C:MxN)(FR→GL)(Warp)
wmma::store_matrix_sync(
 &(C[...]), CFR, N, wmma::mem_col_major);
```

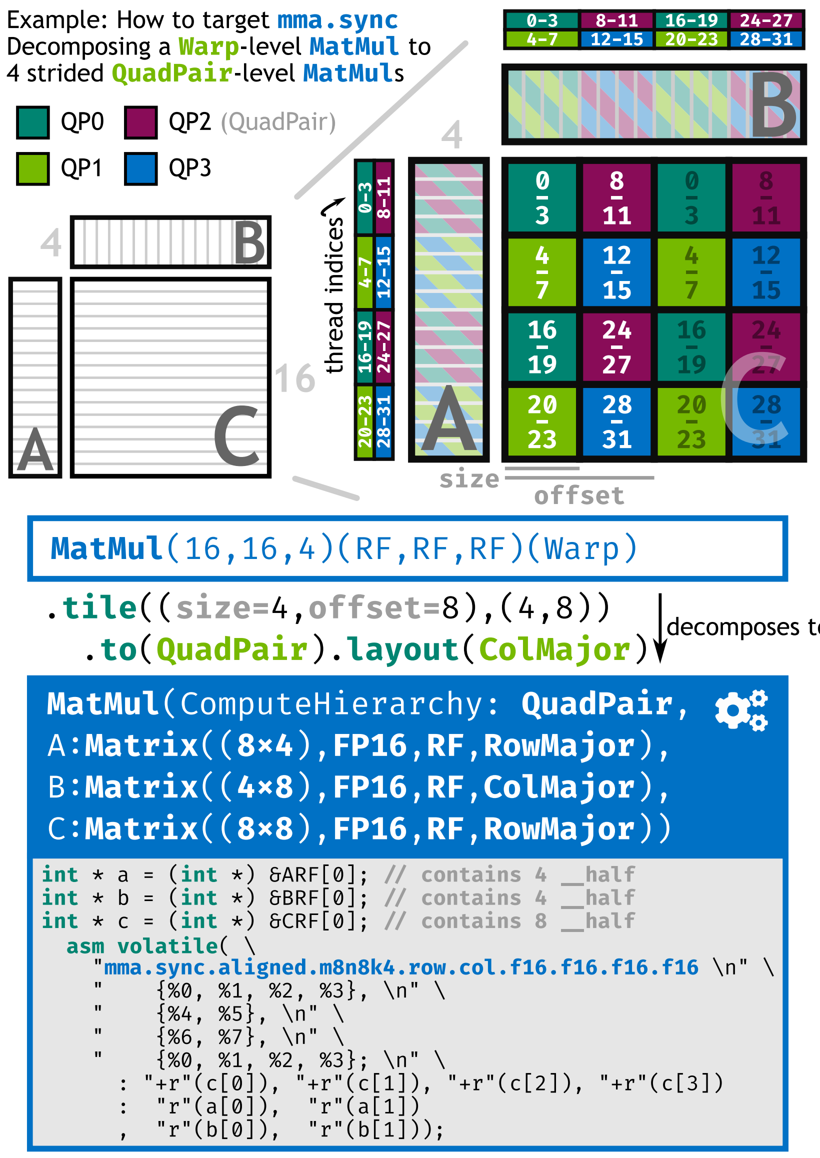
Figure 3.11: Supporting the CUDA WMMA API in Fireiron by adding new warp-level executable specifications.

```
val simpleWMMA = MatMul(M,N,K)(GL,GL,GL)(Kernel)
  .tile(64,64).to(Block)
  .epilog(FR, // accumulate in warp-level fragments
    Init.tile(16,16).to(Warp).done, //  init strategy
    Move.tile(16,16).to(Warp).done) // store strategy
  .split(16)
  .tile(16,16).to(Warp)
  .move(MatMul.A, FR, Move.done) // 16x16 A tiles to FR
  .move(MatMul.B, FR, Move.done) // 16x16 B tiles to FR
  .done // => residual: MatMul(16,16,16)(FR,FR,FR)(Warp)
```

Listing 3.3: Simple Fireiron WMMA decomposition describing the implementation of the first `cudaTensorCoreGemm` kernel shown in the CUDA samples [115].

supporting `mma.sync` in ptx Using the `mma.sync` PTX instructions [113] allows even more fine-grained control over how Tensor Cores are used. First, we define the different `mma.sync` variants as executable `QuadPair`-level specs, as shown at the bottom of Figure 3.12. We can then flexibly target the new executable specs in multiple ways. Figure 3.12 shows one possible decomposition of a contiguous `Warp`-level `MatMul`-spec to four strided executable `QuadPair`-level specs (indicated by different colors). Here, every thread of a `QuadPair` stores four elements from each input operand and, after collectively executing the `mma.sync` instruction, contains eight elements of the C matrix in its registers. We use the `layout` *refinement* for `.tile`, which we explain shortly, to assign tiles to quad-pairs in a column-major order.

As the two Tensor Core examples show, supporting new instructions in Fireiron requires only small changes. It allows us to target complex low-level PTX instructions using simple high-level abstractions. With the introduction of new instructions, e.g., the Turing GPU architecture contains even wider `mma` instructions, new executable specs can be added in a similar way.



```
MatMul(16,16,4)(RF,RF,RF)(Warp)
 .tile((size=4,offset=8),(4,8))
   .to(QuadPair).layout(ColMajor)
```

```
MatMul(ComputeHierarchy: QuadPair,
A:Matrix((8×4),FP16,RF,RowMajor),
B:Matrix((4×8),FP16,RF,ColMajor),
C:Matrix((8×8),FP16,RF,RowMajor))
```

```
int * a = (int *) &ARF[0]; // contains 4 __half
int * b = (int *) &BRF[0]; // contains 4 __half
int * c = (int *) &CRF[0]; // contains 8 __half
  asm volatile( \
    "mma.sync.aligned.m8n8k4.row.col.f16.f16.f16.f16 \n" \
    "     {%0, %1, %2, %3}, \n" \
    "     {%4, %5}, \n" \
    "     {%6, %7}, \n" \
    "     {%0, %1, %2, %3}; \n" \
      : "+r"(c[0]), "+r"(c[1]), "+r"(c[2]), "+r"(c[3])
      :  "r"(a[0]),  "r"(a[1])
      ,  "r"(b[0]),  "r"(b[1]));
```

Figure 3.12: Supporting `mma.sync` in Fireiron by decomposing `MatMul` to the new `QuadPair`-level executable spec.

3.4.2 *Advanced Optimization using Refinements*

Fireiron provides a set of *refinements*, i.e., optional modifications for decompositions, which expose more fine-grained control required to achieve high performance.

`.tile` REFINEMENTS By default, `tile` creates tiles that will be computed sequentially using nested `for`-loops. The `to` refinement allows us to compute tiles in parallel instead. Internally, Fireiron uses one-dimensional compute hierarchy indices that are mapped in a row-major order to the two-dimensional tile arrangement. The `layout` refinement allows us to change this order to column-major and `swizzle` enables even more complex mappings by first permuting the one-dimensional indices arbitrarily before assigning them to the tiles. Using `unroll` emits `#pragma unroll` above the loops.

`.move` REFINEMENTS The `move` decomposition can be refined as well. Using `pad(n)` modifies the memory allocation for the destination buffer associated with the created `Move` spec and allocates n additional columns to avoid memory bank conflicts. `prefetch` generates double-buffered, prefetched versions of the data movement. We emit `__syncthreads()` if the destination location is shared memory. This can be suppressed using `noSync` in situations in which no explicit synchronization is necessary. Finally, `storageLayout` allows us to specify a row- or column-major storage layout for the destination buffer.

`.split` REFINEMENTS Similar to `tile`, the `split` decomposition can also be refined using `unroll`, to unroll the generated loop. Using `sync` emits `__syncthreads();` as the last statement in the body of the created for-loop, which may be required depending on how shared memory is used in a strategy.

We are aware that some refinements, especially `noSync` and `sync`, allow the specification of incorrect implementation due to race conditions. However, a decomposition without refinements always generates correct code. So far, this has caused no problems as Fireiron is meant to be used by performance experts. We intend to improve the analyses of strategies to ensure these refinements cannot cause correctness issues in the future.

3.5 EXPERIMENTAL EVALUATION

In this section, we seek answers to the following questions: If data movements are as important as we think, how much data movement code is present in high-performance GPU kernels? Are we able to express the optimizations experts apply as strategies using Fireiron's decompositions? Does the code we generate perform as well as the manually written implementations? Furthermore, can Fireiron be used by experts to improve the performance of state-of-the-art implementations?

REFERENCES AND GPU ARCHITECTURES Table 3.1 shows the reference implementations used in this evaluation. We choose these because they apply different optimizations targeting specific GPU architectures. We used three GPUs: GeForce GTX 750 Ti (Maxwell), Quadro GV (Volta), and GeForce RTX 2080 Ti (Turing) because they cover different architectures that need to be optimized differently.

Reference	Description
`maxwell`	Manually-tuned CUDA kernel written by NVIDIA's performance experts targeting the Maxwell architecture (without Tensor Cores)
`wmma`	Publicly available CUDA sample [115] targeting the WMMA Tensor Core API
`cuBLAS`	NVIDIA's high-performance math library (using `cublasGemmEx` with Tensor Cores)

Table 3.1: Reference implementations used in the evaluation.

METHODOLOGY We used CUDA-10.0, Driver Version 425.00 and compiled kernels using `-O3 -use_fast_math -arch= sm_XX` where `XX` = 52, 70, and 75 for Maxwell, Volta, and Turing respectively. We locked the clocks to fixed frequencies, report the minimum kernel runtime of 1000 runs using *nvprof*, and omit data transfer time since we are only interested in the quality of our generated kernel code.

The performance reported in Figure 3.1 was measured using public Halide [72] and TVM [173] code, their best matrix multiplication versions we are aware of. At the time of measuring, the hardware used for the other experiments was not available to us anymore. Instead, we used a Titan XP (Pascal, latest architecture without Tensor Cores) for Halide and a GeForce RTX 2080 (Turing) for TVM because they target Tensor Cores. The TVM code was tuned according to the instructions, and we report the best-found performance.

HYPOTHESIS A: *Code related to data movements makes up a significant fraction in high-performance kernels.*

If this is true, we argue that scheduling languages should treat data movements as first-class concepts. We find that about 2/3 of a kernel (in lines of code) is devoted to optimizing data movements.

We count and label the lines of our reference implementations as either related to data movements or computations. Since there is not always a clear purpose of a single line of code, we made a conservative distinction and only count lines for data movements that are: a) declarations of temporary buffers, b) `__syncthreads()`, c) swizzling index computations solely used to avoid bank conflicts, and finally, loops that *only* copy data in their bodies. Everything else counts as 'computation' lines.

Table 3.2 shows our results. Because `cuBLAS` is closed-source, we additionally analyzed the TVM generated code (Figure 3.1), which contains 49 LoC with a data-movement fraction of 77.6%. We also analyzed the corresponding Fireiron strategies and generated code to show how our generated code relates to manually-written code by performance experts.

	Reference Code	Fireiron Strategy	Fireiron Generated Code
`maxwell`	72 (68.1%)	44 (81.8%)	94 (67.0%)
`wmma`	122 (41.0%)	26 (76.9%)	113 (65.4%)
`cuBLAS`	closed-source	49 (83.7%)	260 (60.4%) (small)
`cuBLAS`		46 (84.8%)	309 (72.2%) (large)

Table 3.2: Lines of Code and data-movement related lines (in %) for references, Fireiron strategies, and generated code. For comparing with cuBLAS, we use two different strategies, one for small input sizes and one for large ones.

We are aware of our inconclusive small sample size. However, these numbers already show that data movement optimizations cannot be neglected as they currently are in existing scheduling languages. This is further underlined by the performance our data-movement-heavy kernels achieve (evaluated in Hypothesis C and D) compared to state-of-the-art implementations.

HYPOTHESIS B: *Fireiron can express optimizations that are applied by experts in manually-tuned code.*

We find that this is *mostly* true and that inlining micro-kernels for sub-specs can circumvent limitations of our scheduling language.

Listing 3.1 implementing the data movement shown in Figure 3.3, showed how Fireiron allows to describe complex optimizations as high-level strategies. The optimized data movement is used in one of our `cuBLAS`-strategies. Listing 3.4 and Listing 3.5 show two Fireiron strategies expressing the `maxwell`, and the `wmma` reference, respectively. In the `maxwell`-strategy, for example, we use different strategies for moving A (lines 16–20) and B (lines 22–27) to shared memory because considering the storage layouts separately enables coalesced global memory loads for both operands. We use swizzling (line 2) [128], and specify which loops to unroll and where to add or avoid synchronization with refinements. We also use vectorized loads (lines 37 and 38) and strided tiles (line 32).

However, the `maxwell` kernel uses a clever trick in its epilog: It streams data through shared memory in a way that allows to allocate less memory than we currently do. We cannot yet express this optimization in Fireiron. However, the overall epilog is still precisely described by a **`Move`** specification, allowing us to inline a micro-kernel during code generation instead (line 13). Having specifications describing every decomposed sub-problem enables a fine-grained reuse of efficient implementations as inlined micro-kernels during code generation.

```
val swizz: Swizzle = id => // permutation of thread-ids
  ((id >> 1) & 0x07) | (id & 0x30) | ((id & 0x01) << 3)
val storeCUDA: String = //* CUDA Epilog Micro Kernel  *//
// MATMUL-KERNEL // GEMM_NT (A: ColMajor Layout, B: RowMajor) ////////////
val maxwellOptimized = MatMul(M,N,K)(GL,GL,GL)(Kernel)
///// BLOCK-LEVEL ///////////////////////////////////////////////////////
  .tile(128,128).to(Block).layout(ColMajor)
//--- epilog: store results RF => GL ----------------------------------//
  .epilog(RF, Init           // accumulate results in registers
     .tile(64,32).to(Warp)   // allocate 64*32 register per warp
     .tile(8,  8).to(Thread) // allocate 64 registers per thread
     .tile(1,  1).unroll.done,
    Move.done(storeCUDA))    // use microkernel (18 additional LoC)
  .split(8).sync             // Block-tile: 128 x 128 x 8 (M x N x K)
//--- move A to SH ----------------------------------------------------//
  .move(MatMul.A, SH, Move(A:128x8)(GL->SH)(Block)
    .tile(128, 1).to(Warp) // every warp moves a full column
    .tile(64,  1).unroll   // in two sequential steps (upper/lower half)
    .tile(2,   1).to(Thread).layout(ColMajor) // vect. 2 values per thread
    .done).storageLayout(ColMajor).noSync     // noSync: We move B next
//--- move B to SH ----------------------------------------------------//
  .move(MatMul.B, SH, Move(B:8x128)(GL->SH)(Block)
    .tile(8, 16).to(Warp)   // use 8 warps in a 1x8 arrangement
    .tile(8,  4).unroll     // each warp moves 4 chunks sequentially
    .tile(1,  1).to(Thread).layout(ColMajor).done
  ).storageLayout(RowMajor) // for improving thread-level accesses
   .pad(4)                  // for avoiding load/store bank conflicts
///// WARP-LEVEL ////////////////////////////////////////////////////////
  .tile(64,32).to(Warp) // 8 warps per block, 2x4 arrangement
///// THREAD-LEVEL //////////////////////////////////////////////////////
  .tile((4,32),(4,16))  // use strided tiles for improved memory accesses
    .to(Thread)         // assign tiles to threads
    .swizzle(swizz)     // permute thread ids for optimizing accesses
    .layout(ColMajor)   // assign permuted 1D ids in col-major order
  .split(1).unroll
// move A and B to RF--(omit Move details for brevity)------------------//
  .move(MatMul.A, RF, Move.tile(4,1).unroll.done)  // moving 128 bit ...
  .move(MatMul.B, RF, Move.tile(1,4).unroll.done)  // (4 * FP32)
//--- perform computation using FMA -----------------------------------//
  .tile(1,1).unroll.done // residual spec: MatMul(1,1,1)(RF,RF,RF)(Thread)
```

Listing 3.4: Fireiron strategy expressing the `maxwell` reference implementation. This implementation uses different strategies for moving both operands to shared memory, strided tiles, and vectorized loads and stores.

```
val cudaWMMASample = MatMul(M,N,K)(GL,GL,GL)(Kernel) // using FP16
////// BLOCK-LEVEL /////////////////////////////////////////////////////////
 .tile(128, 128).to(Block)     // each block computes a 128x128 output tile
 .epilog(FR, Init              // accumulate in 16x16 fragments
     .tile(64,32).to(Warp)     // assign 64x32 tile to a warp ...
     .tile(16,16).unroll.done, // that contains 4x2 16x16-WMMA-fragments
   Move(C:128x128)(FR->GL)(Block) // Epilog in 2 steps: FR -> SH -> GL
     .move(Move.src, SH, Move // Step 1: FR -> SH
       .tile(64,32).to(Warp)
       .tile(16,16).unroll.done) // end Step 1 - results are now in SH ...
     .tile(16, 128).to(Warp)         // Step 2: SH -> GL
     .tile(1,  128).unroll           // achieving coalesced accesses
     .tile(1,    4).to(Thread).done) // vectorized: 4 values per thread
 .split(128).sync.unroll      // Block-tile 128 x 128 x 128 (M x N x K)
//--- move A to SH ------------------------------------------------------//
 .move(MatMul.A, SH, Move(A:128x128)(GL->SH)(Block) // A in GL: RowMajor
   .tile(16,128).to(Warp)                       // 16 rows per warp
   .tile(2, 128).unroll                         // 2 rows at a time
   .tile(1,   8).to(Thread).done).noSync.pad(8) // 8 elements per thread
//--- move B to SH ------------------------------------------------------//
 .move(MatMul.B, SH, Move(B:128x128)(GL->SH)(Block) // B in GL: ColMajor
   .tile(128,16).to(Warp) // similar move strategy but considering ...
   .tile(128, 2).unroll   // B's ColMajor layout for achieving coalescing
   .tile(8,   1).to(Thread).layout(ColMajor).done).pad(8)
///// WARP-LEVEL ///////////////////////////////////////////////////////////
 .tile(64, 32).to(Warp)// Warp-tile: 64 x 32 x 16 (including next line) ...
 .split(16).unroll     // -> 4*2*1 (=8) 16x16x16-WMMA computations per Warp
//--- fill WMMA fragments for A and B-----------------------------------//
 .move(MatMul.A, FR, Move.tile(16, 16).unroll.done)//target new specs: ...
 .move(MatMul.B, FR, Move.tile(16, 16).unroll.done)//wmma::load_matrix_sync
//--- perform WMMA computation -----------------------------------------//
 .tile(16,16).done // residual spec: MatMul(16,16,16)(FR,FR,FR)(Warp)
```

Listing 3.5: Fireiron strategies targeting Tensor Cores using WMMA. Operands are transferred to 16×16-WMMA-Fragments which are also used to accumulate the results.

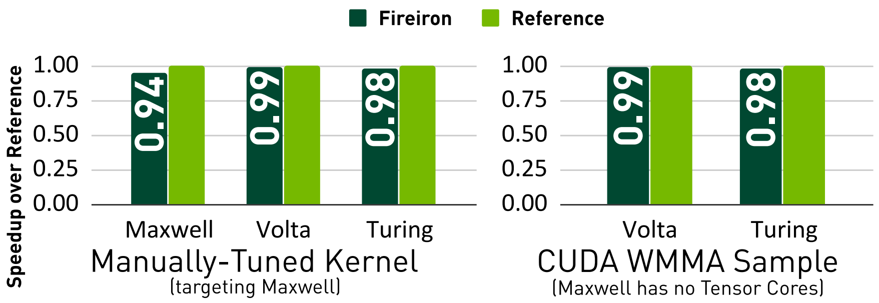


Figure 3.13: Comparing Fireiron generated code against two references. We achieve the same performance while requiring significantly less line of code.

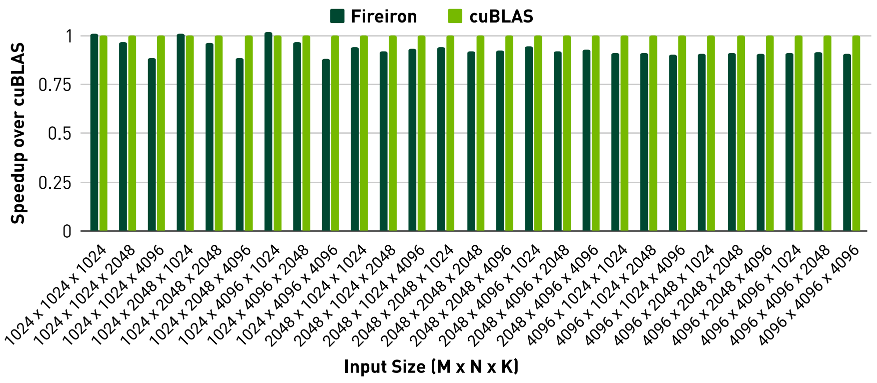


Figure 3.14: Comparing Fireiron-generated code against cuBLAS on large input matrices, both use Tensor Cores.

HYPOTHESIS C: *Fireiron-generated code achieves performance close to expert-tuned code.*

Figure 3.13 shows the performance of our generated kernels using the `maxwell` strategy (left) and the `wmma` strategy (right) shown in Listing 3.4 and Listing 3.5 compared to the reference kernels executed on multiple architectures. Here, we achieve exactly the same performance on Volta and Turing and come very close on the Maxwell architecture compared to the handwritten references while requiring significantly fewer lines of code.

cuBLAS provides the best implementations available written in optimized SASS assembly. It contains multiple differently optimized implementations and chooses one, including tile sizes at runtime, depending on the input sizes and hardware architecture based on internal heuristics. For a fair comparison, we use two parameterized strategies (one more suited for smaller, one for larger inputs), allowing us to explore tile sizes (powers of two: 2^4–2^8) and report the best performance. Figure 3.14 shows the speedup compared to cuBLAS for large inputs. Here, we exactly match the performance in three cases, and on average, we achieve 93.1% of the cuBLAS performance with a minimum of 88.3% in one case and a maximum of 101% in two cases. These results show that Fireiron generates code performing close to the practically achievable peak.

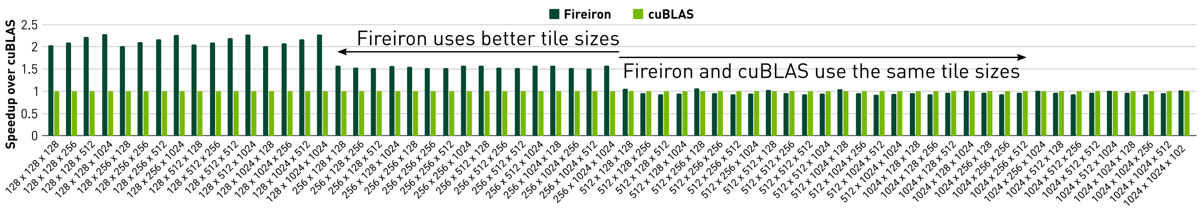


Figure 3.15: Comparing Fireiron-generated code against cuBLAS on small input matrices, both use Tensor Cores.

HYPOTHESIS D: *Experts can write Fireiron strategies that generate code which outperforms the state-of-the-art.*

Experts were able to define Fireiron strategies outperforming the manually optimized cuBLAS code by more than 2×.

Figure 3.15 shows the performance achieved compared to cuBLAS using small input sizes. We can significantly outperform cuBLAS on the smallest input sizes because there we use better tile sizes: We generally found a tile size 16×16 in the M and N dimensions and 64 in the K dimension, computed by two warps per block, to perform best. cuBLAS also chose 64 in the K dimension, but larger sizes for the M and N dimensions, which reduces the available parallelism.

Our high-level schedule language allowed easy experimentation with different tile sizes (it requires changing two lines) whose exploration is a tedious and time-intensive process if kernels are developed in low-level assembly. There, changing tile sizes requires the adjustment of multiple complex index expressions throughout the whole kernel.

3.6 CONCLUSION

In this chapter, we introduced Fireiron, a data-movement-aware scheduling language, IR, and domain-specific compiler for GPUs. Treating data movements as first-class concepts allows the accurate description of high-performance GPU kernels as Fireiron strategies. We introduced specifications for both computations and data movements and decompositions to partially implement and map them to the multi-layered compute and memory hierarchies. Defining low-level PTX assembly as well as macro-instructions like WMMA as executable specs allows us to target specialized hardware like Tensor Cores flexibly.

Using different matrix multiplication implementations, we showed that Fireiron can express optimizations used in hand-tuned kernels written by experts. The code we generate generally matches the performance of hand-tuned implementations and experts can use Fireiron to improve the state-of-the-art by outperforming vendor libraries by more than 2×.

advantages and drawbacks of the fireiron approach Fireiron makes use of domain-specific knowledge in two main ways: Our approach is highly specialized for both, the target architecture (GPUs) and the target domain (matrix multiplication). First, the architecture of modern GPUs, especially their compute and memory hierarchy, is directly embedded into the core design of Fireiron: For example, specs are assignable to elements of the compute hierarchy using `tile`, and the `move` decomposition explicitly expresses data movement between levels of the memory hierarchy. This specialization allows nested specs to reflect the structure of high-performance GPU programs naturally. Second, the target domain (in this case, a single - though pervasive and essential - operation, matrix multiplication), directly dictates the required optimizations we need to be able to express. Two of the four Fireiron decompositions discussed in Section 3.3.2 (`split` and `epilog`) are specific to matrix multiplication computations. These are required for expressing optimizations, like the optimized epilog data movement discussed in Section 3.2, that are necessary for achieving high performance.

Due to these specializations, we were unable to build upon existing schedule-based compilers like Halide or TVM. Those especially lack the support for expressing data movements as first-class concepts in their compilers. Instead, building a domain-specific compiler from scratch, specialized for our target architecture and domain, allowed us to achieve performance that significantly outperforms the state-of-the-art.

As we have shown in this chapter, achieving high-performance via domain-specific compilation is a desirable approach. Achieving this level of performance might even be worth the effort of developing a new compiler from scratch to target new domains or new hardware. However, it remains a time-intensive process, even for compiler-experts. For example, developing and evaluating the Fireiron prototype compiler took about nine months of full-time work. Therefore, in the following chapters, we show how we can achieve domain-specific compilation *without* the need to always start from scratch. We argue that this is possible by addressing the two challenges defined in the introduction.

In the next chapter, we will address the Intermediate Representation Challenge defined in Section 1.2.1 using an extensible IR design based on functional primitives. Chapter 5 will address the Optimization Challenge defined in Section 1.2.2 using a language allowing to specify optimizations a composable, rewrite-based strategies.

4 A GENERIC IR FOR DOMAIN-SPECIFIC COMPUTATIONS

This chapter is largely based on the publication "High-Performance Stencil Computations with Lift" [70] by Hagedorn, Stoltzfus, Steuwer, Gorlatch, and Dubach published at CGO'18

In this chapter, we address the Intermediate Representation Challenge for domain-specific compilation introduced in Section 1.2.1. We demonstrate how a domain-agnostic, functional intermediate representation can be extended and used to express domain-specific computations. Specifically, we extend the Lift IR [162], which so far has been used for expressing linear algebra computations only, for supporting the expression of stencil computations. Our extensions allow to express complex multi-dimensional stencil computations, and optimizations such as tiling, as compositions of simple 1D Lift primitives. Crucially, our extensions to the IR for expressing stencil computations are not stencil-specific but are instead reusable and repurposable. Our experimental results show that this approach outperforms existing compiler approaches and hand-tuned codes in terms of achieved performance. Ultimately, the work discussed in this chapter demonstrates the possibility of achieving domain-specific compilation without using a domain-specific IR.

The IR Challenge: How to define an IR for high-performance domain-specific compilation that can be reused across application domains and hardware architectures while providing multiple levels of abstraction? (Section 1.2.1)

This chapter is largely based on [70], and the acoustic benchmark introduced in Section 4.3.5 and evaluated in Section 4.5 was contributed by Larisa Stoltzfus.

4.1 INTRODUCTION

Stencils are a family of computations that update elements in a multi-dimensional grid based on neighboring values using a fixed pattern. They are used extensively in various application domains such as medical imaging, e. g., SRAD, numerical methods, e. g., Jacobi or machine learning, e. g., convolution neural networks. Stencils are part of the original "seven dwarfs" [7] and are considered one of the most relevant classes of high-performance computing applications.

The efficient programming of stencils for parallel accelerators such as Graphics Processing Units (GPUs) is challenging even for experienced programmers. Hand-optimized high-performance stencil code is usually written using low-level programming languages like OpenCL or CUDA. Similar to optimizing matrix multiplication computations, as discussed in the previous chapter, achieving high performance for stencils requires expert knowledge to manage low-level hardware details. For instance, special care is required on how parallelism is mapped to GPUs or how data locality is exploited using local memory to maximize performance.

Domain-Specific Languages (DSLs) and high-level library approaches have been successful at simplifying stencil-based application development. These approaches are based on algorithmic skeletons [37] that capture recurring patterns of parallel programming. While these raise the abstraction level, they rely on hard-coded, not performance portable implementations. Alternative approaches like Halide [140] or PolyMage [108] are based on code generation. However, those place a massive burden on the compiler developers who have to reinvent the wheel for each new application domain and target hardware.

Lift [162] is a code generation approach based on a high-level, data-parallel intermediate representation whose central tenet is performance portability. It is designed as a target for DSLs and library authors and is based on functional principles to produce high-performance GPU code. Applications are expressed using a small set of functional primitives, and optimizations are all encoded as formal, semantics-preserving rewrite rules. These rules define an optimization space that is automatically explored for high-performance code [167]. This approach liberates application programmers and compiler developers from the tedious process of re-writing and tuning their code for each new domain or hardware.

This chapter shows how stencil computations and optimizations are expressible in Lift, reusing its existing machinery for managing parallelism, memory hierarchy, and optimizations. We find out that surprisingly, only two new primitives are required for expressing stencil computations, one used for neighborhood gathering and one for expressing boundary condition handling. Every Lift primitive, as we introduce in Section 4.3.1, is defined to transform simple one-dimensional arrays. We demonstrate how complex multi-dimensional stencils are expressible as compositions of simple one-dimensional primitives without introducing specialized higher-dimensional stencil primitives as often used in related approaches. This flexibility emphasizes the extensibility of the Lift approach to new application domains.

This chapter also shows how stencil-specific optimizations, such as overlapped tiling, are expressible using rewrite-rules in Lift. We achieve this by reusing existing rules and adding one new rule that handles the newly introduced primitives. By reusing the existing Lift exploration mechanism, we can automatically generate high-performance stencil code for AMD, NVIDIA, and ARM GPUs. Our results show that this approach is highly competitive with hand-written implementations and the state-of-the-art PPCG polyhedral GPU compiler.

In this chapter, we make the following contributions:

1. We show how complex multi-dimensional stencils are expressible using Lift's existing primitives with the addition of only two new primitives.

2. We formalize and implement a stencil-specific optimization – overlapped tiling – as a rewrite rule.

3. We demonstrate that this approach generates high-performance code for several stencil benchmarks.

The rest of this chapter is organized as follows. Section 4.2 motivates our work. Section 4.3 shows how stencil computations are expressed in Lift. Section 4.4 presents stencil-specific optimizations in Lift expressed as rewrite-rules. Section 4.5 explains how we generate efficient OpenCL code from stencils expressed in Lift and provides experimental evidence that this approach achieves high performance on a selection of GPUs. Finally, Section 4.6 concludes.

4.2 MOTIVATION: ADDRESSING THE IR CHALLENGE

The advent of Graphics Processing Units (GPUs) over the past decade has been the first sign of an increasing trend of diversity in computer hardware. The end of Dennard scaling and Moore's law forces computer architects to specialize their designs for increased performance and efficiency. Traditional multi-core CPUs from Intel and AMD are now challenged by more energy-efficient designs by ARM, massively parallel architectures such as GPUs, and accelerators such as the Xeon Phi. This diversity in hardware requires massive changes for software as traditional, sequential implementations are hard to adapt to this zoo of modern architectures automatically.

4.2.1 *High-Level Abstractions for Stencils*

Domain-specific languages (DSLs) and libraries help application developers target modern hardware, shielding them from the ever-changing landscape. They are commonly accepted as being part of the solution to address the challenge of achieving performance portability. DSLs are widely used for stencil computations, which have been extensively – and successfully – studied in terms of application-specific optimizations in the high performance computing community. High-level frameworks such as Halide [141] are designed specifically to express stencil computations, fuse multiple operations, and generate parallel GPU code automatically. Similarly, PolyMage [108] fuses multiple stencil operations and uses the polyhedral model to produce parallel CPU code. These approaches are

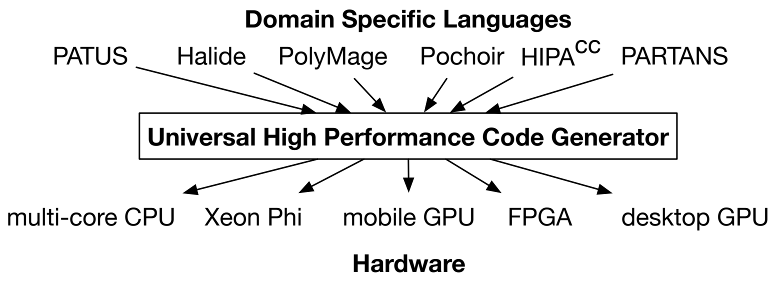


Figure 4.1: Vision of a high-performance code-generator used as a universal interface between DSLs and hardware.

particularly good at optimizing long pipelines of stencil operations typically found in image processing applications.

While DSLs provide an excellent solution for the end-user, they are costly in terms of compiler development. Each new DSL needs to implement its own back-end compiler and optimizer with its own approach to parallelization. This approach is not sustainable, given the number of application domains and the ever-growing hardware diversity.

4.2.2 *Universal High-Performance Code Generation*

What is needed is a compiler approach that can be reused over a wide range of domains and deliver high performance on a broad set of devices. Figure 4.1 shows our vision of a universal compiler between DSLs and hardware, which was first proposed by Delite [170]. Delite advocates the use of a small set of parallel functional primitives upon which DSLs are implemented. A hardware-specific back-end takes care of compiling and optimizing these primitives down to high-performance GPU code, enabling all the DSLs implemented on top of Delite to benefit from these optimizations. This type of approach can lead to excellent performance for many domains on a specific parallel device and, in particular, for stencil code. In this chapter, we build upon this approach but define and extend an IR for which we can use a single back-end that can generate high-performance code for different hardware devices.

Lift [162, 166, 167] has recently emerged as a novel approach to address the performance portability challenge. It follows a similar philosophy as Delite by offering a small set of data-parallel patterns used to express higher-level abstractions. In contrast to Delite, Lift generates high-performance code by encoding algorithmic choices and device-specific optimizations as provably correct rewrite rules. This design makes it easy to extend and add new optimizations into the compiler, in contrast to Delite, where optimizations are hard-coded for each back-end.

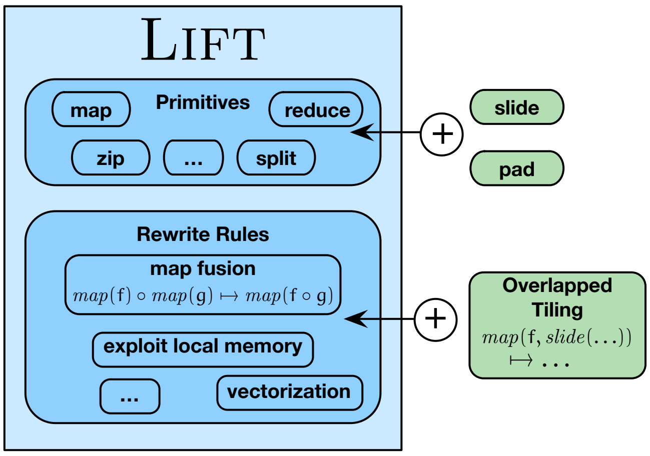


Figure 4.2: Additions to Lift proposed for supporting stencils. Only two new primitives and a rewrite rule implementing the overlapped tiling optimization are added.

LIFT has demonstrated that high performance is achievable for linear algebra operations [166]. In this chapter, we take the LIFT approach a step further and show how it can also be applied, with few modifications, to stencil computations. By successfully using the pattern-based, domain-agnostic LIFT IR for a new domain not targeted so far, we show that this approach is superior to existing domain-specific compilers as it can be simply extended as required. In particular, we show how complex multi-dimensional stencils are expressible by composing a handful of simple 1D primitives. Additionally, we strive to leverage existing functionality in LIFT, inheriting the benefits of automatic exploration of algorithmic and device-specific optimizations.

4.3 A PATTERN-BASED APPROACH FOR EXPRESSING STENCILS

Figure 4.2 shows the new extensions to LIFT for supporting stencil computations. These are described in this and the following section. Only minor additions are required to support stencils and to generate high-performance code across multiple parallel devices.

We begin by describing the existing LIFT primitives we reuse before introducing the two new primitives *slide* and *pad* that allow us to express stencil computations in a functional style. After discussing a one-dimensional stencil example, we introduce the handling of multi-dimensional stencils expressed by composing the fundamental 1D primitives.

4.3.1 *Existing High-Level* Lift *Primitives*

Lift was introduced in [162] and offers a small collection of data-parallel functional primitives. Prior work has shown that it is possible to compile these using rewrite-rules into code that runs effectively on GPUs [167]. The most relevant primitives, which we also reuse to express stencil applications, are shown below with their types to explain how these primitives can be composed.

$$\begin{aligned}
\textbf{\textit{map}} &: (f : T \to U,\ in : [T]_n) \to [U]_n \\
\textbf{\textit{reduce}} &: (init : U,\ f : (U, T) \to U,\ in : [T]_n) \to [U]_1 \\
\textbf{\textit{zip}} &: (in1 : [T]_n,\ in2 : [U]_n) \to [\{T, U\}]_n \\
\textbf{\textit{iterate}} &: (in : [T]_n,\ f : [T]_n \to [T]_n,\ m : \texttt{Int}) \to [T]_n \\
\textbf{\textit{split}} &: (m : \texttt{Int},\ in : [T]_n) \to [[T]_m]_{n/m} \\
\textbf{\textit{join}} &: (in : [[T]_m]_n) \to [T]_{m \times n} \\
\textbf{\textit{at}} &: (i : \mathrm{Cst},\ in : [T]_n) \to T \\
\textbf{\textit{get}} &: (i : \mathrm{Cst},\ in : \{T_1, T_2, \ldots\}) \to T_i \\
\textbf{\textit{array}} &: (n : \texttt{Int},\ f : (i : \texttt{Int},\ n : \texttt{Int}) \to T) \to [T]_n \\
\textbf{\textit{userFun}} &: (s1 : \mathrm{ScalarT},\ s2 : ScalarT',\ldots) \to ScalarU
\end{aligned}$$

We write $[T]_n$ for an array with n elements of type T. Note that arrays can be nested and carry their size in their type. We write $\{T_1, T_2, \ldots\}$ for a tuple with component types T_i. Finally, $T \to U$ denotes a function type expecting a value of type T and returning a resulting value of type U.

MAP, REDUCE, AND ITERATE The *map* primitive applies a function f to all elements of an array and produces a new array of the same length. In Lift, this is the only primitive that expresses data parallelism. *reduce* applies a reduction operator f to an array by traversing it, applying f to the elements, and an accumulator variable initialized with the given $init$ value. *iterate* performs m iterations of a function f reusing the output produced as an input for the next iteration. While this chapter evaluates single iteration stencils only, the *iterate* primitive can be used to perform multiple stencil iterations.

ZIP, SPLIT, AND JOIN *zip* creates an array of tuples $\{T, U\}$ by combining two input arrays of the same length. *split* introduces an additional dimension, by splitting the input array into chunks of size m, where m is a positive number evenly dividing the input size n. *join* performs the opposite operation and flattens two adjacent dimensions by concatenating their elements.

```
for(int i = 0; i < N; i++) {
  int sum = 0;
  for(int j = -1; j <= 1; j++) { // (a) define a neighborhood (nbh)
    int pos = i+j;
    pos = pos < 0   ? 0   : pos; // (b) perform boundary handling
    pos = pos > N-1 ? N-1 : pos;
    sum += A[pos]; }             // (c) compute the output using the nbh
  B[i] = sum; }
```

Listing 4.1: A simple 3-point jacobi stencil code written in C.

ARRAY AND TUPLE ACCESSES The *at* primitive enables the indexing of arrays with constant (literal) indices. For stencils, this is useful for accessing the elements which define the stencil shape. For the rest of this chapter, we write $in[3]$ as syntactic sugar for $at(3, in)$ to express the access of the fourth element in the array in.

Similarly, the *get* primitive provides access to the components of a tuple. For instance, $get(2, in)$ returns the second component of tuple in. For the rest of this chapter, we write $in.2$ as syntactic sugar for $get(2, in)$.

ARRAY CONSTRUCTOR The *array* primitive constructs array elements lazily by invoking the function f with index i and array length n. Later, we show how this primitive is used for creating masks that can be useful for particular multi-dimensional stencils.

USERFUN Finally, *userFun*s define arbitrary functions operating on scalar types such as *Float* or *Int*. These functions are written in C and are embedded in the generated OpenCL code.

4.3.2 *Extensions for Supporting Stencils*

It is not possible to express stencil computations in LIFT using only the existing primitives because they are too restrictive. Therefore, we need to add one or more new primitives to the existing LIFT IR.

The naive way to add support for expressing stencil computations in LIFT would be to use a single new high-level *stencil* primitive. This approach is common and often seen in other high-level approaches, e.g., [31, 32, 164]. However, in LIFT, we avoid adding domain-specific primitives and instead express stencil computations using reusable domain-agnostic building blocks that are not stencil-specific.

Using an example, we show that stencil computations can be decomposed into three fundamental steps. Consider the simple stencil example expressed as C code shown in Listing 4.1. Here a 3-point stencil is applied to a one-dimensional array A of length N that sums the elements for each neighborhood.

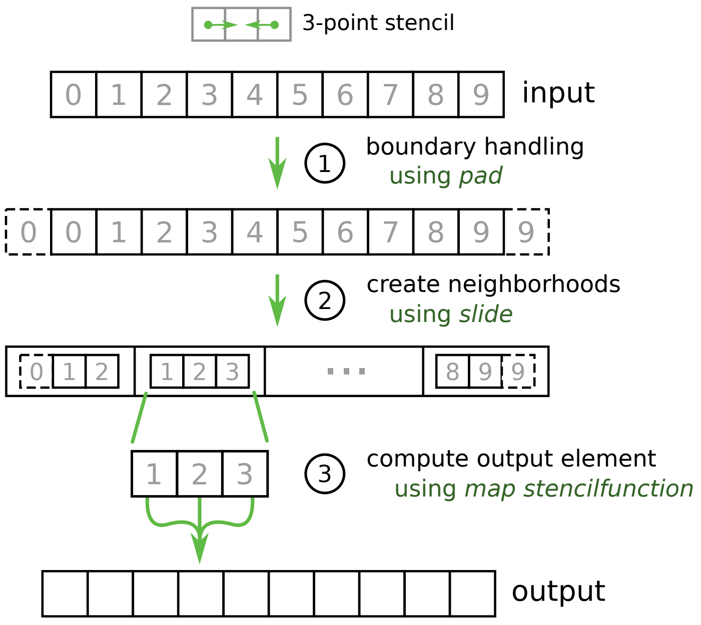

3-point stencil
input
① boundary handling using *pad*
② create neighborhoods using *slide*
③ compute output element using *map stencilfunction*
output

Figure 4.3: Expressing a stencil in LIFT using *pad* for boundary handling, *slide* for creating the neighborhood, and *map* to compute the output elements. These three logical steps are compiled into a single efficient OpenCL kernel by LIFT.

As denoted in the comments, stencil computations generally consist of three fundamental parts:

(a) for every element of the input, a *neighborhood* is accessed specified by the shape of the stencil (line 3);

(b) *boundary handling* is performed that specifies how to handle neighboring values for elements at the borders of the input grid (lines 5 and 6);

(c) finally, *for each* neighborhood, their elements are used to compute an output element (line 7).

For supporting stencils in LIFT, we add two new primitives to perform the first two steps: boundary handling and neighborhood creation. The last step is expressible by reusing an already existing primitive. Following LIFT's design goal, each primitive expresses a single concept, and complex functionality is achieved by composition. The first new primitive *pad* handles boundary conditions, and the second new primitive *slide* expresses element grouping.

BOUNDARY HANDLING WITH *pad* *pad* adds elements to the beginning and end of an array. We define two variants of the *pad* primitive, allowing to add elements in different ways: A first variant reindexes into the input array; a second variant appends values computed by a user-specified function. We write *pad* for both as they can be distinguished based on their unique type.

For stencil computations, *pad* is used to express what happens when we reach the edge of the input array. Step 1 in Figure 4.3 visualizes boundary handling with *pad*. Here, the input array on the top is enlarged with one element on each side, as highlighted with dashed lines. This way of boundary handling is usually referred to as a clamping.

The ***pad*** primitive for reindexing has the following type:

$$\begin{aligned}\textbf{\textit{pad}} : (\quad & l : \texttt{Int},\ r : \texttt{Int},\\ & h : (i : \texttt{Int},\ len : \texttt{Int}) \to \texttt{Int},\\ & in : [T]_n \quad) \to [T]_{l+n+r}\end{aligned}$$

The *pad* primitive adds l and r elements to the beginning and end of the input array in, respectively. It uses the index function h to map indices from the range $[0, l+n+r]$ into the smaller range of the input array $[0, n]$. The elements added at the boundaries are, thus, elements of the input array, and h is used to determine which elements to use. For instance, by defining the following function:

```
clamp(i, n) = (i < 0) ? 0 : ((i >= n) ? n-1 : i)
```

it is possible to express the clamping boundary condition that artificially extends the original $input$ array by two elements to the left and three to the right by repeating the value at the boundary. In the extended version of Lift, we write *pad*$(2, 3, clamp, input)$.

Indexing functions implementing mirroring or wrapping are similarly defined. Indexing functions must not reorder the elements of the input array, but only map indices from outside the array boundaries into a valid array index.

The *pad* primitive to append values has a similar type:

$$\begin{aligned}\textbf{\textit{pad}} : (\quad & l : \texttt{Int},\ r : \texttt{Int},\\ & h : (i : \texttt{Int},\ len : \texttt{Int}) \to T,\\ & in : [T]_n \quad) \to [T]_{l+n+r}\end{aligned}$$

Here, the function h produces a value (i.e., an element of type T), which is added to the ends of the array. This variation of *pad* is used to implement constant boundary conditions to, for example, append zeros at the boundaries as often used in machine learning computations. Another use-case for this version of the *pad* primitive is *dampening* where the out-of-bound values decrease with the distance to the boundary.

CREATING NEIGHBORHOODS WITH *slide* The *slide* primitive creates a nested array of neighboring elements. Conceptually, it uses a sliding window of a particular $size$ that moves with a given $step$ along the input array. For a one-dimensional 3-point stencil we write: *slide*$(3, 1, input)$.

```
1 val sumNbh = fun(nbh => reduce(add, 0.0f, nbh))
2 val stencil =
3   fun( A: Array(Float, N) =>
4     map(sumNbh,                // (c) computing the stencil function
5       slide(3, 1,              // (a) neighborhood creation
6         pad(1, 1, clamp, A)))) // (b) boundary handling
```

Listing 4.2: A 3-point jacobi stencil expressed in LIFT.

Figure 4.3 shows the effect of applying *slide* in the second step. Here, each element of the output array is itself an array of three elements. The second element of the first inner array is hereby also the first element of the second array. This corresponds to the notion that we group the first three elements together before moving the sliding window by one element.

The type of *slide* is defined as:

$$\boldsymbol{slide}: (size\colon \texttt{Int},\ step\colon \texttt{Int},\ in\colon [T]_n) \to [[T]_{size}]_{\frac{n-size+step}{step}}$$

We will later show how this primitive is used to create multi-dimensional neighborhoods. Note that the existing *split* LIFT primitive can now be implemented in terms of the more expressive *slide* primitive: $split(n, in) = slide(n, n, in)$.

COMPUTING THE STENCIL FOR A NEIGHBORHOOD WITH *map*

The *map* primitive is the only way in LIFT to express data parallelism. As stencils are naturally data-parallel, we express the last step of the stencil computation using the *map* primitive. This step takes arrays of neighborhoods as input and performs the stencil computation to produce a single output value for each neighborhood.

4.3.3 *One-dimensional Stencil Example in* LIFT

Listing 4.2 shows a basic 3-point stencil expressed in LIFT. This computation is the same example we saw in Listing 4.1. Due to the functional style, the LIFT expression reads from bottom to top. We can see the decomposition in three logical steps: first, boundary handling is performed (line 6) using *pad*; then, the neighborhoods are created (line 5) using *slide*; finally, *map* is used (line 4) to perform the computation for every created neighborhood using `sumNbh` (line 1).

It is important to emphasize that the logical distinction of these three steps will not be echoed in the generated OpenCL code. The boundary handling and creation of neighborhoods are not performed by copying elements in memory. Instead, they are combined with the last step into a single computation by creating a compiler-internal data structure, called *view* in LIFT [167], which influences how data will be read from memory.

4.3.4 *Multi-dimensional Stencils in* Lift

One of the crucial concepts of our approach is the ability to express complex multi-dimensional stencils as compositions of simple one-dimensional primitives. We will now show how we define n-dimensional versions map_n, pad_n, and slide_n as compositions of the simple *pad*, *slide*, and *map* primitives we have just seen.

Defining these multi-dimensional abstractions allows us to express multi-dimensional stencils following the same structure as the one-dimensional stencil expression:

$$\boldsymbol{map_n}(\mathtt{f},\ \boldsymbol{slide_n}(\mathit{size},\ \mathit{step},\ \boldsymbol{pad_n}(\mathtt{l},\ \mathtt{r},\ \mathtt{h},\ \mathtt{input})))$$

Boundary handling is performed via pad_n using the function $\mathtt{h}$. Here we present the simple case where the same boundary handling strategy is performed in each dimension. It is straightforward – and supported by our implementation – to apply different boundary handlings per dimension. The slide_n creates an n-dimensional neighborhood, which is then processed by map_n.

MULTI-DIMENSIONAL BOUNDARY HANDLING Boundary handling in multiple dimensions follows the same idea as in the one-dimensional case. Using nested *maps*, we apply *pad* to inner dimensions. Thus, pad_n is defined recursively:

$$\begin{aligned}
\mathit{pad}_1(\mathtt{l},\ \mathtt{r},\ \mathtt{h},\ \mathtt{input}) &= \mathit{pad}(\mathtt{l},\ \mathtt{r},\ \mathtt{h},\ \mathtt{input}) \\
\mathit{pad}_n(\mathtt{l},\ \mathtt{r},\ \mathtt{h},\ \mathtt{input}) &= \mathit{map}_{n-1}(\mathit{pad}(\mathtt{l},\ \mathtt{r},\ \mathtt{h}), \\
&\qquad \mathit{pad}_{n-1}(\mathtt{l},\ \mathtt{r},\ \mathtt{h},\ \mathtt{input}))
\end{aligned}$$

where map_n are n nested *maps*:

$$\begin{aligned}
\mathit{map}_1(\mathtt{f},\ \mathtt{input}) &= \mathit{map}(\mathtt{f},\ \mathtt{input}) \\
\mathit{map}_n(\mathtt{f},\ \mathtt{input}) &= \mathit{map}_{n-1}(\mathtt{map(f)},\ \mathtt{input})
\end{aligned}$$

The base case is the simple one-dimensional *pad*. The higher dimensional pad_n case applies a *pad* once in every dimension using nested *maps* to apply it to the inner dimension. The recursive map_n is defined as a nesting of n *maps*.

We provide a simple 2D example for pad_2 using the `clamp` boundary handling, which repeats the values at the boundary. We add explicit parentheses to show the dimensionality of the arrays and additionally arrange the input and output two-dimensional arrays as matrices to visually highlight the effect of multi-dimensional padding.

$$
\begin{aligned}
&\mathit{pad}_2(\mathtt{1,\ 1,\ clamp}, \begin{bmatrix} \mathtt{[a,\ b],} \\ \mathtt{[c,\ d]} \end{bmatrix}) = \\
&\quad \mathit{map}(\mathit{pad}(\mathtt{1,1,clamp}),\ \mathit{pad}(\mathtt{1,1,clamp}, \big[\mathtt{[a,\ b],[c,\ d]}\big])) = \\
&\quad \mathit{map}(\mathit{pad}(\mathtt{1,1,clamp}), \big[\mathtt{[a,\ b],[a,\ b],[c,\ d],[c,\ d]}\big]) = \\
&\qquad \begin{bmatrix} \mathtt{[a,\ a,\ b,\ b],} \\ \mathtt{[a,\ a,\ b,\ b],} \\ \mathtt{[c,\ c,\ d,\ d],} \\ \mathtt{[c,\ c,\ d,\ d]} \end{bmatrix}
\end{aligned}
$$

After expanding the definition of *pad*2, we first apply *pad* to the outer dimension of the two-dimensional array, resulting in an enlarged array where the first and last element – themselves both arrays – are prepended and appended. Then, in the second step, *pad* is applied to the nested dimension using the *map* primitive, which applies *pad* to every nested array resulting in the final two-dimensional array. As expected, the output array contains the input array with additional rows and columns resulting from applying *pad* twice.

This example already demonstrates the expressiveness of composing simple one-dimensional primitives to higher-level abstractions without the need to define them as built-in constructs in the IR themselves.

MULTI-DIMENSIONAL NEIGHBORHOOD CREATION The creation of multi-dimensional neighborhoods is more complicated than the multi-dimensional boundary handling. However, it follows a similar idea: We compose the simple one-dimensional *slide* primitive and apply it once per dimension.

For the two-dimensional case, *slide*2 is defined as:

$$
\begin{aligned}
&\mathit{slide}_2(\mathit{size},\ \mathit{step},\ \mathit{input}) = \\
&\quad \mathit{map}(\mathit{transpose}, \\
&\qquad \mathit{slide}(\mathit{size},\ \mathit{step}, \\
&\qquad\quad \mathit{map}(\mathit{slide}(\mathit{size},\ \mathit{step}),\ \mathit{input})))
\end{aligned}
$$

We explain this definition using an example. In the following, we use a two-dimensional matrix as input and apply *slide*2(2,1,input). This way, we conceptually slide a window of size 2×2 across the input and obtain a four-dimensional result.

$$
\begin{aligned}
&\mathit{slide}_2(2,1,\begin{bmatrix}[a, & b, & c],\\ [d, & e, & f],\\ [g, & h, & i]\end{bmatrix}) = \\
&\quad \mathit{map}(\mathit{transpose},\ \mathit{slide}(2,1,\\
&\qquad \mathit{map}(\mathit{slide}(2,1),\ \Big[\ [a,\ b,\ c],\ [d,\ e,\ f],\ [g,\ h,\ i]\Big]))) = \\
&\quad \mathit{map}(\mathit{transpose},\ \mathit{slide}(2,1,\\
&\qquad \Big[\ [[a,\ b],\ [b,\ c]],\ [[d,\ e],\ [e,\ f]],\ [[g,\ h],\ [h,\ i]]\Big])) = \\
&\quad \mathit{map}(\mathit{transpose},\\
&\qquad \begin{bmatrix}[\ [\ [a,\ b],\ [b,\ c]\ ],\ [\ [d,\ e],\ [e,\ f]\ ]\ ],\\ [\ [\ [d,\ e],\ [e,\ f]\ ],\ [\ [g,\ h],\ [h,\ i]\ ]\ ]\end{bmatrix})) = \\
&\qquad \begin{bmatrix}\left[\begin{bmatrix}[a, & b],\\ [d, & e]\end{bmatrix}, \begin{bmatrix}[b, & c],\\ [e, & f]\end{bmatrix}\right],\\ \left[\begin{bmatrix}[d, & e],\\ [g, & h]\end{bmatrix}, \begin{bmatrix}[e, & f],\\ [h, & i]\end{bmatrix}\right]\end{bmatrix}
\end{aligned}
$$

The resulting four-dimensional array is created out of four 2×2 neighborhoods. These are created by applying *slide* to the inner and then the outer dimension before using *map(transpose)* to swap the two inner dimensions.

Again, we are able to generalize the definition of *slide*$_2$ to a multi-dimensional *slide*$_n$ case that, when applied to an n-dimensional input, creates n-dimensional inner neighborhoods. The general structure remains similar to the two-dimensional case:

```
slide_n(size, step, input) =
  reorderingDimensions(
    slide(size, step,
      map(slide_{n-1}(size, step, input))))
```

We first recursively apply the sliding in all nested dimensions for the n-dimensional input using *map*(*slide*$_{n-1}$). Then we apply *slide* to the last missing outermost dimension. This way, we have now applied *slide* exactly once to all dimensions of the input.

In the last step, we must reorder the dimensions, so that the nested dimensions created by the *slide*s are the innermost ones. In the 2D case, a single *map(transpose)* achieved this; However, it is more complicated in the multi-dimensional case. This required reordering is best understood by looking at the types involved. For a three-dimensional array, after applying *slide* in each dimension, we obtain an array of this type: $[[[[[[T]_{s_o}]_o]_{s_n}]_n]_{s_m}]_m$, where s_m and m are the two dimensions resulting from applying *slide* to the outermost dimension. By rearranging the dimensions, we obtain an array of type: $[[[[[[T]_{s_o}]_{s_n}]_{s_m}]_o]_n]_m$, which corresponds to the desired result: a three-dimensional neighborhood. The rearranging is realized purely as a combination of *map* and *transpose* calls that swap individual dimensions.

4.3.5 *Case Study: Room Acoustics Simulation*

Listing 4.3 shows a real-world stencil application used for modeling room acoustics developed by HPC physicists [186] and models the behavior of a sound wave propagating from a source to a receiver in an enclosed 3-dimensional space.

The two inputs used in this benchmark ($\texttt{grid}_{t-1}$ and $\texttt{grid}_t$ in lines 1- 2) indicate previous and current time steps to update the state of the room across time. This type of inputs is often found in real-world physical simulations, spanning three dimensions for physical space and one dimension for time. The first grid is taken point-by-point; however, the second grid uses *slide3* to form stencil neighborhoods. The number of neighborhoods correctly matches up to the size of the $\texttt{grid}_t$ input array as the $\texttt{grid}_{t-1}$ input is padded using *pad3* first, so that no out-of-bounds accesses occur. These inputs are then zipped together with their number of neighbors resulting in a tuple of: {VALUE$_{t-1}$, NEIGHBORHOOD$_t$, NUMNEIGHBORS} as seen in lines 12- 14.

In lines 4- 6, the stencil is computed by accessing the neighborhood's values using the *at* primitive (written as [ ]). The results are then combined with the other inputs in an equation to model the sound (lines 8–11).

A difficult problem for wave-based simulations is the accurate handling of physical obstacles in the room. The variable coefficients (a. k. a. "loss") at the obstacle's boundary are handled through the use of a mask. This mask returns a different value depending on whether it is on an obstacle or not. In LIFT, this mask is calculated on the fly using the *array3d* generator and contains a value at each point in the grid.

4.3.6 *Summary*

In this section, we have demonstrated how stencils are expressed in LIFT by extending the set of primitives by two new additions: *pad* and *slide*. Together with existing LIFT primitives, this allows for expressing multi-dimensional stencils as compositions of the one-dimensional building blocks. Crucially, neither *pad* nor *slide* are inherently stencil-specific but rather simple primitives that modify one-dimensional arrays in a new way. This design allows us to repurpose and use them for different use cases, as we will see in the next section. This repurposing would not have been possible

```
acousticStencil(grid_{t-1}:[[[Float]_m]_n]_o,
                grid_t:[[[Float]_m]_n]_o) {
 map3(m -> {
   val sumGrid_{t-1} =
    m.1[0][1][1] + m.1[1][0][1] + m.1[1][1][0] +
    m.1[1][1][2] + m.1[1][2][1] + m.1[2][1][1]
   val numNeighbor = m.2
   return getCF(m.2, CSTloss1, 1.0f) * ( (2.0f -
      CST_l2 * numNeighbor) * m.1[1][1][1] +
      CST_l2 * sumGrid_{t-1} - getCF(m.2,
      CSTloss2, 1.0f) * m.0) },
 zip3(grid_t,
      slide3(3, 1, pad3(1,1,1,zero,grid_{t-1})),
      array3(m,n,o,computeNumNeighbors))) }
```

Listing 4.3: A 3D time-stepped room accoustics simulation expressed in Lift.

if we had decided to build a new *stencil* primitive as often found in other domain-specific compilers. Additionally, the parallelism found in stencil applications is expressed using the existing *map* primitive, without introducing a special case for stencils. Rewrite rules explaining how to optimally leverage OpenCL hardware using *map* are then reusable for stencil applications as we show next.

To summarize, so far we have been able to avoid the disadvantages of defining domain-specific IR constructs while still being able to express complex multi-dimensional domain-specific computations in our IR by:

1) decomposing typical stencil computations into three fundamental building blocks (1. neighborhood creation, 2. boundary handling, 3. output computation) and defining flexible and domain-agnostic new *pad* and *slide* primitives for expressing the first two parts.

2) composing simple one-dimensional primitives for expressing higher-dimensional computations, avoiding the need to define specialized higher-dimensional abstractions directly in the IR.

3) reusing the *map* primitive to express data-parallel computations in stencils allowing to reuse existing *map*-related rewrite rules.

4.4 OPTIMIZING STENCILS USING REWRITE RULES

This section discusses how stencil-specific optimizations can be expressed as rewrite rules. We will add a new rewrite rule, which is used together with Lift's existing rules to explore the implementation space of stencil applications. By applying different rewrites, we can tailor programs to target different architectures, thus, achieving performance portability.

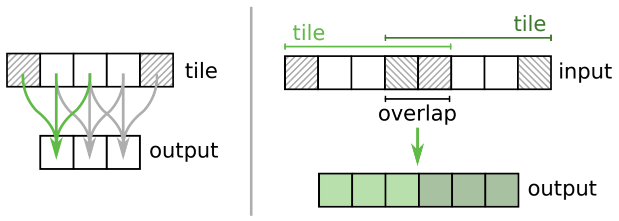


Figure 4.4: Overlapped tiling for a 3-point stencil.

```
fun( A: Array(Float, N) =>
  map(tile => map(sumNbh,  // compute output by accumulating nbh-elements
    slide(3, 1, tile), // use slide to create neighborhoods within tiles
      slide(5, 3, // repurpose slide to create overlapping tiles
        pad(1, 1, clamp, A)))) ) // boundary handling as usual
```

Listing 4.4: A 3-point stencil using overlapped tiling.

4.4.1 *Exploiting Locality through Tiling*

Stencil applications involve local computations that only access elements in a neighborhood. Furthermore, neighboring elements in a grid share large parts of their neighborhoods. Exploiting this locality is the most commonly used and most successful optimization for stencil computations.

On GPUs, the fast but small local memory (shared memory in the CUDA terminology) can be used as a programmer-controlled cache. For stencil computations, this allows us to cache a set of neighborhoods and load elements from the slow global memory only once. Successive accesses by threads that share the same neighborhood elements are then made from the faster local memory.

Traditionally, locality in stencils is exploited using the overlapped tiling optimization [60, 62, 194]. The input grid is divided into tiles that overlap at the edges to allow every grid element to access its neighboring elements. The size of the neighborhood determines the size of the overlap.

Figure 4.4 shows overlapped tiling for a 3-point one-dimensional stencil. The left-hand side shows a single tile of five elements. Here we can see the reuse of data where the highlighted computation on the left shares two elements from the tile with the computation in the middle. Assuming that one thread computes one output element, without overlapped tiling in shared memory, all three threads assigned to this tile would have to access the same three elements in the middle of the tile from the slow global memory. On the right-hand side, we can see the overlap between the left and right tiles. These two elements are available in both tiles.

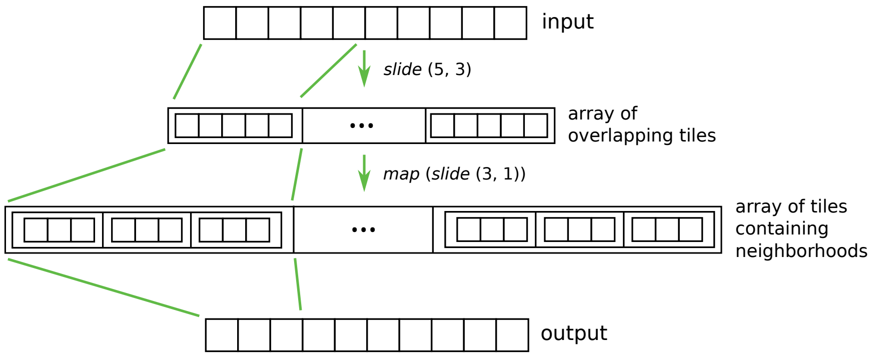

input
slide (5, 3)
array of overlapping tiles
map (slide (3, 1))
array of tiles containing neighborhoods
output

Figure 4.5: Expressing overlapped tiling using *slide*: Applying *slide* to the input creates overlapping tiles. Applying *slide* to every tile creates the required neighborhoods.

REPRESENTING OVERLAPPED TILING IN LIFT We reuse the *slide* primitive to represent overlapping tiles. Listing 4.4 shows the LIFT expression of the 3-point stencil using tiling. The *slide* primitive is used twice: in line 3, a neighborhood is created, as explained earlier, but in line 5, overlapping tiles are created instead of neighborhoods. Due to the parameter choice, (5 and 3) in this case, five elements are grouped in a tile (compare with Figure 4.4), with two elements overlapping with the next tile.

Figure 4.5 shows the creation of tiles in the first step. Afterward, for each tile, we create the local neighborhoods using the *slide* primitive as before.

TILING AS A REWRITE RULE Encoding the tiling optimization as a rewrite rule makes it accessible to LIFT's automatic exploration process. One-dimensional tiling is expressible as follows:

$$\begin{aligned}&\mathit{map}(f,\ \mathit{slide}(\mathit{size},\mathit{step},\mathit{input})) \mapsto\\&\quad \mathit{join}(\mathit{map}(\mathit{tile} \Rightarrow \mathit{map}(f,\mathrm{slide}(\mathit{size},\mathit{step},\mathit{tile})),\\&\qquad\qquad \mathit{slide}(u,v,\mathit{input})))\end{aligned}$$

The parameters u and v have to be selected appropriately, i.e., the difference between the *size* and *step* has to match the difference of u and v: $\mathit{size}-\mathit{step} = u - v$. Figure 4.4 visualizes this constraint for the one-dimensional 3-point Jacobi where the neighborhood *size* is 3 (and the *step* is 1). When choosing the size of the tile u, e.g., 5 in the example, v must be selected to match the formula (i.e., 3 in this case) as $3-1 = 5-3$. This is the only correct choice for v because it determines the overlap created between the tiles, which correlates with the *size* of the original neighborhood. Choosing u and v according to the formula ensures that we create the same number of neighborhoods on both sides of the rewrite rule.

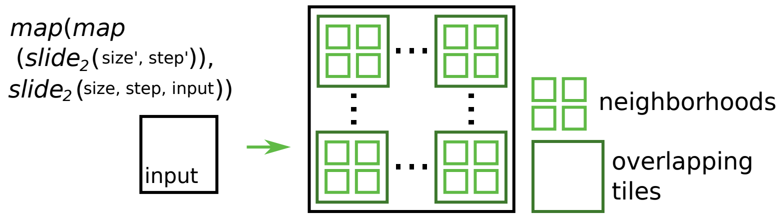


Figure 4.6: Applying overlapped tiling in two dimensions.

For more straightforward reasoning about the tiling rule, we can decompose it into two smaller rules:

$$map(\mathtt{f},\ join(input)) \mapsto join(map(map(\mathtt{f}), input))$$

and

$$\begin{aligned} &slide(size, step, input) \mapsto \\ &\qquad join(tile \Rightarrow map(\mathtt{slide}(size, step, tile)), \\ &\qquad\quad slide(u, v, input)) \end{aligned}$$

Here, on both sides of the first rule, the function f is applied to each element of the two-dimensional *input*. On the left-hand side, this is done by flattening the input and then applying the function, whereas on the right-hand side the function is first applied to each element of the input and then flattened afterwards.

Assuming that u and v are valid parameter choices as described above, the correctness of the second rule is also straightforward. Starting on the right-hand side, we create tiles using the first *slide* primitive. Then, we perform the second *slide* for each created tile, before the *join* removes the outermost dimension and, therefore, resolves the tiles, leaving us with a two-dimensional array equivalent to the array produced by only applying the second *slide*.

MULTI-DIMENSIONAL OVERLAPPED TILING AS REWRITE RULES

Our extension to LIFT fully supports tiling in arbitrarily high dimensions due to the compositional nature of expressing computations and optimizations. Figure 4.6 visualizes the overlapped tiling optimization applied in two dimensions. The optimization rules for tiling higher-dimensional stencils are expressed by reusing the one-dimensional primitives. The rewrite rule implementing two-dimensional tiling looks similar to the one-dimensional case when written with the high-level map_2 and $slide_2$ abstractions introduced previously:

$$\begin{aligned} &map_2(\mathtt{f},\ slide_2(size, step, input)) \mapsto \\ &\quad map(join,\ join(map(transpose, \\ &\quad\ map_2(tile \Rightarrow map_2(\mathtt{f}, \mathtt{slide}_2(size, step, tile)), \\ &\qquad\quad slide_2(u, v, input))))) \end{aligned}$$

Also note that even though we reuse the one-dimensional primitives to express a two-dimensional tiling rule, we do not reuse the one-dimensional tiling rule itself. Instead, we define a new rule specialized for two dimensions. Chapter 5 introduces an approach that allows us to reuse existing rewrite rules by composing rules to what we call *strategies* similar to how we composed primitives to more complex computations in this chapter.

In the following, we briefly discuss two other optimizations encoded as rewrite rules in LIFT. Those are also used in the automatic exploration process. In Chapter 5, we discuss in detail how to implement these and other optimizations as composable and reusable rewrite strategies.

4.4.2 *Usage of Local Memory*

Tiling is used to exploit locality. Modern GPUs have relatively small caches and rely on the programmer explicitly using the fast scratchpad shared memory called local memory in OpenCL. As discussed in Chapter 3, using shared memory efficiently can be cumbersome. Furthermore, it does not always improve the achieved performance, as we will see in the following evaluation. Whether or not the use of local memory is beneficial depends on the hardware architecture and the amount of data reuse in the stencil application.

In LIFT, we address these issues by expressing the local memory usage as a rewrite rule. When exploring the optimization space, this rule will be one of many optimization choices applied in the automatic optimization process.

Besides the high-level primitives introduced in Section 4.3, LIFT also defines OpenCL-specific low-level primitives [69, 162] to exploit particular features of OpenCL, such as the use of the local memory. The *toLocal* primitive, for example, wraps around a function to indicate that this function should write its result into local memory. To copy a single scalar value into local memory, we can use the identity user function *id*, as in: *toLocal(id)*. For copying arrays, we wrap the *map(id)* function in *toLocal*.

As copying data into local memory is always legal inside an OpenCL workgroup, we use the existing LIFT rewrite rule to express this:

$$map(id) \mapsto toLocal(map(id))$$

Together with a rule that introduces *map*(*id*) at any position, it allows the exploration of copying to local memory as an optimization. Currently, heuristics are used to prevent applying this rule at unfavorable locations within a LIFT expression.

4.4.3 *Loop Unrolling*

Unrolling loops is a traditional low-level optimization that can significantly increase performance for some instances. To explore loop unrolling as an optimization for stencil applications, we reuse an existing variation of the *reduce* primitive unrolled by the LIFT compiler. As we saw in the 3-point stencil example in Listing 4.2, the *reduce* pattern is often used in stencil computations to sum up the values in a neighborhood. The unrolled variation of the reduction is called *reduceUnroll* and has a matching rewrite rule providing it as an optimization choice during exploration. Unrolling is only legal if the size of the input array has a length which is known at compile time. For stencils, this is usually the case, as the reduction is applied to a neighborhood which almost always consists of a fixed number of elements.

4.4.4 *Summary*

In this section, we have shown how stencil optimizations are expressed as rewrite rules, which are then applied by the LIFT exploration process. Overlapped tiling in multiple dimensions is expressed by reusing ideas of the simple one-dimensional case. With low-level optimizations, such as local memory usage, we can automatically explore a variety of optimizations for stencil applications.

4.5 EXPERIMENTAL EVALUATION

In this section, we briefly discuss LIFT's code generation and evaluate the performance achieved by the kernels generated using the approach discussed in this chapter.

CODE GENERATION A stencil program expressed using *pad*, *slide*, and *map* is rewritten into a LIFT expression using low-level, OpenCL-specific primitives. These low-level primitives explicitly encode implementation and optimization choices like the use of local memory.

We did not need to extend the original set of existing low-level primitives [162]. We will discuss and introduce these primitives in detail in the following chapter when we discuss how to encode low-level optimizations as rewrite strategies for addressing the Optimization Challenge identified in Section 1.2.2.

The OpenCL code generation is mainly unchanged from the compilation of LIFT programs, as described in [167]. The overall compilation flow consists of multiple stages, including type analysis, memory allocation, array access computation and simplification, and synchronization elimination.

EXTENDING LIFT'S VIEW SYSTEM Lift uses so-called *views* when implementing primitives, which modify data layout without performing computations themselves. These operations are not performed in memory but define how primitives read input data. Examples for such so-called *data-layout primitives* are *split* and *join* whose view construction and consumption is further described in [167]. *Pad* and *slide* are implemented using the same approach [65].

Data-layout primitives are integrated with Lift's view system and are not directly compiled to OpenCL code. For example, *pad* does not initiate the allocation of a new enlarged array in memory. Instead, the reindexing of computations introduced with *pad* is performed when the padded array is read for the first time. Similarly, the *slide* primitive does not physically copy created neighborhoods into memory. *Slide* refers accesses to elements of a neighborhood back to the original array. This way accesses to the same element in different neighborhoods result in memory accesses from the same physical location. This technique allows the expression of complex - potentially multi-dimensional - abstractions in Lift that are nevertheless compiled to efficient OpenCL code.

4.5.1 *Experimental Setup*

METHODOLOGY Experiments are conducted using single precision on a Tesla K20c with CUDA 8.0 driver version 367.48; an AMD Radeon HD 7970 with OpenCL version 1.2 AMD-APP (1912.5); and the SAMSUNG Exynos 5422 ARM Mali GPU with OpenCL 1.2 v1.r17p0. The ARM GPU significantly differs from the two other GPUs because it is mostly designed to be used in low-power embedded systems such as smartphones. Achieving high performance on this set of GPUs thus demonstrates the capability of our approach to be reusable across diverse architectures, one of the main goals identified in the IR Challenge (Section 1.2.1).

The medians of 100 executions are reported measured using the OpenCL profiling API. Data transfer times are ignored since the focus is on the quality of the generated kernel code.

BENCHMARKS The Lift-generated kernels are compared to hand-tuned and automatically-generated kernels from the PPCG [179] state-of-the-art OpenCL polyhedral compiler. We collected hand-written kernels from SHOC (v1.1.5), Rodinia (v3.1), and an OpenCL version of the acoustics simulation code discussed in Section 4.3.5. We hard-coded each benchmark to perform a single iteration of the stencil computation. We also collected a series of single-kernel C codes that work with the PPCG compiler from a recent study [146, 147], provided by the authors. Table 4.1 lists these benchmarks along with their key characteristics.

Benchmark	Dim	Pts	Input size	#grids
Stencil2D [40]	2D	9	4098×4098	1
SRAD1 [27]	2D	5	504 × 458	1
SRAD2 [27]	2D	3	504 × 458	2
Hotspot2D [27]	2D	5	8192×8192	2
Hotspot3D [27]	3D	7	512×512×8	2
Acoustic [168]	3D	7	512×512×404	2
Gaussian [147]	2D	25	4096^2 / 8192^2	1
Gradient [147]	2D	5	4096^2 / 8192^2	1
Jacobi2D [147]	2D	5/9	4096^2 / 8192^2	1
Jacobi3D [147]	3D	7/13	256^3 / 512^3	1
Poisson [146]	3D	19	256^3 / 512^3	1
Heat [146]	3D	7	256^3 / 512^3	1

Table 4.1: Benchmarks used in the evaluation.

4.5.2 *Auto-Tuning*

As explained in the previous sections, the Lift approach exposes optimization choices via rewrite rules, which leads to several differently optimized low-level Lift expressions per benchmark. Each of these low-level expressions contains several tunable parameters. These parameters control, for instance: local/global thread counts (local-size and global-size in the *OpenCL* terminology), tile sizes, how much work a thread performs (sometimes also called the *thread-coarsening factor*), or how memory accesses are reordered. The parameters of each Lift expression are fine-tuned using the ATF auto-tuning framework [142, 143]. ATF builds on top of OpenTuner [6], a classic auto-tuner that uses an ensemble of effective search techniques to optimize a user-specified metric (execution time in our case). Additionally, ATF allows the specification of inter-parameter constraints in the search space. For example, when tuning OpenCL thread sizes, the local-size must evenly divide the global-size, which can be specified as a constraint on the parameters in ATF. The auto-tuner was used for a maximum of three hours per benchmark for tuning all low-level expressions.

The PPCG compiler used in our comparison exposes global/local thread counts and tile sizes as tunable parameters in each dimension. We also use ATF and OpenTuner for finding the best combination of these parameters, with again a maximum tuning time of three hours per benchmark. For both Lift and PPCG, the auto-tuner has been configured to take the OpenCL specific constraints into account (e. g., global thread counts should be a multiple of local thread counts).

4.5.3 *Comparing against Handwritten Benchmarks*

This section presents the results of the exploration and auto-tuning process. It also shows the performance achieved by handwritten optimized kernels from the benchmark suites or HPC experts, as explained previously. Performance is reported as elements updated per second, which we define simply as the output size divided by the execution time. This metric is often used for stencil computations and is sometimes called *LUPs - Lattice Updates per second* [187].

Figure 4.7 shows the performance for the six benchmarks for which we have handwritten reference implementations. As can be seen, in most cases, the Lift generated kernels are comparable to their handwritten counterparts. This performance shows that our domain-agnostic extensions to the Lift IR for expressing stencil computations can be used to generate code that performs as well as code developed by human domain experts.

The benchmarks `srad1` and `srad2` seem to under-perform compared to the other benchmarks on the AMD and NVIDIA platforms. This performance is low because the input sizes are too small to saturate the large server-class AMD and NVIDIA GPUs. On the smaller ARM GPU, these benchmarks perform as good as the others.

The `Hotspot2D` benchmark is also a clear outlier on the AMD and ARM platforms. On the ARM GPU, the Lift generated version is $2\times$ faster than the handwritten version. On the AMD platform, the performance of the handwritten version is clearly under-performing, especially compared to the performance of the other benchmarks. The Lift generated kernel achieves similar performance compared to the other benchmarks while being $15\times$ faster than the handwritten version, which originally targets NVIDIA GPUs. These differences in performance clearly demonstrate the need for flexible and hardware-agnostic code-generation techniques that still can compile code specifically tuned for a particular device.

4.5.4 *Comparing against Polyhedral Compilation*

This section compares our approach towards expressing stencil computations in Lift with the state-of-the-art PPCG polyhedral GPU compiler [179]. Similar to Lift, PPCG is an approach for generating optimized data-parallel programs starting from a single unoptimized program.

Figure 4.8 shows the relative performance of Lift-generated kernels over PPCG-generated kernels. As explained in Section 4.5.2, both Lift and PPCG use the same auto-tuning mechanism for a fair comparison. As can be seen, in most cases, the Lift generated code is on-par or significantly outperforms PPCG.

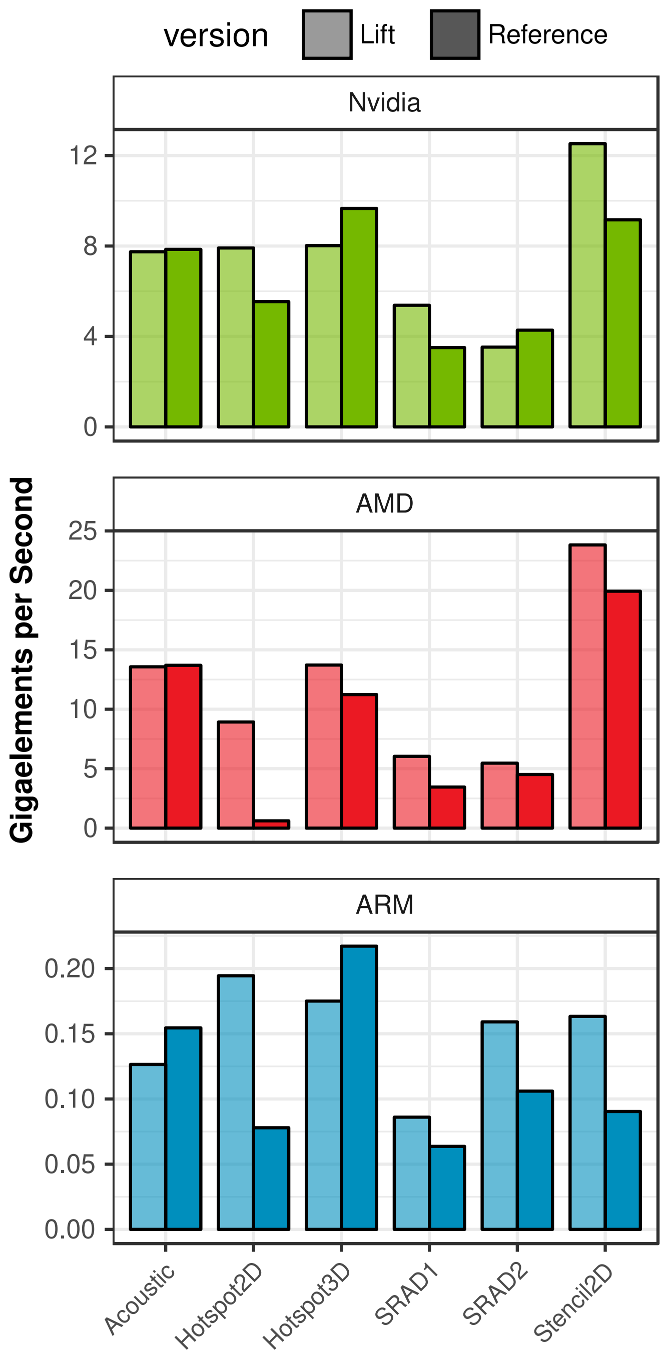


Figure 4.7: Performance of the Lift generated code and hand-optimized kernels reported as Giga-elements updated per second.

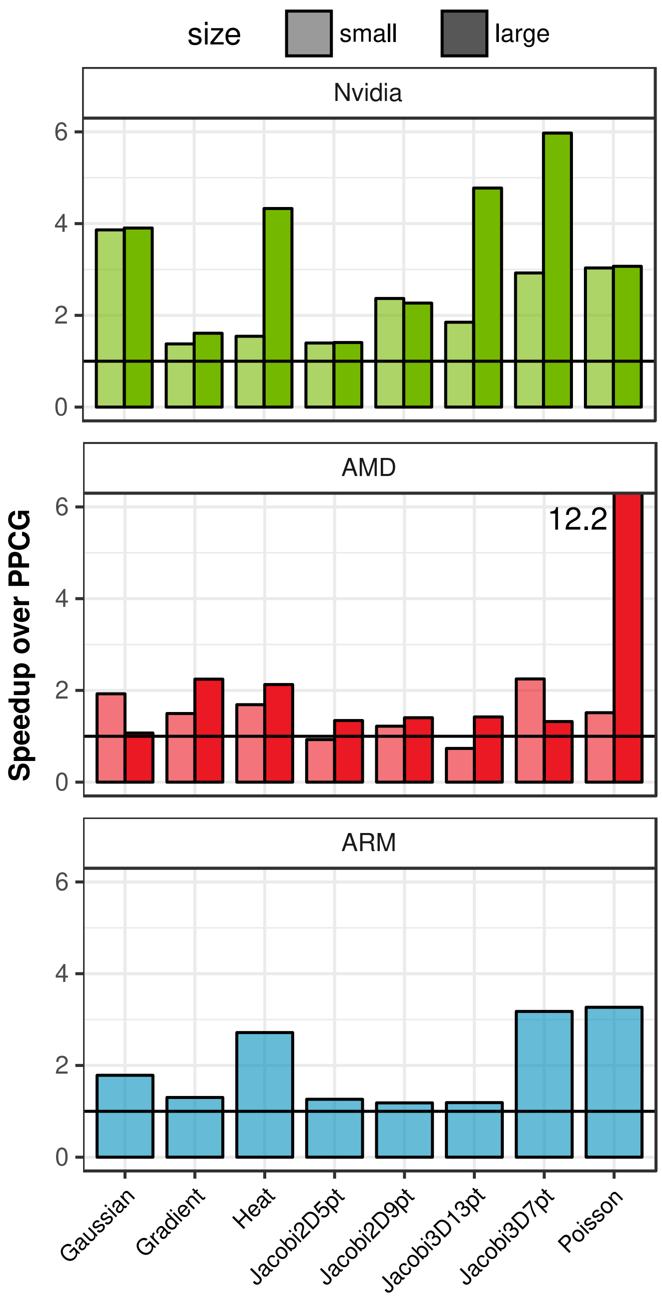


Figure 4.8: Performance of Lift-generated kernels compared to PPCG-generated kernels. Both approaches auto-tune the kernels for up to three hours per benchmark/input/device. Large input sizes did not fit onto the ARM GPU.

On NVIDIA, many benchmarks achieve a speedup of up to 4× over PPCG, such as the `Heat` benchmark using large input sizes, where Lift is 4.3× faster. In this case, the best Lift kernel performs no tiling, and each thread computes only two output elements. On the contrary, the PPCG version looks very different and uses tiling, with each thread processing 512× more elements sequentially than Lift's best version. For the `Gradient` benchmark using the smaller input size, the PPCG performance is almost as good as the performance achieved by our approach. In this case, both versions are similar, use tiling and the difference between the amount of sequential work is only 4×.

On AMD, the results are more uniform, except for the `Poisson` benchmark using the large input size. Here again, the best Lift kernel does not use tiling, while the PPCG compiler generates a tiled version of the benchmarks. On the ARM GPU, Lift and PPCG are much closer than on the other platforms. Here, most of the performance improvements come again from not using tiling.

Interestingly, none of the Lift kernels generated for the ARM or AMD GPU use tiling, however, on NVIDIA 33% of the best Lift versions use the tiling optimization. This confirms that different optimization strategies are required for varying program/input sizes as well as for different hardware.

4.6 CONCLUSION

This chapter showed how stencils and their optimizations are expressible in the domain-agnostic intermediate representation Lift. We extended the Lift IR with two new primitives: One for gathering neighboring elements (*slide*) and one for defining boundary conditions (*pad*). With these extensions, Lift can express and generate code for even complex multi-dimensional stencil computations (like acoustics simulations) *without* requiring domain-specific IR constructs.

This chapter also discussed how stencil-specific optimizations are encoded as rewrite rules. This approach requires only a few new rules explaining how the newly introduced primitives interact with one another. Due to the modular design of Lift, we can leverage existing optimizations, which are also directly applicable to stencil computations. We have shown that Lift is easily extensible to new domains with little effort required and without the need to introduce domain-specific constructs.

Finally, experimental results provide evidence that this approach generates high-performance stencil code on GPUs. On three platforms, we see that performance is on par with hand-optimized reference implementations. We also show that Lift outperforms the PPCG polyhedral compiler in many cases.

The extensions we described in this chapter, and the achieved performance ultimately show that the domain-agnostic LIFT IR is a viable solution to the IR challenge introduced in Section 1.2.1. In the next chapter, we discuss our approach towards providing a solution for the Optimization Challenge defined in Section 1.2.2, which allows us to reuse and compose rewrite rules encoding optimizations just like we composed computational primitives in this chapter.

5 A LANGUAGE FOR DEFINING OPTIMIZATION STRATEGIES

This chapter is largely based on the publication "Achieving High-Performance the Functional Way" [68] by Hagedorn, Lenfers, Koehler, Qin, Gorlatch, and Steuwer published at ICFP'20.

In this chapter, we address the Optimization Challenge introduced in Section 1.2.2. Optimizing programs to run efficiently on modern parallel hardware is hard but crucial for many applications. The predominantly used imperative languages - like C, CUDA, or OpenCL- force the programmer to intertwine the code describing functionality and optimizations. This approach results in a portability nightmare that is particularly problematic, given the accelerating trend towards specialized hardware devices to further increase efficiency.

The Optimization Challenge: How can we encode and apply domain-specific optimizations for high-performance code generation while providing precise control and the ability to define custom optimizations, thus achieving a reusable optimization approach across application domains and hardware architectures? (Section 1.2.2)

Many emerging DSLs used in performance demanding domains, such as deep learning or high-performance image processing, attempt to simplify or even fully automate the optimization process. Using a high-level - often functional - language, programmers focus on describing the functionality in a declarative way. In schedule-based compilers, including Fireiron (Chapter 3), Halide [140], or TVM [29], a separate *schedule* specifies how the program should be optimized. Unfortunately, these schedules are not written in well-defined programming languages. Instead, they are implemented as ad-hoc predefined APIs that the compiler writers have exposed.

In this chapter, we present ELEVATE, a functional language for describing optimization strategies. ELEVATE is inspired by prior systems that express optimizations as compositions of rewrites in the domain of term rewriting systems. With ELEVATE, we show that these well-known techniques, which are largely neglected in state-of-the-art domain-specific compilers, can be successfully used to achieve high-performance domain-specific compilation. Compared to compilers with scheduling APIs, with ELEVATE, programmers are not restricted to a set of predefined, built-in optimizations. Instead, they can define their own strategies freely in a composable way. We show how optimization strategies expressed using ELEVATE enable high-performance code generation for RI**SE** programs (RI**SE** is the spiritual successor to LIFT), and achieve competitive performance with Halide and TVM. Furthermore, we show that ELEVATE is flexible and not specialized for one specific target language by also providing a brief case study in which we use ELEVATE to optimize automatically differentiated $\widetilde{\mathbf{F}}$ [160] programs.

This chapter is primarily based on [67] and [68], and the results for comparing against Halide, discussed in Section 5.6.4, have been contributed by Thomas Koehler.

5.1 INTRODUCTION

The tremendous gains in performance and efficiency that computer hardware continues to make are a vital driving force for innovation in computing. This gain enables entire new areas of computing, such as deep learning, to deliver applications unthinkable even just a few years ago. Moore's law and Dennard scaling describe the exponential growth of transistor counts leading to improved performance, and the exponential growth in performance per watt leading to improved energy efficiency. Unfortunately, these laws are coming to an end, as observed in the 2017 ACM Turing Lecture by Hennessy and Patterson [76]. As a result, the performance and energy efficiency gains no longer come for free for software developers. Programs have to be optimized for an increasingly diverse set of hardware devices by exploiting many subtle details of the computer architecture, as, for example, discussed in Chapter 3. Therefore, performance portability has emerged as a crucial concern as software naturally outlives the faster cycle of hardware generations. The accelerating trend towards specialized hardware emphasizes this and has proven to offer extreme benefits for performance - if the specially optimized software exploits it.

The predominant imperative and low-level programming approaches such as C, CUDA, or OpenCL force programmers to intertwine the code describing the program's functional behavior with optimization decisions. This way of developing programs is – by design – non performance portable. As an alternative, higher-level domain-specific approaches have emerged that allow programmers to declaratively describe the functional behavior without committing to a specific implementation. Popular examples of this approach are virtually all machine learning systems such as TensorFlow [2, 3] or PyTorch [125]. For these approaches, the compilers and runtime systems are responsible for optimizing the computations expressed as data-flow graphs. Programmers have limited control over the optimization process. Instead, large teams of engineers at Google and Facebook provide fast implementations for the most common hardware platforms, for TensorFlow including Google's specialized TPU hardware. This labor-intensive support of new hardware devices is currently only sustainable for the biggest companies in the market – and even they struggle as highlighted by Barham and Isard [11], two of the original authors of TensorFlow.

TVM [29] and Halide [140, 141] are two high-performance, and domain-specific compilers used in machine learning and image processing. Both attempt to tackle the performance portability challenge by separating the program into two parts: schedules and algorithms. A *schedule* describes the optimizations to apply to an *algorithm* that defines the functional behavior of the computation. Schedules are implemented using a set of predefined ad-hoc APIs that expose a fixed set of optimization options. TVM's and Halide's authors describe these APIs as a scheduling *language*, but they lack many desirable properties of a programming language. Most crucially, programmers are not able to define their own abstractions. Even the composition of existing optimization primitives is, in some cases, unintuitive due to the lack of precise semantics, and both compilers have default and implicit behavior limiting experts' control. All of these reasons make writing schedules significantly harder than writing algorithms. Furthermore, for some desirable optimizations, it is not sufficient to change the schedule, but the algorithm itself has to be redefined, which violates the promise of separation between algorithm and schedule. To overcome the innovation obstacle of manually optimizing for specialized hardware, we will need to rethink how we separate, describe, and apply optimizations in a more principled way.

TOWARDS REWRITE-BASED OPTIMIZATION STRATEGIES Encoding program transformations as *rewrite rules* has been a long-established idea that emerged from the functional programming community. Bird and Moor [14] studied an algebraic programming approach where functional programs are rewritten by exploiting algebraic properties. The Glasgow Haskell Compiler allows the specification of rewrite rules for program optimizations [127]. More recently, LIFT [162] encodes optimization and implementation choices as rewrite rules for optimizing a high-level pattern-based data-parallel functional language using an automated stochastic search method applying the rewrites. Rewrite based approaches, such as LIFT, have the advantage of being easily extensible towards new application domains (such as stencils [70] as discussed in Chapter 4) and supporting new hardware features. For example, specialized vector instructions can be encoded as low-level patterns and are introduced by defining a new rewrite rule [166]. Unfortunately, these rewrite approaches are so far limited in their practicality to deliver the high level of performance required in many applications and achieved by current imperative approaches. They especially lack control over the rewriting process, and automated rewriting using stochastic search processes takes a long time to find a high-performance implementation. We aim to address these practical limitations of rewrite-based approaches by defining a strategy language that allows the definition of optimization strategies to precisely control the rewrite process achieving high-performance compilation.

In this chapter, we introduce ELEVATE, a functional language for describing optimization strategies as composable rewrite rules. ELEVATE is heavily inspired by research on strategy languages for term rewrite systems used in other contexts – and largely unknown to

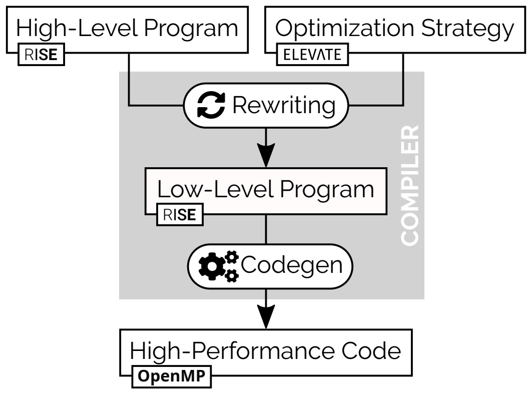


Figure 5.1: Overview of our compilation approach. We use use ELEVATE to optimize RISE programs: Computations are expressed as *High-Level Programs* written in the data-parallel language RISE. These programs are rewritten following the instructions of an *Optimization Strategy* expressed in the strategy language ELEVATE. From the rewritten *Low-Level Programs* that encode optimizations explicitly, *High-Performance Code* is generated.

the high-performance compilation community – such as Stratego by Visser [181]. ELEVATE is a flexible language based on functional programming techniques. It allows programmers to define their own abstractions for building optimization strategies while providing precise control about where and how optimizations are applied in the target program. As the primary target language in this chapter, we use RISE [8]. RISE provides well known functional data-parallel patterns for expressing computations at a high-level and is inspired by languages such as Lift [162, 167], Accelerate [23], and Futhark [77]. We provide a brief introduction to RISE in Section 5.3.1.

While the individual components of our approach are not necessarily novel, our overall design demonstrates a novel application of functional-programming techniques for achieving high-performance compilation. As we will see in our experimental results, our approach provides competitive performance compared to the imperative state-of-the-art while being built with and leveraging functional principles resulting in an elegant and composable design.

Figure 5.1 shows an overview of the compilation flow from a high-level program and an optimization strategy to high-performance code. In this case, the high-level target language is RISE that will be compiled to high-performance OpenMP code. In Section 5.2, we first motivate the need for a more principled way to separate, describe, and apply optimizations. RISE and its compilation to high-performance code are explained in Section 5.3 before focussing on ELEVATE (Section 5.4) and how high-performance optimization strategies are expressed in it (Section 5.6). To demonstrate that ELEVATE is applicable to more than just RISE programs, Section 5.7 presents a brief case study in which we use ELEVATE to optimize automatically differentiated $\widetilde{\mathbf{F}}$ programs.

5.2 MOTIVATION AND BACKGROUND

We motivate the need to define and apply optimizations in a more principled way by again taking a closer look at TVM, the current state-of-the-art in high-performance domain-specific compilation for machine learning. Specifically, we revisit the TVM matrix multiplication example that was already briefly discussed in Section 3.2 and extend the discussion by analyzing a high-performance TVM schedule. We then argue for achieving high-performance compilation by using well-known functional programming techniques instead. This approach allows us to express optimization strategies as programs in a functional programming language instead of using the fixed scheduling APIs.

5.2.1 *Scheduling Languages for High-Performance Code Generation*

Halide by Ragan-Kelley et al. [140] has introduced the concept of decoupling a program into two parts into the domain of high-performance code generation: the *algorithm*, describing the functional behavior, and the *schedule*, specifying how the underlying compiler should optimize the program. It has been initially designed to generate high-performance code for image processing pipelines [141]. It has since inspired similar approaches in other contexts such as TVM by Chen et al. [29] in deep learning.

Figure 5.2 shows two snippets of TVM code for generating matrix-matrix multiplication implementations. TVM is a DSL embedded in Python, so the syntax used here is Python. Listing 5.1 shows a simple version. The lines 2–5 in Listing 5.1 define the matrix-matrix multiplication computation: A and B are multiplied by performing the dot product for each coordinate pair (x, y). The dot product is expressed as pairwise multiplications and reducing over the reduction domain k using the `tvm.sum` operator (line 5). Line 7 in the listing instructs the compiler to use the default schedule, which generates code to compute the output sequentially in a row-major order.

Listing 5.2 in Figure 5.2 shows an optimized version of the same computation. The schedule in lines 10–23 specifies multiple program transformations, including tiling (line 12), vectorization (line 19), and loop unrolling (line 18) for optimizing the performance on multi-core CPUs. However, by carefully analyzing the optimized algorithm and schedule, we identify the following problems with the current schedule-based approach for achieving optimization in high-performance domain-specific compilation. These limitations (that all schedule-based compilers share) motivate five specific goals that we aim to achieve with the design of ELEVATE.

```
1  # Naive algorithm
2  k = tvm.reduce_axis((0, K), 'k')
3  A = tvm.placeholder((M, K), name='A')
4  B = tvm.placeholder((K, N), name='B')
5  C = tvm.compute((M, N),lambda x,y: tvm.sum(A[x,k]*B[k,y], axis=k),name='C')
6  # Default schedule
7  s = tvm.create_schedule(C.op)
```

Listing (5.1) Matrix matrix multiplication in TVM. Lines 2–5 define the computation $A \times B$, line 7 instructs the compiler to use the default schedule computing the output matrix sequentially in a row-major order.

```
1  # Optimized algorithm
2  bn = 32
3  k  = tvm.reduce_axis((0, K), 'k')
4  A  = tvm.placeholder((M, K), name='A')
5  B  = tvm.placeholder((K, N), name='B')
6  pB = tvm.compute((N / bn, K, bn), lambda x,y,z: B[y, x*bn+z], name='pB')
7  C  = tvm.compute((M,N), lambda x,y:tvm.sum(A[x,k] * pB[y//bn,k,
8        tvm.indexmod(y,bn)], axis=k),name='C')
9  # Parallel schedule
10 s = tvm.create_schedule(C.op)
11 CC = s.cache_write(C, 'global')
12 xo, yo, xi, yi = s[C].tile(C.op.axis[0], C.op.axis[1], bn, bn)
13 s[CC].compute_at(s[C], yo)
14 xc, yc = s[CC].op.axis
15 k, = s[CC].op.reduce_axis
16 ko, ki = s[CC].split(k, factor=4)
17 s[CC].reorder(ko, xc, ki, yc)
18 s[CC].unroll(ki)
19 s[CC].vectorize(yc)
20 s[C].parallel(xo)
21 x, y, z = s[pB].op.axis
22 s[pB].vectorize(z)
23 s[pB].parallel(x)
```

Listing (5.2) Optimized Matrix matrix multiplication in TVM. Lines 2–8 define an optimized version of the algorithm in Listing 5.1, the other lines define a schedule specifying the optimizations for targeting CPUs.

Figure 5.2: Two different versions of specifying and optimizing matrix matrix multiplication in TVM.
From: https://docs.tvm.ai/tutorials/optimize/opt_gemm.html

BLURRED SEPARATION OF CONCERNS In order to optimize the memory access pattern, the algorithm has to be changed. In this example, a copy of the `B` matrix (`pB`) is introduced in line 6 (and used in line 8), whose elements are reordered depending on the tile size. This optimization is not expressible with scheduling primitives and, therefore, requires the modification of the algorithm – clearly violating the promise of separating algorithm and schedule.

LIMITED REUSE OF SCHEDULES Even for optimizations that do not require to change the algorithm, the separation between algorithm and schedule is blurred because both share the same Python identifiers and must, therefore, live in the same scope. This restriction limits the reuse of schedules across algorithms.

SCHEDULE LANGUAGES ARE HARD TO EXTEND The parallel schedule uses eight built-in optimization primitives (`cache_write`, `tile`, `compute_at`, `split`, `reorder`, `unroll`, `vectorize`, `parallel`). Scheduling primitives provide high-level abstractions for common program transformations aiming to optimize performance. Some are specific for the targeted hardware (like `vectorize`), some are generally useful algorithmic optimizations for many applications (like `tiling` to increase data locality), and others are low-level optimizations (like `unroll` and `reorder` that transform loop nests). However, TVM's scheduling language is not easily extensible. Adding a new optimization primitive to the existing schedule API requires extending the underlying TVM compiler. The same is true for other schedule-based compilers, including Halide or Fireiron. Even a primitive like `tile`, which can be implemented as a specific composition of `split` and `reorder` [73], is instead provided as a built-in abstraction. Modern scheduling languages are not extensible with user-defined abstractions without extending the whole compiler.

REASONING ABOUT SCHEDULES IS DIFFICULT The behavior of some of the existing primitives is not intuitive, and the documentation provides only informal descriptions. For example, the documentation for `cache_write` is *"Create a cache write of original tensor, before storing into tensor"*. Reasoning about schedules is difficult due to the lack of clear descriptions of the provided optimization primitives.

IMPLICIT DEFAULT BEHAVIOR If no schedule is provided (as in Listing 5.1), the TVM compiler employs a set of implicit default optimizations out of reach for the user's control. This sometimes leads to the surprising behavior that algorithms without a schedule perform better (e.g., due to auto-vectorization) than ones where a schedule is provided.

```
// Matrix Matrix Multiplication in RISE
val dot = fun(as, fun(bs,
  zip(as)(bs) |> map(fun(ab, mult(fst(ab))(snd(ab)))) |> reduce(add)(0) ) )
val mm  = fun(a : M.K.float, fun(b : K.N.float,
  a |> map(fun(arow,                     // iterating over M
    transpose(b) |> map(fun(bcol,        // iterating over N
      dot(arow)(bcol) )))) ) )           // iterating over K
```

```
// Optimization Strategy in ELEVATE
val tiledMM = (tile(32,32) ‘@‘ outermost(mapNest(2)) ‘;‘ lowerToC)(mm)
```

Figure 5.3: Matrix matrix multiplication in RISE (top) and the tiling optimization strategy in ELEVATE (bottom).

5.2.2 *A Principled Way to Separate, Describe, and Apply Optimizations*

Out of the shortcomings of the scheduling API approach, we identify the following desirable features for a more principled way to separate, describe, and apply optimizations for high-performance code generation. Our approach aims to:

1. *Separate concerns*: Computations should be expressed at a high abstraction level only. They should not be changed to express optimizations;

2. *Facilitate reuse*: Optimization strategies should be defined clearly separated from the computational program facilitating reusability of computational programs and strategies;

3. *Enable composability*: Computations *and* strategies should be written as compositions of user-defined building blocks (possibly domain-specific ones); *both languages* should facilitate the creation of higher-level abstractions;

4. *Allow reasoning*: Computational patterns and especially strategies, should have a precise, well-defined semantics that allow reasoning about them;

5. *Be explicit*: Implicit default behavior should be avoided to empower users to be in control.

Fundamentally, we argue that a more principled high-performance compilation approach should consider computations and optimization-strategies equally important. Consequently, a strategy language should be built with the same standards as a language describing computation.

In this paper, we present such an approach by introducing the strategy language ELEVATE. Combining ELEVATE with a computational language like RISE or $\widetilde{\mathbf{F}}$ achieves precisely the desired goals, as we show in the following.

A MOTIVATIONAL EXAMPLE Figure 5.3 shows an example of a RISE program defining a matrix multiplication computation as a composition of well-known data-parallel functional patterns. Below is an ELEVATE strategy that defines one possible optimization by applying the well-known tiling optimization that improves memory usage by increasing spatial and temporal locality of the data. The optimization strategy is a sequential composition (`;`) of user-defined strategies that are themselves defined as compositions of simple rewrite rules giving the strategy a precise semantics. We do not employ implicit behavior and instead generate low-level code according to the optimization strategy specified.

In the remainder of the chapter, we describe how to define optimizations typically provided in high-performance scheduling languages as optimization strategies in ELEVATE defined as compositions of simple rewrite rules for data-parallel functional programs. We start by briefly introducing the computational language RISE and its compilation to parallel imperative code because we will use it as the primary target language in this chapter.

5.3 RISE: A LANGUAGE FOR DATA PARALLEL COMPUTATIONS

In this section, we briefly introduce the RISE [8] programming language (Section 5.3.1). We use RISE in the remainder of this chapter to demonstrate how programs are optimized with ELEVATE. We explain how low-level patterns represent hardware features (Section 5.3.2), and finally describe how imperative code is generated (Section 5.3.3).

5.3.1 *A Brief Introduction to RISE*

RISE is a functional programming language that uses data-parallel patterns to express computations over multi-dimensional arrays. RISE is a spiritual successor of LIFT, which was initially introduced by Steuwer et al. [162]. LIFT has demonstrated that functional, high-performance code generation is feasible for different domains, including dense linear algebra, sparse linear algebra, and stencil computations (see [167] and Chapter 4). RISE is implemented as an embedded DSL in Scala and generates parallel OpenMP code for CPUs and OpenCL for GPUs.

Figure 5.4 shows the abstract syntax of RISE expressions and types as well as the provided high-level and low-level primitives for expressing data-parallel computations. RISE provides the usual λ-calculus constructs of abstraction (written `fun(x, e)`), application (written with parenthesis), identifiers, and literals (underlined). The type system separates data types from function types to prevent functions from being stored in memory. RISE uses a restricted form of dependent function types for types that contain expressions of kind

RISE Syntax of Expressions and Types:

$$
\begin{aligned}
e &::= \texttt{fun}(x, e) \mid e(e) \mid x && \text{(Abstraction, Application, Identifier)} \\
&\mid \underline{l} \mid P && \text{(Literal, Primitives)} \\
\kappa &::= \mathrm{nat} \mid \mathrm{data} && \text{(Natural Number Kind, Datatype Kind)} \\
\tau &::= \delta \mid \tau \to \tau && \text{(Data Type, Function Type)} \\
&\mid (x : \kappa) \to \tau && \text{(Dependent Function Type)} \\
n &::= \underline{0} \mid n + n \mid n \cdot n \mid \ldots && \text{(Nat. Number Literals, Binary Operations)} \\
\delta &::= n.\delta \mid \delta \times \delta \mid \mathrm{idx}[n] && \text{(Array Type, Pair Type, Index Type)} \\
&\mid \texttt{float} \mid n\texttt{<float>} && \text{(Scalar Type, Vector Type)}
\end{aligned}
$$

High-Level Primitives:

$$
\begin{aligned}
\texttt{id} &: (\delta : \mathrm{data}) \to \delta \to \delta \\
\texttt{add} \mid \texttt{mult} \mid \ldots &: (\delta : \mathrm{data}) \to \delta \to \delta \to \delta \\
\texttt{fst} &: (\delta_1\ \delta_2 : \mathrm{data}) \to \delta_1 \times \delta_2 \to \delta_1 \\
\texttt{snd} &: (\delta_1\ \delta_2 : \mathrm{data}) \to \delta_1 \times \delta_2 \to \delta_2 \\
\texttt{map} &: (n : \mathrm{nat}) \to (\delta_1\ \delta_2 : \mathrm{data}) \to \\
&\quad (\delta_1 \to \delta_2) \to n.\delta_1 \to n.\delta_2 \\
\texttt{reduce} &: (n : \mathrm{nat}) \to (\delta : \mathrm{data}) \to \\
&\quad (\delta \to \delta \to \delta) \to \delta \to n.\delta \to \delta \\
\texttt{zip} &: (n : \mathrm{nat}) \to (\delta_1\ \delta_2 : \mathrm{data}) \to \\
&\quad n.\delta_1 \to n.\delta_2 \to n.(\delta_1 \times \delta_2) \\
\texttt{split} &: (n\ m : \mathrm{nat}) \to (\delta : \mathrm{data}) \to nm.\delta \to n.m.\delta \\
\texttt{join} &: (n\ m : \mathrm{nat}) \to (\delta : \mathrm{data}) \to n.m.\delta \to nm.\delta \\
\texttt{transpose} &: (n\ m : \mathrm{nat}) \to (\delta : \mathrm{data}) \to n.m.\delta \to m.n.\delta \\
\texttt{generate} &: (n : \mathrm{nat}) \to (\delta : \mathrm{data}) \to (\mathrm{idx}[n] \to \delta) \to n.\delta
\end{aligned}
$$

Low-Level Primitives:

$$
\begin{aligned}
\texttt{map}\{\texttt{Seq} \mid \texttt{SeqUnroll} \mid \texttt{Par}\} &: (n : \mathrm{nat}) \to (\delta_1\ \delta_2 : \mathrm{data}) \to \\
&\quad (\delta_1 \to \delta_2) \to n.\delta_1 \to n.\delta_2 \\
\texttt{reduce}\{\texttt{Seq} \mid \texttt{SeqUnroll}\} &: (n : \mathrm{nat}) \to (\delta_1\ \delta_2 : \mathrm{data}) \to \\
&\quad (\delta_1 \to \delta_2 \to \delta_1) \to \delta_1 \to n.\delta_2 \to \delta_1 \\
\texttt{toMem} &: (\delta_1\ \delta_2 : \mathrm{data}) \to \delta_1 \to (\delta_1 \to \delta_2) \to \delta_2 \\
\texttt{mapVec} &: (n : \mathrm{nat}) \to (\delta_1\ \delta_2 : \mathrm{data}) \to \\
&\quad (\delta_1 \to \delta_2) \to n\texttt{<}\delta_1\texttt{>} \to n\texttt{<}\delta_2\texttt{>} \\
\texttt{asVector} &: (n\ m : \mathrm{nat}) \to (\delta : \mathrm{data}) \to \\
&\quad nm.\delta \to n.m\texttt{<}\delta\texttt{>} \\
\texttt{asScalar} &: (n\ m : \mathrm{nat}) \to (\delta : \mathrm{data}) \to \\
&\quad n.m\texttt{<}\delta\texttt{>} \to nm.\delta
\end{aligned}
$$

Figure 5.4: The syntax of expressions and types of RISE as well as high- and low-level primitives.

nat representing natural numbers or *data* for type-level variables ranging over data types. Natural numbers are used to represent the length of arrays in the type and might consist of arithmetic formulae with binary operations such as addition and multiplication. Data types are array types, pair types, index types representing array indices up to n, scalar types, or vector types that correspond directly to the SIMD vector types of the underlying hardware. Precise typing rules for such a type system are given by Atkey et al. [8].

In this chapter, we use the following syntactic sugar: we write reverse function application in the style of F-sharp as `e |> f` (equivalent to `f(e)`); function composition is written as `g << f` and in the reverse form as `f >> g`, both meaning `f` is applied before `g`; we may write `+` and `*` as inline binary operators instead of calling the equivalent functions `add` and `mult`.

RISE defines a set of high-level primitives that are used to describe computations over multi-dimensional arrays. These primitives are well-known in the functional programming community: **`id`**, **`fst`**, **`snd`**, and the binary functions **`add`** and **`mult`** have their obvious meaning. **`map`** and **`reduce`** are the well-known functions operating on arrays and allowing for easy parallelization. **`zip`**, **`split`**, **`join`**, and **`transpose`** shape multi-dimensional array data in various ways. RISE also provides **`pad`** and **`slide`** (see Chapter 4), which we omit here for brevity because we are not targeting stencil computations in this chapter. Finally, **`generate`** creates an array based on a generating function. Since RISE does not support general recursion, every RISE program terminates.

5.3.2 *A Functional Representation of Hardware Features*

More interesting are the low-level primitives that RISE offers to indicate how to exploit the underlying hardware. The different variations of the **`map`** pattern indicate if the given function is applied to the array using a sequential loop, by unrolling this loop, or using a parallel loop where each iteration might be performed in parallel.

Similarly, the **`reduce`** variations indicate if the reduction loop should be unrolled or not. A parallel reduction is not provided as a given building block, but must be expressed using the other low-level primitives such as the **`mapPar`**.

The expression `toMem(a)(fun(x, b))` indicates that the value `a` will be stored in memory and that the stored value can be accessed in the expression `b` with the name `x`. The remaining three low-level patterns **`mapVec`**, **`asVector`**, and **`asScalar`** enable the use of SIMD-style vectorization. The low-level primitives presented here are OpenMP-specific and aimed at parallelization for CPUs, a similar set of low-level primitives exists for targeting the OpenCL programming language for GPUs.

5.3.3 *Strategy Preserving Code Generation from RISE*

The compilation of RISE programs is slightly unusual. A high-level program is rewritten using a set of rewrite-rules into the low-level patterns. This process was initially proposed by Steuwer et al. [162] in Lift. From the low-level representation, imperative parallel code is generated. All optimization decisions, such as how to parallelize the `reduce` primitive, must be made in the rewriting stage before the code generation. The code generation process is deterministic and only translates the annotated implementation strategy into the target imperative language such as OpenMP or OpenCL. Atkey et al. [8] describe a compilation process that is guaranteed to be *strategy preserving*; that is, no implicit implementation decisions are performed by the compiler. Instead, it respects the implementation, and optimization decisions are explicitly encoded in the low-level RISE program, which compiles straight into high-performance code.

Lift promises the rewriting process to be fully automatic using a stochastical search method. However, there are many cases where this is either impractical: Either because the rewriting process takes too long, or because expert programmers want precise control over the optimizations applied to a particular program targeting a particular hardware device. Therefore, we introduce a language that allows a programmer to specify optimization strategies as compositions of rewrite rules.

5.4 ELEVATE: A LANGUAGE FOR OPTIMIZATION STRATEGIES

In this section, we introduce ELEVATE, a functional language for describing optimization strategies. ELEVATE is heavily inspired by earlier works on strategy languages for term rewriting systems, e.g., Stratego [184]. Kirchner [87] provides a recent overview of the rewriting community's research. We define the notion of a *Strategy* and make use of previously proposed combinators and traversal operators while extending this set slightly. Our key contribution is not the design of ELEVATE itself but rather its application to formally define optimizations required in high-performance compilation.

5.4.1 *Language Features and Types*

ELEVATE is a functional language with a standard feature set, including function recursion, algebraic data types, and pattern matching. Besides the standard scalar data types such as `int`, types of interests are function types and pair types. Our current implementation is an embedded DSL in Scala, and we use Scala-like notation for ELEVATE strategies in the chapter.

5.4.2 *Strategies*

A *strategy* is the fundamental building block of ELEVATE. Strategies encode program transformations and are modeled as functions with the following type:

```
type Strategy[P] = P => RewriteResult[P]
```

Here, `P` is the type of the rewritten program. `P` could, for example, be `Rise` for RISE programs, or `FSmooth` for $\widetilde{\mathbf{F}}$ programs. A `RewriteResult` is an applicative error monad encoding the success or failure of applying a strategy to a program:

```
RewriteResult[P] = Success[P](p: P)
                 | Failure[P](s: Strategy[P])
```

In case of a successful application, `Success` contains the transformed program, in case of a failure, `Failure` contains the unsuccessful strategy. Carrying the failing strategy along in the `Failure` case is beneficial in cases where a stateful strategy is used.

The simplest example of a strategy is the `id` strategy. It always succeeds and returns its input program `p` unchanged:

```
def id[P]: Strategy[P] = (p: P) => Success(p)
```

The `fail` strategy does the opposite and always fails while recording that it was the failing strategy:

```
def fail[P]: Strategy[P] = (p: P) => Failure(fail)
```

5.4.3 *Rewrite Rules as Strategies*

In ELEVATE, we do not differentiate between rewrite rules and strategies; hence, rewrite rules are also strategies, i.e., functions satisfying the same type given above. Let us suppose we want to apply some well-known rewrite rules such as the fusion of two `map` calls:

```
map(f) << map(g) ⇝ map(f << g)
```

In RISE, the left-hand side of the rule is expressed as:

```
val p: Rise = fun(xs, map(f)(map(g)(xs)))
```

The RISE AST representation of the `fun`-body is shown in Figure 5.5 on the left. Here, function applications appear explicit as `app` nodes. The fusion rule that rewrites this AST representation is implemented in ELEVATE as follows:

```
def mapFusion: Strategy[Rise] = p => p match {
  case app(app(map, f), app(app(map, g), xs)) =>
    Success( map(fun(x, f(g(x))))(xs) )
  case _ => Failure(mapFusion) }
```

Note that we are mixing RISE (i.e., `map(f)`) and ELEVATE expressions. We write `app(f, x)` to pattern match the function application that is

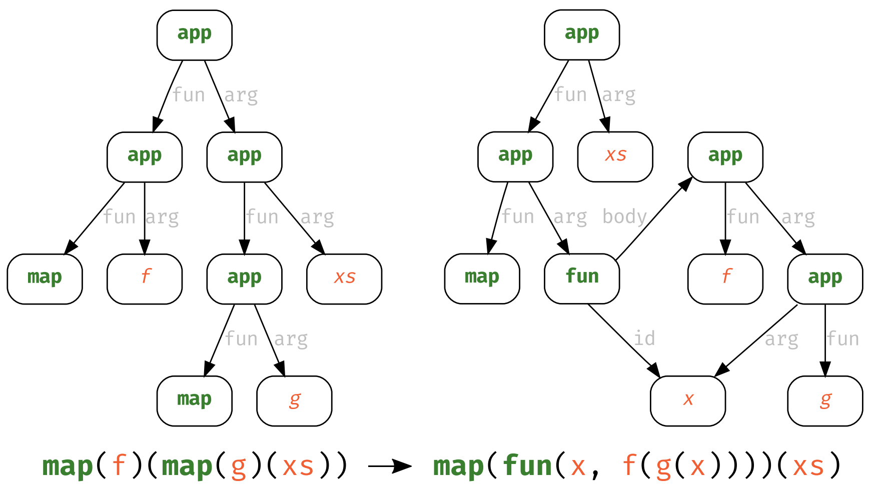


Figure 5.5: RISE's *map-fusion* rule as an AST transformation.

written as `f(x)` in RISE. The expression nested inside `Success` is the rewritten expression shown in Figure 5.5 on the right.

Figure 5.6 shows the RISE rewrite rules that are used as basic building blocks in this chapter. Each of these rules is implemented as an ELEVATE strategy similar to the map-fusion example. We use these rules as building blocks for expressing more complex optimizations such as tiling as compositions in ELEVATE.

5.4.4 *Strategy Combinators*

An idea that ELEVATE inherits from Stratego [182] is to describe strategies as compositions – one of the key aims that we set out for our approach. Therefore, we introduce strategy combinators.

The `seq` combinator is given two strategies `fs` and `ss` and applies the first strategy to the input program `p`. Afterward, the second strategy is applied to the result.

```
def seq[P]: Strategy[P] => Strategy[P] => Strategy[P] =
    fs => ss => p => fs(p) »= (q => ss(q))
```

The `seq` strategy is only successful when both strategies are successfully applied in succession; otherwise, `seq` fails. In the implementation of our combinators, we make use of the monadic interface of `RewriteResult` and use the standard Haskell infix operators `»=` for monadic `bind`, `<|>` for `mplus`, and `<$>` for `fmap`. This way, the definition of the `seq` combinator is read as follows:

1. Apply the first strategy (`fs(p)`), which yields a `RewriteResult`.

2. In case the `RewriteResult` is a `Success(x)`, bind the rewritten program `x` to the name `q` and apply the second strategy (`»= (q =>ss(q))`). Otherwise, return `Failure`.

$$\epsilon \rightsquigarrow \texttt{id} \qquad (\texttt{addId})$$

$$(\texttt{id} : m.n.\delta \to m.n.\delta) \rightsquigarrow \texttt{transpose >> transpose} \qquad (\texttt{idToTranspose})$$

$$\texttt{transpose >> map(map(f))} \rightsquigarrow \texttt{map(map(f)) >> transpose} \qquad (\texttt{transposeMove})$$

$$\texttt{map(f)} \rightsquigarrow \texttt{split(n) >> map(map(f)) >> join} \qquad (\texttt{splitJoin})$$

$$\texttt{map(f >> g)} \overset{\rightsquigarrow}{\leftsquigarrow} \texttt{map(f) >> map(g)} \qquad (\texttt{mapFission/mapFusion})$$

$$\texttt{map(f) >>reduce(fun((acc,y),op(acc)(y)))(init)} \overset{\rightsquigarrow}{\leftsquigarrow} \texttt{reduce(fun((acc,y),op(acc)(f(y))))(init)} \qquad (\texttt{fuseReduceMap/fissionReduceMap})$$

Figure 5.6: All RISE-specific rewrite rules for high-level expressions that are used for composing more complex optimizations in this chapter.

The `lChoice` combinator is given two strategies and applies the second one only if the first failed.

```
def lChoice[P]:Strategy[P] => Strategy[P] => Strategy[P] =
    fs => ss => p => fs(p) <|> ss(p)
```

We make heavy use of both combinators in the following and therefore use `<+` as short-form notation for `lChoice` and ‘`;`‘ for `seq`. Additionally, we define two more combinators inhereted from Stratego: The `try` combinator applies a strategy and, in case of a failure, applies the identity strategy. Therefore, `try` never fails.

```
def try[P]: Strategy[P] => Strategy[P] =
    s => p => (s <+ id)(p)
```

Finally, the `repeat` combinator applies the given strategy until it is no longer applicable.

```
def repeat[P]: Strategy[P] => Strategy[P] =
    s => p => try(s ‘;‘ repeat(s) )(p)
```

5.4.5 *Traversals as Strategy Transformers*

The `mapFusion` strategy we saw in the previous subsection is implemented as a function in ELEVATE. Therefore, its `match` statement will try to pattern match its argument – the entire program. This means that a strategy on its own is hard to reuse in different circumstances because it will always be applied at the root of the AST representation.

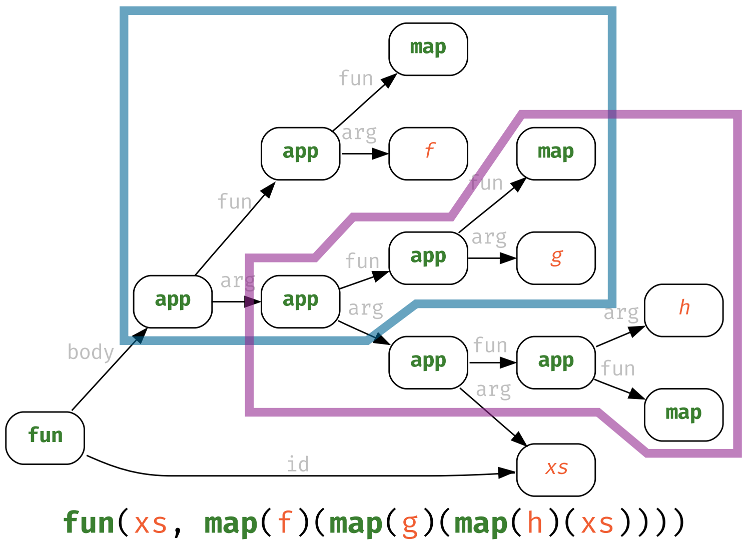


Figure 5.7: Two possible locations for applying the *map-fusion* rule within the same RISE program.

In addition, a strategy is often applicable at multiple places within the same program or only applicable at a specific location. For example, the `mapFusion` strategy is applicable twice in the following RISE program:

```
val threemaps = fun(xs, map(f)(map(g)(map(h)(xs))))
```

We may fuse the first or last two `map`s, as shown in Figure 5.7.

In ELEVATE, we use *traversals* to describe at which exact location a strategy is applied. Luttik and Visser [98] proposed three basic traversals encoded as strategy transformers:

```
type Traversal[P] = Strategy[P] => Strategy[P]
def all[P]:  Traversal[P]
def one[P]:  Traversal[P]
def some[P]: Traversal[P]
```

`all` applies a given strategy to all sub-expressions of the current expression and fails if it does not apply to all sub-expressions. `one` applies a given strategy to exactly one sub-expression and fails if the strategy is not applicable to any sub-expression. `some` applies a given strategy to at least one sub-expression but potentially more if possible. `one` and `some` are allowed to choose sub-expressions non-deterministically.

In ELEVATE, we see these three basic traversals as a type class: an interface that has to be implemented for each program type `P`. The implementation for RISE is straightforward. RISE programs are represented by ASTs such as the one in Figure 5.7. Therefore, `all`, `one`, and `some` correspond to the obvious implementations on the tree-based representation.

DESCRIBING LOCATIONS USING TRAVERSALS By default, every strategy is applied at the root of the AST. To apply a strategy further down in the AST, we can wrap it inside one of the basic traversal strategies. For example, to fuse the first two `maps` in Figure 5.7, we may use the `one` traversal:

```
one(mapFusion)(threemaps)
```

Using `one` will apply the `mapFusion` strategy not at the root of the AST, but instead one level down, first trying to apply the strategy (unsuccessfully) to the function parameter; then (successfully) to the function body, as highlighted in the upper-right blue box.

To fuse the last two `maps`, we use the `one` traversal twice to apply `mapFusion` two levels down in the AST:

```
one(one(mapFusion))(threemaps)
```

This successfully applies the fusion strategy to the expression highlighted in the lower-right purple box shown in Figure 5.7.

5.4.6 *RISE-Specific Traversal Strategies*

The traversals we have discussed so far are generic, meaning they can be applied to any target language. Therefore, these traversals are flexible but offer only limited control For `one` and `some`, the selection of sub-expressions is either non-deterministic or implementation-dependent (as for RISE). Especially in the context of program optimization it rarely makes sense to apply a strategy to `all` sub-expressions. Instead, we often want to apply an optimization at a specific position in the AST (e.g., to tile two specific loops). Since ASTs of real-world programs are significantly larger than the one shown in Figure 5.7, simply nesting the `all`, `one` and `some` traversals does not offer the required precision to describe such locations.

In ELEVATE, one can easily specify language-specific traversals. As we have seen in the previous section, RISE is a functional language using λ-calculus as its representation. Therefore, it makes sense to introduce traversals that navigate the two core concepts of λ-calculus: **fun**ction abstraction and **app**lication.

To apply a strategy to the body of a function abstraction, we define the following traversal:

```
def body: Traversal[Rise] = s => p => p match {
   case fun(x,b) => (nb => fun(x,nb)) <$> s(b)
   case _ => Failure(body(s)) }

def <$>[P]: (P => P) => RewriteResult[P] => RewriteResult[P] =
  f => r => r match {
     case Success(x) => Success(f(x))
     case Failure(y) => Failure(y) }
```

A strategy `s` is applied to the function body (`s(b)`), and if successful, the transformed body is bound to the name `nb` around which we construct a new function.

Similarly, we define traversals `function` and `argument` for function applications:

```
def function: Traversal[Rise] = s => p => p match {
   case app(f,a) => (nf => app(nf, a)) <$> s(f)
   case _ => Failure(function(s)) }

def argument: Traversal[Rise] = s => p => p match {
   case app(f,a) => (na => app(f, na)) <$> s(a)
   case _ => Failure(argument(s)) }
```

For the RISE program shown in Figure 5.7, we can precisely describe a traversal path in the AST. To fuse the first two `maps` we may write `body(mapFusion)(threemaps)`, and to fuse the others, we write `body(argument(mapFusion))(threemaps)`. Both versions describe the precise path from the root to the sub-expression at which the fusion rule is applicable.

5.4.7 *Complete Expression Traversal Strategies*

All of the traversal primitives introduced so far apply their given strategies only to immediate sub-expressions. Therefore, they are sometimes called one-level traversals. Using strategy combinators and traversals, we can define recursive strategies which traverse entire expressions:

```
def topDown[P]: Traversal[P] =
        s => p => (s <+ one(topDown(s)))(p)

def bottomUp[P]: Traversal[P] =
        s => p => (one(bottomUp(s)) <+ s)(p)

def allTopDown[P]: Traversal[P] =
        s => p => (s ‘;‘ all(allTopDown(s)))(p)

def allBottomUp[P]: Traversal[P] =
        s => p => (all(allBottomUp(s)) ‘;‘ s)(p)

def tryAll[P]: Traversal[P] =
        s => p => (all(tryAll(try(s))) ‘;‘ try(s))(p)
```

`topDown` and `bottomUp` are useful strategies traversing an expression either from the top or bottom, trying to apply a strategy at every sub-expression and stopping at the first successful application. If the strategy is not applicable at any sub-expression, `topDown` and `bottomUp` fail. `allTopDown` and `allBottomUp` do not use `lChoice`, insisting on applying the given strategy to every sub-expression. The `tryAll` strategy is often more useful as it wraps its given strategy in a `try` and thus never fails but applies it wherever possible. Also, note that the `tryAll` strategy traverses the AST bottom-up instead of top-down. Visser [182] has also proposed these traversals, and we use them here with slightly different names more fitting for our use case of achieving high-performance domain-specific compilation.

5.4.8 *Normalization*

When implementing rewrite rules, such as the `mapFusion` rule as strategies, the match statement expects the input expression to be in a particular syntactic form. For a functional language like RISE, we might, for example, expect that expressions are fully β-reduced. To ensure that expressions satisfy a *normal-form* we define:

```
def normalize[P]: Strategy[P] => Strategy[P] =
  s => p => repeat(topDown(s))(p)
```

The `normalize` strategy repeatedly applies a given strategy at every possible sub-expression until it can not be applied anymore. Therefore, after `normalize` successfully finishes, it is impossible to apply the given strategy to any sub-expression anymore.

BETA-ETA-NORMAL-FORM λ-calculus (and RISE) allows for semantically equivalent but syntactically different expressions. For example, `fun(x => f(x))` is equivalent to `f` *iff* `x` does not appear free in `f`. Transforming between these representations is called η-*reduction* and η-*abstraction*, which we also implement as ELEVATE strategies:

```
def etaReduction: Strategy[Rise] = p => p match {
  case fun(x1, app(f, x2))
    if x1 == x2 && not(contains(x1))(f) => Success(f)
  case _ => Failure(etaReduction)}

def etaAbstraction: Strategy[Rise] = p => p match {
  case f if hasFunctionType(f) => Success(fun(x, f(x)))
  case _ => Failure(etaAbstraction) }
```

Note that we can use two ELEVATE strategies, `not` and `contains`, in the pattern guard of the `etaReduction` strategy:

```
def not: Strategy[P] => Strategy[P] = s => p => s(p) match {
  case Success(_) => Failure(not(s))
  case Failure(_) => Success(p) }

def contains[P]: P => Strategy[P] = r => p =>
  topDown(isEqualTo(r))(p)

def isEqualTo[P]: P => Strategy[P] = r => p =>
  if(p == r) Success(p) else Failure(isEqualTo(r))
```

The `RewriteResult` obtained by applying `not(contains(x1))` to `f` is implicitly cast to a `Boolean` eventually. The `contains` strategy traverses `f` from top to bottom and checks if it finds `x1` using the `isEqualTo` strategy.

```
def DFNF = BENF `;` // (1) normalize using beta-eta-normal-form
  // (2) ensure that the argument of a map is a function abstraction
  normalize(argOf(map, not(isFun) `;` etaAbstraction))`;`
  // ...similar normalization for reduce primitive (left out for brevity)
  // (3) ensure every map is provided two arguments
  // ... (and every reduce is provided three arguments)
  normalize(
    // (3.1) if there is a map in 2 hops (or a reduce in three hops) ...
    one(function(isMap) <+ one(function(isReduce))) `;`
    // (3.2) ... and the current node is not an apply ...
    not(isApp) `;`
    // (3.3) ... we need to eta-abstract
    one((function(isMap) <+ one(function(isReduce))) `;` etaAbstraction)
```

Listing 5.3: Definition of the Data-Flow-Normal-Form (DFNF).

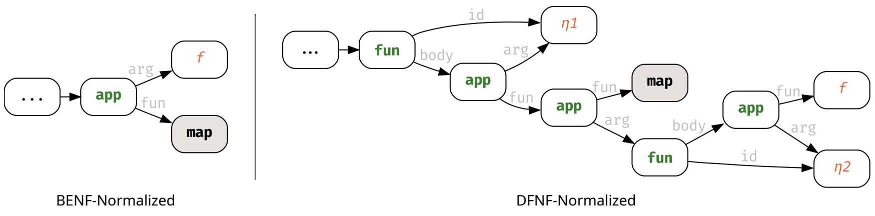


Figure 5.8: Two semantically equivalent but syntactically different versions of the RISE expression `map(f)`. The left version is beta-eta normalized using BENF, and the right version is in data-flow-normal-form (DFNF).

The simplest normal-form we often use in the following is the βη-*normal-form* (BENF) which exhaustively applies β- and η-reduction: `def BENF = normalize(betaReduction <+ etaReduction)`. Since not every function abstraction is η-reducible, the function arguments of RISE's higher-order primitives `map` and `reduce` might have different syntactic forms. Varying syntactic forms complicate the development of rewrite rules because a rule is always defined to match one particular syntactic structure. To simplify the application of strategies and the development of new rules, we make heavy use of an additional normal-form, which unifies the syntactic structure of function arguments to higher-order primitives.

DATA-FLOW-NORMAL-FORM One important normal-form for RISE programs is the Data-Flow-Normal-Form (DFNF) because it ensures a specific syntactic structure that we can rely on during rewriting. Listing 5.3 shows the definition of DFNF, which makes the data flow in a RISE program explicit in two ways: First, by ensuring a function abstraction is present in every higher-order primitive, and second, by ensuring every higher-order primitive is fully applied. For example, DFNF ensures that a `map` is always provided two arguments: a function and an input array, even if it could be β-reduced.

Figure 5.8 shows the result of applying the `DFNF` to the RISE expression `map(f)`. First, the input expression is normalized using `BENF` (line 1). Applying `BENF` generally decreases the size of the AST.

Second, we unify the syntactic form of the function arguments of higher-order primitives `map` and `reduce`. Listing 5.3 shows this unification for the `map` primitive in line 3; the definition for the `reduce` primitive is similar. The `argOf` traversal is similar to `argument` we already introduced; however, it only traverses to the argument of the function application if the applied function matches the given input (`map` in this case). Essentially, whenever the function argument of a `map` primitive is not already a function abstraction (as on the left side of Figure 5.8), we eta-abstract it. We describe the desired form in a declarative way, using the `not` and `isFun` predicates.

Third, we ensure that higher-order primitives are fully applied. For example, the `map` primitive on the left side of Figure 5.8 is not applied to an array argument. By applying `DFNF`, the array argument η1 is added by eta-abstracting again, as shown on the right hand side of the figure. This normalization is also naturally expressed using traversals and predicates: Whenever, we can reach a `map` node in two hops (`one(function(isMap))`), and the current node is not already an `app`-node (`not(isApp)`), we know that the `map` primitive is not applied to an array argument, and thus, we apply eta-abstraction.

Even though the size of the AST potentially increases significantly by applying `DFNF` instead of only `BENF`, we now have a unified syntactic structure. This structure immensely simplifies the traversal and implementation of more complex optimization strategies, as we will see in the following section.

CONFLUENCE AND TERMINATION Confluence (multiple non-deterministic rewrite paths eventually produce the same result) and termination are desirable properties for normal-forms in term rewriting systems. In ELEVATE, confluence only becomes a factor when the implementations of `one` and `some` are non-deterministic. For RISE, we are not interested in having multiple non-deterministic rewrite paths but instead need precise control over where, when, and in which order specific rules are applied. Therefore, we avoid non-determinism and do not need to worry about confluence.

Termination of normal-forms, and ELEVATE programs in general, must be evaluated on a case by case basis as it critically depends on the chosen set of strategies. For example, it is trivial to build a non-terminating normal-form using the `id` strategy that is always applicable. We currently do not prevent creating non-terminating strategies similarly to how almost all general-purpose languages do not prevent writing non-terminating programs. In the future, we are interested in introducing a more powerful type system for ELEVATE to better assist the user in writing well-behaved strategies.

5.5 EXPRESSING OPTIMIZATIONS AS REWRITE STRATEGIES

In the domain of deep learning, high-performance optimizations are particularly important. While Visser, Benaissa, and Tolmach [184] showed that strategy languages can be used to build program optimizers, the optimizations implemented as strategies were not targeted towards high-performance code generation but rather to optimize a functional ML-like language. To the best of our knowledge, this work is the first to describe a holistic functional approach for high-performance compilation that implements optimizations for achieving high performance as rewrite strategies and can compete with state-of-the-art imperative solutions.

In this section, we explore how ELEVATE is used to encode high-performance optimizations by leveraging its ability to define custom abstractions. We use TVM [29] as a comparison for a state-of-the-art imperative optimizing deep learning compiler with a scheduling API implemented in Python. TVM allows programmers to express computations using a DSL (embedded in Python) and control the application for optimizations using a separate scheduling API. In our case, we use RI**SE** as the language to express computations and develop separate strategies in ELEVATE, implementing the optimizations equivalent to those available in TVM's scheduling API.

We start by expressing basic scheduling primitives such as **`parallel`** and **`vectorize`** in ELEVATE. We then explore the implementation of more complex scheduling primitives such as **`tile`** by composition in ELEVATE, whereas it is a built-in optimization in TVM. Following our functional approach, we express sophisticated optimization strategies as compositions of a small set of general rewrite rules resulting in a more principled and even more powerful design. Specifically, the tiling optimization strategy in ELEVATE can tile arbitrary many dimensions instead of only two, while being composed of only five RI**SE**-specific rewrite rules.

5.5.1 *Basic Scheduling Primitives as ELEVATE Strategies*

TVM's basic scheduling primitives **`parallel`**, **`split`**, **`vectorize`**, and **`unroll`** specify loop transformations targeting a single loop. These are implemented as simple rewrite rules for RI**SE**.

PARALLEL The **`parallel`** scheduling primitive indicates that a particular loop shall be computed in parallel. In RI**SE**, this is indicated by transforming a high-level **`map`** into its low-level **`mapPar`** version. We express this rewrite rule as a simple ELEVATE strategy:

```
def parallel: Strategy[Rise] = p => p match {
  case map => Success( mapPar )
  case _   => Failure( parallel ) }
```

A rewrite rule that translates `map` into the sequential variant `mapSeq` is defined in the same way:

```
def sequential: Strategy[Rise] = p => p match {
  case map => Success( mapSeq )
  case _   => Failure( sequential) }
```

SPLIT TVM's `split` scheduling primitive implements loop-blocking (also known as strip-mining). In RISE, this is achieved by transforming the computation over an array expressed by `map(f)`: first the input is split into a two-dimensional array using `split(n)`, then `f` is `map`ped twice to apply the function to all elements of the now nested array, and finally, the resulting array is flattened into the original one-dimensional form using `join`.

```
def split(n: Int): Strategy[Rise] = p => p match {
  case app(map, f) =>
    Success( split(n) >> map(map(f)) >> join )
  case app(app(reduce, op), init) =>
    Success( split(n) >> reduce(
      fun(a, fun(y, op(a, reduce(op)(init)(y)) )))(init) )
  case _ => Failure( split(n) ) }
```

It is important to note that RISE does not materialize the intermediate two-dimensional array in memory. Instead, we only use this representation inside the compiler for code generation. In TVM, the `split` scheduling primitive can also be used to block reduction loops for which we use the `reduce` primitive in RISE. To make the `split` strategy applicable to both `map` and `reduce` primitives, we add a second case to the strategy that blocks a single `reduce` into two nested reductions.

Note that there is now a difference between the ELEVATE strategy `split` and the RISE primitive `split`. This ambiguity arises because we aim to build one-to-one strategy correspondences to TVM's scheduling primitives. The use of whether the strategy or the primitive is used is distinguished by the syntax highlighting and also clear from the context.

VECTORIZE The `vectorize` scheduling primitive indicates that a loop shall be computed in a SIMD-fashion. Its equivalent ELEVATE `vectorize` strategy implementation is similar to the `split` strategy:

```
def vectorize(n: Int): Strategy[Rise] = p => p match {
  case app(map, f) if isScalarFun(f) =>
    Success(asVector(n) >> map(mapVec(f)) >> asScalar)
  case _ => Failure( vectorize(n) ) }
```

First, it splits the input of scalars into an array of vectors using the `asVector` primitive. Then, `f` is mapped twice and transformed to perform vectorized computations using `map(mapVec(f))` and finally, the resulting array of vectors is transformed to an array of scalars.

Vectorization is most efficient when applied to the innermost loop of a loop-nest. In RISE, this corresponds to applying the `vectorize` strategy to the innermost `map` of potentially nested `map`s. Applying a strategy to the innermost `map` of nested `map`s is achieved in ELEVATE by traversing the expression beginning from the bottom of the AST (for example using `bottomUp(vectorize)`). The additional constraint `isScalarFun(f)` ensures that only functions operating on scalars are vectorized by inspecting `f`'s type. The restriction to scalar functions for `vectorize` is a current limitation of RISE.

UNROLL The `unroll` strategy rewrites the high-level `map` and `reduce` primitives into RISE low-level primitives that will be unrolled by the RISE compiler during code generation.

```
def unroll: Strategy[Rise] = p => p match {
  case map    => Success( mapSeqUnroll )
  case reduce => Success( reduceSeqUnroll )
  case _      => Failure( unroll ) }
```

5.5.2 *Multi-dimensional Tiling as an ELEVATE Strategy*

Tiling is a crucial optimization improving the cache hit rate by exploiting locality within a small neighborhood of elements. TVM's `tile` is a more complicated scheduling primitive to implement because it is essentially a combination of two traditional loop transformations: loop-blocking and loop-interchange. In fact, `tile` in TVM is a built-in combination of `split` for loop-blocking and `reorder` for loop-interchange. We already saw how to implement `split` using ELEVATE. We will now discuss how to implement a `tile` strategy using a combination of rules, normal-forms, and domain-specific traversals. We construct a generalized strategy out of a few simple building blocks that can tile an arbitrary number of dimensions, whereas TVM only implements 2D tiling.

Five RISE-specific rules are required for expressing our multi-dimensional tiling strategy: `idToTranspose`, `transposeMove`, `splitJoin`, `addId`, and `mapFission` (all shown in Figure 5.6). We implement these rules as basic ELEVATE strategies, as shown in the previous sections. In addition, we require three standard λ-calculus-specific transformations: η- and β-reduction, and η-abstraction.

MULTI-DIMENSIONAL RECURSIVE TILING Our tiling strategy expects a list of tile sizes, one per tiled dimension:

```
def tileND: List[Int] => Strategy[Rise]
```

The two-dimensional tiling, which is equivalent to TVM's built-in `tile` scheduling primitives, is expressed as `tileND(List(x,y))(mm)`. For this two-dimensional case, we also write `tile(x,y)(mm)`.

```
def tileND(n: List[Int]): Strategy[Rise] = DFNF ‘;‘ (n.size match {
    case 1 =>  function(split(n.head))          // loop-blocking
    case i =>  fmap(tileND(d-1)(n.tail)) ‘;‘   // recurse
               function(split(n.head))   ‘;‘   // loop-blocking
               interchange(i)     })           // loop-reorder
```

Listing 5.4: ELEVATE strategy implementing tiling recursively for arbitrary dimensions.

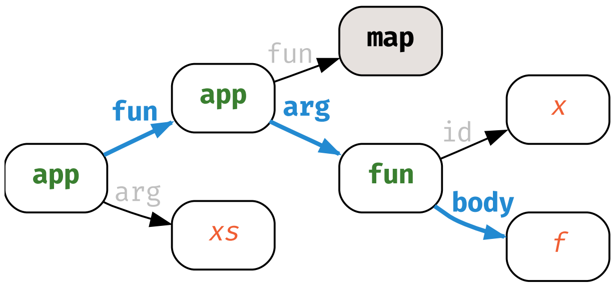


Figure 5.9: Visualization of the `fmap` traversal which traverses to the function argument of a RISE-`map` primitive.

Listing 5.4 shows the ELEVATE implementation of the tiling optimization. The intuition for our `tileND` implementation is simple: First, we ensure that the required rules are applicable to the input expression by normalizing the expression using the `DFNF` normal-form. Then, we apply the previously introduced `split` strategy to every `map` to be blocked, recursively going from innermost to outermost. Finally, we interchange dimensions accordingly.

We start to explain how we recursively traverse (using `fmap`) to apply loop-blocking and then discuss how we interchange dimensions in RISE (`interchange`).

RECURSIVELY APPLYING LOOP-BLOCKING In order to recursively apply the loop blocking strategy to nested `map`s, we make use of the RISE-specific traversal `fmap`:

```
def fmap: Traversal[Rise] = s => function(argOf(map, body(s)))
```

Figure 5.9 visualizes the traversal of the `fmap` strategy, the traversed path is highlighted in blue. `fmap` traverses to the function argument of a `map` primitive and applies the given strategy `s`. Note that the strategy requires the function argument of a `map` primitive to be a function abstraction. This syntactic structure can be assumed because we normalized the expression using `DFNF`. The `fmap` strategy is useful because it can be nested to "push" the application of the given strategy inside of a `map`-nest. For example,

```
fmap(fmap(split(n)))(DFNF(map(map(map(f)))))
```

skips two `map`s and applies loop-blocking to the innermost `map`. In Listing 5.4 line 3, we use `fmap` to recursively call `tileND` applying loop-blocking first to the inner `map`s before to the outer `map`:

```
// apply loop-blocking to inner map
fmap(tileND(d-1)(n.tail)) `;`
// apply loop-blocking to outer map
function(split(n.head)) `;` ...
```

LOOP-INTERCHANGE IN `tile` After blocking all `map`s recursively, we use `interchange` to rearrange the dimensions in the correct order. For simplicity, we describe the two-dimensional case of tiling matrix multiplication. The untiled matrix multiplication implementation contains a loop nest of depth three, corresponding to a `reduce` primitive nested in two `map` primitives in RISE. For brevity, we write this loop-nest as (M.N.K), indicating the dimensions each loop iterates over from outermost to innermost. After applying loop-blocking to the outermost two loops, the loop-nest has been transformed into a 5-dimensional loop-nest (M.*mTile*.N.*nTile*.K). To create the desired tiling iteration order (M.N.*mTile*.*nTile*.K), we need to swap two inner loops. To achieve this, we introduce two `transpose` patterns inside the `map` nest using the rules shown in Figure 5.6:

```
// interchange-strategy used for 2D-tiling
val loopInterchange2D =
fmap(              // in: map(map(map(map(dot)))) ...
  addId `;`        // map(id « map(map(map(dot))))
  idToTranspose `;`// map(transpose « transpose « map(map(map(dot))))
  DFNF `;`         // normalize intermediate expression
  // ... map(transpose « map(map(map(dot))) « transpose)
  argument(transposeMove)) `;`
// out: map(transpose) « map(map(map(map(dot)))) « map(transpose)
normalize(mapFission)
```

Creating the two `transpose` patterns inside the `map` nest swaps the iteration order in the desired way. The general `interchange` case adds multiple `transpose` pairs in the required positions.

Using normalization, domain-specific traversals, and five RISE-specific rewrite rules, we were able to define a multi-dimensional tiling strategy.

5.5.3 *Reordering as an ELEVATE Strategy*

Finally, we briefly discuss the implementation of TVM's `reorder` strategy, which enables arbitrary loop-interchanges. Generally, TVM's `reorder` is a generalization of the loop-interchange optimization we discussed in the previous subsection. Due to the loopless nature of RISE, implementing TVM's `reorder` primitive as a strategy is slightly more involved. Instead of merely interchanging perfectly nested loops, the same optimization effect is achieved in RISE by interchanging the nesting of **map** and **reduce** patterns. Therefore, there are multiple possible combinations to consider.

The most straightforward case is two nested **map**s, which correspond to a two-dimensional loop-nest. To interchange the loops created by the two **map**s, we introduce two **transpose** primitives and move one before and one after the **map**-nest, as discussed in the previous subsection (`loopInterchange2D`). In addition to interchanging loop-nests created by nested **map**s, we also need to consider interchanging nested **map** and **reduce** primitives. For computations including matrix multiplication it is often beneficial to hoist reduction loops higher up in a loop nest as shown in the following listings:

```
for (int i = 0; i < M; i++) {     /* map    */
  float acc = 0.0f;               /* reduce */
  for (int j = 0; j < N; j++) {
    acc += xs[j + (N * i)]; }
  out[i] = acc; }
```

Listing 5.5: Reducing the rows of a matrix `xs` with the reduction as the *inner* loop.

```
float acc[M];                                    /* reduce */
for (int i = 0; i < M; i++) { acc[i] = 0.0f; }
for (int j = 0; j < N; j++) {
  for (int i = 0; i < M; i++) {                  /* map    */
    acc[i] += xs[j + (N * i)]; }}
for (int i = 0; i < M; i++) { out[i] = acc[i]; }
```

Listing 5.6: Reducing the rows of a matrix `xs` with the reduction as the *outer* loop.

Transforming the code in Listing 5.5 into the code in Listing 5.6 enables different opportunities for parallelizing the reduction. For example, the computation in Listing 5.6 can now be easily vectorized.

The following rule implements this interchange of nested `map` and `reduce` primitives.

```
map(reduce(+)(0̲))(xs :: M.N.δ)
⇝
reduce(fun(acc, fun(y,
  map(fun(x, fst(x) + snd(x)))(zip(acc)(y))))) // reduce-op
 (generate(N)(0̲))                              // reduce-init
 (transpose(xs))                               // reduce-input
```

By transposing the input and modifying the operator of the **reduce** primitive we can interchange the nesting. Instead of reducing scalars as in the input expression, the **reduce**-operator is transformed to reduce arrays using the inner **map**. Due to the more complex AST structure after performing such a transformation, the strategy producing and traversing this tree is similarly complicated and will not be discussed in detail. While it is possible to implement TVM's **reorder** primitive, this particular loop transformation is not a good fit for the pattern-based abstractions in the RISE language.

5.5.4 *Abstractions for Describing Locations in RISE*

In TVM, named identifiers describe the location at which the optimization should be applied. For example, TVM's `split` is invoked with an argument specifying the loop to block:

```
k,     = s[C].op.reduce_axis
ko, ki = s[C].split(k, factor=4)
```

Using identifiers ties schedules to computational expressions and makes reusing schedules hard. ELEVATE does not use names to identify locations, but instead uses the traversals defined in Section 5.4. Traversals are another example of how our approach facilitates reuse – one of the key aims that we set out for our approach.

By using ELEVATE's traversal strategies, we apply the basic scheduling strategies in a much more flexible way: e.g., `topDown(parallel)` traverses the AST from top to bottom and will thus always parallelize the outermost `map`, corresponding to the outermost for loop. `tryAll(parallel)` traverses the whole AST instead, and all possible `map`s are parallelized.

To apply optimizations on large ASTs, it is often not sufficient to use the `topDown` or `tryAll` traversals. For example, we might want to block a specific loop in a loop-nest. Using `topDown(split)` always blocks the outermost loop, and `tryAll(split)` blocks every loop in the loop nest. Similarly, none of the introduced traversals so far allow to describe a precise loop conveniently , or rather a precise location, required for these kinds of optimizations.

STRATEGY PREDICATES Strategy *predicates* allow us to describe locations in a convenient way. A strategy predicate checks the program for a syntactic structure and returns `Success` without changing the program if the structure is found. Two simple examples of strategy predicates are `isReduce` and `isApp` that check if the current node is a `reduce` primitive or an applied function respectively:

```
def isReduce: Strategy[Rise] = p => p match {
    case reduce => Success(reduce)
    case _      => Failure(isReduce) }

def isApp(funPredicate: Strategy[Rise]): Strategy[Rise] = p =>
     p match {
    case app(f,_) => (_ => p) <$> funPredicate(f)
    case _        => Failure(isApp(s)) }
```

These predicates can be composed with the regular traversals to define precise locations.

The '`@`' strategy allows to describe the application of strategies at precise locations conveniently:

```
def '@'[P](s: Strategy[P], t: Traversal[P]) = t(s)
```

We write this function in infix notation and use Scala's implicit classes for this in our implementation. The left-hand side of the '@' operator specifies the strategy to apply, and the right-hand side specifies the precise location as a traversal. This design visually separates the strategy to apply from the traversal describing the location. This abstraction is especially useful for sophisticated optimization strategies with nested location descriptions. For RISE, we specify multiple traversals and predicates, which can be extended as needed. Two useful ones are `outermost` and `mapNest` that are defined as follows:

```
def outermost: Strategy[Rise] => Traversal[Rise] =
  predicate => s => topDown(predicate ‘;‘ s)
```

```
def mapNest(d: Int): Strategy[Rise] = p => (d match {
  case x if x == 0 => Success(p)
  case x if x < 0  => Failure(mapNest(d))
  case _ => fmap(mapNest(d-1))(p)                   })
```

`outermost` traverses from top to bottom, and visits nested primitives from outermost to innermost, trying to apply the predicate. Only if the predicate is applied successfully, it applies the given strategy `s`. Similarly, we define an `innermost` function which uses **`bottomUp`** instead of **`topDown`**. The `mapNest` predicate recursively traverses inside a `DFNF`-normalized **`map`**-nest of a given depth using nested `fmap` traversals. If the traversal is successful, a **`map`**-nest of depth `d` has been found.

By combining these abstractions, we conveniently describe applying the tiling optimization to the two outermost loop nests elegantly in ELEVATE:

```
(tile(32,32) ‘@‘ outermost(mapNest(2)))(mm)
```

5.6 EVALUATING SCHEDULING STRATEGIES

In this section, we evaluate our functional approach targeting the Optimization Challenge. We combine the strategies discussed in the previous section to more sophisticated strategies that describe optimizations equivalent to TVM schedules. Specifically, we use matrix-matrix multiplication as our primary case study. We analyze the performance achieved using code generated by the RISE compiler compared to TVM generated code. Afterward, we briefly investigate using ELEVATE to optimize programs of a different domain and compare it to Halide, the state-of-the-art compiler for image processing pipelines.

```
// matrix multiplication in RISE
val dot = fun(as, fun(bs, zip(as)(bs) |>
  map(fun(ab, mult(fst(ab))(snd(ab)))) |> reduce(add)(0) ) )
val mm = fun(a, fun(b,
  a |> map( fun(arow, transpose(b) |> map( fun(bcol, dot(arow)(bcol) ))
)) ))
```

```
// baseline strategy in ELEVATE
val baseline = ( DFNF ‘;‘ fuseReduceMap ‘@‘ topDown )
(baseline ‘;‘ lowerToC)(mm)
```

Listing 5.7: RISE matrix multiplication (top) and *baseline* strategy in ELEVATE (bottom).

```
1 # Naive matrix multiplication algorithm
2 k = tvm.reduce_axis((0, K), 'k')
3 A = tvm.placeholder((M, K), name='A')
4 B = tvm.placeholder((K, N), name='B')
5 C = tvm.compute((M, N),lambda x,y: tvm.sum(A[x,k]*B[k,y], axis=k),name='C')
6
7 # TVM default schedule
8 s = tvm.create_schedule(C.op)
```

Listing 5.8: TVM matrix multiplication algorithm and *baseline* (default) schedule.

5.6.1 *Optimizing Matrix Multiplication with* ELEVATE *Strategies*

For our case study of matrix multiplication, we follow a tutorial from the TVM authors [172] that discusses seven differently optimized versions: *baseline*, *blocking*, *vectorized*, *loop permutation*, *array packing*, *cache blocks*, and *parallel*. Each version is designed to improve the previous version progressively. For each TVM schedule, we develop an equivalent strategy implemented with ELEVATE and evaluate the performance achieved. Using the TVM-like scheduling abstractions implemented as strategies and the traversal utilities, we now discuss how to describe entire schedules in ELEVATE.

BASELINE For the *baseline* version, TVM uses a default schedule (Listing 5.8), whereas ELEVATE describes the implementation decisions explicitly (Listing 5.7) – one of our key aims.

The TVM algorithm computes the dot product in a single statement in Listing 5.8, line 5. The RISE program is shown at the top of Listing 5.7 describes the dot product with separate **map** and **reduce** primitives which are fused as described in the ELEVATE program below using the fuseReduceMap rewrite rules from Figure 5.6. The lowerToC strategy rewrites **map** into **mapSeq** and **reduce** into **reduceSeq**. Both systems generate equivalent C code of two nested loops iterating over the output matrix and a single nested reduction loop performing the dot product. For the following optimized versions, we do not repeat the RISE and TVM programs if they are similar to the previous version

```
val appliedReduce = isApp(isApp(isApp(isReduce)))
val blocking = ( baseline ‘;;‘ // ‘;;‘ == (‘;‘ DFNF ‘;‘)
  tile(32,32)      ‘@‘ outermost(mapNest(2))   ‘;;‘
  fissionReduceMap ‘@‘ outermost(appliedReduce)‘;;‘
  split(4)         ‘@‘ innermost(appliedReduce)‘;;‘
  reorder(List(1,2,5,6,3,4)))
(blocking ‘;‘ lowerToC)(mm) // apply strategy to mm and lower for codegen
```

```
# blocking version
xo, yo, xi, yi = s[C].tile(C.op.axis[0], C.op.axis[1], 32, 32)
k,             = s[C].op.reduce_axis
ko, ki         = s[C].split(k, factor=4)
s[C].reorder(xo, yo, ko, ki, xi, yi)
```

Listing 5.9: ELEVATE *blocking* strategy (top). TVM *blocking* schedule (bottom).

```
val loopPerm = (
  tile(32,32)      ‘@‘ outermost(mapNest(2))    ‘;;‘
  fissionReduceMap ‘@‘ outermost(appliedReduce) ‘;;‘
  split(4)         ‘@‘ innermost(appliedReduce) ‘;;‘
  reorder(List(1,2,5,3,6,4))                     ‘;;‘
  vectorize(32) ‘@‘ innermost(isApp(isApp(isMap))))
(loopPerm ‘;‘ lowerToC)(mm)
```

```
xo, yo, xi, yi = s[C].tile(C.op.axis[0], C.op.axis[1], 32, 32)
k,             = s[C].op.reduce_axis
ko, ki         = s[C].split(k, factor=4)
s[C].reorder(xo, yo, ko, xi, ki, yi)
s[C].vectorize(yi) # added already in 'vectorized'
```

Listing 5.10: ELEVATE *loop permutation* strategy (top) and TVM's schedule (bottom).

BLOCKING For the *blocking* version, we reuse the `lowerToC` strategy and the abstractions built in the previous sections, emulating the TVM schedule, as shown in Listing 5.9. First, we `tile`, then we `split`, and then we `reorder`, just as specified in the TVM schedule. To `split` the reduction, we first need to fission the fused map and reduce primitives again using `fissionReduceMap`. We describe locations using `outermost` and `innermost`, applying `tile` to the outermost **`map`**s and `split` to the nested reduction. In contrast to TVM, for `reorder`, we identify dimensions by index rather than by name. We introduce the `‘;;'` combinator for convenience. It denotes that we apply `DFNF` to normalize intermediate expressions between each step.

VECTORIZED AND LOOP PERMUTATION For brevity, we summarize the *vectorized* and the *loop permutation* version. The *vectorized* version vectorizes the innermost loop. We achieve this using `(vectorize(32)'@'innermost(isMap))`, as discussed earlier. The *loop permutation* version shown in Listing 5.10 performs the same vectorization and additionally applies a different reordering of loops compared to the prior *blocking* version.

```
val appliedMap = isApp(isApp(isMap))
val isTransposedB = isApp(isTranspose)

val packB = storeInMemory(isTransposedB,
 permuteB ‘;;‘
 vectorize(32) ‘@‘ innermost(appliedMap) ‘;;‘
 parallel      ‘@‘ outermost(isMap)
) ‘@‘ inLambda

val arrayPacking = packB ‘;;‘ loopPerm
(arrayPacking ‘;‘ lowerToC )(mm)
```

```
 1  # Modified algorithm
 2  bn = 32
 3  k  = tvm.reduce_axis((0, K), 'k')
 4  A  = tvm.placeholder((M, K), name='A')
 5  B  = tvm.placeholder((K, N), name='B')
 6  pB = tvm.compute((N / bn, K, bn), lambda x,y,z: B[y, x*bn+z], name='pB')
 7  C  = tvm.compute((M,N), lambda x,y:
 8        tvm.sum(A[x,k] * pB[y//bn,k, tvm.indexmod(y,bn)],axis=k),name='C')
 9
10  # Array packing schedule
11  s = tvm.create_schedule(C.op)
12  xo, yo, xi, yi = s[C].tile(C.op.axis[0], C.op.axis[1], bn, bn)
13  k,             = s[C].op.reduce_axis
14  ko, ki         = s[C].split(k, factor=4)
15  s[C].reorder(xo, yo, ko, xi, ki, yi)
16  s[C].vectorize(yi)
17  x, y, z = s[pB].op.axis # prepare scheduling optimized pB stage
18  s[pB].vectorize(z)
19  s[pB].parallel(x)
```

Listing 5.11: ELEVATE *array packing* strategy (top), TVM *array packing* schedule and optimized algorithm (bottom).

ARRAY PACKING So far, ELEVATE's strategies and TVM's schedules were reasonably similar. The *array packing* version is the first to emphasize the flexibility of our holistic functional approach. As already discussed in the motivation section, some optimizations are not expressible in TVM's scheduling API without changing the algorithm – This clearly violates the separation between specifying computations and optimizations. Here specifically, the TVM *array packing* version in Listing 5.11 (bottom) permutes the elements of the B matrix in memory to improve the memory access patterns by introducing an additional computation `pB` in lines 6-6, before using it in the computation in line 7.

For our implementation of the *array packing* version in Listing 5.11 (top), we are not required to change the RISE program but define and apply the array packing of matrix B simply as another rewrite step in ELEVATE using the `storeInMemory` strategy described below. To complete the entire strategy for this version, we compose the array packing together with `permuteB` that uses **`transpose`** primitives to implement the permutation similarly to `interchange` for the `tile`

```
def storeInMemory(what: Strategy[Rise],
                  how : Strategy[Rise]): Strategy[Rise] = { p =>

  extract(what)(p) »= (extracted => { how(extracted) »= (storedSubExpr => {

    val idx = Identifier(freshName("x"))
    replaceAll(what, idx)(p) match {
      case Success(replaced) =>
        Success(toMem(storedSubExpr)(fun(idx, replaced)))
      case Failure(_) => Failure(storeInMemory(what, how))
  }})}}

  // helper-functions
  def replaceAll(exprPredicate: Strategy[Rise],
                 withExpr: Rise): Strategy[Rise] =
    p => tryAll(exprPredicate ‘;‘ insert(withExpr))(p)
  def insert(expr: Rise): Strategy[Rise] = p => Success(expr)
  // find and return Rise expr which matches the exprPredicate
  def extract(exprPredicate: Strategy[Rise]): Strategy[Rise] = ...
}

def inLambda(s: Strategy[Rise]): Strategy[Rise] =
  isLambda ‘;‘ ((p:Rise) => body(inLambda(s))(p)) <+ s
```

Listing 5.12: ELEVATE strategy to store RISE subexpressions to memory.

strategy. Finally, we can reuse the prior `loopPerm` strategy to complete this version.

The strategy for storing sub-expressions in memory uses the `toMem` primitive of RISE and is defined as shown in Listing 5.12. The `storeInMemory` strategy expects two arguments: `what` - a strategy predicate describing the sub-expression to store and, `how` - the strategy specifying how to perform the copy. In the `arrayPacking` strategy, we want to store a permuted version of the transposed B (described by the `isTransposedB` predicate) to memory. Since every RISE sub-expression can be stored in memory at any time, the `storeInMemory` strategy only fails if the desired sub-expression (described by `what`) cannot be found or cannot be stored as specified in `how`.

The `toMem` primitive is introduced in two steps: First, the sub-expression needs to be removed from the original expression (`p`) and be replaced with the new identifier `x`. Second, the value for the new identifier `x` needs to be extracted from the original expression `p`.

CACHE BLOCKS AND PARALLEL The TVM version in Listing 5.13 changes the algorithm yet again (not shown for brevity) to introduce a temporary buffer (`CC`) for the accumulation along the K-dimension to improve the cache writing behavior and unrolls the inner reduction loop. The RISE code generator makes accumulators for reductions always explicit. Therefore, we reuse the *array packing* version adding strategies for unrolling the innermost reduction and parallelizing the outermost loop.

```
val par = (
  arrayPacking                       `;;`
  (parallel `@` outermost(isMap))   `;;`
            `@` outermost(isToMem)  `;;`
  unroll    `@` innermost(isReduce))

(par `;` lowerToC)(mm)
```

```
# Another change to the algorithm omitted here: Introducing CC stage
s = tvm.create_schedule(C.op)
CC = s.cache_write(C, 'global')
xo, yo, xi, yi = s[C].tile(C.op.axis[0], C.op.axis[1], bn, bn)
s[CC].compute_at(s[C], yo)
xc, yc = s[CC].op.axis
k, = s[CC].op.reduce_axis
ko, ki = s[CC].split(k, factor=4)
s[CC].reorder(ko, xc, ki, yc)
s[CC].unroll(ki)
s[CC].vectorize(yc)
s[C].parallel(xo)
x, y, z = s[pB].op.axis
s[pB].vectorize(z)
s[pB].parallel(x)
```

Listing 5.13: ELEVATE *parallel* strategy (top). TVM *parallel* schedule (bottom).

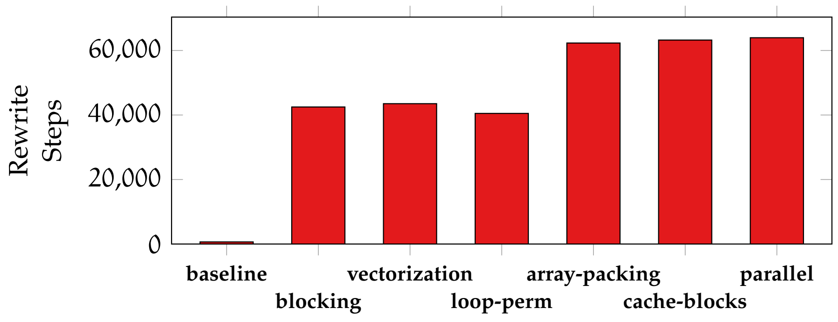


Figure 5.10: Total number of successful rewrite steps when applying different optimization strategies.

5.6.2 *Scalability and Overhead of Rewriting*

We have demonstrated that it is feasible to implement a TVM-like scheduling language by expressing schedules as compositions of reusable strategies. Using our holistic approach, we express the computation only once in RISE and express all optimizations as ELEVATE strategies. In this section, we are interested in the scalability and overhead of our functional rewrite-based approach using matrix multiplication as a case study of high-performance compilation.

We are counting the number of successfully applied rewrite steps performed when applying a strategy to the RISE matrix multiply expression. Every intermediate step is counted, which includes traversals as these are implemented as rewrite steps. For example, `id(fun(x,x))` would be counted as one rewrite step whereas `(body` `(id))(fun(x,x))` would be counted as two steps because we also count the traversal into the function body as one step.

Figure 5.10 shows the number of rewrites for each version. No significant optimizations are applied to the *baseline* version, and 657 rewrite steps are performed. However, as soon as interesting optimizations are applied, we reach about 40,000 steps for the next three versions and about 63,000 for the most complicated optimizations. Applying the strategies to the unoptimized RISE expression took less than two seconds per version on a commodity notebook.

These high numbers clearly show the scalability of our compositional approach in which complex optimizations are composed out of a small set of fundamental building blocks. It also shows that abstractions are required to control this many rewrite steps. The high-level strategies encode practical optimizations and hide massive numbers of individual rewrite steps that are actually performed. Simultaneously, developing and debugging such sophisticated strategies using ELEVATE is still possible as there is a clear pathway towards developing debugging tools for recording traces or rewrites or reporting which strategy was not applicable. Additionally, due to the compositional nature of our strategy approach, one can easily inspect intermediate RISE expressions. In contrast, debugging TVM or Halide schedules is much harder as the scheduling primitives are provided as black-box function calls operating on the internal intermediate program representation.

5.6.3 *Performance Comparison against TVM*

In this section, we are interested in the performance achieved when optimizing RISE programs with ELEVATE compared to TVM. Ideally, the RISE code optimized with ELEVATE should be similar to the TVM-generated code and achieve competitive performance. We generated LLVM code with TVM (version 0.6.dev) and C code for RISE annotated with OpenMP pragmas for the versions which include parallelization or vectorization. The RISE generated C code was compiled with clang (v.9.0.0) using `-Ofast -ffast-math -fopenmp`, which echoes the settings used by TVM and Halide [71]. We used an Intel core i5-4670K CPU (frequency at 3.4GHz) running Arch Linux (kernel 5.3.11-arch1-1). We report wall-time for RISE-generated code and used TVM's built-in measurement API. We performed 100 iterations per version reporting the median runtimes in milliseconds.

Figure 5.11 shows the performance of RISE and TVM generated code. The code generated by RISE controlled by the ELEVATE optimization strategies performs competitively with TVM. Our experiment indicates a matching trend across versions compared to TVM, showing that our ELEVATE strategies encode the same optimizations used in the TVM schedules. The most optimized parallel RISE generated

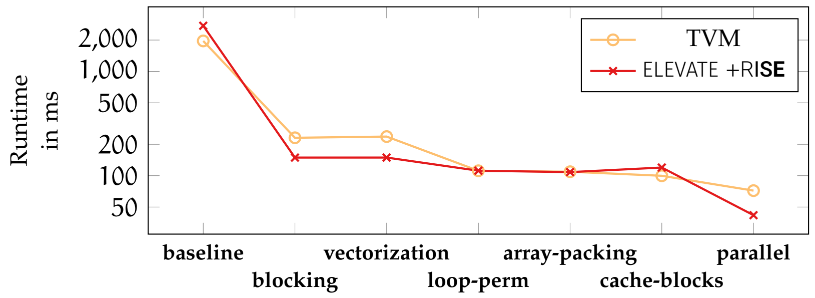


Figure 5.11: Performance of TVM vs. RISE generated code that has been optimized by ELEVATE strategies.

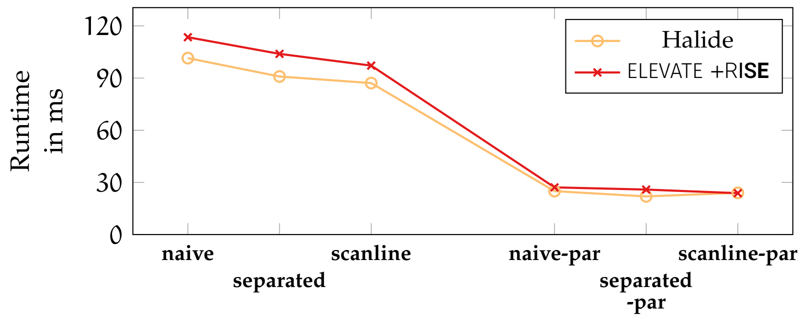


Figure 5.12: Performance evaluation of Halide and RISE generated code for the binomial filter application. Optimization decisions for RISE are implemented as ELEVATE strategies.

version improves the performance over the baseline by about 110×. The strategies developed in an extensible style by composing individual rewrite steps using ELEVATE, are practically usable and provide competitive performance for important high-performance code optimizations.

5.6.4 *Performance Comparison against Halide*

Since the scheduling languages of Halide and TVM are very similar, we additionally performed a third experiment comparing against the Halide compiler that is specialized in high-performance code generation for image processing. Specifically, we implemented a binomial image processing filter in RISE, corresponding to the example described in [141]. As with the comparison against TVM, we used ELEVATE to describe three different optimization strategies (each with a sequential and a parallel version), which are equivalent to three different schedule programs using Halide. Optimizing the binomial filter application also required to modify the algorithm in Halide, similar to the *array packing* and *parallel* example in TVM, whereas we were again able to express all optimizations using ELEVATE strategies only.

Figure 5.12 shows the performance of the Halide and RISE generated code measured on a ARM Cortex A7 quad-core (The 4 LITTLE cores of a Samsung Exynos 5 Octa 5422). We can see – not surprisingly – that the non-parallel versions on the left are significantly slower than the parallel versions. The Halide generated code is 10-15% faster than the RISE generated code. Improvements to the RISE code generator might close this gap in the future.

Crucially, we again observe the same trend for performance improvements across versions. These results demonstrate that our extensible and rewrite based approach is capable of achieving competitive performance to state-of-the-art compilers used in production. Encoding optimizations as extensible building blocks using rewrite rules thus poses a viable solution to the Optimization Challenge. As a last case study, we use ELEVATE to optimize a different programming language to emphasizing that ELEVATE is not specialized only to optimize RISE programs.

5.7 CASE STUDY: AUTOMATIC DIFFERENTIATION

So far, we used RISE as the only example of a language transformed with ELEVATE. However, ELEVATE and its rewrite-based optimization approach are flexible and not restricted to a single language. Therefore, in this final case study, we will additionally target the $\widetilde{\mathbf{F}}$ language that has been introduced in [160].

$\widetilde{\mathbf{F}}$ is a small functional language capable of automatically computing the derivative of arbitrary $\widetilde{\mathbf{F}}$ functions. Automatic differentiation is often required for back-propagation computations to adjust weight matrices when training neural networks. Implementing automatic differentiation requires following precise rules and is thus not too difficult; however, generating efficient differentiated programs is non-trivial. A common problem for achieving high-performance is an explosion in the code size of differentiated programs, and $\widetilde{\mathbf{F}}$ achieves efficiency by rewriting the differentiated code using several simplification rules.

Shaikhha et al. [160] introduce multiple rewrite rules for optimizing $\widetilde{\mathbf{F}}$ programs and provide instructive examples. For example, they show that an $\widetilde{\mathbf{F}}$ program transposing a matrix twice can be rewritten into a more concise program without transposition (see Figure 5.13) by systematically applying the $\widetilde{\mathbf{F}}$ simplification rules. The authors do not provide a derivation or explanation for how exactly the rewriting between these programs is specified. Instead, they state that: *"by applying the loop fusion rules and performing further partial evaluation, the expression is derived"*.

We are interested in exploring the flexibility of ELEVATE and if we can specify the rewrite rule applications programmatically. Therefore, we implemented $\widetilde{\mathbf{F}}$ as an embedded language in Scala using

```
let MT = build (length M[0]) (fun i ->
          build (length M) (fun j -> M[j][i] ) ) in
         build (length MT[0])(fun i ->
          build (length MT) (fun j -> MT[j][i] ) )
```

```
build (length M) (fun i ->
 build (length M[0]) (fun j -> M[i][j] ) )
```

Figure 5.13: Transposing a matrix twice in $\widetilde{\mathbf{F}}$ (top) and the rewritten program without transposition (bottom). This example is taken from [160].

an algebraic data type `FSmooth`. We also implemented the provided rewrite rules for simplifying (differentiated) $\widetilde{\mathbf{F}}$ programs, such as the fusion rules:

`build` constructs an array in $\widetilde{\mathbf{F}}$ given a size and a function for generating values of type `M` [160]: *`build: Card => (Index => M) => Array<M>`*

$$\begin{array}{lll} \texttt{(build } e_0\ e_1\texttt{)[}e_2\texttt{]} & \rightsquigarrow & e_1\ e_2 \\ \texttt{length (build } e_0\ e_1\texttt{)} & \rightsquigarrow & e_0 \end{array}$$

For example, the former rule states the following identity: Constructing an array of size e_0 that is initialized with values according to e_1 and accessing its value at position e_2 ($\texttt{(build } e_0 e_1\texttt{)[}e_2\texttt{]}$) is equivalent to merely calling the value-generating function e_1 with e_2 as argument, as shown on the right-hand side of the rewrite rule. The latter rule states that constructing an array of size e_0 ($\texttt{build } e_0 e_1$), and immediately querying its `length` is equivalent to the simplified expression on the right-hand side (e_0).

We implement these fusion rules and all other $\widetilde{\mathbf{F}}$ rules as ELEVATE strategies, as described in the previous sections. For example, the fusion rules are implemented as follows:

```
def buildGet(p: FSmooth): Strategy[FSmooth] = p match       {
  case app(get,(app(build,(e0,e1)),e2)) => Success(app(e1,e2))
  case _ =>                                Failure(buildGet)  }

def lenBuild(p: FSmooth): Strategy[FSmooth] = p match       {
  case app(length, app(build, (e0, e1))) => Success(e0)
  case _                                => Failure(lenBuild) }
```

Encoding the $\widetilde{\mathbf{F}}$ rules as ELEVATE strategies allows us to use `normalize` and `lChoice` to specify a simple normal-form that exhaustively applies a given set of simplification rules:

Here, `letPartialEval`, `letApp`, and `funToLet` refer to other $\widetilde{\mathbf{F}}$ simplification rules, as described in [160].

```
normalize(buildGet <+ lenBuild <+ letPartialEval <+
          letApp <+ funToLet)
```

This ELEVATE strategy successfully rewrites the doubly-transposed $\widetilde{\mathbf{F}}$ program shown in Figure 5.13 (top) into the non-transposed form, as shown on the bottom. Tracing the execution of the ELEVATE strategy allows us to obtain the exact sequence of rewrite steps. In this case, the doubly-transposed expression requires 12 rewrite steps to be transformed into the non-transposed version.

Additionally, we implemented the other $\widetilde{\mathbf{F}}$ simplification examples using ELEVATE strategies and, due to our programmatic approach, we even identified a minor bug in one of their examples. Generally, this brief case-study shows ELEVATE's flexibility to implement and optimize another existing rewrite system, and that ELEVATE is not restricted to rewrite RISE programs.

5.8 CONCLUSION

In this chapter, we presented ELEVATE, a language for defining optimization strategies. With ELEVATE, we aim to address the Optimization Challenge defined in Section 1.2.2. We showed that our high-performance program optimization approach successfully:

The Optimization Challenge: How can we encode and apply domain-specific optimizations for high-performance code generation while providing precise control and the ability to define custom optimizations, thus achieving a reusable optimization approach across application domains and hardware architectures? (Section 1.2.2)

- *separates concerns* by truly separating the computation and strategy languages;
- *facilitates reuse* of computational patterns and optimizations encoded as rewrite rules;
- *enables composability* by building programs and rewrite strategies as compositions of a small number of fundamental building blocks;
- *allows reasoning* about programs and strategies since both are expressed as functional programs with well-defined semantics; and
- *is explicit* by empowering users to control the optimization strategy that is applied by our compiler.

In contrast to existing imperative systems with scheduling APIs such as Halide, Fireiron, or TVM, programmers are not restricted to apply set of built-in optimizations but can define their own optimization strategies. We also showed that our optimization approach is applicable to different programming languages such as RISE or $\widetilde{\mathbf{F}}$. Finally, our experimental evaluation demonstrates that our holistic functional approach achieves competitive performance compared to the state-of-the-art code generators Halide and TVM.

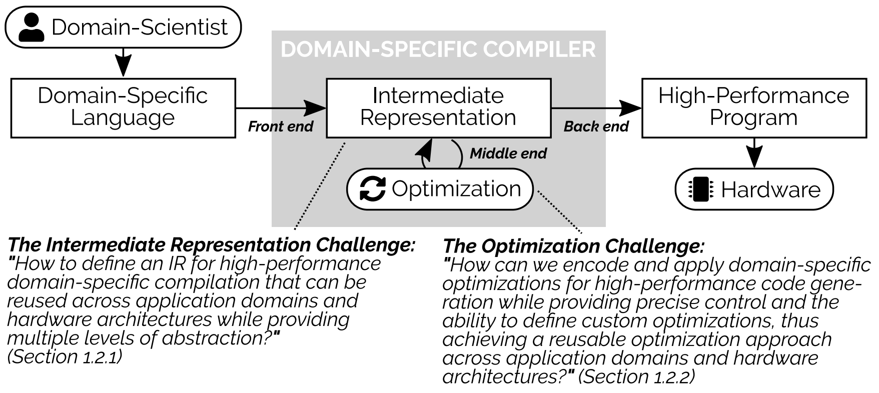
6 TOWARDS A UNIFIED COMPILATION APPROACH

In the three previous technical chapters, we have demonstrated the potential of high-performance domain-specific compilation. However, due to limited reusability between existing compilers, building a domain-specific compiler for targeting a new application domain or hardware architecture is still a complicated and time-intensive process. Figure 6.1 visualizes this traditional process of achieving domain-specific compilation and summarizes its challenges, as identified in Chapter 1.

In this chapter, we begin by summarizing our individual contributions presented in the previous chapters. Afterward, we present our vision of achieving high-performance domain-specific compilation *without* constructing a domain-specific compiler by combining the individual contributions to a novel unified compilation approach. Finally, we discuss how the work presented in this thesis can be further extended in the future.

6.1 SUMMARY: HIGH-PERFORMANCE DOMAIN-SPECIFIC COMPILATION WITHOUT DOMAIN-SPECIFIC COMPILERS

In the following sections, we briefly summarize our contributions and refer back to the advantages and challenges related to domain-specific compilation, as introduced in Chapter 1.



The Intermediate Representation Challenge:
"How to define an IR for high-performance domain-specific compilation that can be reused across application domains and hardware architectures while providing multiple levels of abstraction?"
(Section 1.2.1)

The Optimization Challenge:
"How can we encode and apply domain-specific optimizations for high-performance code generation while providing precise control and the ability to define custom optimizations, thus achieving a reusable optimization approach across application domains and hardware architectures?" (Section 1.2.2)

Figure 6.1: The traditional way of using a domain-specific compiler to achieve high-performance domain-specific compilation and its associated challenges, as identified and discussed in Chapter 1 of this thesis.

6.1.1 *Demonstrating the Potential of Domain-Specific Compilation*

In Chapter 3, we introduced Fireiron, a domain-specific compiler for generating fast matrix multiplication kernels for NVIDIA GPUs.

FIREIRON: A DATA-MOVEMENT-AWARE-COMPILER FOR GPUS Existing high-performance domain-specific compilers generally treat optimizations for data-movements as second-class citizens. With Fireiron, we introduced a compiler IR and scheduling language in which both computations *and data movements* are first-class citizens, meaning that they can be scheduled with the same primitives. Fine-grained control over data movement optimizations is especially crucial for generating high-performance implementations targeting GPUs. In order to achieve near peak performance, programmers must use the GPU's multi-layered compute and memory hierarchies as efficiently as possible. Doing this requires precise mappings of which element of the compute hierarchy processes which part of the computation, and fine-grained control over the coordination of data movements through the various memory hierarchy levels. The increasing diversity in memory hierarchy levels demonstrates the importance of optimizing data movements as first-class concepts in a scheduling language of a domain-specific compiler targeting GPUs. For example, the introduction of the warp-wide register fragments, which are required for Tensor Core computations on modern NVIDIA GPUs, highlights this challenge.

The two major concepts we introduced with Fireiron are *Specifications* and *Decompositions*. Specifications provide high-level abstractions for gradually decomposing computations and data movements (using the Decompositions) until they match assembly instructions, accelerator primitives, or predefined microkernels. Representing each (instructions, primitives, and microkernels) using precise specifications as a unifying concept enables this gradual decomposition.

We have shown that Fireiron's specifications and decompositons allow performance engineers to precisely describe high-performance implementations that must be written so far in low-level assembly instead. The performance achieved by the Fireiron-generated kernels shows 1) that Fireiron can accurately capture the optimizations that humans typically apply by hand by in low-level code by matching the performance of handwritten implementations on different GPU architectures, and 2) that Fireiron can be used by performance engineers to find and express implementations that even outperform high-performance vendor library routines.

The design and implementation of the Fireiron domain-specific compiler is the first major contribution of this thesis. It demonstrates the potential of domain-specific compilation and the challenges of designing a domain-specific compiler at the same time.

6.1.2 *Addressing the Intermediate Representation Challenge*

The Intermediate Representation Challenge: How to define an IR for high-performance domain-specific compilation that can be reused across application domains and hardware architectures while providing multiple levels of abstraction? (Section 1.2.1)

Chapter 4 discussed how to express stencil computations by extending the generic, functional, and pattern-based IR LIFT.

HIGH-PERFORMANCE STENCIL CODE GENERATION WITH LIFT By extending the existing LIFT IR with two new generic primitives *pad* and *slide*, we showed that it is possible to express domain-specific computations (stencils) with domain-agnostic primitives. Instead of adding a single stencil-specific new primitive, we showed that stencil computations, in general, can be conceptually decomposed into three fundamental steps: 1) boundary handling, 2) neighborhood creation, and 3) output computation. Each of these steps is expressible with a single LIFT primitive: *pad*, *slide*, and *map*.

To express arbitrary multi-dimensional stencil computations, we showed that it is possible to define higher-dimensional stencil abstractions by simply nesting and composing the existing one-dimensional primitives. We did not need to introduce specialized primitives for higher dimensions as often done in related approaches. Since we can reuse the existing *map* primitive and only use one-dimensional primitives, optimizing stencil computations expressed in LIFT requires only minor additions. We introduced rules to rewrite expressions using the two new primitives and showed how we can reuse the remaining existing LIFT approach for optimizing programs. The performance achieved by the LIFT-generated stencil code matches and even outperforms handwritten benchmarks and a state-of-the-art polyhedral compiler on different GPU architectures.

At the time of this work, LIFT had only been used for expressing and optimizing dense linear algebra computations. Our work on stencil computations showed that LIFT's approach is far more powerful and that its generic, domain-agnostic IR can be used to express computations of different domains, such as stencil computations. In fact, with the work described in Chapter 4, we showed that the LIFT approach is a viable solution to the IR Challenge identified in Section 1.2.1. Today, LIFT and RISE (LIFT's spiritual successor, see Section 5.3.1) are used to express computations of many more domains, including sparse linear algebra, image processing, machine learning, and high-level synthesis for FPGAs. Many computations of these domains expressed in LIFT reuse the discussed *pad* and *slide* primitives in different contexts, which shows that they too are not stencil-specific but rather as flexible as the already existing primitives.

Demonstrating that the LIFT IR successfully solves the IR Challenge by expressing domain-specific multi-dimensional stencil computations that require only small extensions to the existing generic IR is the second major contribution of this thesis.

6.1.3 *Addressing the Optimization Challenge*

The Optimization Challenge: How can we encode and apply domain-specific optimizations for high-performance code generation while providing precise control and the ability to define custom optimizations, thus achieving a reusable optimization approach across application domains and hardware architectures? (Section 1.2.2)

In Chapter 5, we introduced ELEVATE, a language for describing program optimization strategies as compositions of rewrite rules.

A LANGUAGE FOR DESCRIBING OPTIMIZATION STRATEGIES
Inspired by earlier work on strategy languages for term rewriting systems, we introduced ELEVATE, a language for encoding and applying optimization strategies as composable rewrites to address the Optimization Challenge defined in Section 1.2.2. ELEVATE allows us to define AST transformations as simple rewrite rules for arbitrary target languages and enables us to combine them into sophisticated optimization strategies using different combinators and traversals.

In a detailed case study, we demonstrated how to encode and apply domain-specific machine learning optimizations as ELEVATE strategies for rewriting RI**SE** matrix-multiplication expressions. Using TVM's state-of-the-art scheduling language as an example, we showed how to decompose and implement each scheduling primitive as a composition of simple rewrite rules in ELEVATE. For example, we showed how the two-dimensional TVM `tile` primitive is expressible as a composition of only a few rewrite rules, traversals, and normal-forms. Our resulting tiling strategy is even more expressive than TVM's built-in primitive as ours allows us to tile arbitrary dimensions due to being recursively defined based on common functional programming techniques.

Chapter 4 discussed how we decomposed stencil computations to gain expressiveness by re-composing simpler computational primitives. Similarly, with ELEVATE, we demonstrated the possibility of decomposing optimizations typically applied in domain-specific compilers for achieving reusability by composing simple rewrite rules that are easy to reason about. Expressing domain-specific optimizations as compositions of simple rules leads to strategies that transparently perform thousands of rewrite steps. With ELEVATE, we can hide those behind high-level abstractions that are as easy to use as state-of-the-art scheduling languages while being significantly more flexible due to the possibility to redefine or extend the set of rewrite rules.

By applying ELEVATE to simplify automatically differentiated $\widetilde{\mathbf{F}}$ programs, we demonstrated ELEVATE's flexibility to optimize programs from different domains, one of the main goals we defined in the Optimization Challenge.

The design, implementation, and application of the ELEVATE strategy language to achieve high-performance domain-specific compilation, targeting the Optimization Challenge, is the third major and final contribution of this thesis.

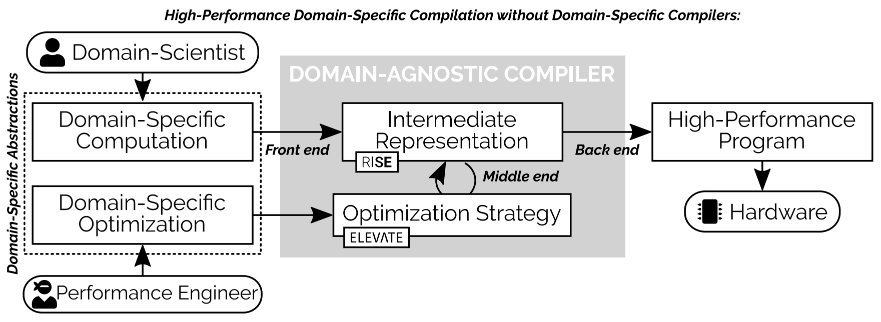

High-Performance Domain-Specific Compilation without Domain-Specific Compilers:
Domain-Scientist
Domain-Specific Abstractions
Domain-Specific Computation
Domain-Specific Optimization
Performance Engineer
DOMAIN-AGNOSTIC COMPILER
Front end
Intermediate Representation
RISE
Middle end
Optimization Strategy
ELEVATE
Back end
High-Performance Program
Hardware

Figure 6.2: Our approach for achieving high-performance domain-specific compilation without domain-specific compilers. We use RISE as a flexible and domain-agnostic intermediate representation and ELEVATE for encoding high-performance program optimizations as composable rewrite strategies. This design enables the development of different domain-specific abstractions and *reuses* the same domain-agnostic compiler for high-performance code generation.

6.2 TOWARDS A UNIFIED APPROACH FOR HIGH-PERFORMANCE DOMAIN-SPECIFIC COMPILATION

In this section, we introduce our vision of a holistic and unified approach for achieving high-performance domain-specific compilation without domain-specific compilers. Specifically, we discuss how combining the individual contributions summarized in the previous section leads to a reusable compiler design for leveraging the benefits of domain-specific compilation without requiring the construction of domain-specific compilers.

Figure 6.2 presents our novel compilation approach. This design is discussed in detail in the following paragraphs. Generally, we envision a domain-agnostic compiler that is reusable across different application domains and hardware architectures and that is capable of generating high-performance code.

A REUSABLE DOMAIN-AGNOSTIC COMPILER The primary motivation for the work in the thesis is to leverage the advantage of high-performance domain-specific compilation (as demonstrated in Chapter 3) without requiring the construction of a new domain-specific compiler for every new application domain or hardware architecture. Therefore, we require a *reusable* and *domain-agnostic* compilation approach.

In Chapter 1, we analyzed existing state-of-the-art general-purpose and domain-specific compilers. We identified two challenges that must be addressed for achieving our goal: We need a reusable IR that is capable of providing domain-specific abstractions while being itself domain-agnostic (Section 1.2.1). In addition, we identified the need for a reusable optimization approach that provides precise control over where, how, and when domain-specific optimizations are applied (Section 1.2.2).

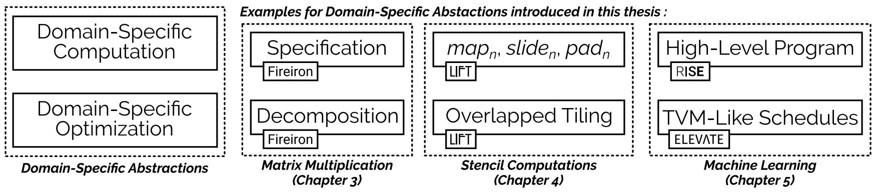

Examples for Domain-Specific Abstactions introduced in this thesis :
Domain-Specific Computation
Domain-Specific Optimization
Domain-Specific Abstractions
Specification
Fireiron
Decomposition
Fireiron
Matrix Multiplication (Chapter 3)
map_n, $slide_n$, pad_n
LIFT
Overlapped Tiling
LIFT
Stencil Computations (Chapter 4)
High-Level Program
RISE
TVM-Like Schedules
ELEVATE
Machine Learning (Chapter 5)

Figure 6.3: Examples of possible domain-specific abstractions that already are, or potentially could be lowered into, RISE programs and ELEVATE optimization strategies.

We argue that using RISE as intermediate representation and ELEVATE for controlling the optimization process successfully addresses exactly these two challenges. Therefore, the envisioned reusable domain-agnostic compiler shown in Figure 6.2 uses RISE and ELEVATE as its two core components. Both languages, RISE and ELEVATE, follow the same design principle of providing an intentionally small set of generic and composable building blocks. RISE provides computational building blocks in the form of functional primitives for manipulating arrays; and ELEVATE provides reusable primitives for composing rewrite rules that encode program optimizations to sophisticated optimization strategies.

external and extensible domain-specific abstractions

To leverage the benefits of domain-specific compilation, our compiler needs to be able to represent domain-specific computations and optimize those with domain-specific optimizations. We achieve this by providing both domain-specific computations and optimizations as external high-level abstractions, as shown in Figure 6.2. This modular design allows us to design different abstractions for different domains and hardware architectures that are still all compiled using the same high-performance domain-agnostic compiler.

Figure 6.3 shows examples of domain-specific abstractions discussed in this thesis that already are, or could be lowered to, RISE programs and ELEVATE strategies. Generally, in Chapters 4 and 5, we demonstrated how the building blocks provided by Lift (and thus RISE), as well as ELEVATE, are combined to define high-level domain-specific abstractions. Crucially, we showed that they provide the same level of abstraction as existing state-of-the-art domain-specific languages and compilers. In fact, Chapter 5 already presented a prototype implementation of the compilation approach shown in Figure 6.2: In our case study on the machine learning compiler TVM, we used high-level RISE programs to express the computations and ELEVATE for implementing TVM's scheduling primitives for guiding the optimization. Similarly to how TVM provides its computational building blocks and scheduling primitives to domain-scientists and performance engineers, we expose our RISE and ELEVATE abstractions as machine learning domain-specific abstractions in a library, as shown on the right-hand side of Figure 6.3. Crucially, in contrast to TVM, our abstractions are not built into the compiler. Instead, our domain-agnostic compiler is only aware of the small set of core RISE and ELEVATE building blocks; all high-level domain-specific abstractions are external and replaceable.

This allows us to reuse the same domain-agnostic compiler not only to generate high-performance machine learning code but also to generate high-performance stencil computations. The domain-specific stencil abstractions discussed in Chapter 4, namely expressing stencil computations using *pad*, *slide*, and *map*, and optimizations such as overlapped tiling, can be defined as RISE expressions and ELEVATE strategies.

As shown in Figure 6.3, we also believe that we can reuse the same domain-agnostic compilation approach for compiling Fireiron's matrix multiplication specific abstractions into high-performance code instead of using Fireiron's own domain-specific code generator. Specifically, this requires a lowering of Fireiron's Specifications and Decompositions into RISE programs and ELEVATE strategies, similar to the work presented in the TVM case study.

CONCLUSION Our vision of a unified approach to domain-specific compilation, as shown in Figure 6.2, will greatly improve the state-of-the-art in developing optimizing compilers for achieving high performance. We presented a novel approach for achieving the benefits of high-performance domain-specific compilation without the need to develop domain-specific compilers. The key idea of our approach is decomposing both domain-specific computations and their optimizations into a small set of generic building blocks. In contrast to existing domain-specific compilers, this decomposition allows us to avoid relying on compiler-internal built-in domain-specific abstractions for achieving high performance. Instead, we simply express domain-specific computations and their optimizations as *compositions* of those generic building blocks. Using RISE as intermediate representation and ELEVATE for controlling the optimization process enables the development of external and replaceable high-level domain-specific abstractions, as shown in Figure 6.3. Ultimately, this unified compilation approach allows us to reuse the same domain-agnostic compiler for many application domains and hardware architectures, as we have shown in the previous chapters.

6.3 FUTURE WORK

As discussed in the previous section, our three individual contributions can naturally be combined to achieve a holistic and unified compilation approach. Figure 6.4 shows a more detailed view of this approach and includes possible future extensions. Here, we use

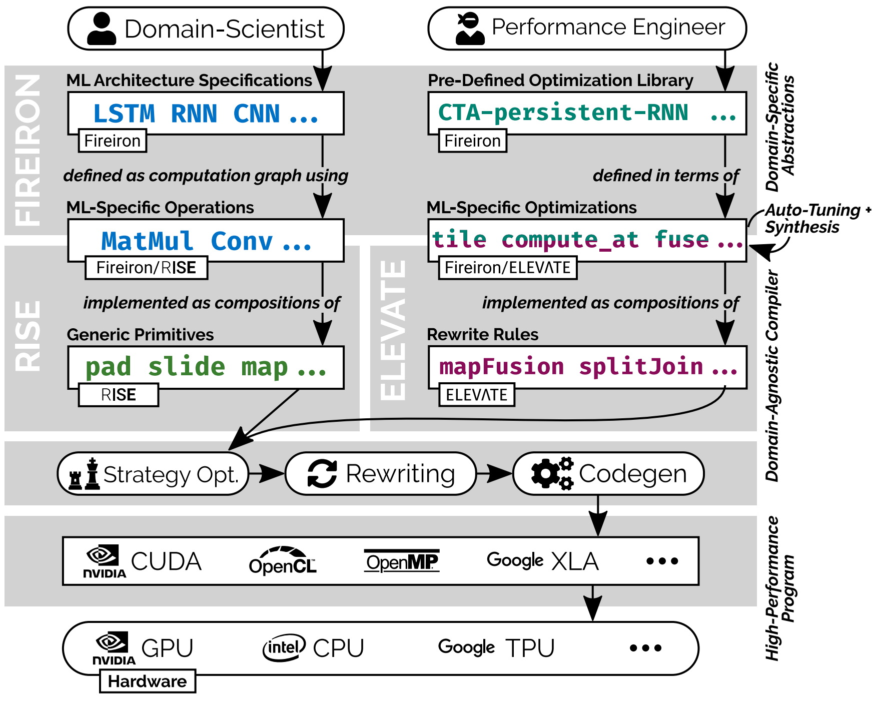


Figure 6.4: One possibility to combine and extend the approaches introduced in this thesis (extensions are shown in gray). Extending the set of Fireiron specifications and decompositions and implementing them in terms of RISE expressions and ELEVATE strategies allows us to leverage the advantages of each separate approach.

Fireiron for providing high-level domain-specific abstractions and use deep learning as an example application domain. We briefly outline this extended approach in the following and discuss the single extensions in more detail in the subsequent subsections.

Fireiron's specifications and decompositions already provide easy-to-use high-level abstractions for domain-scientists and performance engineers. Raising the abstraction level by defining higher-level constructs such as complete neural network architectures like Long-Short-Term Memory (LSTM), Recurrent Neural Networks (RNN), or Convolutional Neural Networks (CNN) could even improve Fireiron's usability. Each of these architectures defines a computations graph performing typical machine learning computations, including matrix multiplications or convolutions in a specific sequence. Similarly to how high-performance libraries such as cuBLAS provide multiple implementations for the same computation, we can imagine providing predefined optimization strategies as high-performance Fireiron decompositions in a library, as shown in Figure 6.4 (top right).

Instead of using Fireiron's code generator, a useful next step would be to implement Fireiron's specs, e.g., `MatMul`, in terms of RISE (or LIFT) expressions, and Fireiron's decompositions as ELEVATE strategies. This way, Fireiron acts as a high-level library providing easy-to-use abstractions for both domain-scientists as well as performance engineers. At the same time, we would make use of the more principled strategic rewriting of RISE programs using ELEVATE that is based on a solid foundation of fundamental functional programming techniques. However, the RISE code generator needs to be adjusted to generate CUDA code that uses Tensor Cores to achieve the same performance as Fireiron's current code generator.

In the following, we discuss our envisioned extensions and enhancements in more detail.

6.3.1 *Enhancing the Fireiron Domain-Specific Compiler*

In the following, we discuss several possible extensions and enhancements for Fireiron.

IMPLEMENTING FIREIRON USING RISE AND ELEVATE The first possible enhancement for Fireiron would be to implement its two core concepts - specifications and decompositions - in terms of RISE and ELEVATE, respectively. Currently, as described in Chapter 3, Fireiron is implemented as a standalone domain-specific compiler with its own ad-hoc code generation implementation. Implementing Fireiron in terms of RISE and ELEVATE would allow us to replace the existing code generator and reuse the RISE code generator while driving the optimization process using formal rewrite rules as specified in ELEVATE strategies. As we have seen in the previous chapters, both RISE and ELEVATE are based on fundamental functional programming techniques to achieve high-performance compilation from first principles.

We believe there is a strong correspondence between Fireiron specifications and RISE expressions as well as between Fireiron's decompositions and ELEVATE strategies. For example, a Fireiron `MatMul` specification, as discussed in Chapter 3, is implementable using a RISE matrix multiplication expression, as seen in Chapter 5. Simultaneously, a Fireiron decomposition could be implemented as an ELEVATE strategy, similarly to how we discussed the implementation of TVM's scheduling primitives as strategies. However, RISE does not yet support the generation of CUDA code. Notably, it does not support using specialized Tensor Core instructions required to achieve the same performance as the code generated by Fireiron's ad-hoc code generator. It also remains an interesting research question if and how Fireiron's data-movement related optimizations are expressible as RISE programs and whether these programs are derivable by applying ELEVATE strategies.

EXTENDING THE SET OF FIREIRON COMPUTE SPECIFICATIONS So far, Fireiron has only been used to generate efficient matrix-matrix multiplication kernels using the `MatMul` and `Move` specifications. We believe that the Fireiron's general approach applies to a broader set of computations beyond matrix multiplication kernels. Extending the set with additional compute specifications, such as supporting convolutions, point-wise matrix additions, activation functions like rectified linear unit, or other typical machine learning operations, is an obvious and beneficial extension to the Fireiron compiler. Generally, each building block provided in today's high-performance libraries, such as cuBLAS or cuDNN, could be defined as a new specification in Fireiron and optimized similarly to how we developed optimizations for the `MatMul` spec.

By extending the set of existing compute specifications, an interesting research question naturally emerges: It is still unclear if and how the set of existing decompositions would have to be adjusted or generalized to provide useful optimizations for the new computation specs. We believe that Fireiron's `tile` decomposition, which maps computations to the parallel compute hierarchy, is generally applicable because every efficient implementation must make use of the available hierarchy. We also believe that we can reuse most or all data-movement related functionality in Fireiron (i.e., the `move` decomposition and `Move` specs) because every high-performance GPU kernel must efficiently move data through the memory hierarchy.

However, whether the `split` and `epilog` decompositions are useful for optimizing other computational specs than `MatMul` remains to be seen. Most likely, `tile` and `split` are generalizable to decompose tensors of arbitrary dimensionality. These generalized decompositions then allow us to specify which dimensions to process in parallel (`tile`) and which dimensions are reduction dimensions (`split`). Similarly, the `epilog` decomposition, which is currently only defined for `MatMul` specs, is likely generalizable for decomposing other compute specs: Every high-performance implementation on a GPU, regardless of what operation exactly is implemented, must compute its results in registers and eventually move those results back to global memory again - this is the purpose of the `epilog` decomposition.

SUPPORT THE OPTIMIZATION OF COMPUTATION GRAPHS In its current version, Fireiron is only capable of generating code for a single input spec, e.g., a single matrix multiplication kernel for a given kernel-level `MatMul` spec as input. Even if we extend the set of existing specs, as discussed in the previous paragraph, this fact remains a limitation because most real-world computations require performing multiple operations in a sequence (e.g., artificial neural networks or image processing pipelines). Supporting the expression and optimization of computation graphs will, therefore, significantly extend Fireiron's capability to generate high-performance programs for computations occurring in domains such as machine learning.

Assume that we extend the set of available computation specs with an element-wise matrix addition and element-wise matrix multiplication (also called Hadamard-Product, its operator is written as $\circ$). Supporting the expression of computation graphs in Fireiron would then allow us to optimize more sophisticated programs such as a complete LSTM cell [79] expressed as follows:

LSTM – Long Short-Term Memory [79]: A Recurrent Neural Network (RNN) architecture commonly used in deep learning

$$f_t = \sigma_g(W_f * x_t + U_f * h_{t-1} + V_f \circ c_{t-1} + b_f)$$

Here, x is the input vector, h is the output of the previous iteration, W, U, V are weight matrices, and b is the bias vector. An LSTM cell is a regularly used high-level building block in more complex recurrent neural network architectures.

The optimization of computation graphs like an LSTM cell opens up the possibility to define and explore inter-operation optimizations like operator-fusion. For example, instead of computing the two matrix multiplications ($W_f * x_t$ and $U_f * h_{t-1}$) followed by the addition using three separate kernels, the whole computation can be fused into a single kernel. Operator fusion significantly reduces the amount of data moved to and from the GPU and, therefore, improves the overall performance.

We will have to extend Fireiron's set of decompositions with scheduling primitives similar to Halide's `compute_at` and `store_at` to support inter-operator optimizations like fusion. These primitives allow programmers to specify producer-consumer relationships in the overall computation graph.

Generally, we believe that having implemented Fireiron's specifications as RI**SE** expressions, as discussed in the previous paragraphs, would significantly simplify supporting computation graphs and their optimization. Since each specification is representable as a RI**SE** expression, we can express a computation graph by merely using the standard function composition. ELEVATE's rewrite rules like the `mapFusion` rule can then be used to define strategies implementing optimizations such as operator fusion.

PROVIDE A PRE-DEFINED OPTIMIZATION LIBRARY Fireiron allows us to specify typically applied high-performance optimizations as concise decomposition strategies. One advantage of Fireiron is that it allows experts to specify implementations at various granularities. For example, a performance engineer can use Fireiron to specify the optimization of a complete matrix multiplication kernel by fully decomposing a kernel-level **`MatMul`** spec. Alternatively, Fireiron also allows the decomposition of smaller parts of a kernel like data movements between levels of the memory hierarchy by decomposing the associated **`Move`** spec.

We argue that it is possible to reuse decompositions in different circumstances, which motivates providing a library of high-performance decompositions. For example, moving a buffer from global memory to shared memory on a GPU with the Maxwell architecture always has to be implemented in a specific way: For achieving coalesced global memory accesses, adjacent threads must load from adjacent memory addresses; to achieve efficient shared memory stores, the threads have to avoid bank conflicts using architecture-specific swizzles. Similarly, every hardware architecture poses specific constraints for data movement that must be satisfied for achieving high performance, regardless of which kind of computation uses the buffer that is being moved. Assume a performance engineer defines a strategy that moves a buffer from global memory to shared memory in such a way that loads and stores are performed efficiently on the Maxwell architecture (e.g., the strategies we discussed in Section 3.5). If we provide this strategy in a library, a Fireiron user can reuse this data-movement implementation when optimizing matrix multiplication or convolution computations without having to optimize the data-movement herself, a complicated task that may lie outside her expertise.

With the support of computation graphs, even more complex optimizations could be pre-defined by performance engineers. Optimizations across single operations such as operator fusion or persistent weight matrices are two popular techniques for optimizing machine learning sub-graphs that span multiple operations. Operator fusion fuses the computation of two operations that are typically executed using two separate kernels into one kernel. Fusion is typically applied to reduce the kernel-launching overhead and the need to move data to and from the slow off-chip global memory.

More advanced techniques such as persistent weight matrices could also be developed and provided as pre-defined Fireiron decompositions. In recurrent neural networks, the output of one iteration serves as the input of the next iteration while the weight matrices are reused across iterations. A common optimization for computing recurrent neural networks is keeping the weight matrices, whose values do not change across iterations, persistently in lower memory-hierarchy levels such as shared memory to reduce data movements. Similar to how we define common optimizations for matrix multiplications as decompositions in Fireiron, one could provide a whole set of advanced optimization techniques for computation graphs as a pre-defined library.

AUTOMATIC OPTIMIZATION EXPLORATION Finally, another useful extension to Fireiron would be the application of automated search techniques to automatically find efficient decompositions for a given specification or computation graph. Techniques like sketch-based program synthesis, as used in SwizzleInventor [128], Evolutionary Algorithms, Deep Reinforcement Learning, or basic auto-tuning techniques for tuning numerical parameters such as tile sizes are applicable at multiple levels of our optimization pipeline. Related domain-specific compilers, including Halide [141], TVM [29], and Tensor Comprehension [175], already apply some of these techniques to optimize their computations automatically.

The work on optimizing stencil computations in LIFT, which we discussed in Chapter 4, already applied basic auto-tuning techniques to tune numerical parameters such as tile sizes. We believe that it is possible to further automate the exploration process beyond tuning numerical parameters by, for example, letting a reinforcement learning agent experiment with applying decompositions in different ways while using the generated program's runtime as the reward in the learning process. Enabling automated exploration of high-performance decomposition strategies for Fireiron would significantly decrease the need for human experts to develop the strategies in a trial and error process.

6.3.2 *Enhancing the* ELEVATE *Language*

In the following, we discuss potential enhancements for ELEVATE.

A TYPE SYSTEM FOR STRATEGY LANGUAGES Currently, we implemented ELEVATE using a shallow embedding in Scala without a dedicated type system. One of the key aims of our ELEVATE design is the ability to reason about strategies. All of the basic rewrite rules we implemented in ELEVATE are easy to prove correct (i.e., all rules preserve the semantics of the input program, as discussed in [68]). Combining semantics-preserving rewrite rules to more sophisticated strategies using ELEVATE's traversals and combinators thus also always results in a semantics-preserving strategy.

Still, it is hard to reason about the result of applying a strategy to an arbitrary program due to a missing type system. Furthermore, it is straightforward to define completely useless strategies. Two simple examples are the `fail` strategy and `repeat(id)`. The `fail` strategy always fails regardless of the input program, and the `repeat (id)` strategy never terminates as it is always possible to apply the `id` strategy. Both strategies are semantics-preserving as they do not change the input program. However, failure and non-termination are generally undesired properties of strategies in the process of program optimizations.

Mametjanov, Winter, and Lämmel [101] introduce a type systems in the domain of term-rewriting systems that allows compilers to statically check the behavior of a strategy before applying it to a program. Developing such a type system for ELEVATE would allow to statically reason about the effect of applying a strategy and potentially statically rule out the definition or application of useless strategies like the two discussed examples.

OPTIMIZING STRATEGIES: OPTIMIZING ELEVATE WITH ELEVATE We have shown in Section 5.6.1 that defining complex optimizations as rewrite strategies quickly leads to the application of thousands of rewrite steps. In our case studies for optimizing matrix multiplications, the time to perform these was not a limiting factor yet. However, using ELEVATE to optimize more complex computations like full image-processing pipelines or large neural networks will likely increase the number of required rewrite steps.

One possibility to reduce the required steps to rewrite a program is to optimize the optimization strategy itself. Since ELEVATE can be used to rewrite programs in any programming language, ELEVATE can also be used to rewrite ELEVATE programs - performing meta optimizations. For example, assume we apply the following strategy to a simple RISE program:

```
(body(id) ‘;‘ body(id))(fun(x,x))
```

This strategy requires four rewrite steps: 1) traverse to the body of the function abstraction, 2) apply the `id` strategy, 3) traverse to the body again, 4) apply the `id` strategy again. However, without changing the strategy's effect we could also apply the following strategy which requires one step less:

```
(body(id ‘;‘ id))(fun(x,x))
```

Note the similarity to the `mapFusion` strategy for RISE programs. Here, we fused the two traversals to the body of a function abstraction. The following rewrite rule implements this kind of transformation in ELEVATE:

```
def bodyFusion: Strategy[Elevate] = e => e match {
  case seq(body(f), body(g)) => Success(body(seq(f,g)))
  case _                     => Failure(bodyFusion) }
```

Similarly, we can define more simplification rules that reduce the number of performed rewrite steps:

```
def idSimplification: Strategy[Elevate] = e => e match {
  case seq(id, id)     => Success(id)
  case lChoice(id,id) => Success(id)
  case body(id)        => Success(id)
  case _               => Failure(idSimplification) }
```

Applying this rule exhaustively to our original input strategy reduces the performed rewrite steps from four to only a single one:

```
normalize(idSimplification)(body(id) ‘;‘ body(id)) // == id
```

As soon as the time required to rewrite becomes infeasible due to requiring too many rewriting steps during program optimization with ELEVATE, meta-optimizing the strategies themselves might be a potential solution for making rewriting itself more efficient. This type of optimization can also be regarded as a kind of compile-time optimization for ELEVATE.

6.3.3 *Enhancing the Expression and Optimization of Stencils in* Lift

Finally, we discuss potential extensions to our high-performance compilation approach for stencil computations in Lift and to generating code using Lift and its spiritual successor RISE in general.

ADDING OTHER BACK-ENDS AND HOST-CODE GENERATION
Both, Lift and RISE, currently are only capable of generating sequential C code (optionally annotated with OpenMP pragmas for parallelization) or parallel OpenCL code. It makes sense to extend the set of back-ends for Lift or RISE to support a broader range of possible target architectures. For example, adding a CUDA back-end would enable targeting NVIDIA GPUs specifically to target CUDA's WMMA-API to use Tensor Cores, which is not possible when generating OpenMP or OpenCL code. A CUDA back-end would also enable the use of inline PTX assembly to target the lower-level `mma.sync` instructions, which expose more fine-grained control over the Tensor Cores. To target Google's TPU, we would have to add an XLA back-end because XLA is currently the only publicly available API that allows to target these accelerators.

Additionally, instead of only generating kernel code, generating the host code required to launch those kernels as well would open up even more opportunities. For example, in cases where a single kernel does not fully saturate the targeted CPU or GPU, it is beneficial to run additional independent kernels concurrently on the same device to make use of the idle hardware resources. When executing computations graphs like neural networks, multiple operations can often be computed concurrently because they do not depend on each other. For example, in CUDA, this is achieved by launching each kernel using a separate so-called stream. So far, this must be done manually because neither RISE nor Lift is capable of generating the required host code.

EXPRESSING AND OPTIMIZING ITERATIVE STENCILS In Chapter 4, we only expressed and optimized single stencil iterations. Typically, stencil computations are performed iteratively to, for instance, compute multiple time steps in a physics simulation. Performing the same stencil computation iteratively enables further potential for advanced optimizations. Popular techniques for optimizing iterative stencil computations especially include various versions of tiling to exploit the temporal locality across stencil iterations. A vast amount of work exists on different ways to tile iterative stencil computation, including diamond, hexagonal, skewed and other versions of overlapped tiling, that aim to improve the performance of applying multiple time steps.

Expressing iterative-stencil computations in Lift does not require to add more algorithmic primitives. As we have shown, a single stencil iteration is expressible using *pad*, *slide*, and *map*. The most straightforward way to express multiple stencil iterations is to unroll them manually:

```
// a simple 3-pt stencil computation expressed in lift/rise
val stencil = pad(1,1,clamp) |> slide(3,1) |> map(reduce(0,+))
// performing three stencil iterations
input |> stencil |> stencil |> stencil
```

Another way to express iterative stencil computation is to use Lift's *iterate* primitive that expects a natural number specifying the number of iterations and the function describing the stencil computation that is applied to apply to the input. The *iterate* primitive applies the given function to the input and uses the computed output as input for the next iterations. Using *iterate*, the three stencil iterations are expressible in Lift as follows:

```
// performing three stencil iterations
input |> iterate(3,stencil)
```

Implementing tiling optimizations for iterated stencil computations requires to rewrite and fuse multiple separate time steps. There exists a wide variety of multi-dimensional tiling approaches for iterative stencils, as we will discuss in Section 7.4. We believe that again, multi-dimensional tiling for iterative stencil computation is expressible using simple rewrite rules similar to how we express overlapped tiling, as discussed in Chapter 4.

6.3.4 *Using MLIR as a Unifying Framework*

Google has recently introduced the Open Source project MLIR (Multi-Level Intermediate Representation) [93]. MLIR is part of the LLVM project and is a framework and toolkit for building compilers. One of the key features of MLIR is the ability to define custom so-called *dialects*. A dialect is essentially an IR implemented in the MLIR framework and consists of a set of custom operations and types. MLIR already provides a set of dialects, including a TensorFlow IR, an XLA IR, a polyhedral-inspired IR, and an LLVM IR. Additionally, MLIR allows compiler developers to define transformations that translate between dialects to gradually lower computations expressed in one IR to another IR.

Instead of implementing the intermediate representations and languages introduced in this thesis (Fireiron, Lift, RISE, and ELEVATE), as embedded domain-specific languages in Scala, we believe that it would be a signifcant improvement to reimplement those as MLIR dialects. The first distinct advantage is that this enables the use of the existing MLIR front- and back-ends. For example, instead of developing our own TensorFlow front end, which translates TensorFlow computations into Fireiron specifications, we could define MLIR transformations that translate computations expressed in the already existing TensorFlow dialect into our Fireiron dialect. Similarly, we could then define additional transformations to lower our Fireiron dialect into RI**SE** and ELEVATE dialects that eventually are lowered into the existing LLVM dialect. MLIR's LLVM dialect can ultimately be compiled to code targeting a wide variety of hardware accelerators, including CPUs, GPUs, and Google's TPU. Essentially, the whole compilation infrastructure shown in Figure 6.4 is implementable within the MLIR framework.

Using MLIR as the unifying underlying infrastructure would also significantly reduce the effort required for potential users to benefit from and adapt the concepts introduced in this thesis. For example, it would be interesting to try to apply an ELEVATE-like approach towards program optimizations by composing transformations as rewrites for dialects in MLIR. ELEVATE's capability of being able to rewrite arbitrary programming languages (i.e., dialects) is especially useful in the context of MLIR, where one can expect to find a plethora of different dialects in the near future. A single principled mechanism to drive the translation between MLIR dialects will be useful. Our performed case-study shows that ELEVATE is able to deliver precise control over transformations while providing easy-to-use high-level abstractions.

Finally, similar to how LLVM significantly influenced the construction of general-purpose compilers, we believe MLIR can influence the construction of domain-specific compilers. We already see many similarities between concepts implemented in MLIR and the work discussed here. For example, the proposed *linalg* MLIR dialect [100] exhibits similarities with our design of Fireiron, especially its ability to gradually decompose computations to smaller sub-computations, as we will discuss in the final Chapter 7. Generally, we believe that making the concepts and ideas discussed in this thesis available by implementing them as dialects in a shared MLIR infrastructure accelerates the ability of a broader audience to achieve high-performance domain-specific compilation without constructing domain-specific compilers.

7 RELATED WORK

In this final chapter, we compare our work presented in the previous chapters with existing related work. Specifically, we discuss related domain-specific compilers, high-performance code generation approaches, rewrite-based optimization approaches, and high-performance frameworks for stencil computations.

We begin with a discussion of various high-performance domain-specific compilation approaches (Section 7.1) and discuss how they relate, how they influence, or how they differ compared to the contributions made in this thesis. In Section 7.2, we briefly discuss approaches for simplifying the development of domain-specific languages and compilers. In Section 7.3, we introduce approaches to automating the optimization process and how to potentially apply those approaches to the compilers described in the previous chapters. We conclude this chapter with a comparison of existing stencil-specific frameworks and our contributions made in Chapter 4 (Section 7.4), and a comparison of related rewriting-based approaches (Section 7.5), where we especially highlight the differences to our contributions made in Chapter 5.

7.1 HIGH-PERFORMANCE DOMAIN-SPECIFIC COMPILATION

We begin by discussing existing related approaches to achieving high-performance domain-specific compilation and compare those to our work introduced in the previous chapters.

7.1.1 LIFT *and* RISE

The contributions made in this thesis, especially in Chapter 4 and Chapter 5, extensively build upon earlier work on LIFT [162] and RISE [8] – LIFT's spiritual successor. LIFT was introduced in 2015 [161] as a functional, pattern-based approach for achieving performance-portability on accelerators. Before our work on extending LIFT for expressing domain-specific stencil computations [70] (Chapter 4), LIFT has mainly been used for generating high-performance dense linear algebra code [150, 152, 162, 166]. Steuwer, Remmelg, and Dubach [167] describe LIFT's compilation process for generating high-performance OpenCL code.

Our work on expressing stencil computations in Lift has shown that its functional IR is suitable for expressing computations of various application domains. Lift has since been used for expressing computations of multiple other domains, including sparse linear algebra [74, 132], machine learning [103, 104], high-level synthesis targeting FPGAs [89], acoustics simulations [168], and Fast-Fourier Transformations [88]. Many of these extensions to Lift use the *pad* and *slide* primitives introduced in Section 4.3.2. Additionally, Remmelg et al. [151] discuss how to predict the performance of Lift programs, and Stoltzfus et al. [169] present an extended discussion about stencil computations in Lift, including expressing advanced tiling optimizations. Lift has also been used for the parallelization of legacy code [58].

RISE was introduced in 2017 as the spiritual successor to Lift and is used in Chapter 5 as the target language to be optimized by ELEVATE rewrite strategies. RISE introduces a *strategy-preserving compilation* approach [8]: The IR contains all relevant information about low-level code-generation details such as memory locations and parallelism to alleviate the compiler from making implicit implementation decisions. Position-dependent arrays in RISE [131] allow the computation of triangular matrix-vector multiplication and the avoidance of unnecessary out-of-bound checks for specific stencil computations.

7.1.2 *Schedule-Based Compilation*

HALIDE [138–141] is a domain-specific schedule-based compiler for generating high-performance image-processing pipelines. Halide has pioneered the development of schedule-based compilers that separate computation (typically called the *algorithm*) from its optimization described in a so-called *schedule*. We already introduced and compared our work extensively against Halide, especially in Chapter 1, Chapter 3, and Chapter 5. In the following, we briefly summarize our findings.

The Fireiron compiler, introduced in Chapter 3, follows Halide's schedule-based design but targets the optimization of matrix multiplications on NVIDIA GPUs specifically. Optimizing data movements, which are particularly important for achieving high performance on GPUs, is challenging in Halide because it treats data movements as second-class citizens. Fireiron (introduced in Chapter 3) is the first domain-specific schedule-based compiler that treats data movement as first-class citizens, which allows the optimization of both computations and data movements using scheduling primitives.

Halide's IR is -by design- specific to the domain of image processing pipelines. However, being domain-specific restricts its reusability and forces compiler developers to start from scratch for targeting new application domains. In Chapter 4, we introduced a generic IR for expressing domain-specific computations. We specifically demonstrated how to extend it for expressing stencil computations, which are an integral part of image processing pipelines.

In Chapter 5, we demonstrated how to implement scheduling languages as optimization strategies composed of simple rewrite rules. In contrast to using a pre-defined scheduling API for expressing program optimizations like Halide provides, defining composable rewrite strategies leads to a more flexible, scalable, and extensible program optimization approach.

TVM [29] is another domain-specific compiler inspired by Halide; it provides an end-to-end compilation stack for deep learning applications. TVM contains front-ends for popular machine learning frameworks, including TensorFlow [2, 3], and PyTorch [126], and provides back-ends for generating high-performance code targeting various hardware architectures including CPUs and GPUs.

In Chapter 3, we compared Fireiron against TVM and especially demonstrated the need for first-class data movement optimizations in scheduling languages; those are currently missing in the design of existing schedule based compilers such as Halide and TVM. In Chapter 5, we conducted an in-depth case study where we demonstrated how to implement TVM's scheduling primitives as ELEVATE strategies targeting RI**SE** programs. By implementing TVM's scheduling primitives as composable and extensible rewrite strategies, we showed that it is possible to design similar or even more expressive scheduling primitives (e.g., multi-dimensional tiling) from a small collection of simple rewrite rules. The performance achieved by the code optimized with our rewrite optimization strategies is confirmed to be competitive to code optimized and generated by TVM.

TIRAMISU [10] is a schedule-based polyhedral compiler for optimizing machine learning, dense- and sparse linear algebra, and image processing pipelines on different hardware architectures. We separately compare against polyhedral compilation in Section 7.1.3. In Section 6.3, we discussed how we could extend Fireiron to support the expression of more computations; however, data-dependent computations such as sparse linear algebra, as supported by Tiramisu, lie outside the scope of Fireiron's design. Tiramisu's IR is separated into four distinct layers: *Abstract Algorithm*, *Computation Management*, *Data Management*, and *Communication Management* for separating algorithms, loop-transformations, data layouts, and communication. In contrast to this separated design, in Chapter 4 and Chapter 5, we introduced LIFT and RI**SE**. These two functional pattern-based approaches capture all information required for high-performance code generation in the same IR, without such separation. This design enables *strategy-preserving code generation* [8] in a compiler that does not need to make implicit optimization choices because all required information is already present in the IR.

Another key difference to the work discussed in this thesis is Tiramisu's capability to target distributed machines. All approaches introduced in the previous chapters generate single compute kernels only, and are thus not yet suitable for generating code that runs on multiple compute nodes in a distributed machine.

GRAPHIT [192] is a domain-specific language and compiler developed by MIT and Adobe. In contrast to the work described in the previous chapters, GraphIt's IR and scheduling primitives are designed to express and optimize graph algorithms. Their high-level algorithm language enables the expression of edge processing code, edge traversals, and vertex trackings; the scheduling language exposes a graph iteration space model that enables the composition of graph-algorithm optimizations. GraphIt (as well as Halide [107] and TVM) provides an auto-tuner to find high-performance schedules automatically. As described in Section 6.3, both Fireiron and ELEVATE could be extended with auto-tuning techniques for enabling the automatic optimization of programs similar to GraphIt's approach.

CHILL [28] was introduced in 2008 before Halide and is a framework for composing traditional loop transformations. It provides a so-called *transformation script* for prescribing how to optimize the target code. CHiLL's transformation scripts are closely related to today's scheduling primitives, and CHiLL can be regarded as a precursor to today's popular schedule-based compilers in general. CUDA-CHiLL [159] is an extension to CHiLL that allows programmers to target GPUs. Similar to Fireiron's `.move` decomposition, CUDA-CHiLL provides a `datacopy` primitive that initiates a data movement to a different level in the memory hierarchy. CUDA CHiLL's `datamovent`, however, relies on fixed built-in strategies to move the data through the memory hierarchy.

SCHEDULE TREES [178] are an explicit representation of the execution order of program statements in the polyhedral model. Modifying the execution order of a program is expressed by modifying the schedule tree using predefined graph-transformations such as *tile* or *split*.

We separately discuss *Polyhedral Compilation* in the following section, but briefly mention schedule trees here already because of their similarity to schedule-based compilers, which also allow programmers to describe how a program is executed explicitly.

SUMMARY Generally, when comparing existing domain-specific schedule-based compilers with our approaches to achieving domain-specific compilation, three significant differences persist with varying degrees:

- Fireiron's ability to precisely orchestrate data movements using high-level decompositions is unmatched compared to existing scheduling languages.
- By design, the IRs of existing domain-specific compilers are specific for one or only a few application domains. With LIFT, we demonstrated the possibility to express domain-specific computations using a generic, pattern-based, functional IR.
- Similarly, the scheduling primitives provided by schedule-based compilers are inherently specific for the applications they aim to optimize and hard to extend. With `ELEVATE`, we demonstrated how to define domain-specific scheduling languages as compositions of simple rewrite rules.

7.1.3 *Polyhedral Compilation*

Polyhedral compilation is a technique for representing and optimizing programs that contain nested loops and arrays as parametric polyhedra [51, 97, 189] or Pressburger relations [86, 177]. This mathematical representation of programs enables compilers to perform combinatorial and geometrical transformations for optimizing the performance.

A comprehensive summary of polyhedral compilation and related publications can be found online at `polyhedral.info` *[59].*

PPCG [179] (Polyhedral Parallel Code Generation) is a parallelizing polyhedral compiler that generates CUDA. In Section 4.5.4, we compared LIFT-generated stencil kernels against code parallelized using PPCG. Specifically, PPCG is a source-to-source compiler that translates annotated C-code into CUDA host and device code. PPCG compiles loop-nests annotated with `pragma scop start/end` into CUDA kernels running on a GPU, and it inserts host code for data transfers to and from the device into the remaining C code. The approaches introduced in the previous chapters all focus on the generation of high-performance device code In contrast to PPCG, they do not generate nor optimize the host code required to execute the generated kernels.

DIESEL [49] is a domain-specific language and compiler for generating high-performance linear algebra and neural network codes. In Diesel, computations are expressed as a Directed Acyclic Graph (DAG), which is compiled to an internal polyhedral IR that allows to perform optimizations, including the fusion of multiple operations into a single kernel. As mentioned in Section 6.3, operator-fusion is a promising optimization to explore for the compilers introduced in this thesis. The polyhedral model is particularly suitable for implementing such optimizations because it operates by purely transforming nested loops. Diesel's fusion approach could be adapted by expressing the operator-fusion transformations as ELEVATE rewrite rules that operate on nested RISE primitives. Like Fireiron, Diesel also generates high-performance CUDA matrix multiplication implementations. However, Diesel has just recently been updated to support using Tensor Cores on modern NVIDIA architectures [12].

TENSOR COMPREHENSIONS [175, 176] As already discussed in Section 1.2.1, Tensor Comprehensions (TC) is a domain-specific language and compiler for the generation of high-performance machine learning kernels. Tensor Comprehension's DSL allows programmers to specify computations using a tensor notation close to the mathematics of deep learning. The underlying compiler uses a hierarchical IR that includes features from both Halide as well as the polyhedral model. TC provides a fully automatic optimization approach that includes traditional auto-tuning and evolutionary algorithms. In Section 4.5.2, we briefly explained how we automatically optimize LIFT programs for generating high-performance OpenCL kernels. TC's automatic optimization approach is much more sophisticated and could be adapted for the compilation of LIFT or RISE programs. However, the tools and compilers presented in this thesis, especially Fireiron and ELEVATE, aim to empower the user to decide how to optimize programs, rather than applying optimizations transparently.

7.1.4 *Algorithmic Skeletons*

Algorithmic Skeletons have been introduced in 1988 [37] and describe often-recurring patterns in parallel programming. Essentially, an algorithmic skeleton can be viewed as a higher-order function used in imperative parallel programming models: A skeleton describes the overall structure of the computation, and it is parameterized by one or more other functions specifying the problem-specific computation.

For example, one of the originally proposed skeletons is the *divide & conquer* skeleton that describes the well-known partitioning of a problem into multiple sub-problems that can be solved in parallel. This particular skeleton expects four functions that describe the problem-specific instance: *indivisible*, *f*, *split*, and *join*. Here, *indivisible* is a predicate determining whether the current element can be further divided; *f* is the computation performed in parallel on the indivisible elements; *split* specifies how to divide the current element, and *join* is its inverse.

Algorithmic skeletons are often categorized as either data-parallel, i.e., performing the same task on different data, or task-parallel, i.e., performing multiple different tasks in parallel. In the following, we briefly introduce libraries based on algorithmic skeletons aiming to achieve high-performance on different hardware architectures.

SKELCL [165] is an OpenCL-based skeleton library for high-level GPU programming that provides four basic skeletons: *map*, *zip*, *reduce*, and *scan*. For each of these skeletons, SkelCL provides an efficient parameterizable OpenCL implementation targeting GPUs. As we have shown in the previous technical chapters, there is often no one-size-fits-all implementation that achieves high performance across different hardware architectures. However, SkelCL, like other skeleton libraries, relies on a fixed implementation per skeleton designed to achieve high-performance on the targeted architecture but fails to achieve similar performance on different architectures. Another restriction of the SkelCL approach is the missing support for nesting skeletons. In SkelCL, every skeleton is compiled to a separate OpenCL kernel. Due to being designed as actual functional programming languages, LIFT and RISE allow the expression of similar computations (they support the same skeletons/higher-order functions except *scan*), but allow arbitrary nestings and compositions of primitives.

Steuwer and Gorlatch [163] describe an extension to SkelCL for supporting stencil computations by introducing a new *stencil* and *mapOverlap* skeleton. In Section 4.3.2, we described how a stencil computation (and therefore the stencil algorithm skeleton) is decomposable into fundamental and generic building blocks. This approach achieves more flexibility by using domain-agnostic primitives for expressing domain-specific computations.

Unlike the compilers introduced in this thesis, SkelCL is capable of targeting multi-GPU systems.

MUSKET [155, 190] is a domain-specific language based on algorithmic skeletons for targeting distributed systems. Musket is embedded in C++ and provides popular data-parallel skeletons, including *map*, *fold*, *mapFold*, or *zip*. Here, *mapFold* is a special-case skeleton for expressing the composition of *map* and *fold*. A key difference to the approaches discussed in the previous chapter is Musket's ability to target distributed systems by generating MPI code and its Eclipse IDE integration for simplified development.

MUESLI [90] is another typical skeleton library for high-level programming of heterogeneous clusters that contain multi-core CPUs, GPUs, or Xeon Phi co-processors. MUESLI provides a high-level interface to the programmers and encapsulates the low-level skeleton implementations in library functions. Since its introduction, it has been extended to support both data-parallel and task-parallel skeletons [91], including skeletons for sparse computations [33].

ACCELERATE [23] is a high-level domain-specific language embedded in Haskell for generating high-performance CUDA code. Accelerate exposes typical data-parallel skeletons as higher-order functions in Haskell that can be parameterized with scalar code to instantiate the CUDA templates designed for each skeleton. Due to being embedded in the pure functional language Haskell , Accelerate has an interesting compilation workflow that differs from traditional skeleton approaches. The surface language facing the Accelerate programmer is implemented using type families and GADT's (Generalized Algebraic Data Types) and is eventually compiled into a nameless de Bruijn representation [41].

MDH [144, 145] is another promising skeleton-based approach that exposes only a single pattern (or skeleton) to the user, a so-called *multi-dimensional homomorphism (MDH)*. MDH is based on the Bird-Meertens Formalism [13, 14, 102] and generalizes their one-dimensional *list homomorphisms* to work on multi-dimensional arrays (MDA). The multi-dimensional homomorphism pattern (combined with a specific *view* pre-processing for gathering required data elements) is capable of expressing the typically used data-parallel skeletons *map* and *reduce*. Additionally, more complicated computations such as matrix multiplications, generalized tensor contractions, or stencil computations are expressible using MDH too. Following the algorithmic skeleton approach, MDH is implemented using a fixed code template that, however, exposes several tunable parameters that can be tuned at runtime once the skeleton is instantiated for the specific problem to compute.

OTHER ALGORITHMIC SKELETON LIBRARIES. Similar to the previously discussed skeleton-based approaches, there exist several other skeleton libraries that aim at achieving high performance on various parallel architectures. They differ in the programming languages they support or are embedded in, the skeletons they provide, and the architectures they support. Notable examples of further skeleton libraries are SkePU [50], FastFlow [4].

7.1.5 *Other High-Performance Code Generators and Libraries*

In the following, we discuss other related compilers targeting high-performance code generation that are neither schedule-based nor based on the polyhedral model. Additionally, we analyze related libraries for achieving high performance on different architectures.

FUTHARK [77] is a functional data-parallel array language that compiles to high-performance OpenCL code. Futhark is based on list homomorphisms pioneered by Bird and Meertens [13]. Like RISE and Lift, it provides built-in primitives (second-order array combinators) like *map* or *reduce* for expressing parallel computations. Futhark's compilation is based on rewrite rules for exploiting parallelism by flattening nested data structures to optimize the locality of reference. Generally, Futhark's compilation follows a built-in strategy, e.g., based on the *Incremental Flattening* algorithm [78], that could also be implemented using ELEVATE. A key difference to our work is the expression of stencil computation, which relies on unsafe explicit array indices (e.g., `#[unsafe] input[i-1,j+1]`) instead of using primitives like *pad* or *slide*. Futhark's approach allows programmers to accidentally perform out-of-bounds memory accesses, which are prevented by design using our functional and pattern-based approach.

NOVA [38] is a functional language and compiler targeting multicore CPUs and GPUs. NOVA is statically typed, polymorphic, and similar to Lift and RISE, provides a set of well-known higher-order functions such as `map`, `reduce`, or `scan`, for expressing parallelism. Another similarity to RISE and Lift is the support of nested data-structures and nested parallelism. NOVA's compiler follows a more traditional pass-style optimization pipeline design, and it applies optimizations such as *inlining* and *common subexpression elimination*.

The key difference to the work introduced in this thesis is NOVA's fixed compilation strategies for higher-order primitives. For example, the `scan` operation is always computed in three steps: 1) a partial reduction, an intermediate scan, and a third step to compute the final result. In Fireiron, decompositions allow programmers to specify precisely how an operation shall be computed. In Lift and RISE, low-level primitives introduced by rewrite rules (potentially using ELEVATE) allow programmers to specify how computations shall be parallelized.

SPIRAL [122, 137] AND FFTW [53] are two automatic high-performance approaches in the domain of digital signal processing (DSP), which includes Fast-Fourier Transformations (FFT). Like Lift and RISE, Spiral attempts to achieve performance portability by expressing computations in an IR (the so-called *operator language*) that is optimized using semantics-preserving rewrite rules. In an automated search process, Spiral generates efficient DSP code targeting hardware devices ranging from mobile devices to supercomputers.

FFTW is a C subroutine library for computing the discrete Fourier Transform. FFTW applies an automated optimization process in which an *executor* creates a so-called *plan*. It describes how the algorithm is executed by combining composable blocks of C code (*codelets*) for minimizing the runtime on the targeted hardware.

In contrast to the approaches in this thesis, both Spiral and FFTW are specific to the domain of digital signal processing.

CUTLASS [116] is a CUDA template library developed by NVIDIA, which provides high-performance matrix multiplication implementations. Like Fireiron, CUTLASS decomposes matrix multiplication into reusable sub-problems and exposes tunable parts (such as tile sizes) as composable C++ templates. In contrast to Fireiron, the strategies implemented in CUTLASS are pre-defined. The user has only limited control over how exactly the algorithm shall be decomposed and executed on the GPU.

CUTLASS supports Tensor Cores and a wide variety of different data types beyond what is supported in the current implementation of Fireiron. Huang, Yu, and Geijn [80] describe an implementation of the high-performance *Strassen* algorithm for computing matrix multiplications in CUTLASS.

TRADITIONAL HIGH-PERFORMANCE LIBRARIES Every hardware vendor typically provides a collection of high-performance libraries that enable users to achieve high performance for common computational building blocks. Those libraries contain manually tuned algorithm implementations written in low-level machine assembly for the targeted hardware architecture. For example, NVIDIA provides cuBLAS that contains high-performance implementations for common dense linear algebra building blocks such as matrix multiplication, or cuDNN, a high-performance library for machine learning computations. Similarly, Intel provides the *Math Kernel Library* (MKL [82]), and AMD provides a collection of optimized CPU libraries (AMD Optimizing CPU Libraries (AOCL) [1]).

7.2 BUILDING DOMAIN-SPECIFIC LANGUAGES AND COMPILERS

In this section, we discuss related approaches for constructing domain-specific languages and compilers.

MLIR [93] (Multi-Layer Intermediate Representation) is a novel approach for building reusable and extensible compilers recently introduced by Google. We already extensively discussed MLIR in Section 1.2.1 and Section 6.3.4, and we only briefly summarize the discussion here. MLIR allows compiler developers to define domain-specific IRs as so-called MLIR *dialects* that can interact (hence, multi-level) with other existing dialects. Declarative rewrite rules encode optimizations and are used to transform dialects.

Especially MLIR's *linalg* dialect [100] is closely related to our work on Fireiron because it allows programmers to strategically decompose linear algebra computations, such as matrix multiplications, into tiled smaller versions of the original problem. Bondhugula [17] presents an extensive case study about optimizing matrix multiplications targeting CPUs in MLIR. The optimizations applied are similar to those discussed and applied in our case study (Section 5.6) on optimizing R**ISE** matrix multiplication expression with ELEVATE.

The work described in this thesis has been conducted before the introduction of MLIR. In Section 6.3.4, we already highlighted the potential benefits of reimplementing our approaches in MLIR to use it as a common compiler infrastructure. Lücke, Steuwer, and Smith [99] describe the implementation of a restricted version of the R**ISE** IR and compilation approach in MLIR. More recently, Gysi et al. [64] describe the design and implementation of a stencil-specific dialect and its optimization in MLIR. This work is closely related to our approach to expressing and optimizing stencil computations in LIFT as described in Chapter 4.

ANYDSL [94] is a programming system that generally targets a similar goal as described in this thesis: Achieving high performance on parallel hardware without the need to construct domain-specific compilers. AnyDSL heavily relies on *partial evaluation* [55, 56], a technique for evaluating programs in two stages, leading to program optimization by specialization: In the first stage, the program is evaluated on its static inputs (e.g., values for variables known at compile-time are directly injected in the program). The first stage produces a residuum that is fully evaluated on the remaining inputs in the second stage. Partial Evaluation is also closely related to metaprogramming (e.g., Template Meta Programming (TMP) in C++ or Scala Lightweight Modular Staging [156, 157] (LMS)), or the shallow embedding of domain-specific languages [81].

AnyDSL is built upon Impala [105], a general-purpose functional language that essentially acts as syntactic sugar for AnyDSL's CPS-based (Continuation-Passing Style [154, 171]) IR Thorin [95]. Relying on a partially-evaluable language (Impala) allows a programmer to implement desired functionality as an Impala library, rather than defining a new DSL and an associated domain-specific compiler. The advantage of this approach is that the library functions are evaluable in the first partial-evaluation stage, which is similar but more powerful than using compiler pragmas where code is simply inlined but not necessarily evaluated. For example, AnyDSL provides so-called *generators*, i.e., functions known to the Impala compiler, such as `parallel` or `vectorize` that a programmer can use to annotate an Impala program. These generators are closely related to the ELEVATE `parallel` and `vectorize` strategies introduced in Section 5.5.1 that rewrite RISE programs into parallel versions.

In AnyDSL, the programmer generally decides which functions are partially evaluated using different annotations. The authors also demonstrate the implementation of various DSLs using AnyDSL, including a Halide-like DSL for image processing. Özkan et al. [124] extend AnyDSL for targeting FPGAs.

Optimizing programs by using partial evaluation in AnyDSL or by using rewrite strategies as introduced in this thesis are two similar approaches aiming to achieve the same goal by leveraging fundamentally different techniques.

DELITE [170] is a general-purpose programming language embedded in Scala that aims at simplifying the process of DSL development by providing parallel patterns, optimizations, and code generators. Programs written in Delite are automatically optimized and compiled to parallel C++, OpenCL, or CUDA programs and can simultaneously run on CPUs and GPUs. Delite's parallel patterns largely overlap with the patterns available in LIFT and RISE, or languages like Futhark or NOVA. Delite automatically performs generic optimizations like Dead Code Elimination or Common Subexpression Elimination and encodes domain-specific optimizations as rewrite rules similar to LIFT and RISE.

The programmer's control over Delite's optimization approach is limited. One can programmatically disable specific pre-defined generic optimizations (e.g., Dead Code Elimination) and can define custom rewrite rules. The rewrite rule application is only controllable by implementing a fixed interface consisting of *Traversals* and *Transformers*. With ELEVATE, we formalized and generalized the concepts of transformers (i.e., combinators, see Section 5.4.4) and traversals (see Section 5.4.5). In contrast to Delite, our approach offers unlimited control over the rewrite process to the programmer.

SPOOFAX [85] is a platform for developing DSLs. Spoofax originates from ASF+SDF [21] (see Section 7.5) and incorporates multiple tools for simplifying the development of DSLs, associated domain-specific compilers, and even IDE integrations. For example, one of the included tools is Stratego [184], which has significantly inspired the design of ELEVATE, and which will be discussed in more detail in Section 7.5. With Spoofax, compiler developers still have to construct a new domain-specific compiler from scratch for every new domain-specific language. However, Spoofax provides a reusable set of common tools for simplifying this task. In this thesis, we have shown a generic compiler IR and extensible optimization strategies that can be reused for achieving domain-specific compilation without constructing new compilers for every new language.

7.3 AUTOMATIC PROGRAM OPTIMIZATION

Besides using basic auto-tuning for optimizing LIFT programs as described in Section 4.5.2, none of the approaches introduced in this thesis uses techniques for automatically achieving high performance on parallel hardware. Nevertheless, as discussed in Section 6.3.1, our approaches are amenable for extending and applying automated search and tuning techniques for generating high-performance programs. In the following, we briefly discuss such related approaches towards automatically optimizing programs.

OPENTUNER [6] AND ATF [143] OpenTuner is an extensible framework for program auto-tuning. It enables the description of search spaces and provides an ensemble of search techniques that work together to find optimal points in the search space with respect to a user-provided cost function. The Auto-Tuning Framework (ATF) builds on top of OpenTuner and enables the expression of inter-parameter dependencies in the search space. Expressing dependencies between tuning parameters is particularly useful for tuning OpenCL or CUDA programs. Here, the number of threads in a block (one tunable parameter) must evenly divide the overall number of threads launched (another tunable parameter). We used ATF to tune thread and block sizes as well as tile sizes and the PPCG code generation in the evaluation of stencil computations expressed in LIFT, as discussed in Section 4.5.2.

There exists a wide variety of other classical auto-tuning approaches, including ATLAS [188] for tuning linear algebra applications, HyperMapper [16] for design space exploration in 3D scene understanding, or CLTune [121], a generic auto-tuner for OpenCL kernels.

PETABRICKS [108] AND TANGRAM [26] Petabricks is a parallel language and compiler that provides algorithmic choice (i.e., how an algorithm is implemented) as a first-class concept available to the user. In Petabricks, a user informs the underlying compiler about different implementation choices for a given algorithm by implementing an interface consisting of *transforms* and *rules*. Here, transforms describe specific algorithms, and rules describe various implementation choices. At compile time, the Petabricks compiler constructs a choice-dependency graph and explores various implementation choices automatically. Additionally, it tunes numerical parameters such as block sizes or user-specified tuning parameters.

Tangram follows a similar approach compared to Petabricks and enables the user to specify so-called *codelets*. A codelet represents a snippet of code that can have several semantics-preserving implementations for the Tangram compiler to explore at compile time.

Additionally, codelets can contain tunable parameters that allow architecture-specific parameterization, such as tile sizes that need to be adjusted for targeting different architectures.

In contrast to Petabricks and Trangram, our high-performance compilation approach allows programmers to derive optimized implementations by applying rewrite rules instead of replacing unoptimized code with user-defined high-performance implementations.

TASO [83] (Tensor Algebra SuperOptimizer) is a DNN computation graph optimizer for automatically generating graph substitutions. TASO expects a set of machine learning operators as input (e.g., `matmul`, `conv`, or `transpose`) and automatically generates graph substitutions for optimizing machine learning computations graphs. By enumerating all possible compositions of operators up to a fixed depth and comparing the computed output using random input values, TASO generates potential graph substitution candidates. The found candidates are verified with respect to the operator's properties. Verified substitutions are used for optimizing real-world machine learning computation graphs such as ResNet-50 [75].

TASO's approach could be adapted to work on LIFT (or RISE) primitives instead of machine learning operators by synthesizing all possible rewrite rules for sub-expressions of a fixed length. The well-defined semantics of LIFT and RISE primitives are similar to the defined properties for the machine learning operators used by TASO. In addition, the intentionally small number of existing primitives roughly matches the number of supported machine learning operators.

SWIZZLE INVENTOR [128] is a sketch-based program-synthesis approach to optimizing data movements in GPU kernels. Swizzle Inventor allows programmers to develop CUDA-style GPU kernels in the Racket programming language. Here, a kernel is allowed to have unspecified parts; for example, the index expression for accessing elements of an array might be left unspecified. A kernel is accompanied by a sequential program acting as a *specification* that describes how to compute a correct output matrix.

Swizzle Inventor automatically synthesizes program fragments for the unspecified parts of the kernel such that the computed output agrees with the output of the specification while optimizing the kernel's execution time. Swizzle Inventor's optimization approach is orthogonal to the approaches described in this thesis. However, sketch-based program synthesis, and Swizzle Inventor specifically, are directly applicable to, for example, synthesize high-performance swizzle expression in Fireiron decompositions.

AUTOMATICALLY GENERATING EFFICIENT SCHEDULES Unlike Fireiron, many schedule-based compilers often provide or are extended with automatic approaches for finding efficient schedules. For example, both TVM [30] and Halide [107] provide such functionality to liberate experts from developing high-performance schedules. Halide's auto-scheduling approach relies heavily on the bound inference analysis already present in the original Halide compiler [140]. After initial pre-processing, the auto-scheduling algorithm groups and tiles computations specified in the input program by enumerating and evaluating potential merging opportunities.

TVM's auto-scheduling approach (AutoTVM) rather relies on machine learning techniques. A statistical cost model estimates the cost of low-level TVM programs, and an exploration module produces new schedule candidates to evaluate.

Both approaches, algorithmic and machine-learning-based auto-scheduling, are similarly applicable for generating high-performance Fireiron decompositions because the scheduling languages of those compilers follow a similar design.

7.4 HIGH-LEVEL ABSTRACTIONS FOR STENCIL COMPUTATIONS AND HIGH-PERFORMANCE STENCIL OPTIMIZATIONS

In the following, we compare our work on stencil computations presented in Chapter 4 against existing domain-specific frameworks for expressing and optimizing stencil computations.

OPTIMIZATIONS FOR PARALLEL STENCIL COMPUTATIONS Before discussing specific high-performance stencil frameworks, we briefly discuss conventional optimization techniques typically applied for optimizing the parallel performance of stencil computations on different hardware architectures.

One essential optimization for improving the performance by exploiting locality is tiling. For example, overlapped tiling [62] describes a tiling scheme in which tiles overlap according to the stencil shape. In Section 4.4.1, we have shown how to encode this optimization as a rewrite rule in LIFT. There exists a wide variety of different tiling schemes [194] suitable for particular stencil shapes or hardware architectures that especially improve performance for iterative stencil computations. The polyhedral model discussed earlier is especially suitable for expressing different tiling schemes. Its mathematical polyhedron representation of loops allows simple transformations to perform computations using tiling schemes, including overlapped-tiling [62], diamond-tiling, parallelogram-tiling [195], split-tiling using trapezoidal tiles [60], or *flextended* tiles [193]. Stoltzfus et al. [169] extend the work on optimizing stencil computations described in Chapter 4 and introduce how to express 2.5D tiling, a tiling technique particularly suitable for 3D stencils, as rewrite rules in LIFT.

Additionally, there exists a large body of work on optimizing stencil computations for particular hardware architectures. Rawat et al. [146, 147, 149] describe optimizations for targeting GPUs specifically. GPU-specific stencil optimizations include approaches for minimizing register usage [148] or reducing global memory access [120]. Similarly, optimizations for stencil computations have been developed for targeting other architectures, including FPGAs [45, 185], POWER8 [191], heterogeneous hybrid CPU-GPU architectures [63], or multi-core clusters [47].

POLYMAGE [108] is a domain-specific language and compiler for image processing pipelines. Users of PolyMage express image processing pipelines, i.e., a graph of connected operations that typically include various stencil computations, in a high-level language. The underlying compiler aims to generate high-performance code targeting CPUs automatically. The design of PolyMage's DSL is inspired by Halide and is inherently domain-specific. With our extensions to the functional LIFT IR, we showed how to express similar domain-specific stencil computations in a generic IR.

PolyMage's compilation process relies heavily on polyhedral compilation techniques for optimizing the image processing pipeline performance. The compiler applies optimizations, including overlapped tiling, as discussed in Section 4.4.1, but it covers a broader selection of stencil-specific optimizations than we implemented as rewrite rules in our case study of supporting stencils in LIFT. Examples for such optimizations are parallelogram or split tiling schemes.

PASTHA [96] is a parallel Haskell library for stencil computations. PASTHA defines a stencil computation as a 5-tuple containing the dimensions and numbers of input matrices, their initial values, a tuple consisting of stencil shape, a function describing the computation per iteration, and a convergence condition for terminating the time-iterated computation.

In contrast to our way of expressing stencil computations in the functional LIFT IR, PASTHA provides a domain-specific *stencil* primitive that only allows the definition of two-dimensional stencil computations. However, PASTHA can be used for both Jacobi-type stencils (neighboring values for the current iteration are taken from the previous iteration), and Gauss-Seidel-type stencils (neighboring values for the current iteration are taken from the current iteration). So far, we only used LIFT for optimizing the more often used Jacobi-type stencils.

PATUS [31, 32] (Parallel AutoTUned Stencils) is a code generation and auto-tuning framework for stencil computations targeting CPUs and GPUs. Patus provides a high-level C-like domain-specific language for specifying stencil computations. Additionally, Patus provides a second small DSL for specifying parallelization strategies. Unlike the rewrite-based strategy languages like ELEVATE, Patus' strategy language is also a small C-like language. It contains specific extensions for describing how and when individual results in a loop-nest are computed, similar to schedule trees used in polyhedral compilation. The mapping of computation and data to the parallel hardware is implicitly defined in Patus. However, in Chapter 3, we specifically argued for making all mapping decisions, including data movements, explicit in a scheduling language.

POCHOIR [174] is a domain-specific language and compiler embedded in C++ for expressing generic multi-dimensional stencil computations. Pochoir targets multicore processors by generating high-performance Cilk code, Intel's C++ language extension for fork-join based code parallelization. The Pochoir compiler, written in Haskell, applies sophisticated domain-specific transformations such as computing stencils using a cache-oblivious algorithm based on a *trapezoidal decomposition* [54]. Pochoir is an excellent example of a typical domain-specific compiler that leverages domain-specific knowledge by applying specialized compilation strategies to achieve high performance. With the contributions made in this thesis, we have shown that it is possible to design similar high-performance domain-specific compilation approaches without requiring domain-specific solutions.

SUMMARY Similar to PASTHA's *stencil* function, there exist many frameworks for high-level stencil programming that provide a restricted domain-specific stencil construct. Other frameworks following the same approach are, for example, SkelCL [164] (providing a *MapOverlap* and a *Stencil* skeleton), HLSF [48] (providing a `makeStencil` function), or SkePU [50] (using *MapOverlap*).

Another popular approach for describing stencil computations is to use explicit indices for describing the neighborhood accesses defined by the stencil shape. Using explicit indices leads to potential out-of-bounds accesses that must be dealt with, for example, using `unsafe` annotations as used by Futhark [77] or by complex shape-inference algorithms as used in Halide [140].

The advantage of our approach is that by decomposing stencil computations in three fundamental building blocks that each can be expressed with a simple, functional primitive, we neither have to introduce domain-specific stencil primitives nor need to deal with potentially unsafe index expressions.

7.5 REWRITING IN COMPILERS AND STRATEGY LANGUAGES

Rewrite rules and rewriting strategies have long been used to build compilers. For example, the Glasgow Haskell Compiler [127] uses rewrite rules as a practical way to optimize Haskell programs. Other areas include building interpreters [46], instruction selection [22] or constant propagation [123].

In the following, we briefly introduce rewriting approaches closely related to the work in this thesis. LIFT and MLIR have already been discussed earlier and will not be repeated in detail below. LIFT [70, 162, 167] showed how to use rewrite rules to generate high-performance code targeting accelerators. Google recently introduced MLIR [93] with declarative rewrite rules for specifying dialect transformations in optimizing domain-specific compilers.

Visser [180, 183] and Kirchner [87] provide surveys covering term rewriting, strategy languages and their application domains. Tactic languages, including [42] and [52], are related to strategy languages for rewriting systems but are designed for specifying proofs in theorem provers.

Generally, the idea of program optimization via rewrite rules reaches as far back as the introduction of functional programming languages. For example, in his 1978 Turing Lecture, John Backus introduced the concept of an *algebra of programs* [9] in which programs are computed by applying simple algebraic rules to a specification describing the computation to perform. The same idea is further developed in the Bird-Meertens Formalism [13, 14, 102], which represents one concrete approach for *computing* programs according to algebraic rules.

In the following, we mainly focus on approaches for controlling the application of rewrite rules by using strategy languages. These are especially related to our work on LIFT and ELEVATE, as discussed in Chapter 4 and Chapter 5.

STRATEGO [181, 184] significantly inspired the design of ELEVATE, our language for expressing optimizations as rewriting strategies, as presented in Chapter 5. We directly adopted parts of ELEVATE's core design from Stratego. This includes the core combinators such as `seq` and `lChoice` (Section 5.4.4), one-level traversals [98] such as `one` and `all` (Section 5.4.5), as well as whole tree traversals such as `topDown` (Section 5.4.7).

In contrast to Stratego, in ELEVATE, we do not model strategies as functions from programs to programs, but rather as functions from programs to an applicative error monad `RewriteResult` for explicitly encoding success and failure of strategy applications. Stratego differentiates between rewrite rules and strategies (ELEVATE does not), and Stratego's rewrite rules are further decomposed into three parts (local variables, matching conditions, application) for expressing contexts and conditional rules. In ELEVATE, we achieve this using precise location descriptions and strategy predicates. ELEVATE's domain-specific traversal strategies relate to Stratego's implicitly defined *congruence* operators, which allow to traverse children of user-defined terms.

Finally, Visser, Benaissa, and Tolmach [184] describe how to build program optimizers using rewriting strategies; however, on a significantly smaller scale, not focusing on high-performance code generation. With ELEVATE, we have shown how to use and extend the techniques developed in Stratego for the optimization of programs allowing to achieve high performance that competes with state-of-the-art optimizing compilers.

MAUDE [34, 35] is a high-level language for declarative programming in rewriting logic. Maude defines two kinds of rewrite rules: *equations* and *rules*, which are applied using fixed but different traversal strategies. Maude itself does not allow programmers to define strategies composed of rewrite rules nor to define to alter or define new traversal strategies. Instead, strategies for applying rewrite rules are made internal to Maude's implementation and can be changed using reflection [36].

PORGY [5, 129, 130] is a visual and interactive graph rewriting system to apply graph transformations encoded as rewrite rules using strategies. PORGY defines two core concepts: *port graphs* for encoding different graphs with nodes containing explicitly labeled connection points per edge (called *ports*) and a *port graph rewriting relation* for encoding rewrite rules on port graphs. PORGY's port graphs could be used for rewriting ASTs similar to the rewriting performed in ELEVATE. However, their main application domains are biochemical networks and interaction nets. Similar to ELEVATE and Stratego, PORGY provides basic combinators for composing rewrite rules, including sequential composition and basic strategies such as `id` and `fail`. Additionally, PORGY provides probabilistic combinators that non-deterministically pick from a given set of strategies with different probabilities. Such combinators do not exist in ELEVATE but are frequently used in PORGY because they are particularly suited for biochemical models.

ASF+SDF [21, 44] is a meta-environment for the development of interactive environments for programming languages. ASF+SDF defines two formalisms: the *Algebraic Specific Formalism* that provides a notation for specifying programming languages, and the *Syntax Definition Formalism* that allows the concrete and abstract language syntax specification. ASF+SDF provides term-rewriting functionality and was one of the first frameworks to provide a limited set of traversal functions for controlling the application of rewrite rules. ELEVATE's traversal functionality exposes more fine-grained control to the user as required for achieving high-performance domain-specific compilation.

ELAN [18] is a framework for expressing a logic using syntax and inference rules and relies on rewriting guided by strategies. In contrast to ELEVATE, the application of a rewrite rule leads to multiple results (e.g., if a rule is applicable at multiple locations of a term). Returning the empty set after applying a rewrite rule or strategy is considered a *failure*. In ELEVATE, we model the success and failure of strategy applications explicitly using the **`RewriteResult`** monad.

ELAN allows a restricted definition of strategies for specifying where a rule is applied, which differentiates between two levels. The first level consists of basic regular expressions based on a fixed rule alphabet and the second level that allows to apply labeled rules using a fixed traversal. In contrast, ELEVATE allows users to freely define their own strategies, including advanced abstractions for describing precise locations in a program.

TAMPR [19, 20] is an automatic rewrite-based transformation system developed in the seventies to optimize numerical computations. TAMPR enables programmers to define syntax-based rewrite rules and provides a simple *control language* for controlling the application of rewrite rules. This language consists of a small set of control structures that, for example, apply a given set of rules exhaustively or in another case at most one rule to any program fragment. In contrast to traditional term rewriting systems that often exhaustively apply a given set of rewrite rules to normalize (also known as *canonicalize*) terms, TAMPR uses sequences of canonical forms. This is similar to our approach of optimizing RISE matrix multiplication expressions with ELEVATE. We also rely on and apply multiple normal-forms such as `DFNF` or βη-*normal-form*, as discussed in Section 5.4.8.

In this thesis, we introduced a novel approach to achieving

High-Performance Domain-Specific Compilation
without
Domain-Specific Compilers.

Our approach achieves the benefits of domain-specific compilation by expressing both domain-specific computations and their optimizations as compositions of reusable and generic building blocks instead of constructing a domain-specific compiler from scratch.

BIBLIOGRAPHY

[1] AMD. *AMD Optimizing CPU Libraries (AOCL)*. 2020. URL: https://developer.amd.com/amd-aocl/.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. "TensorFlow: A System for Large-Scale Machine Learning." In: *OSDI*. USENIX Association, 2016, pp. 265–283.

[4] Marco Aldinucci, Massimo Torquati, and Massimiliano Meneghin. "FastFlow: Efficient Parallel Streaming Applications on Multi-core." In: *CoRR* abs/0909.1187 (2009).

[5] Oana Andrei, Maribel Fernández, Hélène Kirchner, Guy Melançon, Olivier Namet, and Bruno Pinaud. "PORGY: Strategy-Driven Interactive Transformation of Graphs." In: *TERMGRAPH*. Vol. 48. EPTCS. 2011, pp. 54–68.

[6] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman P. Amarasinghe. "OpenTuner: an extensible framework for program autotuning." In: *PACT*. ACM, 2014, pp. 303–316.

[7] Krste Asanovic, Rastislav Bodík, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David A. Patterson, Koushik Sen, John Wawrzynek, David Wessel,

and Katherine A. Yelick. "A view of the parallel computing landscape." In: *Commun. ACM* 52.10 (2009), pp. 56–67.

[8] Robert Atkey, Michel Steuwer, Sam Lindley, and Christophe Dubach. "Strategy Preserving Compilation for Parallel Functional Code." In: *CoRR* abs/1710.08332 (2017).

[9] John W. Backus. "Can Programming Be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs." In: *Commun. ACM* 21.8 (1978), pp. 613–641.

[10] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman P. Amarasinghe. "Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code." In: *CGO*. IEEE, 2019, pp. 193–205.

[11] Paul Barham and Michael Isard. "Machine Learning Systems are Stuck in a Rut." In: *HotOS*. ACM, 2019, pp. 177–183.

[12] Somashekaracharya G Bhaskaracharya, Julien Demouth, and Vinod Grover. "Automatic Kernel Generation for Volta Tensor Cores." In: *arXiv preprint arXiv:2006.12645* (2020).

[13] Richard S. Bird. "Algebraic Identities for Program Calculation." In: *Comput. J.* 32.2 (1989), pp. 122–126.

[14] Richard S. Bird and Oege de Moor. "The algebra of programming." In: *NATO ASI DPD*. 1996, pp. 167–203.

[15] OpenMP Architecture Review Board. "OpenMP Application Programming Interface (Version 5.0)." In: *OpenMP Specification* (2018). URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

[16] Bruno Bodin, Luigi Nardi, M. Zeeshan Zia, Harry Wagstaff, Govind Sreekar Shenoy, Murali Krishna Emani, John Mawer, Christos Kotselidis, Andy Nisbet, Mikel Luján, Björn Franke, Paul H. J. Kelly, and Michael F. P. O'Boyle. "Integrating Algorithmic Parameters into Benchmarking and Design Space Exploration in 3D Scene Understanding." In: *PACT*. ACM, 2016, pp. 57–69.

[17] Uday Bondhugula. "High Performance Code Generation in MLIR: An Early Case Study with GEMM." In: *CoRR* abs/2003.00532 (2020).

[18] Peter Borovanský, Claude Kirchner, Hélène Kirchner, and Christophe Ringeissen. "Rewriting with Strategies in ELAN: A Functional Semantics." In: *Int. J. Found. Comput. Sci.* 12.1 (2001), pp. 69–95.

[19] James M Boyle. "Abstract programming and program transformation—An approach to reusing programs." In: *Software reusability: vol. 1, concepts and models*. 1989, pp. 361–413.

[20] James M. Boyle, Terence J. Harmer, and Victor L. Winter. "The TAMPR Program Transformation System: Simplifying the Development of Numerical Software." In: *SciTools*. Birkhäuser, 1996, pp. 353–372.

[21] Mark van den Brand, Arie van Deursen, Jan Heering, H. A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. "The ASF+SDF Meta-environment: A Component-Based Language Development Environment." In: *CC*. Vol. 2027. Lecture Notes in Computer Science. Springer, 2001, pp. 365–370.

[22] Martin Bravenboer and Eelco Visser. "Rewriting Strategies for Instruction Selection." In: *RTA*. Vol. 2378. Lecture Notes in Computer Science. Springer, 2002, pp. 237–251.

[23] Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. "Accelerating Haskell array codes with multicore GPUs." In: *DAMP*. ACM, 2011, pp. 3–14.

[24] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. "Parallel Programmability and the Chapel Language." In: *Int. J. High Perform. Comput. Appl.* 21.3 (2007), pp. 291–312.

[25] Donald D. Chamberlin and Raymond F. Boyce. "SEQUEL: A Structured English Query Language." In: *SIGMOD Workshop, Vol. 1*. ACM, 1974, pp. 249–264.

[26] Li-Wen Chang, Abdul Dakkak, Christopher I Rodrigues, and Wen-mei Hwu. "Tangram: a high-level language for performance portable code synthesis." In: *Programmability Issues for Heterogeneous Multicores* (2015).

[27] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. "Rodinia: A benchmark suite for heterogeneous computing." In: *IISWC*. IEEE Computer Society, 2009, pp. 44–54.

[28] Chun Chen, Jacqueline Chame, and Mary Hall. *CHiLL: A framework for composing high-level loop transformations*. Tech. rep. Citeseer, 2008.

[29] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning." In: *OSDI*. USENIX Association, 2018, pp. 578–594.

[30] Tianqi Chen, Lianmin Zheng, Eddie Q. Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. "Learning to Optimize Tensor Programs." In: *NeurIPS*. 2018, pp. 3393–3404.

[31] Matthias Christen, Olaf Schenk, and Helmar Burkhart. "PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures." In: *IPDPS*. IEEE, 2011, pp. 676–687.

[32] Matthias Christen, Olaf Schenk, and Yifeng Cui. "Patus for convenient high-performance stencils: evaluation in earthquake simulations." In: *SC*. IEEE/ACM, 2012, p. 11.

[33] Philipp Ciechanowicz. "Algorithmic skeletons for general sparse matrices on multi-core processors." In: *Proceedings of the 20th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS)*. 2008, pp. 188–197.

[34] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. "Maude: specification and programming in rewriting logic." In: *Theor. Comput. Sci.* 285.2 (2002), pp. 187–243.

[35] Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. "Principles of Maude." In: *WRLA*. Vol. 4. Electronic Notes in Theoretical Computer Science. Elsevier, 1996, pp. 65–89.

[36] Manuel Clavel and José Meseguer. "Reflection and strategies in rewriting logic." In: *WRLA*. Vol. 4. Electronic Notes in Theoretical Computer Science. Elsevier, 1996, pp. 126–148.

[37] Murray Cole. "Algorithmic skeletons : a structured approach to the management of parallel computation." PhD thesis. University of Edinburgh, UK, 1988.

[38] Alexander Collins, Dominik Grewe, Vinod Grover, Sean Lee, and Adriana Susnea. "NOVA: A Functional Language for Data Parallelism." In: *ARRAY@PLDI*. ACM, 2014, pp. 8–13.

[39] Patrick Cousot and Nicolas Halbwachs. "Automatic Discovery of Linear Restraints Among Variables of a Program." In: *POPL*. ACM Press, 1978, pp. 84–96.

[40] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. "The Scalable Heterogeneous Computing (SHOC) benchmark suite." In: *GPGPU*. Vol. 425. ACM International Conference Proceeding Series. ACM, 2010, pp. 63–74.

[41] Nicolaas Govert De Bruijn. “Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem.” In: *Indagationes Mathematicae (Proceedings)*. Vol. 75. 5. North-Holland. 1972, pp. 381–392.

[42] David Delahaye. “A Tactic Language for the System Coq.” In: *LPAR*. Vol. 1955. Lecture Notes in Computer Science. Springer, 2000, pp. 85–95.

[43] RH Dennard, FH Gaensslen, H Yu, VL Rideout, E Bassous, and AR LeBlanc. *Design of ion-implanted MOSFET's with very small physical dimensions. In proceedings of IEEE Solid-State Circuits, vol. 12 (1)*. 2007.

[44] Arie van Deursen, Jan Heering, and Paul Klint. *Language Prototyping: An Algebraic Specification Approach*. Vol. 5. AMAST Series in Computing. World Scientific, 1996.

[45] Keisuke Dohi, Kota Fukumoto, Yuichiro Shibata, and Kiyoshi Oguri. “Performance modeling and optimization of 3-D stencil computation on a stream-based FPGA accelerator.” In: *ReConFig*. IEEE, 2013, pp. 1–6.

[46] Eelco Dolstra and Eelco Visser. “Building Interpreters with Rewriting Strategies.” In: *Electron. Notes Theor. Comput. Sci.* 65.3 (2002), pp. 57–76.

[47] Hikmet Dursun, Manaschai Kunaseth, Ken-ichi Nomura, Jacqueline Chame, Robert F. Lucas, Chun Chen, Mary W. Hall, Rajiv K. Kalia, Aiichiro Nakano, and Priya Vashishta. “Hierarchical parallelization and optimization of high-order stencil computations on multicore clusters.” In: *J. Supercomput.* 62.2 (2012), pp. 946–966.

[48] Fabian Dütsch, Karim Djelassi, Michael Haidl, and Sergei Gorlatch. “HLSF: A high-level; C++-based framework for stencil computations on accelerators.” In: *Proceedings of the second workshop on optimizing stencil computations*. 2014.

[49] Venmugil Elango, Norm Rubin, Mahesh Ravishankar, Hariharan Sandanagobalane, and Vinod Grover. “Diesel: DSL for linear algebra and neural net computations on GPUs.” In: *MAPL@PLDI*. ACM, 2018, pp. 42–51.

[50] August Ernstsson, Lu Li, and Christoph W. Kessler. “SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems.” In: *Int. J. Parallel Program.* 46.1 (2018), pp. 62–80.

[51] Paul Feautrier. “Parametric Integer Programming.” In: *RAIRO Recherche opérationnelle* 22.3 (1988), pp. 243–268.

[52] Amy P. Felty. “Implementing Tactics and Tacticals in a Higher-Order Logic Programming Language.” In: *J. Autom. Reasoning* 11.1 (1993), pp. 41–81.

[53] Matteo Frigo and Steven G. Johnson. “The Design and Implementation of FFTW3.” In: *Proceedings of the IEEE* 93.2 (2005). Special issue on “Program Generation, Optimization, and Platform Adaptation”, pp. 216–231.

[54] Matteo Frigo and Volker Strumpen. “Cache oblivious stencil computations.” In: *ICS*. ACM, 2005, pp. 361–366.

[55] Yoshihiko Futamura. “Parital Computation of Programs.” In: *RIMS Symposium on Software Science and Engineering*. Vol. 147. Lecture Notes in Computer Science. Springer, 1982, pp. 1–35.

[56] Yoshihiko Futamura. “Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler.” In: *High. Order Symb. Comput.* 12.4 (1999), pp. 381–391.

[57] Michael Garland and David B. Kirk. “Understanding throughput-oriented architectures.” In: *Commun. ACM* 53.11 (2010), pp. 58–66.

[58] Philip Ginsbach, Toomas Remmelg, Michel Steuwer, Bruno Bodin, Christophe Dubach, and Michael F. P. O’Boyle. “Automatic Matching of Legacy Code to Heterogeneous APIs: An Idiomatic Approach.” In: *ASPLOS*. ACM, 2018, pp. 139–153.

[59] Tobias Grosser. *polyhedral.info*. 2020. URL: https://polyhedral.info/.

[60] Tobias Grosser, Albert Cohen, Paul H. J. Kelly, J. Ramanujam, P. Sadayappan, and Sven Verdoolaege. “Split tiling for GPUs: automatic parallelization using trapezoidal tiles.” In: *GPGPU@ASPLOS*. ACM, 2013, pp. 24–31.

[61] Khronos OpenCL Working Group. “The OpenCL Specification (Version 1.2).” In: *Khronos Group* (2012). URL: https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf.

[62] Jia Guo, Ganesh Bikshandi, Basilio B. Fraguela, and David A. Padua. “Writing productive stencil codes with overlapped tiling.” In: *Concurr. Comput. Pract. Exp.* 21.1 (2009), pp. 25–39.

[63] Tobias Gysi, Tobias Grosser, and Torsten Hoefler. “MODESTO: Data-centric Analytic Optimization of Complex Stencil Programs on Heterogeneous Architectures.” In: *ICS*. ACM, 2015, pp. 177–186.

[64] Tobias Gysi, Christoph Müller, Oleksandr Zinenko, Stephan Herhut, Eddie Davis, Tobias Wicky, Oliver Fuhrer, Torsten Hoefler, and Tobias Grosser. “Domain-Specific Multi-Level IR Rewriting for GPU.” In: *CoRR* abs/2005.13014 (2020).

[65] Bastian Hagedorn. "An Extension of a Functional Intermediate Language for Parallelizing Stencil Computations and its Optimizing GPU Implementation using OpenCL." MA thesis. University of Münster, 2016.

[66] Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. "Fireiron: A Data-Movement-Aware Scheduling Language for GPUs." In: *PACT*. ACM, 2020, (accepted).

[67] Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Sergei Gorlatch, and Michel Steuwer. "A Language for Describing Optimization Strategies." In: *arXiv preprint arXiv:2002.02268* (2020).

[68] Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. "Achieving High-Performance the Functional Way - A Functional Pearl on Expressing High-Performance Optimizations as Rewrite Strategies." In: *ICFP*. ACM, 2020, (accepted).

[69] Bastian Hagedorn, Michel Steuwer, and Sergei Gorlatch. "A Transformation-Based Approach to Developing High-Performance GPU Programs." In: *Ershov Informatics Conference*. Vol. 10742. Lecture Notes in Computer Science. Springer, 2017, pp. 179–195.

[70] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. "High performance stencil code generation with lift." In: *CGO*. ACM, 2018, pp. 100–112.

[71] Halide. *Issue*. describes use of "-Ofast -ffast-math". 2018. URL: https://github.com/halide/Halide/issues/2905.

[72] Halide. *MatMulGenerator*. 2020. URL: https://github.com/halide/Halide/blob/master/apps/cuda_mat_mul/mat_mul_generator.cpp.

[73] Halide. *Tutorial: Lesson 5 - Scheduling*. 2020. URL: https://halide-lang.org/tutorials/tutorial_lesson_05_scheduling_1.html.

[74] Adam Harries, Michel Steuwer, Murray Cole, Alan Gray, and Christophe Dubach. "Compositional Compilation for Sparse, Irregular Data Parallelism." In: *Workshop on High-Level Programming for Heterogeneous and Hierarchical Parallel Systems (HLPGPU)*. 2016.

[75] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition." In: *CVPR*. IEEE Computer Society, 2016, pp. 770–778.

[76] John L. Hennessy and David A. Patterson. "A new golden age for computer architecture." In: *Commun. ACM* 62.2 (2019), pp. 48–60.

[77] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. “Futhark: purely functional GPU-programming with nested parallelism and in-place array updates.” In: *PLDI*. ACM, 2017, pp. 556–571.

[78] Troels Henriksen, Frederik Thorøe, Martin Elsman, and Cosmin E. Oancea. “Incremental flattening for nested data parallelism.” In: *PPoPP*. ACM, 2019, pp. 53–67.

[79] Sepp Hochreiter and Jürgen Schmidhuber. “LSTM can Solve Hard Long Time Lag Problems.” In: *NIPS*. MIT Press, 1996, pp. 473–479.

[80] Jianyu Huang, Chenhan D. Yu, and Robert A. van de Geijn. “Implementing Strassen’s Algorithm with CUTLASS on NVIDIA Volta GPUs.” In: *CoRR* abs/1808.07984 (2018).

[81] Paul Hudak. “Modular domain specific languages and tools.” In: *ICSR*. IEEE Computer Society, 1998, pp. 134–142.

[82] Intel. *Intel Math Kernel Library*. 2020. URL: https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html.

[83] Zhihao Jia, Oded Padon, James J. Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. “TASO: optimizing deep learning computation with automatic generation of graph substitutions.” In: *SOSP*. ACM, 2019, pp. 47–62.

[84] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. “In-Datacenter Performance Analysis of a Tensor Processing Unit.” In: *ISCA*. ACM, 2017, pp. 1–12.

[85] Lennart C. L. Kats and Eelco Visser. “The spoofax language workbench: rules for declarative specification of languages and IDEs.” In: *OOPSLA*. ACM, 2010, pp. 444–463.

[86] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and Dave Wonnacott. “The Omega Calculator and Library, Version 1.1.0.” In: (1996). URL: http://www.cs.utah.edu/~mhall/cs6963s09/lectures/omega.ps.

[87] Hélène Kirchner. “Rewriting Strategies and Strategic Rewrite Programs.” In: *Logic, Rewriting, and Concurrency*. Vol. 9200. Lecture Notes in Computer Science. Springer, 2015, pp. 380–403.

[88] Bastian Köpcke, Michel Steuwer, and Sergei Gorlatch. “Generating efficient FFT GPU code with Lift.” In: *FHPNC@ICFP*. ACM, 2019, pp. 1–13.

[89] Martin Kristien, Bruno Bodin, Michel Steuwer, and Christophe Dubach. “High-level synthesis of functional patterns with Lift.” In: *ARRAY@PLDI*. ACM, 2019, pp. 35–45.

[90] Herbert Kuchen. “A Skeleton Library.” In: *Euro-Par*. Vol. 2400. Lecture Notes in Computer Science. Springer, 2002, pp. 620–629.

[91] Herbert Kuchen and Murray Cole. “The Integration of Task and Data Parallel Skeletons.” In: *Parallel Process. Lett.* 12.2 (2002), pp. 141–155.

[92] Chris Lattner and Vikram S. Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation.” In: *CGO*. IEEE Computer Society, 2004, pp. 75–88.

[93] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. “MLIR: A Compiler Infrastructure for the End of Moore’s Law.” In: *CoRR* abs/2002.11054 (2020).

[94] Roland Leißa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt. “AnyDSL: a partial evaluation framework for programming high-performance libraries.” In: *PACMPL* 2.OOPSLA (2018), 119:1–119:30.

[95] Roland Leißa, Marcel Köster, and Sebastian Hack. “A graph-based higher-order intermediate representation.” In: *CGO*. IEEE Computer Society, 2015, pp. 202–212.

[96] Michael Lesniak. “PASTHA: parallelizing stencil calculations in Haskell.” In: *DAMP*. ACM, 2010, pp. 5–14.

[97] Vincent Loechner. *PolyLib: A library for manipulating parameterized polyhedra*. 1999. URL: https://repo.or.cz/polylib.git/blob_plain/HEAD:/doc/parampoly-doc.ps.gz.

[98] Sebastiaan P Luttik and Eelco Visser. "Specification of Rewriting Strategies." In: *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications* 2. 1997, pp. 1–16.

[99] Martin Lücke, Michel Steuwer, and Aaron Smith. "A functional pattern-based language in MLIR." In: *AccML@HiPEAC*. 2020.

[100] MLIR. *Linalg Dialect*. 2020. URL: https://mlir.llvm.org/docs/Rationale/RationaleLinalgDialect/.

[101] Azamatbek Mametjanov, Victor L. Winter, and Ralf Lämmel. "More precise typing of rewrite strategies." In: *LDTA*. ACM, 2011, p. 3.

[102] Lambert G. L. T. Meertens. "Constructing a Calculus of Programs." In: *MPC*. Vol. 375. Lecture Notes in Computer Science. Springer, 1989, pp. 66–90.

[103] Naums Mogers, Valentin Radu, Lu Li, Jack Turner, Michael F. P. O'Boyle, and Christophe Dubach. "Automatic generation of specialized direct convolutions for mobile GPUs." In: *GPGPU@PPoPP*. ACM, 2020, pp. 41–50.

[104] Naums Mogers, Aaron Smith, Dimitrios Vytiniotis, Michel Steuwer, Christophe Dubach, and Ryota Tomioka. "Towards Mapping Lift to Deep Neural Network Accelerators." In: *Workshop on Emerging Deep Learning Accelerators (EDLA)*. 2019.

[105] Simon Moll and Sebastian Hack. "Partial control-flow linearization." In: *PLDI*. ACM, 2018, pp. 543–556.

[106] Gordon E Moore. "Cramming more components onto integrated circuits." In: *Proceedings of the IEEE* 86.1 (1998), pp. 82–85.

[107] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. "Automatically scheduling halide image processing pipelines." In: *ACM Trans. Graph.* 35.4 (2016), 83:1–83:11.

[108] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. "PolyMage: Automatic Optimization for Image Processing Pipelines." In: *ASPLOS*. ACM, 2015, pp. 429–443.

[109] NVIDIA. "NVIDIA Parallel Thread Execution ISA." In: *NVIDIA PTX ISA* (2017). URL: https://docs.nvidia.com/pdf/ptx_isa_7.0.pdf.

[110] NVIDIA. "NVIDIA Tesla V100 GPU Architecture (2017)." In: *NVIDIA white paper* (2017). URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[111] NVIDIA. *CUDA Toolkit Documentation: CUDA C++ Programming Guide*. 2020. URL: https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf.

[112] NVIDIA. *CUDA Toolkit Documentation: Inline PTX Assembly in CUDA*. 2020. URL: https://docs.nvidia.com/cuda/inline-ptx-assembly/index.html.

[113] NVIDIA. *CUDA Toolkit Documentation: PTX MMA Instructions*. 2020. URL: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html\#warp-level-matrix-instructions-mma.

[114] NVIDIA. *CUDA Toolkit Documentation: Warp-level matrix instructions*. 2020. URL: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html\#warp-level-matrix-instructions.

[115] NVIDIA. *CUDA WMMA Sample Kernel*. 2020. URL: https://github.com/NVIDIA/cuda-samples/blob/master/Samples/cudaTensorCoreGemm/cudaTensorCoreGemm.cu.

[116] NVIDIA. *CUTLASS: Fast Linear Algebra in CUDA C++*. 2020. URL: https://devblogs.nvidia.com/cutlass-linear-algebra-cuda/.

[117] NVIDIA. *NVIDIA Ampere Architecture In-Depth*. 2020. URL: https://devblogs.nvidia.com/nvidia-ampere-architecture-in-depth/.

[118] NVIDIA. "NVIDIA Ampere Architecture Whitepaper (2020)." In: *NVIDIA white paper* (2020). URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.

[119] NVIDIA. *Programming Tensor Cores*. 2020. URL: https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/.

[120] Thiago Carrijo Nasciutti, Jairo Panetta, and Pedro Pais Lopes. "Evaluating optimizations that reduce global memory accesses of stencil computations in GPGPUs." In: *Concurr. Comput. Pract. Exp.* 31.18 (2019).

[121] Cedric Nugteren and Valeriu Codreanu. "CLTune: A Generic Auto-Tuner for OpenCL Kernels." In: *MCSoC*. IEEE Computer Society, 2015, pp. 195–202.

[122] Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. “Spiral in scala: towards the systematic construction of generators for performance libraries.” In: *GPCE*. ACM, 2013, pp. 125–134.

[123] Karina Olmos and Eelco Visser. “Strategies for Source-to-Source Constant Progagation.” In: *Electron. Notes Theor. Comput. Sci.* 70.6 (2002), pp. 156–175.

[124] M. Akif Özkan, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, Roland Leißa, Sebastian Hack, Jürgen Teich, and Frank Hannig. “AnyHLS: High-Level Synthesis with Partial Evaluation.” In: *CoRR* abs/2002.05796 (2020).

[125] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. “Automatic differentiation in PyTorch.” In: (2017).

[126] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. “PyTorch: An Imperative Style, High-Performance Deep Learning Library.” In: *NeurIPS*. 2019, pp. 8024–8035.

[127] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. “Playing by the rules: rewriting as a practical optimisation technique in GHC.” In: *2001 Haskell Workshop*. ACM SIGPLAN. 2001.

[128] Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodík. “Swizzle Inventor: Data Movement Synthesis for GPU Kernels.” In: *ASPLOS*. ACM, 2019, pp. 65–78.

[129] Bruno Pinaud, Oana Andrei, Maribel Fernández, Hélène Kirchner, Guy Melançon, and Jason Vallet. “PORGY : a Visual Analytics Platform for System Modelling and Analysis Based on Graph Rewriting.” In: *EGC*. Vol. E-33. RNTI. Éditions RNTI, 2017, pp. 473–476.

[130] Bruno Pinaud, Jonathan Dubois, and Guy Melançon. “PORGY: Interactive and visual reasoning with graph rewriting systems.” In: *IEEE VAST*. IEEE Computer Society, 2011, pp. 293–294.

[131] Federico Pizzuti, Michel Steuwer, and Christophe Dubach. "Position-dependent arrays and their application for high performance code generation." In: *FHPNC@ICFP*. ACM, 2019, pp. 14–26.

[132] Federico Pizzuti, Michel Steuwer, and Christophe Dubach. "Generating fast sparse matrix vector multiplication from a high level generic functional IR." In: *CC*. ACM, 2020, pp. 85–95.

[133] LLVM Compiler Infrastructure Project. *Clang: a C language family frontend for LLVM*. URL: http://clang.llvm.org/.

[134] LLVM Compiler Infrastructure Project. *Clang 11 Documentation*. 2020. URL: https://clang.llvm.org/docs/CommandGuide/clang.html.

[135] LLVM Compiler Infrastructure Project. *LLVM Language Reference*. 2020. URL: https://llvm.org/docs/LangRef.html\#abstract.

[136] LLVM Compiler Infrastructure Project. *LLVM's Analysis and Transform Passes*. 2020. URL: https://llvm.org/docs/Passes.html.

[137] Markus Püschel, José M. F. Moura, Jeremy R. Johnson, David A. Padua, Manuela M. Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. "SPIRAL: Code Generation for DSP Transforms." In: *Proceedings of the IEEE* 93.2 (2005), pp. 232–275.

[138] Jonathan Ragan-Kelley. "Decoupling algorithms from the organization of computation for high performance image processing." PhD thesis. Massachusetts Institute of Technology, Cambridge, MA. USA, 2014.

[139] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman P. Amarasinghe, and Frédo Durand. "Decoupling algorithms from schedules for easy optimization of image processing pipelines." In: *ACM Trans. Graph.* 31.4 (2012), 32:1–32:12.

[140] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman P. Amarasinghe, and Frédo Durand. "Halide: decoupling algorithms from schedules for high-performance image processing." In: *Commun. ACM* 61.1 (2018), pp. 106–115.

[141] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines." In: *PLDI*. ACM, 2013, pp. 519–530.

[142] Ari Rasch and Sergei Gorlatch. "ATF: A generic directive-based auto-tuning framework." In: *Concurr. Comput. Pract. Exp.* 31.5 (2019).

[143] Ari Rasch, Michael Haidl, and Sergei Gorlatch. "ATF: A Generic Auto-Tuning Framework." In: *HPCC/SmartCity/DSS*. IEEE Computer Society, 2017, pp. 64–71.

[144] Ari Rasch, Richard Schulze, and Sergei Gorlatch. "Developing High-Performance, Portable OpenCL Code via Multi-Dimensional Homomorphisms." In: *IWOCL*. ACM, 2019, 4:1.

[145] Ari Rasch, Richard Schulze, and Sergei Gorlatch. "Generating Portable High-Performance Code via Multi-Dimensional Homomorphisms." In: *PACT*. IEEE, 2019, pp. 354–369.

[146] Prashant Singh Rawat, Changwan Hong, Mahesh Ravishankar, Vinod Grover, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. "Resource Conscious Reuse-Driven Tiling for GPUs." In: *PACT*. ACM, 2016, pp. 99–111.

[147] Prashant Singh Rawat, Changwan Hong, Mahesh Ravishankar, Vinod Grover, Louis-Noël Pouchet, and P. Sadayappan. "Effective resource management for enhancing performance of 2D and 3D stencils on GPUs." In: *GPGPU@PPoPP*. ACM, 2016, pp. 92–102.

[148] Prashant Singh Rawat, Fabrice Rastello, Aravind Sukumaran-Rajam, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. "Register optimizations for stencils on GPUs." In: *PPOPP*. ACM, 2018, pp. 168–182.

[149] Prashant Singh Rawat, Miheer Vaidya, Aravind Sukumaran-Rajam, Mahesh Ravishankar, Vinod Grover, Atanas Rountev, Louis-Noël Pouchet, and P. Sadayappan. "Domain-Specific Optimization and Generation of High-Performance GPU Code for Stencil Computations." In: *Proceedings of the IEEE* 106.11 (2018), pp. 1902–1920.

[150] Toomas Remmelg. "Automatic performance optimisation of parallel programs for GPUs via rewrite rules." PhD thesis. The University of Edinburgh, 2019.

[151] Toomas Remmelg, Bastian Hagedorn, Lu Li, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. "High-level hardware feature extraction for GPU performance prediction of stencils." In: *GPGPU@PPoPP*. ACM, 2020, pp. 21–30.

[152] Toomas Remmelg, Thibaut Lutz, Michel Steuwer, and Christophe Dubach. "Performance portable GPU code generation for matrix multiplication." In: *GPGPU@PPoPP*. ACM, 2016, pp. 22–31.

[153] Facebook Research. *Announcing Tensor Comprehensions*. 2018. URL: https://research.fb.com/blog/2018/02/announcing-tensor-comprehensions/.

[154] John C. Reynolds. "The Discoveries of Continuations." In: *LISP Symb. Comput.* 6.3-4 (1993), pp. 233–248.

[155] Christoph Rieger, Fabian Wrede, and Herbert Kuchen. "Musket: a domain-specific language for high-level parallel programming with algorithmic skeletons." In: *SAC*. ACM, 2019, pp. 1534–1543.

[156] Tiark Rompf and Martin Odersky. "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs." In: *GPCE*. ACM, 2010, pp. 127–136.

[157] Tiark Rompf and Martin Odersky. "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs." In: *Commun. ACM* 55.6 (2012), pp. 121–130.

[158] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "Global Value Numbers and Redundant Computations." In: *POPL*. ACM Press, 1988, pp. 12–27.

[159] Gabe Rudy. "CUDA-CHiLL: A programming language interface for GPGPU optimizations and code generation." PhD thesis. School of Computing, University of Utah, 2010.

[160] Amir Shaikhha, Andrew Fitzgibbon, Dimitrios Vytiniotis, and Simon Peyton Jones. "Efficient differentiable programming in a functional array-processing language." In: *PACMPL* 3.ICFP (2019), 97:1–97:30.

[161] Michel Steuwer. "Improving programmability and performance portability on many-core processors." PhD thesis. University of Münster, 2015.

[162] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. "Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code." In: *ICFP*. ACM, 2015, pp. 205–217.

[163] Michel Steuwer and Sergei Gorlatch. "SkelCL: a high-level extension of OpenCL for multi-GPU systems." In: *J. Supercomput.* 69.1 (2014), pp. 25–33.

[164] Michel Steuwer, Michael Haidl, Stefan Breuer, and Sergei Gorlatch. "High-Level Programming of Stencil Computations on Multi-GPU Systems Using the SkelCL Library." In: *Parallel Process. Lett.* 24.3 (2014).

[165] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. "SkelCL - A Portable Skeleton Library for High-Level GPU Programming." In: *IPDPS Workshops*. IEEE, 2011, pp. 1176–1182.

[166] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. "Matrix multiplication beyond auto-tuning: rewrite-based GPU code generation." In: *CASES*. ACM, 2016, 15:1–15:10.

[167] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. "Lift: a functional data-parallel IR for high-performance GPU code generation." In: *CGO*. ACM, 2017, pp. 74–85.

[168] Larisa Stoltzfus, Alan Gray, Christophe Dubach, and Stefan Bilbao. "Performance Portability For Room Acoustics Simulations." In: *DAFx-17*. 2017.

[169] Larisa Stoltzfus, Bastian Hagedorn, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. "Tiling Optimizations for Stencil Computations Using Rewrite Rules in Lift." In: *TACO* 16.4 (2020), 52:1–52:25.

[170] Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. "Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages." In: *ACM Trans. Embedded Comput. Syst.* 13.4s (2014), 134:1–134:25.

[171] Gerald Jay Sussman and Guy L Steele. "Scheme: A interpreter for extended lambda calculus." In: *Higher-Order and Symbolic Computation* 11.4 (1998), pp. 405–439.

[172] TVM. *How to optimize GEMM on CPU*. 2020. URL: https://docs.tvm.ai/tutorials/optimize/opt_gemm.html.

[173] TVM. *How to optimize matmul with Auto TensorCore CodeGen*. 2020. URL: https://docs.tvm.ai/tutorials/optimize/opt_matmul_auto_tensorcore.html.

[174] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. "The pochoir stencil compiler." In: *SPAA*. ACM, 2011, pp. 117–128.

[175] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. "Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions." In: *CoRR* abs/1802.04730 (2018).

[176] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. "The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically." In: *TACO* 16.4 (2020), 38:1–38:26.

[177] Sven Verdoolaege. “isl: An Integer Set Library for the Polyhedral Model.” In: *Mathematical Software (ICMS'10)*. Ed. by Komei Fukuda, Joris Hoeven, Michael Joswig, and Nobuki Takayama. LNCS 6327. Springer-Verlag, 2010, pp. 299–302.

[178] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. “Schedule Trees.” In: *IMPACT*. 2014.

[179] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. “Polyhedral parallel code generation for CUDA.” In: *TACO* 9.4 (2013), 54:1–54:23.

[180] Eelco Visser. “A Survey of Strategies in Program Transformation Systems.” In: *Electron. Notes Theor. Comput. Sci.* 57 (2001), pp. 109–143.

[181] Eelco Visser. “Stratego: A Language for Program Transformation Based on Rewriting Strategies.” In: *RTA*. Vol. 2051. Lecture Notes in Computer Science. Springer, 2001, pp. 357–362.

[182] Eelco Visser. “Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in Stratego/XT 0.9.” In: *Domain-Specific Program Generation*. Vol. 3016. Lecture Notes in Computer Science. Springer, 2003, pp. 216–238.

[183] Eelco Visser. “A survey of strategies in rule-based program transformation systems.” In: *J. Symb. Comput.* 40.1 (2005), pp. 831–873.

[184] Eelco Visser, Zine-El-Abidine Benaissa, and Andrew P. Tolmach. “Building Program Optimizers with Rewriting Strategies.” In: *ICFP*. ACM, 1998, pp. 13–26.

[185] Hasitha Muthumala Waidyasooriya, Yasuhiro Takei, Shunsuke Tatsumi, and Masanori Hariyama. “OpenCL-Based FPGA-Platform for Stencil Computation and Its Optimization Methodology.” In: *IEEE Trans. Parallel Distrib. Syst.* 28.5 (2017), pp. 1390–1402.

[186] Craig J. Webb. “Parallel computation techniques for virtual acoustics and physical modelling synthesis.” PhD thesis. University of Edinburgh, UK, 2014.

[187] Gerhard Wellein, Georg Hager, Thomas Zeiser, Markus Wittmann, and Holger Fehske. “Efficient Temporal Blocking for Stencil Computations by Multicore-Aware Wavefront Parallelization.” In: *COMPSAC (1)*. IEEE Computer Society, 2009, pp. 579–586.

[188] R. Clinton Whaley and Jack J. Dongarra. “Automatically Tuned Linear Algebra Software.” In: *SC*. IEEE Computer Society, 1998, p. 38.

[189] Doran K. Wilde. *A Library for Doing Polyhedral Operations*. Tech. rep. 785. IRISA, 1993.

[190] Fabian Wrede, Christoph Rieger, and Herbert Kuchen. "Generation of high-performance code based on a domain-specific language for algorithmic skeletons." In: *J. Supercomput.* 76.7 (2020), pp. 5098–5116.

[191] Jingheng Xu, Haohuan Fu, Lin Gan, Yu Song, Hongbo Peng, Wen Shi, and Guangwen Yang. "Performance optimization of Jacobi stencil algorithms based on POWER8 architecture." In: *ASAP*. IEEE Computer Society, 2016, pp. 221–222.

[192] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman P. Amarasinghe. "GraphIt: a high-performance graph DSL." In: *Proc. ACM Program. Lang.* 2.OOPSLA (2018), 121:1–121:30.

[193] Jie Zhao and Albert Cohen. "Flextended Tiles: A Flexible Extension of Overlapped Tiles for Polyhedral Compilation." In: *TACO* 16.4 (2020), 47:1–47:25.

[194] Xing Zhou. "Tiling Optimizations For Stencil Computations." PhD thesis. University of Illinois at Urbana-Champaign, 2013.

[195] Xing Zhou, María Jesús Garzarán, and David A. Padua. "Optimal Parallelogram Selection for Hierarchical Tiling." In: *TACO* 11.4 (2014), 58:1–58:23.

LEBENSLAUF

ZUR PERSON

	Bastian Hagedorn
	geboren am 03.10.1990 in Ibbenbüren
Familienstand:	verheiratet
Nationalität:	deutsch
Name des Vaters:	Wolfgang Hagedorn
Name der Mutter:	Marianne Hagedorn geb. Block

SCHULBILDUNG

08/1997 – 07/2001	Grundschule, Ibbenbüren
08/2001 – 06/2010	Gymnasium, Ibbenbüren
	Abitur am 26.06.2010

STUDIUM

10/2011 – 09/2014	Bachelorstudiengang Informatik Westfälische Wilhelms-Universität Münster
	Bachelor of Science am 12.08.2014
10/2014 – 09/2016	Masterstudiengang Informatik Westfälische Wilhelms-Universität Münster
	Master of Science am 20.09.2016
seit 10/2016	Promotionsstudiengang Informatik, Westfälische Wilhelms-Universität Münster

TÄTIGKEITEN

09/2010 – 08/2011	Zivildienst, Ibbenbüren
10/2013 – 03/2016	Studentische Hilfskraft, Westfälische Wilhelms-Universität Münster
seit 10/2016	Wissenschaftlicher Mitarbeiter, Westfälische Wilhelms-Universität Münster

BEGINN DER DISSERTATION

seit 10/2016	Institut für Informatik, Westfälische Wilhelms-Universität Münster
	betreut durch Prof. Dr. Sergei Gorlatch

colophon

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography *"The Elements of Typographic Style"*. `classicthesis` is available for both LaTeX and LyX:

`https://bitbucket.org/amiede/classicthesis/`