Kœhler, Thomas (2022) *A domain-extensible compiler with controllable automation of optimisations.* PhD thesis.

A Domain-Extensible Compiler with Controllable Automation of Optimisations

Thomas Kœhler

Submitted in fulfilment of the requirements for the
Degree of Doctor of Philosophy

School of Computing Science
College of Science and Engineering
University of Glasgow

2022

Abstract

In high performance domains like image processing, physics simulation or machine learning, program performance is critical. Programmers called *performance engineers* are responsible for the challenging task of optimising programs. Two major challenges prevent modern compilers targeting heterogeneous architectures from reliably automating optimisation. First, domain-specific compilers such as Halide for image processing and TVM for machine learning are difficult to extend with the new optimisations required by new algorithms and hardware. Second, automatic optimisation is often unable to achieve the required performance, and performance engineers often fall back to painstaking manual optimisation.

This thesis shows the potential of the SHINE compiler to achieve domain-extensibility, controllable automation, and generate high performance code. *Domain-extensibility* facilitates adapting compilers to new algorithms and hardware. *Controllable automation* enables performance engineers to gradually take control of the optimisation process.

The first research contribution is to add 3 code generation features to SHINE, namely: synchronisation barrier insertion, kernel execution, and storage folding. Adding these features requires making novel design choices in terms of compiler extensibility and controllability. The rest of this thesis builds on these features to generate code with competitive runtime compared to established domain-specific compilers.

The second research contribution is to demonstrate how extensibility and controllability are exploited to optimise a standard image processing pipeline for corner detection. SHINE achieves 6 well-known image processing optimisations, 2 of them not being supported by Halide. Our results on 4 ARM multi-core CPUs show that the code generated by SHINE for corner detection runs up to $1.4\times$ faster than the Halide code. However, we observe that controlling rewriting is tedious, motivating the need for more automation.

The final research contribution is to introduce *sketch-guided equality saturation*, a semi-automated technique that allows performance engineers to guide program rewriting by specifying rewrite goals as sketches: program patterns that leave details unspecified. We evaluate this approach by applying 7 realistic optimisations of matrix multiplication. Without guidance, the compiler fails to apply the 5 most complex optimisations even given an hour and 60GB of RAM. With the guidance of at most 3 sketch guides, each 10 times smaller than the complete program, the compiler applies the optimisations in seconds using less than 1GB.

Contents

Acknowledgements

This thesis is the result of 3 short years of research, and 6 long months of write-up. I am grateful to those who helped me throughout this journey.

Thanks to my supervisors, Michel Steuwer and Phil Trinder, for their encouragements, guidance, and constructive feedback. Michel, you always cared about my personal growth, I owe you for attending many interesting events and learning to confidently promote my work. Phil, your helicopter view helped me to focus on the bigger picture, strengthened my research and writing. I am still using the mug from your last supply drop.

Thanks to everybody in the School of Computing Science in Glasgow for creating such a welcoming environment. The pandemic made me realise how much I missed the social lunches, coffee breaks and pub nights. I had the pleasure of many refreshing interactions with Adrian Ramsingh, Dejice Jacob, Cristian Urlea and Kyle Simpson in our shared F101 office.

Thanks to the Halide developers and egg developers for maintaining open-source code, to Max Willsey for sharing his insights on equality saturation.

Thanks to the entire Lift, Rise and Elevate teams for their work and our shared moments. Bastian Köpcke and Federico Pizzuti, you developed the Shine compiler at my side, and shared many pre-pandemic whiteboard sessions. Christof Schlaak, you unexpectedly shared the post-pandemic write-up experience with me (or was it a table tennis competition?).

Above all, thanks to my loved ones, who gave me the strength to withstand this journey. Gözel Shakeri, my partner, you gave meaning to work-life balance, celebrated my successes, and relativized my failures. Madeleine Nœuveglise and François Kœhler, my parents, you supported me unconditionally as I moved far away from home. My friends, stay who you are. My levain, may you live a long and sour life.

Declaration

The work reported in this thesis is primarily my own unless otherwise explicitly stated, and has not been submitted for any other degree or qualification. This thesis unifies and expands work reported in the following publications:

[1] Thomas Kœhler and Michel Steuwer. "Towards a Domain-Extensible Compiler: Optimizing an Image Processing Pipeline on Mobile CPUs". In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2021, pp. 27–38

[2] Thomas Kœhler, Phil Trinder, and Michel Steuwer. *Sketch-Guided Equality Saturation: Scaling Equality Saturation to Complex Optimizations in Languages with Bindings*. 2021. arXiv: `2111.13040` `[cs.PL]`

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

[3] Bastian Hagedorn, Johannes Lenfers, Thomas Kœhler, 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: *Proc. ACM Program. Lang.* 4.ICFP (2020), 92:1–92:29. DOI: `10.1145/3408974`

[4] Michel Steuwer, Thomas Kœhler, Bastian Köpcke, and Federico Pizzuti. *RISE & Shine: Language-Oriented Compiler Design*. 2022. arXiv: `2201.03611` `[cs.PL]`

[5] Jackson Woodruff, Thomas Kœhler, Alexander Brauckmann, Sam Ainsworth, Michel Steuwer, and Michael O'Boyle. *Rewriting History: Repurposing Domain-Specific Accelerators with Rewrite Exploration*. under submission. 2022

Chapter 1

Introduction

Breakthroughs in artificial intelligence keep making our machines smarter: they become better at diagnosing diseases, driving vehicles, predicting the weather, understanding natural languages, and more. Part of this success is due to machine learning models like artificial neural networks that have increasingly more neural connections (aka parameters). Networks went from millions of parameters (e.g. ResNet) to billions of parameters (e.g. Microsoft Turing), and are anticipated to reach trillions of parameters. Such models require massive computing power, which is a major issue [6] since physical resources are limited (left of Figure 1.1). In the cloud, huge data centres are taking an increasing toll on our planet: a recent study estimates that training a single natural language processing neural network has a carbon footprint equivalent to driving 4,311 km in a European car and would take 823 tree-months to offset [7]. At the edge, running advanced neural networks on small devices with low energy supply is not yet possible. To sustain progress, we need to use resources more efficiently.

In high performance domains like image processing, physics simulation or machine learning, resources are used more efficiently through continuous algorithm, hardware and program optimisation (right of Figure 1.1). Here domain scientists are responsible for optimising algorithms, hardware architects optimise hardware, and *performance engineers* are programmers
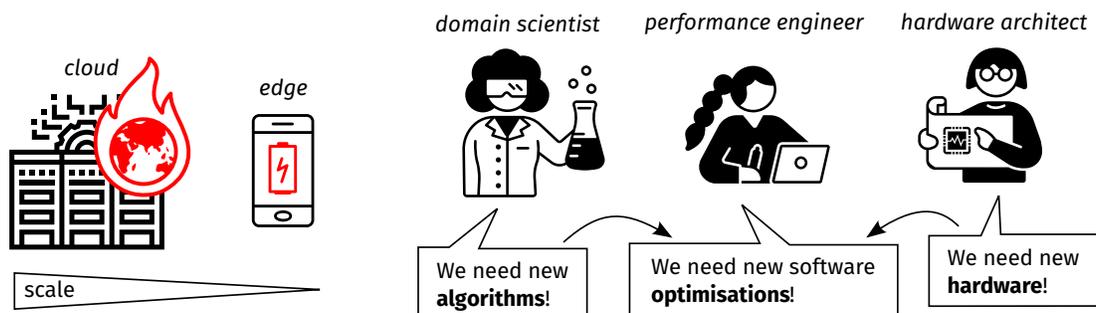

Figure 1.1: (Left-to-Right) Physical resources are limited at all scales: from the impact of the cloud on our planet, to battery limitations at the edge. Different actors optimise resource usage: while domain scientists design new algorithms, hardware architects design new hardware, and performance engineers design new program optimisations.

responsible for optimising programs. The three optimisation aspects are inter-dependent: in particular, for each new algorithm and hardware architecture, new specialised program optimisations are required.

Optimising programs is challenging. On one side, skilled performance engineers perform optimisations manually on low-level code (e.g. C, OpenCL), which takes months and risks introducing bugs, slowing down algorithm and hardware innovation. On the other side, modern compilers targeting heterogeneous and parallel hardware do not reliably automate program optimisation, facing the two following major challenges.

Challenge 1) Extending compilers with new program optimisations is hard.

Traditional *domain-agnostic* compilers such as LLVM [8] or GCC only support a fixed, generic set of program abstractions and optimisations. As a result, they do not automate optimisations for specific domains and performance engineers must perform them manually (Section 2.1).

Established *domain-specific* compilers such as Halide [9] for image processing or TVM [10] for machine learning only support a fixed, specialised set of program abstractions and optimisations. Although they successfully automate many optimisations, extending them with new optimisations as the domain evolves is difficult [11]. Domain evolution includes changes in algorithms, changes in program optimisations, but also changes in specialised hardware (e.g. machine learning accelerators [12]). Performance engineers are not necessarily compiler writers, and falling back to manual optimisation is often easier for them than modifying a compiler to achieve their goals. In consequence, many state-of-the-art high-performance kernel libraries (e.g., BLAS, cuDNN, MKL) are still predominantly written by hand [13].

Recently, *domain-extensible* compilers such as Delite [14], LIFT [15, 16], or AnyDSL [17] provide an extensible set of program abstractions and optimisations, showing potential to mitigate *Challenge 1* (Section 2.2). However, domain-extensible compilers remain relatively immature and lack adoption compared to established domain-specific compilers. For example, previous work does not investigate whether LIFT is able to generate code with competitive runtime performance compared to established domain-specific compilers.

Challenge 2) Manual optimisation is tedious, automatic optimisation is unsatisfactory.

Compilers such as LIFT [15] or PolyMage [18] aim for *full automation* of optimisations, removing performance engineers from the optimisation process. This is highly desirable in scenarios where performance engineers are not available. However, full automation of optimisations is not always feasible or even desirable as it may result in poor performance or may be too time-consuming [19, 20, 21]. When compiler optimisations are unsatisfactory, performance engineers often fall back to manual optimisation in order to achieve their performance goals [22, 23, 24].

Compilers such as Halide offer *control* of optimisations through schedules [9, 10, 25]. However, schedules are challenging to write [26]. Successful research goes into automatically generating schedules [27, 28, 29], but sacrifices control for full automation of optimisations.

This thesis explores *controllable automation* (Section 2.3.3) of optimisations that embraces trade-offs between full automation and precise control of optimisations. Controllable automation enables performance engineers to gradually take control, instead of abruptly falling back to manual optimisation when compiler optimisations are unsatisfactory, and has the potential to mitigate *Challenge 2* (Section 2.3).

Thesis Statement   This thesis shows the potential of a novel compiler design to achieve domain-extensibility, controllable automation, and generate high performance code. Domain-extensibility is combined with controlled rewriting in an image processing case study to generate faster code than the established domain-specific compiler Halide. Controllable automation is exploited in a linear algebra case study to automatically explore an optimisation space while providing the performance engineer with control over the optimisation outcome.

This thesis makes the following research contributions:

1. **Enhancing Code Generation in a Domain-Extensible Compiler** (Chapter 3).
   Three important code generation features are added to a domain-extensible compiler with controllable automation of optimisations called SHINE, making it possible to:

   - generate correct and efficient synchronisation barriers
   - generate multiple computation kernels and the host code launching them
   - generate folded storage for temporary arrays

   Adding these features requires making novel design choices in terms of compiler extensibility and controllability. Crucially the following chapters require these features to generate imperative code with competitive runtime performance compared to established domain-specific compilers such as Halide.

2. **Going Beyond Halide Scheduling with Controlled Rewriting** [1] (Chapter 4).
   Domain-extensibility is combined with controlled rewriting to optimise a standard image processing pipeline: the Harris corner detection [30]. We show how rewriting is controlled in SHINE to achieve 6 well-known image processing pipeline optimisations, including 2 optimisations that are not supported by Halide schedules. Our results on four mobile ARM multi-core CPUs, often used for image processing tasks, show that the code generated using SHINE for the Harris corner detection is up to 1.4× (geomean of 1.27×) faster than Halide. However, we also observe that controlling rewriting is tedious, motivating the following chapter that aims to lower performance engineer effort with semi-automation.

3. **Proposing a Novel Semi-Automatic Optimisation Technique** [2] (Chapter 5).

   A new semi-automatic optimisation technique called *sketch-guided equality saturation* is developed, based on a fully automated technique called equality saturation. Sketch-guiding allows performance engineers to guide program rewriting by specifying rewrite goals as *sketches*: program patterns that leave details unspecified. We evaluate sketch-guided equality saturation by applying 7 realistic optimisations of matrix multiplication. Unguided equality saturation alone does not scale to the 5 most complex optimisations, even given an hour and 60GB of RAM. With the guidance of at most 3 sketch guides, each 10 times smaller than the complete program, the compiler applies the optimisations in seconds using less than 1GB of RAM. We also explore how to efficiently encode a polymorphically typed lambda calculus for equality saturation. The runtime and memory consumption of unguided equality saturation over lambda terms is reduced by orders of magnitude, which is also beneficial for sketch-guided equality saturation.

Chapter 2

Motivating Background

This chapter motivates the contributions of this thesis by presenting technical background and related work. The remaining background will be introduced separately in the technical chapters. Similarly, more specific related work will be presented throughout the thesis and discussed in Chapter 6.

Section 2.1 provides background on how performance engineers traditionally program and optimise performance-demanding applications in low-level languages like C and OpenCL. Performance engineers typically achieve orders of magnitude performance improvements by manually exploring potential optimisations using their expert knowledge. Unfortunately this optimisation process is time consuming, risks introducing bugs, and needs to be repeated for each new algorithm and hardware target.

To improve over this traditional optimisation process, we would like to combine higher-level programming models with compilers that automate program optimisations, but this is a challenging task.

Section 2.2 motivates research on *domain-extensible* compilers. The compiler *extensibility challenge* is introduced: extending compilers with new optimisations is hard. Existing compilers are categorised into traditional domain-agnostic compilers; established domain-specific compilers; and emerging domain-extensible compilers. The advantages and disadvantages of each category of compilers are highlighted.

Section 2.3 motivates research on *controllable automation* of optimisations. The compiler *controllability challenge* is introduced: even though manual optimisation is tedious for performance engineers, automatic optimisation is not always satisfactory. The advantages and disadvantages of automated optimisation and controlled optimisation are discussed before introducing the concept of controllable automation of optimisations.

2.1 High Performance Programming

2.1.1 Hardware Architectures are Evolving

In the quest to maximise runtime performance and energy efficiency, hardware architects continuously come up with significant architectural changes. First, hardware architectures became highly parallel [31, 32]. Nowadays, hardware architectures are increasingly heterogeneous [33, 34]. Such hardware evolution seriously impacts high performance programming practices. Performance engineers must consider hardware parallelism and heterogeneity to achieve high performance when optimising software [35].

In particular, various forms of parallelism are exploited:

- *instruction-level parallelism*, where multiple instructions are executed simultaneously. The hardware commonly exploits this parallelism implicitly at runtime (e.g. instruction pipelining, superscalar execution), but some hardware explicitly exposes this parallelism to software instead (e.g. Very Long Instruction Word).

- *vector-level parallelism*, where multiple vector elements are processed simultaneously. This is also referred to as Single Instruction, Multiple Data (SIMD). Many processors explicitly expose this parallelism. As a result, software must use explicit instructions to exploit this parallelism.

- *thread-level parallelism*, where multiple software threads are executed simultaneously. A thread is a programmed sequence of instructions that can be scheduled independently to other threads on the available hardware cores. Care must be taken when developing explicitly multi-threaded software, as it is a difficult and error-prone task [36].

Two common hardware architectures are often combined in heterogeneous systems: multi-core Central Processing Units (CPUs) and many-core Graphics Processing Units (GPUs).

Multi-Core CPUs Optimised for individual thread performance and low latency, multi-core CPUs have a few sophisticated cores able to run relatively heavyweight threads. CPUs dynamically adapt to various usage patterns through complex execution, control flow and memory management. To exploit locality of memory accesses, CPUs have a hierarchy of caches that automatically stores recently accessed data closer to the cores. Three levels of caches named L1, L2 and L3 are commonly used; where L1 is the closest to the cores. The closer to a core a cache is, the smaller and faster it is. Even though data is moved implicitly between caches, program optimisations can still improve cache usage by improving the locality of accesses.

While CPUs focus on individual thread performance, they still expose significant parallelism. It is standard for CPUs to execute 8 to 16 threads simultaneously and to provide 128 to 256 bits of vector parallelism (i.e. 4 to 8 32-bit values) on top of instruction-level parallelism.

Many-Core GPUs   Optimised for high throughput, many-core GPUs hide high latency operations by overlapping the execution of many relatively lightweight threads. Originally specifically designed to accelerate 3D graphics rendering, GPUs have been generalised to support more use cases and are becoming increasingly flexible [37]. GPUs are now popular for massively parallel and regular computations in many performance demanding domains.

It is standard for GPUs to contain hundreds or thousands of cores, grouped hierarchically to share hardware resources during execution. For example, while GPUs have a cache hierarchy, they also typically provide scratchpad memory which is shared by a group of threads. A scratchpad is similar to a cache, but is less sophisticated because data transfers are explicitly programmed instead of implicit. In general, GPUs tend to avoid sophisticated hardware techniques with high hardware cost and instead favor simpler techniques with low hardware cost. This shift from implicit hardware logic to explicit software programming makes program optimisation both more important and difficult.

Other Processors   While this thesis focuses on CPUs and GPUs, heterogeneous systems also include reconfigurable hardware such as Field-Programmable Gate Arrays (FPGAs) or Coarse-Grained Reconfigurable Architectures (CGRAs); as well as highly specialised hardware such as Digital Signal Processors (DSPs) or Tensor Processing Units (TPUs). The contributions of this thesis may be combined with related work that tackles such hardware [38, 39, 40]. For example, the D2A methodology [41] addresses compilation challenges that are specific to accelerators, such as the mismatch between fine-grained compiler IR (Intermediate Representation) operations and coarse-grained accelerator operations, and the need for a formal hardware specification. In the future, processors are likely to become even more diverse, complicating program optimisation tasks: performance engineers may need to exploit quantum [42], neuromorphic [43] or optical processors [44].

2.1.2   The OpenCL Programming Model is too Low-Level

To achieve high performance on parallel and heterogeneous hardware, the industry standard is to use relatively low-level programming models. In this thesis, *low-level code* is code that is specialised to a given hardware for performance reasons. By extension, a *low-level programming model* is a programming model where non-specialised code performs poorly, i.e. the programming model fails to provide what is also called performance portability [45]. We specifically provide more background on the OpenCL programming model [46, 47], an open standard targeting diverse processors that we will use in this thesis.

The OpenCL standard consists of a C API to orchestrate computation from the *host* and a programming language to express computation on *OpenCL devices*.

Compute Model   Programs executed on a device are called *kernels*. A kernel is a program that each thread executes in a *Single Program, Multiple Data* (SPMD) fashion. The host program may submit a kernel for execution on a device by specifying an N-dimensional index space (where N goes from 1 to 3). For each point in the index space, an instance of the kernel program called *work-item* is executed. Each point in the index space is a global identifier for the corresponding work-item and can be queried by the kernel program to inform its execution.

The index space is decomposed into *work-groups* of multiple work-items that may share hardware resources for cooperation. Each work-group has a corresponding identifier, and each work-item has a local identifier within its work-group. This thread hierarchy closely corresponds to the compute hierarchy found in most GPU architectures, allowing efficient runtime scheduling, and is compatible with execution on other architectures such as CPUs.

Memory Model   The OpenCL memory model also closely corresponds to the memory hierarchy of common GPU architectures, exposing different address spaces:

- *Global memory* is accessible to all work-items in all work-groups, and also to the host program. On CPUs and GPUs, it typically corresponds to device RAM.

- *Local memory* is only accessible to work-items of the same work-group. It usually corresponds to RAM on CPUs, and faster scratchpad memory on GPUs.

- *Private memory* is only accessible to a single work-item. On CPUs and GPUs, it typically corresponds to registers, the fastest possible memory.

OpenCL also provides *constant memory*, read-only global memory that remains constant during kernel execution, however we will not exploit this type of memory in this thesis.

Kernel Programming   The OpenCL C programming language, used to define kernels, is derived from the C99 specification. On one hand, OpenCL C disallows pointers to functions, recursion, and dynamic memory allocation. On the other hand, OpenCL C introduces address space qualifiers, custom data types and built-in functions.

Listing 2.1 shows a simple implementation of point-wise addition of two arrays using an OpenCL kernel. In line 1, the kernel signature consists of pointers referring to arrays in global memory (`a`, `b`, `output`) and an integer representing the size of these arrays (`n`). The kernel program is executed by multiple threads, and starts by querying the global identifier of the current thread in line 2 (`get_global_id(0)`). Depending on the number of available threads (`get_global_size(0)`), the program may loop depending on how many elements the current thread needs to compute. An element is computed inside the loop using indexing in line 3.

```
1  kernel void add_v1(global float* a, global float* b, global float* output, int n) {
2    for (int gid = get_global_id(0); gid < n; gid += get_global_size(0)) {
3      output[gid] = a[gid] + b[gid];
4    }
5  }
```

Listing 2.1: Point-wise addition of two arrays in OpenCL

In practice, optimised kernel implementations are significantly more complex. Listing 2.2 shows another implementation of the same computation. The program leverages vector-level parallelism using `vload4` and `vstore4` to transfer vectors of 4 single-precision floating-point values, as well as a vectorised addition. The program loops are tiled to increase single-thread workload and potentially improve data locality: chunks of 4 vectors are computed together.

```
1  kernel void add_v2(global float* a, global float* b, global float* output, int n) {
2    for (int gid = get_global_id(0); gid < n/16; gid += get_global_size(0)) {
3      for (int i = 0; i < 4; i += 1) {
4        float4 av = vload4(gid * 4 + i, a);
5        float4 bv = vload4(gid * 4 + i, b);
6        vstore4(av + bv, gid * 4 + i, output);
7      }
8    }
9  }
```

Listing 2.2: Point-wise addition of two arrays in OpenCL: tiled, vectorized.

Optimised code is typically orders of magnitude faster than naive code, but is also more complex and specialised to a given hardware. For performance engineers, manually optimising low-level code using techniques such as tiling or vectorisation is time-consuming and risks introducing bugs. This is aggravated by the fact that many optimisation decisions require global thinking: naive compositions of locally optimised code often perform poorly.

Other Industrial Programming Standards   The *C* and *C++* languages are commonly used to program CPUs, in combination with primitives such as *POSIX threads* for multi-threading and *SIMD instructions* (e.g. SSE, AVX, NEON) for vectorisation. *CUDA* is a popular closed framework developed by NVIDIA for NVIDIA GPUs. It is also common to build convenient abstractions on top of these standards. *SYCL* allows single-source programming in C++ for OpenCL, avoiding the need to write separate OpenCL C code, and providing implicit memory transfers. *OpenMP* and *OpenACC* are used for quick code acceleration with compiler directives, such as parallelising a for loop by preceding it with `#pragma omp parallel for`. All of these programming models suffer from similar performance engineering productivity and portability issues as OpenCL: non-specialised code performs poorly compared to specialised code.

2.1.3 Higher-Level Programming Models are Challenging to Compile

Higher-level programming models and languages are a promising way to increase productivity, performance and portability. We give concrete examples with domain-specific languages and array programming languages before mentioning common compilation challenges.

Domain-Specific Languages   Domain-Specific Languages (DSLs) enable combining convenient, hardware-agnostic programming with high-performance for specific domains. This is a proven success for domains such as image processing [9, 48, 18], signal processing [49], linear algebra [50], machine learning [10] or partial differential equations [51, 52].

For example, the point-wise addition of two arrays can be defined in the Halide image processing DSL by writing a high-level *algorithm* as in Listing 2.3. Crucially, DSLs separate concerns: domain scientists can focus on writing high-level programs (what to compute) while compilers and performance engineers optimise programs for performance (how to compute). For example, Listing 2.3 could be compiled by Halide into an OpenCL implementation (e.g. Listing 2.1, Listing 2.2) or into some other optimised implementation, depending on the target hardware.

```
Var i; Func output;

output(i) = a(i) + b(i);
```

Listing 2.3: Point-wise addition of two arrays in Halide

Array Programming Languages   Array programming languages such as Accelerate [53], SaC [54], NOVA [55], Lift [15], Futhark [56], or Dex [57] provide a middle-ground between highly specialised languages and general-purpose languages. Multi-dimensional arrays are a key abstraction for many performance-demanding application domains (e.g. physics, chemistry, mechanics, image processing, data processing, linear algebra, machine learning), as they can be used to represent widely used tensors [58, 59].

Some array programming languages focus on index-based notations [54, 57], similar to how the i index is used in the Halide algorithm of Listing 2.3. Other array programming languages focus on collective array operations such as map or reduce [53, 55, 15], that can also be seen as algorithmic skeletons [60]. For example, the point-wise addition of two arrays can be defined in the Lift language as in Listing 2.4. **zip** a b combines two arrays a and b whose elements are added pairwise using **map**, and no explicit indexing is required.

```
map (λx. (fst x) + (snd x)) (zip a b)
```

Listing 2.4: Point-wise addition of two arrays in Lift

Array programming languages offer the potential to share common infrastructure across multiple application domains while still being convenient and hardware-agnostic.

Developing Compilers for High-Level Programs   Whether high-level programs are written in domain-specific or array languages, building and maintaining compilers that generate high-performance code for a variety of algorithmic domains and hardware architectures is challenging.

Industrial-strength compilers such as Halide for image processing and TVM for machine learning are the result of years of engineering effort from multiple contributors. The Halide repository contains 278k lines of code written by 167 contributors.[1] The TVM repository contains 794k lines of code written by 752 contributors.[2] Further, both use other compiler technologies such as LLVM as backends. LLVM 8.0.1 comprises 6,887k lines of code written by 1,210 contributors, with an estimated development cost of 529M$ [61].

The following sections motivate the need to address two specific compiler challenges: the *extensibility challenge* and the *controllability challenge*.

2.2   The Compiler Extensibility Challenge

2.2.1   Domain-Agnostic Compilers

Traditional, general-purpose compilers such as LLVM [8] or GCC are *domain-agnostic*: they only support a fixed, generic set of program abstractions and optimisations.

LLVM is a widely adopted compiler framework consisting of production-quality libraries for modular compiler construction. The LLVM IR (Intermediate Representation) provides many generic abstractions that are useful across multiple programming languages and hardware targets, such as types, functions and exceptions. LLVM also provides many powerful optimisation passes, such as dead code elimination, common subexpression elimination, function inlining, strength reduction or target-specific instruction selection. LLVM is successfully used as part of compilers for the C/C++ (clang compiler), Rust and Swift languages, to name a few.

While the LLVM framework is modular and extensible with custom compiler analyses and compiler passes, the LLVM IR is generic and even lower-level than languages like C or OpenCL. Higher-level program constructs must be lowered by compilers targeting LLVM, erasing domain-specific information which can be hard or impossible to recover. As a result, domain-agnostic compilers typically do not automate optimisations for specific domains. Instead, performance engineers must perform them manually as seen in Section 2.1.2.

---

[1]`https://github.com/halide/Halide/commit/820ec1f963f06f53a1808eb9b4631f2031be7468`
[2]`https://github.com/apache/tvm/commit/178f82dc481bf31961206412c22dd5519a245b49`

Optimised Libraries   Numerous *libraries of optimised functions* (e.g. BLAS, cuDNN, MKL, NVIDIA Performance Primitives, ARM Compute Library, Intel IPP, OpenCV) provide optimised implementations of common computations. However, they are developed by performance engineers at high cost. Limited engineering budget leads to limited functionality and limited support for certain use cases or hardware. Moreover, the composition of optimised functions through library calls is often far from optimal because many optimisations cannot be applied across library calls. For example, Weld [62] achieves order of magnitude performance improvements by optimising across libraries and functions using a common intermediate representation.

2.2.2   Domain-Specific Compilers

Established *domain-specific* compilers only support a fixed, specialised set of program abstractions and optimisations. Examples include Halide [9] and PolyMage [18] for image processing; TVM [10] for machine learning; SPIRAL for signal processing [49]; Firedrake for partial differential equations [52].

Domain-specific compilers are successfully established in industry where they automate many domain-specific optimisations and achieve impressive performance results. Halide is used for some processing tasks in the Google Pixel camera, Adobe Photoshop, and YouTube. TVM is used and developed by companies like AMD, ARM, Microsoft, NVIDIA, and Samsung.

However, extending domain-specific compilers with new optimisations as the domain evolves is difficult [11]. Domain evolution includes changes in algorithms, changes in program optimisations, but also changes in specialised hardware.

Compiler extensions are particularly difficult when they have an impact on the entire compilation stack, requiring simultaneous expertise of high-level algorithms, multiple compilation aspects, and low-level hardware targets. Separating concerns facilitates extension, for example by separating the definition of correct transformations from the search for best transformations as shown later in this thesis.

Extending Halide is Hard   The Halide Development Roadmap from 2020[3] highlights that extending Halide is hard, listing unsolved questions such as:

- "How do we make Halide easier to use for researchers wanting to [...] extend it [...]?"
- "How do we make Halide more useful on current and upcoming hardware?"
- "How do we make Halide more useful for new types of application?"

In fact, solutions to the extensibility challenge have been independently researched in the course of this thesis by Halide authors [63, 13], showing that our research direction is valuable and our motivations shared by the broader community.

---

[3]`https://github.com/halide/Halide/issues/5055`

Extending TVM is Hard   The TVM Unity vision for 2022[4] demonstrates that, currently, extending TVM is hard. Two problematic boundaries are identified in the compilation stack. First, vertical boundaries between successive abstraction layers. Each local decision made within a layer has a global impact on the following layers. As a result, introducing support for new hardware in the last layer requires re-thinking how decisions are made in all prior layers, requiring global instead of local engineering efforts. Second, horizontal boundaries between lowering strategies. For example, TVM may decide to target optimised libraries or to generate code from scratch, but typically cannot combine both approaches.

> "[...] boundaries are slowing down the pace of innovation in machine learning. New hardware accelerators are emerging with new levels of capability and performance, but harnessing them will require fluid collaboration between ML scientists, ML engineers, hardware vendors that these boundaries prevent. To cope with the rapid pace of change in ML systems, frameworks need to support incremental evolution: Incorporating new capabilities should require effort proportional to the change, not wholesale re-engineering at each level." (TVM Unity vision)

2.2.3   Domain-Extensible Compilers

Building DSLs and their compilers has long been recognized as a highly complex task, which is why many projects aim to simplify the development of DSLs. For example, Spoofax [64] is a language workbench simplyfing the development of DSLs. The idea is to produce parsers, type checkers, compilers, interpreters and even IDE support from declarative language specifications. Gammars are used to declare syntax, rewrite rules to declare semantics. With Spoofax, multiple domain-specific compilers are still constructed even if the task is simplified. The constructed compilers may still be hard to adapt as their domain evolves, as seen for established domain-specific compilers in Section 2.2.2. Another approach is to build *domain-extensible* compilers providing an extensible set of program abstractions and optimisations.

Delite [14] is a DSL framework providing a fixed set of generic parallel patterns that DSLs can target. Domain-specific optimisations can be defined using staging, a technique used to eliminate the cost of DSL abstractions while lowering them to generic parallel patterns. Delite demonstrated that multiple DSLs can be composed together while sharing common infrastructure [65]. The Delite projet also identified that its staging mechanism is not sufficient, and showed that using term rewriting techniques is beneficial for certain use cases [66].

AnyDSL [17] is a more recent approach leveraging partial evaluation to replace DSL compilers with self-specialising library code written in a language called Impala. The idea of implementing languages as libraries has a long strand of research predating AnyDSL, such as the Racket language that leverages macros instead of partial evaluation [67].

---

[4]`https://tvm.apache.org/2021/12/15/tvm-unity`

LIFT [15] combines a high-level functional language with an extensible rewrite system to define an optimisation space and generate high-performance code. LIFT has demonstrated its ability to generate high-performance code by deriving multiple low-level implementations from a single portable high-level program, obtaining performance on par with highly tuned platform-specific libraries on various GPUs [68]. LIFT is extensible with rewrite rules and language primitives by design, and has been successfully extended for stencil computations [16] which are common in multiple domains including image processing.

Overall, domain-extensible compilers show a lot of potential, but they remain relatively immature and lack adoption compared to established domain-specific compilers. This thesis builds on the foundations laid by LIFT and addresses some of its shortcomings. For example, previous work does not investigate whether LIFT is able to generate code with competitive runtime performance compared to Halide for image processing as we do in Chapter 4. LIFT also does not address the compiler controllability challenge that we discuss in the next section.

While we follow a rewrite-based approach, multiple extensible techniques can complement each other, as demonstrated by Delite for staging and rewriting. We believe that research on each technique is useful on its own.

Extending Compilers with Rewrite Rules   Many compilers before LIFT already allowed programmers to express domain-specific optimisations as rewrite rules. The Glasgow Haskell Compiler (GHC) allows this, but only applies rewrite rules according to a simple strategy, assuming that the right-hand side of a rule is always preferable to the left [69]. The following pragma would instruct GHC to always rewrite `map f (map g xs)` into `map (f . g) xs`:

```
{-# RULES
  "map/map" forall f g xs.
    map f (map g xs) = map (f . g) xs
#-}
```

While such a simple strategy can be effective in optimising programs, it falls short for use cases where deciding which rewrite rule is beneficial when is hard.

Stratego [70] is a language for defining customised rewriting strategies, which is included in the Spoofax language workbench. Phobos is a compiler frontend enabling the development of domain-specific languages by combining an open term language with term rewriting [71].

On one hand, tools like GHC, Stratego and Phobos allow domain-extensbility via rewrite rules but do not focus on high-performance code generation. On the other hand, tools like Spiral [72, 49] successfully generate high-performance code for heterogeneous hardware using rewrite rules, but are domain-specific and not extensible. This thesis follows LIFT's ambition of combining domain-extensibility with high-performance code generation for heterogeneous hardware using rewrite rules.

2.3 The Compiler Controllability Challenge

2.3.1 Automatic Optimisation

Many compilers aim for fully automated optimisation, some of them lean towards *greedy optimisation*, while others lean towards *explorative optimisation*.

Greedy Optimisation Greedy optimisation is often achieved using a fixed pipeline of optimisation *passes*, relying on expert heuristics to make local decisions that are assumed to be beneficial for final performance.

For example, the LLVM framework [8] provides many transformation and optimisation passes, and allows defining custom passes. In GHC, user-defined rewrite rules are applied greedily [69]. Many other projects rely on greedy optimisation passes, such as Accelerate [73] and the SHIR compiler that adopts ideas from LIFT to target FPGAs [40].

Typically, greedy optimisation is fast, but may result in poor performance as it gets stuck in local minima and may poorly predict performance benefits. The problem of deciding when to apply which optimisation pass is known as the *phase ordering problem*, and has a huge impact on the performance of the rewritten program.

Explorative Optimisation Explorative optimisation is a more holistic approach, where a space of possible implementations arising from different combinations of optimisations is explored, in order to make a decision.

Random sampling, or Monte Carlo methods, can be used to navigate the space of possible implementations. Using this approach, finding an implementation with high enough performance can be very time consuming. LIFT [15] uses random sampling, and optimising a single convolution takes hours to reach peak performance on a GPU [16, 74].

Performance models can be used to choose from the space of possible implementations. Some performance models are analytical, describing performance as an equation, as done in TyTra [38, 75], SPIRAL [72, 49] and Telamon [76]. Analytical performance models are typically fast to evaluate, estimating a program's performance without executing it. Creating an accurate performance model for complex systems is a challenging task, which is why empirical performance models are also used, predicting future performance by learning from previous performance data [77]. Although performance modelling is making great progress, it remains a standing challenge that is the focus of multiple papers each year [78, 79, 80, 81, 82].

Automatic, explorative optimisation is an active research field and novel techniques are constantly developed to explore various optimisation spaces, such as iterative compilation [83], adaptive compilers [84], or equality saturation [85, 86].

Advantages of Automatic Optimisation   Fully automated optimisation is invaluable if performance engineers are not available, or are allocated to other tasks. State-of-the-art automation of tedious, low-level implementation decisions such as register allocation or instruction selection delivers satisfying performance for most applications, and frees up development time for higher-level optimisations where a performance engineer can have higher impact. Valuable human time is saved, trading it for machine time. Both inside and outside of computer science, automation has the potential to increase task feasibility, productivity and quality [87].

Disadvantages of Automatic Optimisation   Fully automated optimisation may result in poor performance or require too much compilation time [19, 20, 21]. Crucially, control is usually sacrificed. The compilation flags offered by compilers like clang, that uses LLVM for C-like languages, only provide very limited control. Therefore, when compiler optimisations are unsatisfactory, performance engineers often fall back to manual optimisation in order to achieve their performance goals [22, 23, 24].

2.3.2 Controlled Optimisation

Instead of manually optimising low-level code, or entrusting optimisation to a black box automatic compiler, performance engineers may control semi-automatic compilers that effectively act as optimisation assistants. Techniques for controlled optimisation include transformation scripts, scheduling APIs, and rewriting strategies.

Transformation scripts   The polyhedral framework [88] is a powerful technique modelling loop nests as polyhedra, enabling sophisticated dependency analysis, and providing tools to explore the space of valid transformations. URUK *transformation scripts* were proposed to define and apply sequences of polyhedral transformations [89, 90]. The CHiLL framework extended this idea to more complex loop transformations [91]. An example of CHiLL script that applies loop transformations such as `tile(statement_id, loop_level, tile_size)` is given in Listing 2.5. Loo.py [92] is a more recent programming system embedded in Python and inspired by the polyhedral model, that provides a library of loop transformations.

```
permute([3,1,2])
tile(0,2,TJ)
tile(0,2,TI)
tile(0,5,TK)
datacopy(0,3,2,[1])
datacopy(0,4,3)
unroll(0,4,UI)
unroll(0,5,UJ)
```

Listing 2.5: CHiLL transformation script used to optimise matrix multiplication in [91].

Scheduling APIs Halide [9] has popularised the development of compilers that offer scheduling APIs. In this setting, a *schedule* defines how to optimise an *algorithm* that defines what to compute at a high level of abstraction, separating the two concerns. The Halide scheduling API focuses on exposing important trade-offs between parallelism, redundancy, and locality; by defining when and where functions are computed and stored. Listing 2.6 gives a simple example of Halide schedule. Following the success of Halide, the TVM [10], Fireiron [25] and Tiramisu [93] compilers apply the same principle to different domains.

```
output.split(i, gid, i, 16)
  .parallel(gid)
  .vectorize(i, 4);
```

Listing 2.6: Schedule for the point-wise addition Halide algorithm from Listing 2.3, loosely comparable to the OpenCL implementation from Listing 2.2.

Rewriting Strategies As discussed in the previous section, languages such as Stratego [70] enable the definition of custom *rewriting strategies* [94]. As this thesis follows a rewrite-based approach to tackle the extensibility challenge, rewriting strategies are of particular interest to tackle the controllability challenge. Chapter 4 will demonstrate how rewriting strategies, written in a language called ELEVATE [3], can be used to apply transformations that are beyond what is possible with Halide schedules on a case study.

Advantages of Controlled Optimisation Transformation scripts, schedules and rewriting strategies all enable performance engineers to take control over the optimisation process. They expose a structured optimisation space to the performance engineer, who explores it in order to achieve his performance goals. Compared to manual optimisation of low-level code, controlled optimisation saves development time, and given a correct compiler implementation, avoids introducing bugs during optimisation.

Disadvantages of Controlled Optimisation Controlled optimisation remains challenging, as argued for schedules in [26]. Further, systems built around these techniques typically do not allow smooth trade-offs between automation and control, as this trade-off is built-in. Therefore, controlled optimisation may be tedious and take too much performance engineering effort compared to automatic optimisation.

2.3.3 Controllable Automation of Optimisations

This thesis explores *controllable automation* of optimisations that promotes trade-offs between automation and control of optimisations. If a compiler supports both extremes, as well as

a spectrum in-between, performance engineers should be able to gradually take control of optimisations depending on their performance requirements and time budget.

Properties like domain-extensibility and controllable automation are hard to quantify on a scale, as they represent informal design principles. However, it is possible to identify whether different approaches provide controllable automation of specific choices or not, as in Table 2.1.

Approach	Parameter Choice (tile size)	Optimisation Choice (tiling)
Halide Schedule	✗	✗
TVM Schedule Template	✓	✗
This Thesis	✓	✓

Table 2.1: This thesis allows controllable automation of optimisation choices like tiling, as opposed to current Halide or TVM approaches.

Halide schedules neither provide controllable automation of parameters (e.g. tile size), nor of optimisations (e.g. tiling). Successful research goes into *autoscheduling*, the challenge of automatically generating schedules, both for Halide [27, 28, 29] and TVM [95, 96]. However to the best of our knowledge, autoscheduling techniques offer no controllable automation: performance engineers cannot constrain autoscheduling to apply a specific tiling optimisation, or not use specific tile sizes. This contrasts with the conclusion of the first Halide paper: *"the ultimate solution must allow a smooth trade off between inference when it is sufficient, and sparse programmer control when it is necessary"* [9].

TVM *schedule templates* provide controllable automation of parameters, but not of optimisations. For example, performance engineers may delegate the tuning of some numerical parameters, such as tile sizes, while manually specifying other parameters in a schedule template [10]. However, this approach does not allow delegating optimisations that significantly change code structure, such as deciding whether to use tiling, or how to fuse operators [95].

By contrast, the techniques presented in this thesis enable controllable automation of both parameters and optimisations (Chapter 5).

In the polyhedral community, there exists multiple trade-offs between automation and control. Transformation scripts can be used to structure automatic empirical optimisation [97, 98]. Powerful heuristics have also been developed to automate optimisation, without using transformation scripts, in PLuTo [99], PPCG [100] and PolyMage [18, 101]. Ways to increase interactivity have been explored, *"enabling users to examine, refine, replay and even design complex optimizations semi-automatically in partnership with the compiler"* [102]. Program optimisation tactics, loosely comparable to rewriting strategies, have been developed independently during this thesis for the polyhedral model [103, 104]. While the polyhedral framework is amenable to controllable automation, we focus on term rewriting techniques instead in this thesis. Term rewriting is more flexible than the polyhedral framework [105], which gets its strength from a restricted but carefully structured transformation space.

In the previous section, we already identified term rewriting as a compelling technique to tackle the extensibility challenge. Term rewriting is also a compelling technique to tackle the controllability challenge. Indeed, the definition of rewrite rules can be clearly separated from their application during rewriting. Given rewrite rules, we can decide to fully automate their application as done in Lift with random sampling, to precisely control their application as done in Stratego with rewriting strategies as we will leverage in Chapter 4, or trade-off between the two as we will explore in Chapter 5.

2.4 Conclusion

This chapter motivated the contributions of this thesis by presenting technical background and related work.

Section 2.1 discussed how, in a context of evolving hardware architectures, the programming models commonly used in industry, such as OpenCL, are too low-level. Higher-level programming models such as domain-specific languages or array programming languages have been successful at increasing productivity and portability. However, building and maintaining compilers that generate high-performance code for a variety of algorithmic domains and hardware architectures, while minimising engineering efforts, is challenging.

Section 2.2 discussed the compiler *extensibility challenge*. Domain-agnostic compilers such as LLVM are typically not able to automate important domain-specific optimisations. Extending domain-specific compilers such as Halide or TVM with new optimisations is hard. Domain-extensible compilers show potential to mitigate the extensibility challenge, but remain relatively immature compared to domain-specific compilers. Research is needed to further develop domain-extensible compiler technology.

Finally, Section 2.3 discussed the compiler *controllability challenge*. Automatic optimisation is invaluable if performance engineers are not available, but may result in poor performance or be too time consuming for some applications. Alternatively, performance engineers may take control of the optimisation process. However, controlling the optimisation process is challenging as there is typically no smooth transition between automation and control of optimisations. Research is needed to enable performance engineers to gradually take control instead of abruptly falling back to manual optimisation.

Imagination is the beginning of creation. You imagine what you desire, you will what you imagine and at last you create what you will.

— George Bernard Shaw

Chapter 3

Code Generation in a Domain-Extensible Compiler

The previous chapter identified term rewriting as a compelling technique to tackle the compiler extensibility and controllability challenges. LIFT [15] addresses the *extensibility challenge* by combining a high-level functional language with an extensible term rewriting system [68]. However, LIFT does not address the *controllability challenge*, and optimising a single convolution takes hours to reach peak performance on a GPU [16, 74]. SHINE is a novel compiler inspired by LIFT, with the additional goal to provide controllability by exploring trade-offs between automation and control of rewrite rule applications. A key concern for SHINE is how to generate imperative code from rewritten functional programs, with competitive runtime with established domain-specific compilers such as Halide and TVM. Code generation in SHINE does not aim to be extensible and controllable, but rather to predictably preserve the optimisation choices encoded during rewriting.

This chapter presents the design and implementation of three important code generation features. These features are crucial for SHINE to generate faster code than Halide on an image processing case study via controlled rewriting (Chapter 4), and similarly fast code as TVM on a linear algebra case study via semi-automated rewriting (Chapter 5).

Section 3.1 introduces the SHINE compiler and its RISE language, both the result of collaboration between multiple researchers [4, 106]. The rest of this chapter presents work that is solely my own. Section 3.2 contributes a synchronisation *barrier insertion* algorithm that does not need to be modified when extending RISE patterns, contrasting with the barrier elimination algorithm of LIFT [107]. While barrier insertion is implicit and automated, the next two features add new RISE patterns explicitly exposing implementation choices to rewriting, allowing design space exploration and external control. Section 3.3 demonstrates how a RISE pattern is added for explicit *kernel execution*, and how SHINE is modified to generate corresponding imperative code. Section 3.4 discusses how RISE patterns are added for explicit *storage folding*, and how SHINE is modified to generate corresponding imperative code. Section 3.5 concludes.
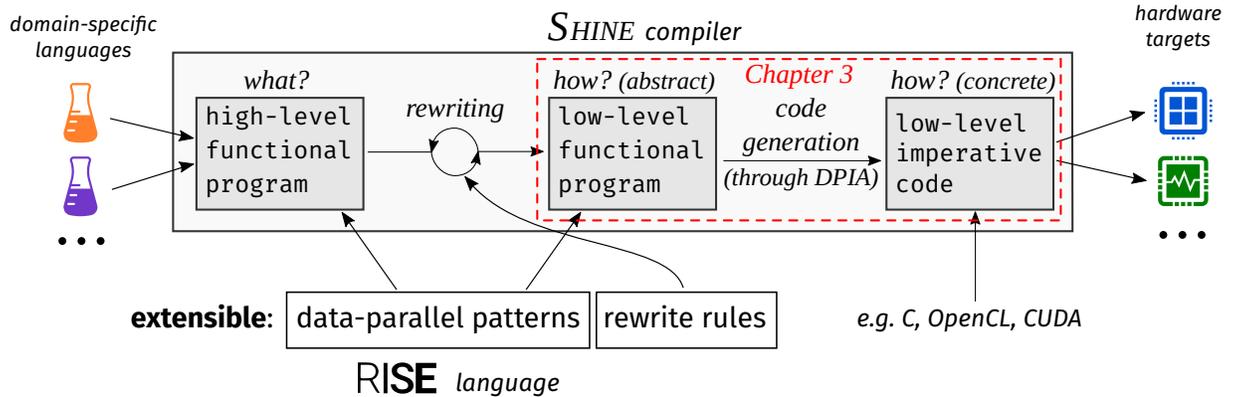
Figure 3.1: RISE & SHINE: A Domain-Extensible Compiler Design

3.1   The RISE Language & SHINE Compiler

SHINE is a domain-extensible compiler inspired by the LIFT compiler, its implementation is open-source.[1] Figure 3.1 gives an overview of the SHINE compiler, which is meant to be a bridge between domain-specific languages (left of Figure 3.1) and hardware targets (right of Figure 3.1). A program written in a domain-specific language is translated into a *high-level RISE program* (this step is left for future work and not the topic of this thesis). Then, the high-level program is rewritten into a *low-level RISE program* where implementation decisions are explicitly encoded (Section 3.1.1). Finally, imperative code is generated and can be executed on a given hardware target (Section 3.1.3). The focus of this chapter (Chapter 3) is to enhance imperative code generation from low-level RISE programs (red rectangle in Figure 3.1).

RISE is a functional data-parallel language inspired by the LIFT language. More precisely, RISE is a lambda calculus extended with higher-order functions called *patterns* (e.g. `map` or `reduce`) and a restricted form of dependent typing (Section 3.1.2). Functional languages are referentially transparent, meaning that an expression is equivalent to its value. There are no side effects, and semantics-preserving rewrite rules can be easily defined to encode program optimisations.

The rest of this section gives a complete overview of the compilation stack, to contextualise the following sections.

Relationship to LIFT   RISE & SHINE started as an effort to re-implement LIFT in a more principled way by following the typed lambda calculi formalism [108]. In SHINE, patterns are implemented as higher-order functions, and functions as expressions. Adding a pattern only requires providing a name and a type. This contrasts from the LIFT implementation, where adding a pattern also requires defining how to infer its type (Table 3.1). Both implementations then evolved separately as the original research group forked to explore two different directions. There are no plans to merge the implementations, although it would be possible.

---

[1]https://github.com/rise-lang/shine

language	functions are expressions	function type	type inference
LIFT	no	no	pattern-specific code
RISE	yes	yes	generic for all functions

Table 3.1: RISE follows the typed lambda calculus formalism more closely than LIFT.

3.1.1   Optimising Programs via Rewriting

The SHINE compiler takes as input a high-level RISE program describing *what* to compute, rather than *how* to compute. For example, a vector dot product is represented as the high-level RISE program dot:

```
def dot a b = zip a b ▷ map (λx. (fst x) × (snd x)) ▷ reduce + 0
```

Listing 3.1: dot program in RISE

The **zip** a b pattern combines two vectors a and b whose elements are multiplied pairwise using **map** before they are summed using **reduce** + 0. The triangle symbol (x ▷ y) indicates the chaining of operations via function application (y x) or composition (λz. y (x z)).

The high-level dot program does not encode how it is executed. We could parallelise the **map**, store the intermediate result, and perform a sequential **reduce**. Alternatively, we could avoid intermediate storage by fusing **map** and **reduce** into a single sequential reduction. Many more options are possible, and such choices are encoded explicitly by applying rewrite rules.

To achieve a fused version avoiding the intermediate results, a compiler writer or performance engineer writes the reduceMapFusion rewrite rule. Currently, the rule is written in an informal syntax, future work may design a formal language for RISE rewrite rules. This rule states that mapping a function f over an array before reducing the array is equivalent to reducing the array while applying f on the go. Note that the reduction must be performed sequentially because the reduction operator is not commutative anymore.

```
rule reduceMapFusion = map f ▷ reduce g init
                    ↦ reduceSeq (λacc x. g acc (f x)) init
```

Listing 3.2: reduceMapFusion rewrite rule.

Deciding which rewrite rule to apply where is challenging, and will be discussed in Chapters 4 and 5. For now, consider that reduceMapFusion is applied to **map** × ▷ **reduce** + 0 in the high-level dot product program, replacing it with the right hand side of the rewrite and producing the following low-level program:

```
def dotSeq a b = zip a b ▷ reduceSeq (λacc x. acc + (fst x) × (snd x)) 0
```

Listing 3.3: dotSeq program in RISE, equivalent to dot from Listing 3.1.

Generating the equivalent C or OpenCL code from this representation is conceptually straightforward, but has some technical challenges that have been explored in prior work [107,

106] and are discussed in Section 3.1.3. Some patterns generate loops (e.g. **reduceSeq**), others generate scalar expressions (e.g. +, ×, 0), and others affect indexing (e.g. **zip**, **fst**, **snd**). The C function dotSeqC is generated from dotSeq, implementing the dot product with a sequential reduction loop as expected:

```c
1  void dotSeqC(float* output, int n, float* a, float* b) {
2      float acc;
3      acc = 0.0f;
4      for (int i = 0; i < n; i++) {
5          acc = acc + (a[i] * b[i]);
6      }
7      output[0] = acc;
8  }
```

Listing 3.4: dotSeqC prorgam in C, generated from Listing 3.3

3.1.2 High-Level and Low-Level RISE Programs

RISE is a functional language with anonymous functions (written as λx. e where x is a variable name and e a RISE expression), familiar function application (written as e e), identifiers and literals. The language is embedded in Scala, allowing meta-programming and the definition of macros, i.e. Scala code that will generate RISE programs. RISE also defines a set of high-level patterns to describe computations as shown in Listing 3.5 together with their types. Formally, patterns are higher-order functions that can depend on natural numbers (**nat**) and data types (**data**). RISE has no support for general recursion or iteration, and instead relies on extensible and higher-level array patterns that explicitly expose data parallelism [109, 110]: **map** applies a function to each element of an array; **reduce** reduces all elements of an array to a single value given a binary reduction function. Multi-dimensional array reshaping is common in array languages [54], RISE has patterns such as **split**, **join**, or **transpose** for this purpose.

```
1   + | ×       : t → t → t
2   map         : (s → t) → n.s → n.t
3   reduce      : (t → t → t) → t → n.t → t
4   split       : (n: nat) → (n × m).t → m.n.t
5   join        : n.m.t → (n × m).t
6   transpose   : n.m.t → m.n.t
7   slide       : (sz sp: nat) → (sp × n + sz - sp).t → n.sz.t
8   zip         : n.s → n.t → n.(s × t)
9   fst         : (s × t) → s
10  snd         : (s × t) → t
```

Listing 3.5: Selection of RISE high-level patterns and their type

We write s $\rightarrow$ t for a function type with input of type s and output of type t, n.t for an array type with n elements of type t, s $\times$ t for a pair type with component types s and t. To avoid the challenge of storing functions (or closures) in heterogeneous hardware memory, the type system ensures that function types cannot be stored in memory, only data types (e.g. it is challenging for GPUs).

The SHINE compiler rewrites a high-level RISE program into a low-level RISE program that describes *how* the result is computed, encoding implementation decisions explicitly. RISE's low-level patterns (Listing 3.6) indicate specific implementation decisions.

```
1  mapSeq           : (s → t) → n.s → n.t
2  reduceSeq        : (s → t → s) → s → n.t → s
3  mapGlobal(dim)   : (s → t) → n.s → n.t
4  toMem            : (a: addr) → t → t
5  asVector         : (m: nat) → (n × m).t → n.<m>t
6  asScalar         : n.<m>t → (n × m).t
7  vectorFromScalar : t → <m>t
```

Listing 3.6: Selection of RISE low-level patterns and their type

For example, **mapSeq** and **reduceSeq** respectively implement **map** and **reduce** with sequential loops. Some low-level patterns are specific to the target programming model (such as OpenCL) or hardware architecture (such as SIMD vector support). For OpenCL, **mapGlobal** introduces parallelism by parallelising across the dimension dim of global threads. The **toMem** pattern is used to explicitly encode storing an expression in the given address space in memory (a: **addr**). Other patterns enable SIMD vectorisation (e.g. **asVector**). A vector type with m elements of type t is written as <m>t.

3.1.3   Code Generation through DPIA

To generate imperative code (e.g. C or OpenCL) from a low-level RISE program, a hybrid functional-imperative language called DPIA is used as an intermediate language. The DPIA language [106] is a variation of idealised ALGOL [111] that separates program *phrases* into three categories: functional *expressions*, imperative *commands*, and imperative *acceptors*. One can think of commands as C statements, and of acceptors as C pointers that can be manipulated and ultimately used for writing to a memory location. DPIA basically uses the same functional constructs and data types as RISE. Imperative constructs are added, such as assignment (p = q), sequential composition (;), memory allocation (**new**), and loops (**for**). The type system reflects the separation into expressions (**exp**[dt, rw]), acceptors (**acc**[dt]), and commands (**comm**). Functions and pairs can freely combine phrases of all categories.

A selection of DPIA patterns is shown in Listings 3.7 and 3.8. While RISE introduces patterns as higher-order functions with implicit type-level arguments more suitable for rewriting,

DPIA introduces patterns as fully applied values with explicit type-level arguments more suitable for translation and code generation.

```
zip(n: nat, s: data, t: data, w: access,
    a: exp[n.s, w], b: exp[n.t, w]): exp[n.(s x t), w]
fst(s: data, t: data, x: exp[(s x t), read]): exp[s, read]
snd(s: data, t: data, x: exp[(s x t), read]): exp[t, read]
mapSeq(n: nat, s: data, t: data, f: exp[s, read] → exp[t, write],
       x: exp[n.s, read]): exp[n.t, write]
reduceSeq(n: nat, a: addr, s: data, t: data,
          f: exp[s, read] → exp[t, read] → exp[s, write],
          init: exp[s, write], x: exp[n.t, read]): exp[s, read]
toMem(a: addr, t: data, x: exp[t, write]): exp[t, read]
idx(n: nat, t: data, i: exp[idx[n], read], x: exp[n.t, read]): exp[t, read]
// ^ written as x[i]
```

Listing 3.7: Selection of functional DPIA patterns

```
assign(t: data, p: acc[t], q: exp[t, read]): comm // written as p = q
seq(c1: comm, c2: comm): comm                     // written as c1 ; c2
new(a: addr, t: data, body: (exp[t, read] x acc[t]) → comm): comm
for(n: nat, body: exp[idx[n], read] → comm): comm
zipAcc1(n: nat, s: data, t: data, x: acc[n.(s x t)]): acc[n.s]
zipAcc2(n: nat, s: data, t: data, x: acc[n.(s x t)]): acc[n.t]
idxAcc(n: nat, t: data, i: exp[idx[n], read], a: acc[n.t]): acc[t]
```

Listing 3.8: Selection of imperative DPIA patterns

The type system of DPIA enforces important invariants and assumptions. Expression types include an **access** annotation, which can either be **read** or **write** and enforces that DPIA programs must explicit encode how to write in what memory. A **read** annotation signifies that the value can be read from memory while a **write** annotation indicates that the value must first be written to memory before being readable. To convert a **write** expression into a **read** expression, the **toMem** pattern can be used and specifies a choice of address space.

Loop patterns like **for** iterate over bounded indices of type **idx**[n], meaning that the index value belongs to the interval $[0; n[$. This is important to statically enforce safe array accesses and to remove the need for runtime checks.

From RISE to DPIA   Going from low-level RISE to functional DPIA requires two steps which we will not detail: (1) inferring access annotations to create DPIA types from RISE types, the work of Bastian Köpcke; (2) translating patterns to be fully applied values with explicit type-level arguments, a personal engineering contribution.

To illustrate the second step, λf x. **mapSeq**(n, s, t, f, x) would result from translating **mapSeq**, extracting n, s, t from the instantiated RISE type.

Translation to Imperative   Translating functional DPIA into imperative DPIA is performed by two mutually recursive translation functions (Listing 3.9).

```
𝒜(e: exp[dt, write], out: acc[dt]): comm
𝒞(e: exp[dt, read], k: exp[dt, read] → comm): comm
```

Listing 3.9: Type of the acceptor and continuation translations

The *acceptor translation* 𝒜 produces a command which writes a functional **write** expression to the memory represented by an acceptor (Listing 3.10). The *continuation translation* 𝒞 produces a command which reads from a functional **read** expression and calls a *continuation* function to continue the translation as required (Listing 3.11). To start the overall translation process, we need to know where to write the program result. For this, we generate an output acceptor according to the data type computed by the functional expression before invoking the 𝒜 translation: imperative = 𝒜(functional, output). The output will later correspond to a runtime parameter in the generated C function (as in Section 3.1.1), or OpenCL kernel.

```
𝒜(zip(n, s, t, write, a, b), out) =
  𝒜(a, zipAcc1(n, s, t, out)) ;
  𝒜(b, zipAcc2(n, s, t, out))

𝒜(mapSeq(n, s, t, f, x), out) = 𝒞(x, λxT.
  for(n λi. 𝒜(f(idx(n, s, i, xT), idxAcc(n, t, i, out)))))

𝒜(reduceSeq(n, a, s, t, f, init, x), out) =
  𝒞(reduceSeq(n, a, s, t, f, init, x))(λr. 𝒜(r)(out))
```

Listing 3.10: Selection of acceptor translations

```
𝒞(zip(n, s, t, read, a, b), k) =
  𝒞(a, λaT. 𝒞(b, λbT. k(zip(n, s, t, read, aT, bT))))

𝒞(fst(s, t, read, x), k) = 𝒞(x, λxT. k(fst(s, t, read, xT)))

𝒞(snd(s, t, read, x), k) = 𝒞(x, λxT. k(snd(s, t, read, xT)))

𝒞(reduceSeq(n, a, s, t, f, init, x), k) =
 𝒞(x, λxT. new(a, s, λ(accE, accA).
   𝒜(init, accA) ;
   for(n, λi. 𝒜(f(accE, idx(n, t, i, xT)), accA)) ;
   k(accE)))

𝒞(toMem(a, t, x), k) = new(a, t, λ(tmpE, tmpA). 𝒜(e, tmpA) ; k(tmpE))
```

Listing 3.11: Selection of continuation translations

For example, translating the dotSeq program (Listing 3.3) from RISE to functional DPIA and further to imperative DPIA leads to the following translation steps (with syntactic sugar):

```
𝒜(reduceSeq (λa x. a + (fst x) × (snd x)) ⊙ (zip a b), output)
=
𝒞(reduceSeq (λa x. a + (fst x) × (snd x)) ⊙ (zip a b))(λr. 𝒜(r)(output))
=
𝒞(zip a b, λz. new (exp[f32, read] x acc[f32]) in λ(accE, accA). (
  𝒜(0, accA);
  for n λi.
    𝒜(accE + (fst z[i]) × (snd z[i]), accA);
  output = accE))
=
new (exp[f32, read] x acc[f32]) in λ(accE, accA). (
  𝒜(0, accA);
  for n λi.
    𝒜(accE + (fst (zip a b)[i]) × (snd (zip a b)[i]), accA);
  output = accE)
=
new (exp[f32, read] x acc[f32]) in λ(accE, accA). (
  accA = 0.0f;
  for n λi.
    (accA = accE + ((fst (zip a b)[i]) * (snd (zip a b)[i])));
  output = accE)
```

Imperative Passes The use of DPIA in the SHINE compiler enables transformation passes to be applied at the imperative DPIA level, as we will show in Section 3.2. This is valuable because imperative DPIA has more convenient abstractions and richer semantics compared to C or OpenCL: for example, multi-dimensional arrays can be used and precise type information has not been erased yet.

Code Generation Finally, code such as C or OpenCL is generated from the imperative DPIA program. One important transformation which happens at this stage is the translation of indexing patterns (e.g. **fst**, **snd**, **zip**) into array indexing, and the flattening [112] (or linearisation [113]) of multi-dimensional arrays into one-dimensional arrays. The C code generated for the dotSeq program (Listing 3.3) was shown in Listing 3.4.

3.2 Implicit Barrier Insertion for SHINE

When multiple threads access the same memory concurrently and at least one thread is writing, synchronisation is required. Without synchronisation, operation ordering is non-deterministic, and the computation may produce incorrect results: this type of bug is called a *data race* [114].

This section contributes DPIA$_{\text{BI}}$, a synchronisation barrier insertion algorithm for SHINE. DPIA$_{\text{BI}}$ transforms imperative DPIA programs and does not need to be modified when extending functional RISE patterns, as opposed to the barrier elimination algorithm of LIFT [107, 115]. Following LIFT's design, barrier insertion is implicit and not controllable by rewriting, allowing RISE programs to ignore low-level synchronisation requirements.

3.2.1 Synchronisation in OpenCL

In OpenCL kernels, calling `barrier(flag)` synchronises work-items from the same work-group. The `flag` argument specifies which address spaces require a memory fence to ensure correct ordering of memory operations (`CLK_LOCAL_MEM_FENCE` for local memory and `CLK_GLOBAL_MEM_FENCE` for global memory). We must also ensure that all work-items execute the barrier, to avoid undefined behaviour and potential *deadlocks* [116] where some work-items are indefinitely waiting for others to encounter the barrier:

> "All the work-items of a work-group must execute the barrier before any are allowed to continue execution beyond the barrier. Note that the work-group barrier must be encountered by all work-items of a work-group executing the kernel or by none at all." (OpenCL 1.2 Specification)

OpenCL does not support synchronisation across work-groups inside a kernel, instead multiple kernels must be launched. We will discuss how to add support for launching multiple kernels in Section 3.3 and focus on work-item barriers in this section.

3.2.2 Barrier Insertion as an Imperative DPIA$_{\text{BI}}$ Transformation

Because functional RISE programs ignore low-level synchronisation requirements, code generation must take care of inserting work-item barriers. Barriers are relatively expensive [117, 118, 119], to maximise performance of the generated code, we seek to minimise the number of barriers executed at runtime while ensuring correct synchronisation.

Previous work on LIFT follows a pessimistic approach, first inserting barriers at the end of all parallel map patterns to avoid any data-race (e.g. after **mapGlobal**, **mapWorkGroup** and **mapLocal** patterns), then eliminating barriers as an optimisation [107, 115]. There are two problems with this approach. First, the pessimistic approach is not sufficient to guarantee correctness because it does not ensure that all work-items execute the barriers. Following the pessimistic approach without any optimisation generates programs with undefined behaviour, as expanded on in Section 3.2.5. Second, LIFT's barrier elimination is analysing how functional patterns such as **mapLocal** are composed, and their interaction with other patterns that might result in data sharing between work-items (**split**, **join**, etc). We argue that this requires

the definition of too many special cases (for example, special treatment is required when encountering several `mapGlobal` inside the arguments of a `zip`, or nested `mapLocal` patterns). Additionally, such an approach may require modifying the barrier elimination optimisation when new functional patterns are added, reducing the ease of compiler extension.

Therefore, we follow instead an optimistic approach, inserting barriers as required in imperative DPIA programs. Before generating OpenCL kernel code from imperative DPIA, the following imperative passes are applied in sequence:

1. Inject work item sizes. The OpenCL number of groups, local size and global size values are inlined if they are statically known.

2. Flag private array loops. Loops iterating over private arrays are flagged to be unrolled, which helps OpenCL perform register allocation.

3. Unroll loops. The unrolled loops have been flagged in the previous pass, or explicitly during rewriting.

4. Simplify nats. Tries to simplify certain arithmetic expressions (e.g. $x \times 0 = 0$), such simplifications are not only applied here but also throughout the compilation process.

5. **Insert memory barriers**. The topic of this section realised in the DPIA$_{\text{BI}}$ algorithm.

6. Hoist memory allocations. In OpenCL, global and local memory cannot be allocated while the kernel is running, it must be allocated upfront. This pass hoists such memory allocations as required (e.g. outside of loops).

7. Adapt kernel parameters. Kernel parameters in global or local memory which have a scalar type in DPIA are represented as arrays of size 1 for OpenCL.

3.2.3 Definition of DPIA$_{\text{BI}}$

DPIA$_{\text{BI}}$ is defined by analysing reads and writes to memory, making conservative barrier insertions to ensure synchronisation according to the observed data dependencies.

The DPIA$_{\text{BI}}$ analysis records memory reads and writes in two mutable dictionaries from memory identifier to address space, one representing reads from outer scope allocations, and one representing work-group parallel writes to outer scope allocations. This corresponds to the following record structure (a case class is used in Scala):

```
record D(reads: MutMap[Ident, Address], wg_writes: MutMap[Ident, Address])
```

DPIA$_{\text{BI}}$ is defined recursively in Listing 3.12. The DPIA$_{\text{BI}}$ function is the entry point and calls DPIA$_{\text{BI}}^{\text{fresh\_rec}}$, which creates empty data of type D before calling DPIA$_{\text{BI}}^{\text{rec}}$. Barriers are represented by the **barrier** DPIA command and may be inserted in-between sequences (inserted for **seq** in Line 30), or at the end of loop bodies (inserted for **for** or **parFor** in Line 41).

```
1  def DPIA_BI(p: DPIA_comm): DPIA_comm = DPIA_BI^fresh_rec(p, Map())_1
2
3  def DPIA_BI^fresh_rec(p: DPIA_comm,
4                 allocs: Map[Ident, Address]): (DPIA_comm, D) =
5    def d = D(MutMap(), MutMap())
6    (DPIA_BI^rec(p, allocs, d), d)
7
8  def DPIA_BI^rec(p: DPIA_comm, allocs: Map[Ident, Address], d: D): DPIA_comm =
9    match p
10   for(n, λx. body) =>
11     for(n, λx. DPIA_BI^loop_body(body, allocs, d, MutMap()))
12   parFor(level, ..., out, λx o. body) =>
13     parFor(level, ..., out, λx o. DPIA_BI^loop_body(body, allocs, d,
14       if level == local then collect writes by inspecting out
15       else MutMap()))
16   new(adr, dt, λx. body) =>
17     def allocs2 = if adr != private then allocs + (x -> adr) else allocs
18     new(adr, dt, λx. DPIA_BI^rec(body, allocs2, d))
19   assign(dt, lhs, rhs) =>
20     collect reads into d.reads by inspecting rhs
21     assign(dt, lhs, rhs)
22   seq(a, b) =>
23     def (a2, ad) = DPIA_BI^fresh_rec(a, allocs)
24     def (b2, bd) = DPIA_BI^fresh_rec(b, allocs)
25     def dependencies = (ad.reads.keys ∩ bd.wg_writes.keys)
26       ∪ (bd.reads.keys ∩ ad.wg_writes.keys)
27     extend d.reads with ad.reads and bd.reads
28     extend d.wg_writes with ad.wg_writes and bd.wg_writes
29     if dependencies non empty then
30       seq(a2, seq(make barrier according to dependencies, b2))
31     else
32       seq(a2, b2)
33   [...]
34
35 def DPIA_BI^loop_body(p: DPIA_comm, allocs: Map[Ident, Address], d: D,
36         outer_wg_writes: MutMap[Ident, Address]): DPIA_comm =
37   def p2 = DPIA_BI^rec(p, allocs, d)
38   def dependencies = d.wg_writes.keys ∩ d.reads.keys
39   if dependencies non empty then
40     clear d.reads and d.wg_writes
41     seq(p2, make barrier according to dependencies)
42   else
43     extend d.wg_writes with outer_wg_writes
44     p2
```

Listing 3.12: Pseudo-code for the DPIA$_{BI}$ barrier insertion algorithm

To analyse dependencies, reads and work-group parallel writes are collected in lines 20 and 14. The `parFor` pattern corresponds to parallel loops (e.g. `parFor(local, ...)` corresponds to `mapLocal`), and differently from sequential loops, explicitly mentions its global output (out) and its local output (o), making analysis easier. Information about memory allocations and their address space is also collected in line 17, since this information is critical to select the right barrier flags (`CLK_LOCAL_MEM_FENCE` and `CLK_GLOBAL_MEM_FENCE`).

3.2.4 Evaluating the Correctness and Efficiency of DPIA$_{BI}$

Experimental Setup The correctness and efficiency of barrier insertion is evaluated on 48 programs (38 unit tests and 10 benchmarks), by observing and comparing the code generated by LIFT and SHINE. The generated code is considered correct if it respects the OpenCL 1.2 specification. The generated code is considered more efficient if it executes less barriers at runtime, or uses less flags.

The 38 unit tests mainly come from the LIFT repository[2], with a few additions. The unit tests compose various patterns in various ways: parallel and sequential map patterns to introduce threads and loops, `toMem` to introduce temporary memory, and indexing patterns to introduce data sharing between threads.

The 10 benchmarks come from the experimental evaluation of LIFT presented in [107]. The benchmarks target GPU hardware and represent different domains including physics simulations, statistics, and linear algebra. Two benchmarks were ported but elided from the results, as they are not fairly comparable: "NBody NVIDIA" and "MM NVIDIA". SHINE allocates additional memory for these programs due to orthogonal issues.

Correctness and Efficiency Results Table 3.2 summarises the results, identifying 6 differences between the code generated by SHINE and the one generated by LIFT:

Impact	Difference	Unit tests	Benchmarks	Total
LIFT incorrect	Incorrect flags	13 (34%)	1 (10%)	14 (29%)
LIFT more efficient	Better analysis	13 (34%)	0 (0%)	13 (27%)
	Single iteration loops	6 (16%)	1 (10%)	7 (15%)
RISE more efficient	Better analysis	0 (0%)	2 (20%)	2 (4%)
	Barrier position	1 (2%)	0 (0%)	1 (2%)
Trade-off	Sync. vs alloc.	7 (18%)	0 (0%)	7 (15%)
Total		38 (100%)	10 (100%)	48 (100%)

Table 3.2: Number and percentage of unit tests and benchmarks that exhibit each difference identified in the code generated by SHINE and LIFT. SHINE generates correct barriers in all cases, which is not the case of LIFT.

---

[2]`https://github.com/lift-project/lift/blob/5e8a18df48ab791ae66016007e18826047a015f7/src/test/opencl/generator/TestBarrier.scala`

```
1  λin : n.m.f32. (
2  in ▷ mapWorkGroup(0) (
3    mapLocal(0) (λx. x) ▷
4    toMem local ▷
5    slide 3 1 ▷
6    mapLocal(0) sum
7  ))
```

```
1  for (int wg0 = get_group_id(0); wg0 < n; wg0 += get_num_groups(0)) {
2    for (int l0 = get_local_id(0); l0 < m; l0 += get_local_size(0)) {
3      [...] // read from global input; write to local memory
4    }
5    barrier(CLK_LOCAL_MEM_FENCE);
6    for (int l1 = get_local_id(0); l1 < m-2; l1 += get_local_size(0)) {
7      [...] // read from local memory; write to global output
8    }
9  - barrier(CLK_GLOBAL_MEM_FENCE);
10 + barrier(CLK_LOCAL_MEM_FENCE);
11 }
```

Figure 3.2: Example low-level Rise program and its generated OpenCL code. In red (-), barrier that would be generated by Lift. In green (+), barrier generated by Shine. The second barrier flag is incorrect with Lift, but correct with Shine.

1. *Incorrect flags (Lift incorrect).* While Shine generates correct barriers for all programs, Lift generates barriers with incorrect flags for 14 programs (29%). For the test shown in Figure 3.2, a barrier on global memory is generated instead of a barrier on local memory. In practice, the program still computes the expected result, however the OpenCL 1.2 specification is not respected and this could lead to undefined behaviour with a different OpenCL implementation. With DPIA$_{BI}$, Shine inserts barriers with correct flags: one in line 5 to protect the read-after-write dependency on local memory, and one in line 10 to protect the write-after-read dependency between outer loop iterations.

2. *Better analysis (Lift more efficient).* DPIA$_{BI}$ only tracks memory accesses for entire allocations. Lift generates fewer barriers for 13 programs (27%) by reasoning about data sharing in the functional program. This difference is not visible in the benchmarks that exclusively use local memory to share data between work-items. However, integrating a more precise dependency analysis into DPIA$_{BI}$, for example by tracking accessed intervals using symbolic natural numbers, would enable Shine to generate as many or fewer barriers than Lift in all observed cases.

3. *Single iteration loops (Lift more efficient).* Lift avoids generating barriers for loops whose body is only executed once in 7 programs (15%), while Shine generates superfluous barriers. Shine could avoid generating such barriers by eliminating single iteration loops before barrier insertion, detecting such loops during barrier insertion, or by differentiating between two scenarios which are currently conflated both in Lift and Rise: (1) the loop will be executed again; or (2) the loop will be exited.

4. *Better analysis (RISE more efficient).* RISE avoids generating barriers that LIFT generates, or uses more precise flags, in 2 programs (4%). This is because LIFT fails to reason about specific combinations of functional patterns, while SHINE handles these combinations naturally at the imperative level, without any special treatment.

5. *Barrier position (RISE more efficient).* LIFT only inserts barriers at the end of parallel map patterns. For 1 program (2%) this strategy is not flexible enough and leads to inefficient barriers compared to SHINE. Figure 3.3 shows an example where SHINE is able to generate barriers at the end of sequential loops, which results in a program executing $2 \times (m - 1) \times n$ times fewer barriers at runtime ($m - 1$ less per sequential loop execution).

6. *Synchronisation vs allocation (Trade-off).* For 7 (15%) programs, LIFT allocates more memory than RISE. As a result, LIFT generates less barriers because the memory is not re-used while RISE generates more barriers because the memory is re-used. It is unclear which trade-off is best without benchmarking a specific program on a specific hardware target. It may even be that a fine-grain combination of different options for different allocations is optimal. Future work may explore this trade-off.

```
1  λin : n.m.o.f32. (
2    in ▷ mapWorkGroup(0) (
3      mapSeq (mapLocal(0) (λx. x)) ▷
4      toMem local ▷
5      map (slide 3 1) ▷
6      mapSeq (mapLocal(0) sum)
7    ))
```

```
1  for (int wg0 = get_group_id(0); wg0 < n; wg0 += get_num_groups(0)) {
2    for (int i = 0; < m; i += 1) {
3      for (int l0 = get_local_id(0); l0 < o; l0 += get_local_size(0)) {
4        [...] // read from global input; write to local memory
5      }
6  -   barrier(CLK_LOCAL_MEM_FENCE);
7    }
8  + barrier(CLK_LOCAL_MEM_FENCE);
9    for (int i = 0; < m; i += 1) {
10     for (int l1 = get_local_id(0); l1 < o-2; l1 += get_local_size(0)) {
11       [...] // read from local memory; write to global output
12     }
13 -   barrier(CLK_GLOBAL_MEM_FENCE);
14   }
15 + barrier(CLK_LOCAL_MEM_FENCE);
16 }
```

Figure 3.3: Example low-level RISE program and its generated OpenCL code. In red (-), barrier that would be generated by LIFT. In green (+), barrier generated by SHINE. The second barrier flag is only correct with SHINE. The code generated by SHINE is also more efficient: $2 \times (m - 1) \times n$ times less barriers will be executed at runtime.

3.2.5 Limitations of DPIA$_{\text{BI}}$

In addition to the "*Better analysis*" and "*Single iteration loops*" issues (Items 2 and 3 above), there are four other limitations worth mentioning.

Barrier Reachability In some cases, the generated barriers might not be reached by all work-items, which would result in undefined behaviour. By optimising barrier placement on the 48 programs from Section 3.2.4, this issue is avoided in 11 unit tests and 3 benchmarks compared to following the naive pessimistic approach of LIFT with barrier elimination optimisations turned off. However, even when optimising barrier placement, the issue remains in 2 unit tests. An example is given in Figure 3.4, where part of the work-items might not enter the loop in line 3. Previous work on LIFT suffered from similar limitations and implemented a mix of compilation time and runtime checks to report the issue. For SHINE, an additional imperative DPIA pass could be implemented to fix the code as illustrated in Figure 3.4.

```
1  λin : n.m.o.p.f32. (
2    in ▷ mapWorkGroup(1) (mapWorkGroup(0) (
3      mapLocal(1) (
4        mapLocal(0) (λx. x) ▷
5        toMem(local) ▷
6        slide 3 1 ▷
7        mapLocal(0) sum
8      ))))
```

```
1  for (int wg0 = get_group_id(1); wg0 < n; wg0 += get_num_groups(1)) {
2    for (int wg1 = get_group_id(0); wg1 < m; wg1 += get_num_groups(0)) {
3  -    for (int l0 = get_local_id(1); l0 < o; l0 += get_local_size(1)) {
4  +    for (int l0 = get_local_id(1); l0 < ctt(o); l0 += get_local_size(1)) {
5  +      if (l0 < o) {
6        for (int l1 = get_local_id(0); l1 < p; l1 += get_local_size(0)) {
7          [...] // read from global input; write to local memory
8        }
9  +      }
10       barrier(CLK_LOCAL_MEM_FENCE);
11  +      if (l0 < o) {
12       for (int l2 = get_local_id(0); l2 < p-2; l2 += get_local_size(0)) {
13         [...] // read from local memory; write to global output
14       }
15  +      }
16       barrier(CLK_LOCAL_MEM_FENCE);
17     }
18   }
19  }
```

Figure 3.4: Example low-level RISE program and its generated OpenCL code where barriers may not be encountered by all work-items. In red (-), buggy code that would be generated by both LIFT and SHINE. In green (+), a potential fix where the `ctt` function rounds up a number to a multiple of the involved work-items.

Extensibility of Indexing Patterns To simplify DPIA$_{\text{BI}}$ further and separate concerns, it would be valuable to eliminate indexing patterns (e.g. **split**, **join**) beforehand. This would en-

able extending indexing patterns without having to adjust the "collect reads" operation, which would only need to deal with `idx` patterns (line 20 of Listing 3.12).

Extensibility of Imperative Patterns Adding a new imperative DPIA pattern requires adding a new case in Listing 3.12, reducing the ease of extensibility. However, we argue that this improves on previous LIFT work where barrier elimination happened at the functional level. In SHINE, it is possible to add new functional patterns without having to adapt DPIA$_{BI}$, as long as no new imperative patterns are required.

Formal Verification No proof of correctness is provided. Future work could provide an algorithm that provably prevents any data race or undefined behaviour for any well-formed low-level RISE program.

3.2.6 Related Work

Barriers for Collective Patterns When using data-parallel collective patterns like `map`, a common approach is to insert synchronisation barriers after collective operations. This is the approach taken by the array-based languages LIFT [15], Futhark [56], and SaC [120].

Barriers for Imperative Programs When transforming imperative code, using dependency analyses to reason about synchronisation barriers is common [121]. Both barrier insertion [117, 122, 123, 124, 125] and barrier elimination [126, 127, 128, 129] have been studied, often using advanced dependency analyses, such as polyhedral analyses. The Tiramisu compiler explicitly exposes synchronisation decisions to the performance engineer, checking validity using the polyhedral model [130].

Barriers in Domain-Specific Compilers The problem of inserting correct and efficient barriers is barely discussed by the Halide and TVM papers. For example, the TVM paper [10] simply mentions that "memory synchronization barriers must be properly inserted to guarantee that shared loaded data is visible to consumers". Following papers show that, for some use cases, the TVM barrier insertion pass is unsatisfactory, leading to poor performance and forcing users to modify the compiler [131].

The novelty of DPIA$_{BI}$ is to insert barriers using imperative dependency analyses in a domain-extensible compiler for a functional language based on data-parallel collective patterns. We showed how, in the SHINE domain-extensible compiler, the design of DPIA$_{BI}$ facilitates extending functional RISE patterns and can even lead to generating more efficient barriers (Figure 3.3).

3.2.7   Summary

This section contributes DPIA$_{BI}$, a synchronisation barrier insertion algorithm for SHINE that transforms imperative DPIA programs. Crucially, DPIA$_{BI}$ does not need to be modified when extending RISE patterns, as opposed to the barrier elimination algorithm of LIFT [107, 115]. Following LIFT's design, barrier insertion is implicit and not controllable by rewriting, allowing RISE programs to ignore low-level synchronisation requirements.

The correctness and efficiency of DPIA$_{BI}$ is evaluated on 38 unit tests and 10 benchmarks, mostly taken from prior LIFT work. We identify 6 differences in the code generated by SHINE and LIFT (Table 3.2), and observe that DPIA$_{BI}$ fixes LIFT bugs in 13 unit tests and 1 benchmark. There is only 1 benchmark where SHINE inserts a barrier that LIFT eliminates, and we provide a clear pathway to improve DPIA$_{BI}$ to generate more efficient barriers than LIFT on all 48 unit tests and benchmarks.

3.3   Explicit Kernel Execution for SHINE

Any high-level language that allows offloading computations using the OpenCL or CUDA programming models must deal with executing kernels from the host. SHINE allows RISE programs to be compiled for CPUs by generating C code, or to various hardware by generating OpenCL kernel code. However, so far RISE programs cannot be compiled for a host CPU while offloading computations to various hardware devices. Even if no computations are performed on the host, host-side calls to the OpenCL C API are tightly coupled to the use of OpenCL kernels. The host is responsible for dynamically allocating global/local memory, launching kernels, and transferring data between host and device. Launching multiple kernels is also the only way to synchronise across work-groups, if strictly following the OpenCL 1.2 specification (Section 3.2.1).

This section enables explicit kernel execution in RISE by adding the `oclRun` pattern. Crucially, this design makes kernel execution controllable by rewriting in the domain-extensible SHINE compiler, enabling to explore different kernel decompositions of a high-level RISE program via rewriting. The `oclRun` pattern allows a single low-level RISE program to express both host-side and device-side computations. SHINE is modified to generate imperative code for multiple OpenCL kernels, as well as the necessary C host code to launch them. To simplify host code generation, we implement a thin runtime called LRA that abstracts over some of the OpenCL runtime details. The focus is to support executing kernels on a single OpenCL device, support for multiple devices is left for future work.

3.3.1 Adding a RISE Pattern for Kernel Execution

To make device execution explicit, while keeping the interface as simple as possible, we introduce an **oclRun** pattern. This pattern is an identity function which takes 3D local and global thread counts as additional input, it has the following type:

---

oclRun : (ls0, ls1, ls2, gs0, gs1, gs2 : **nat**) → t → t

The input expression of **oclRun** will be computed on the device by generating and calling a corresponding kernel. We also provide syntactic sugar to enable providing 1D, 2D or 3D local and global thread counts, where the thread counts of unspecified dimensions are set to 1:

---

oclRun(ls: LocalSize, gs: GlobalSize) = **oclRun** ls.x ls.y ls.z gs.x gs.y gs.z

---

This new pattern can be used to write RISE programs such as the one in Listing 3.13, where **store** is similar to **toMem** without the OpenCL address space parameter, and with a function parameter to provide name binding. First, the value 3 is added to each element of in in parallel on the device (line 4), storing the result into x1. Second, the value 1 is subtracted to each element of x1 sequentially on the host (line 5), storing the result into x2. Finally, the value 1 is added to each element of x2 in parallel on the device (line 6), producing the final output.

```
1  store : (s → t) → s → t
2
3  Λn : nat. λin : n.i32. (
4    oclRun(LocalSize(2), GlobalSize(n/2)) (mapGlobal (add 3) in) ▷ store (λx1.
5    mapSeq (λy. y-1) x1 ▷ store (λx2.
6    oclRun(LocalSize(4), GlobalSize(n/4)) (mapGlobal (add 1) x2)))
7  )
```

Listing 3.13: Example low-level RISE program mixing host computation (middle **mapSeq** call) and device computation (both **oclRun** calls).

3.3.2 Creating a Lightweight Runtime Abstraction (LRA)

Low-level OpenCL host code needs to explicitly manage devices, kernels and memory in great detail. To simplify code generation, a lightweight runtime abstraction called LRA is designed, delegating some implementation decisions to the runtime implementation. This runtime is meant to be simplistic and to enable quick prototyping of our ideas, future work may implement or re-use more sophisticated runtime libraries, if more advanced features are required.

Buffer Abstraction   The most important abstraction provided by LRA is the Buffer abstraction which offers a unified and consistent view on memory that may reside on the host or on

the device, as required during program execution (Listing 3.14). This is not a new idea, as libraries such as SPOC [132], SkePU [133] and SkelCL [134] provide a similar abstraction.

A `Buffer` is created with `createBuffer` and deleted with `destroyBuffer`. Creating a buffer requires specifying its size in bytes and the type of accesses that it should support. `AccessFlags` express a combination of reads/writes from the host/device. To access a buffer, explicit synchronisation functions must be called. `hostBufferSync` allows accessing a buffer from the host, returning a `void*` pointer that may be used in regular C code. `deviceBufferSync` allows accessing a buffer from the device, returning a `DeviceBuffer` structure that may be used as a kernel parameter. The runtime user (e.g. the SHINE compiler) is responsible for avoiding data races while using this interface.[3]

```
1  typedef enum {
2    HOST_READ = 1 << 0,
3    HOST_WRITE = 1 << 1,
4    DEVICE_READ = 1 << 2,
5    DEVICE_WRITE = 1 << 3,
6  } AccessFlags;
7
8  Buffer createBuffer(Context ctx, size_t byte_size, AccessFlags access);
9  void destroyBuffer(Context ctx, Buffer b);
10 void* hostBufferSync(Context ctx, Buffer b,
11                      size_t byte_size, AccessFlags access);
12 DeviceBuffer deviceBufferSync(Context ctx, Buffer b,
13                                size_t byte_size, AccessFlags access);
```

Listing 3.14: Interface for the `Buffer` abstraction.

We provide two different `Buffer` implementations: one with memory copies, and one without. With memory copies, the buffer is allocated both in host memory and in device memory. Memory copies are issued between both allocations as required to maintain consistency. Without memory copies, the buffer is allocated in memory that is accessible by both host and device. Memory synchronizations are issued as required to maintain consistency. In both cases, memory copies and synchronisations are performed lazily by keeping track of dependencies between accesses at runtime in a very simple and coarse-grain manner. Each implementation requires no more than 80 lines of C code, illustrating how lightweight the abstractions are.

Context Abstraction   As already visible in Listing 3.14, LRA uses a `Context` abstraction (Listing 3.15). The context is used to manage global state required for OpenCL kernel execution. A `Context` may be created using `createDefaultContext` or `createContext`, and deleted using `destroyContext`. `createContext` enables specifying the OpenCL platform and OpenCL device desired for execution, if multiple ones are available.

---

[3]reminder: multiple threads should not access the same memory concurrently if at least one thread is writing

```
1  Context createDefaultContext();
2  Context createContext(const char* platform_subname,
3                        const char* device_type_str);
4  void destroyContext(Context ctx);
```

Listing 3.15: Interface for the `Context` abstraction.

Kernel Abstraction   Finally, LRA provides a `Kernel` abstraction (Listing 3.16). A `Kernel` is loaded from OpenCL kernel source code (`loadKernelFrom...`) encoded in a string, or in a file. A `Kernel` is launched using `launchKernel`, which requires specifying global and local thread counts as well as providing kernel arguments (e.g. scalars, device buffers). It is deleted using `destroyKernel`.

```
1  Kernel loadKernelFromSource(Context ctx, const char* name,
2                             const char* source, size_t length);
3  Kernel loadKernelFromFile(Context ctx, const char* name, const char* path);
4  void launchKernel(Context ctx, Kernel k,
5                    const size_t* global_size, const size_t* local_size,
6                    size_t arg_count, const KernelArg* args);
7  void destroyKernel(Context ctx, Kernel k);
```

Listing 3.16: Interface for the `Kernel` abstraction.

In addition to the two possible `Buffer` implementations, we provide a single implementation for `Context` and `Kernel`, which consists of about 300 lines of boilerplate C code.

Code Generation Objective   Given LRA, the code we want to generate from Listing 3.13 is depicted in Listing 3.17. The kernel source codes are provided as string constants (e.g. `k0_source`), each one corresponding to an **`oclRun`** call. A `add3_t` structure holds the execution environment for the computation (i.e. the two `Kernel`s), a `add3_init` function initializes this environment and a `add3_destroy` function deletes it. Finally, a `add3_run` function runs the actual computation: allocating `Buffer`s, performing synchronisations, launching kernels and performing host-side computations.

Crucially, if the RISE program is rewritten differently, the interface of the generated code does not change (i.e. the signature of `add3_init`, `add3_destroy` and `add3_run`). The `Buffer` abstraction is particularly important to enable this: inputs and outputs may freely reside on either host or device without having to generate different code, as the runtime will take care of both cases seamlessly.

```
1  const char k0_source[] =
2  "__kernel __attribute__ ((reqd_work_group_size(2, 1, 1)))"
3  "void k0(global int* restrict output, int n, const global int* restrict in){"
4  "  for (int gl_id = get_global_id(0); gl_id < n; gl_id = gl_id + (n / 2)) {"
5  "    output[gl_id] = ((int)3) + in[gl_id];"
6  "  }"
7  "}";
8
9  const char k1_source[] =
10 "__kernel __attribute__ ((reqd_work_group_size(4, 1, 1)))"
11 "void k1(global int* restrict output, int n, const global int* restrict in){"
12 "  for (int gl_id = get_global_id(0); gl_id < n; gl_id = gl_id + (n / 2)) {"
13 "    output[gl_id] = ((int)1) + in[gl_id];"
14 "  }"
15 "}";
16
17 typedef struct add3_t {
18     Kernel k0;
19     Kernel k1;
20 } add3_t;
21
22 void add3_init(Context ctx, add3_t* self){
23     (*self).k0 = loadKernelFromSource(ctx, "k0", k0_source, sizeof(k0_source) - 1);
24     (*self).k1 = loadKernelFromSource(ctx, "k1", k1_source, sizeof(k1_source) - 1);
25 }
26
27 void add3_destroy(Context ctx, add3_t* self){
28     destroyKernel(ctx, (*self).k0);
29     destroyKernel(ctx, (*self).k1);
30 }
31
32 void add3_run(Context ctx, add3_t* self, Buffer moutput, int n, Buffer min){
33   Buffer mx1 = createBuffer(ctx, ..., HOST_READ | DEVICE_WRITE);
34   {
35     DeviceBuffer b0 = deviceBufferSync(ctx, mx1, ..., DEVICE_WRITE);
36     DeviceBuffer b2 = deviceBufferSync(ctx, min, ..., DEVICE_READ);
37     launchKernel(ctx, (*self).k0, {n / 2, 1, 1}, {2, 1, 1}, 3, {b0, n, b2});
38   }
39   Buffer mx2 = createBuffer(ctx, ..., HOST_WRITE | DEVICE_READ);
40   {
41     int32_t* x2 = (int32_t*)hostBufferSync(ctx, mx2, ..., HOST_WRITE);
42     int32_t* x1 = (int32_t*)hostBufferSync(ctx, mx1, ..., HOST_READ);
43     for (int i = 0; i < n; i = 1 + i) {
44         x2[i] = x1[i] - ((int32_t)1);
45     }
46   }
47   {
48     DeviceBuffer b0 = deviceBufferSync(ctx, moutput, ..., DEVICE_WRITE);
49     DeviceBuffer b2 = deviceBufferSync(ctx, mx2, ..., DEVICE_READ);
50     launchKernel(ctx, (*self).k1, {n / 2, 1, 1}, {4, 1, 1}, 3, {b0, n, b2});
51   }
52   destroyBuffer(ctx, mx2);
53   destroyBuffer(ctx, mx1);
54 }
```

Listing 3.17: Code using LRA that should be generated from Listing 3.13

3.3.3 Compiling Kernel Executing Programs in SHINE

To generate code targeting LRA from kernel executing RISE programs, we take inspiration from partial compilers [135] to modify SHINE. A *partial compiler* delegates intermediate compilation tasks to child (partial) compilers before combining their generated intermediate code into the final code. We implement the partial C+OpenCL SHINE compiler that delegates C host code generation and OpenCL kernel code generation to dedicated child compilers (Figure 3.5).
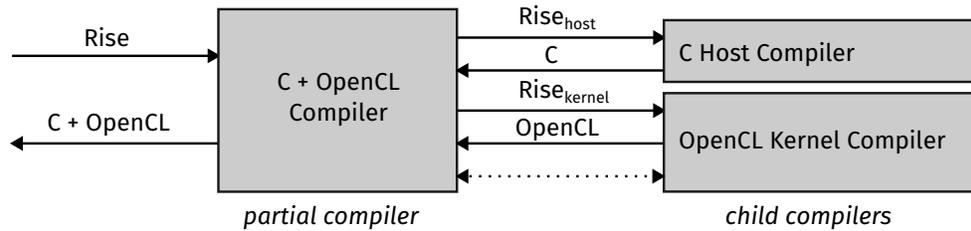

Figure 3.5: The partial C+OpenCL SHINE compiler delegates tasks to child compilers.

Partial C+OpenCL Compiler Programs with explicit kernel executions such as the one from Listing 3.13 can be partially compiled by decomposing them into two simpler compilation tasks: host code generation and kernel code generation. In the initial program, a kernel definition with name ki is generated for the i-th **oclRun** pattern through *lambda lifting* [136]: all free variables in the input expression of **oclRun** are eliminated by introducing explicit parameters (Listings 3.18 and 3.19). Each ki can be compiled separately, while the associated **oclRun** pattern is replaced with a **kernelCall** pattern that treats ki as a regular callable function: producing the host program which can also be compiled separately (Listing 3.20).

Once separately compiled by child compilers, kernel code and host code are put together in a single file as done in Listing 3.17.

```
1  Λn : nat. λin : n.i32. mapGlobal (add 3) in
```

Listing 3.18: RISE k0 kernel program for Listing 3.13

```
1  Λn : nat. λx2 : n.i32. mapGlobal (add 1) x2
```

Listing 3.19: RISE k1 kernel program for Listing 3.13

```
1  Λn : nat. λin : n.i32. (
2    kernelCall("k0", LocalSize(2), GlobalSize(n/2), n, in) ▷ store (
3    mapSeq (λy. y-1) x1 ▷ store (λx2.
4    kernelCall("k1", LocalSize(4), GlobalSize(n/4), n, x2)))
5  )
```

Listing 3.20: RISE host program for Listing 3.13

OpenCL Kernel Compiler   Generating kernel code does not require anything new: the existing SHINE OpenCL kernel compiler is simply re-used.

C Host Compiler   To generate host code, the existing SHINE C compiler requires extensions to target LRA and support compiling the **kernelCall** patterns (which become **kernelCallCmd** imperative DPIA commands after a straightforward translation to imperative).

First, we add an imperative DPIA compilation pass that, given a host program which uses plain arrays, introduces LRA `Buffer`s as needed, using the **newBuffer** pattern:

```
newBuffer(af: AccessFlags, dt: data,
          k: (exp[buffer[dt], read] × acc[buffer[dt]]) → comm): comm
```

Sub-programs that only perform host-side computation are kept as-is, they can still use plain arrays and the pre-existing C compilation, we call them *host execution sections* and introduce a corresponding **hostExecution** imperative DPIA pattern:

```
hostExecution(afs: Map[Ident, AccessFlags], body: comm): comm
```

To identify host execution sections, the algorithm used is fairly similar to the barrier insertion algorithm from Listing 3.12. Host and device side reads/writes are tracked to insert **hostExecution** patterns as required.

Finally, C code generation from imperative DPIA is extended to support the three new patterns. For each **kernelCallCmd**, deviceBufferSync and launchKernel LRA calls are generated in a C block for scoping (`{}`). For each **newBuffer**, createBuffer and destroyBuffer calls are generated in a block. For each **hostExecution**, hostBufferSync calls are generated in a block also containing the code generated for the host execution body. As desired, the code generated for the RISE program in Listing 3.13 corresponds to the C+OpenCL program in Listing 3.17, modulo small syntactic details.

3.3.4   Evaluating the Boilerplate Avoided through Kernel Execution

Experimental Setup   The impact of using the **oclRun** RISE pattern for kernel execution is evaluated by measuring how many lines of host code are automatically generated to launch the kernels, rather than written by hand. As a case study, we consider optimising the Harris corner (and edge) detector [30], a well established image processing pipeline.

Figure 3.6 depicts the Harris algorithm for a grayscale input image. Given the input on the left, point-to-point operators (multiplications $\times$, coarsity) and 3×3 convolutions (sobel operators S_x and S_y, sums $+$) are combined to detect corners and edges highlighted in the output on the right. As a composition of point wise and stencil operators, the Harris detector is more complex than its individual parts, and exposes more optimisation opportunities.

kernel grouping	variants per group	kernel variants
$[S_x], [S_y], [\times], [+], [coarsity]$	5	25
$[S_x, S_y, \times], [+, coarsity]$	5	10
$[S_x, S_y], [\times, +, coarsity]$	5	10
$[S_x, S_y, \times, +, coarsity]$	5	5
total	*total*	50

Table 3.3: Four different kernel groupings (a group is represented as []), and the number of differently optimised kernel variants that are considered (e.g. vectorised, tiled).

`oclRun`?	handwritten lines	generated lines
✗	>1000	0
✓	0	>1200

Table 3.4: Without `oclRun`, 1K lines of host code are handwritten to launch each kernel for the Harris corner detection design space exploration case study.

We consider well-known optimisations presented in [23]. Table 3.3 shows how many different OpenCL kernels need to be implemented to explore a space of important optimisations. In particular, different decisions for an optimisation called operator fusion lead to different kernel groupings. 4 different ways to fuse operators are considered, each variant offering a different trade-off between memory accesses and arithmetic complexity. 5 different ways to optimise each resulting kernel are also considered (e.g. tiling, vectorisation).

In total, benchmarking this relatively small optimisation space requires implementing 50 different kernels, each associated with slightly different host code to launch them.

Avoided Boilerplate Results   Table 3.4 shows how many lines of code are handwritten and generated with or without the kernel execution feature (i.e. the `oclRun` pattern). Without `oclRun`, 1K lines of host code are handwritten to launch the generated kernels. This corresponds to about 20 lines per kernel launch, and does not even count the code required to compose the kernels into the entire Harris processing pipeline. With `oclRun`, 1.2K lines of host code are automatically generated. Therefore, the kernel execution feature enables exploring the same optimisation space without having to write 1K lines of boilerplate code.
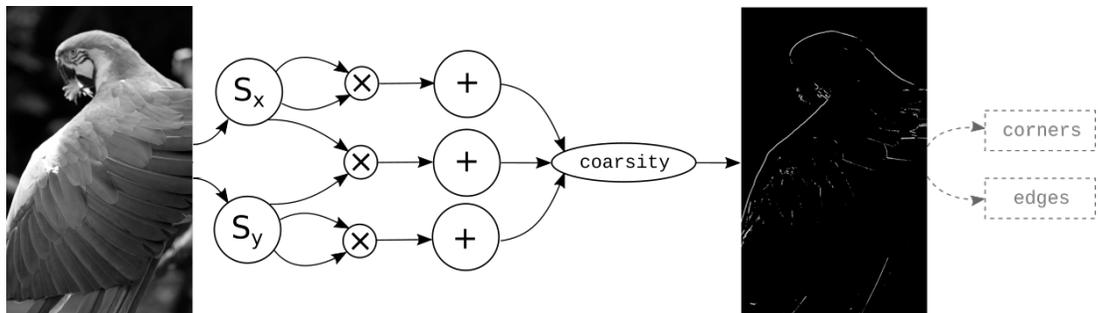

Figure 3.6: Harris corner detection computation flow, example image taken from Halide.

3.3.5   Related work

Kernel Execution in Domain-Specific Compilers   Halide and PolyMage support kernel executions, including exploring different kernel groupings [137, 101], but are domain-specific compilers for image processing.

Kernel Execution in General-Purpose Languages   Many general-purpose languages support executing kernels to offload computations on GPUs. OCaml has the SPOC library [132], Haskell has Accelerate [53], C/C++ has SYCL, OpenMP and OpenACC.

Typically, the host code is directly written in such languages, leveraging runtime libraries without necessarily generating C host code. The RISE language is not general-purpose, which is why we generate C host code. Additionally, RISE and the **oclRun** pattern are designed for rewrite-based design space exploration, which is not a design goal for the acceleration libraries mentioned above.

Implicit Kernel Execution in High-Level Languages   Many high-level languages support executing kernels to accelerate computations. The common strategy is to make kernel execution implicit and automated by heuristics, as in MapCG [138], NOVA [55] SaC [139, 140], X10 [141] and Lime [142]. MATLAB is an exception, as it allows the manual insertion of compiler directives in imperative code, similar to OpenACC [143, 144]. Both approaches differ from our explicit and functional **oclRun** pattern, designed for rewrite-based design-space exploration.

LIFT   Host code generation support has been independently added to LIFT in [145]. However, the approach taken introduces two challenges for rewriting without discussing them, as the paper focuses on manually written low-level programs. First, the introduced equivalent to our **oclRun** pattern (called OclKernel) does not take a single input expression, but rather a function as well as n explicit arguments. It is unclear how a rewrite system would introduce such a function and its arguments. By contrast, our design abstracts over the kernel function, inferring it via lambda lifting, and allowing a simple rewrite rule definition:

```
rule introOclRun(ls: LocalSize, gs: GlobalSize) =
  (x : dt) ↦ oclRun(ls, gs) x
  where dt : data
```

Second, explicit data transfers (as well as in-place computations) are introduced. By contrast, our design keeps data transfers implicit at the functional level for simplicity, and rewriting is not burdened with introducing efficient and correct transfers.

3.3.6   Summary

In this section, we demonstrate how the `oclRun` pattern is added to RISE to enable explicit kernel execution. To the best of our knowledge, SHINE is the first domain-extensible compiler that makes kernel execution controllable by rewriting high-level functional programs. Different kernel decompositions of a high-level RISE program can be explored via rewriting, producing a single low-level RISE program representing both host-side and device-side computations. SHINE is modified to generate imperative code for multiple OpenCL kernels, as well as the necessary C host code to launch them. Using `oclRun` in a relatively simple Harris corner detection case study enables replacing 1K lines of previously handwritten host code with 1.2K lines of automatically generated code (Table 3.4). The kernel execution feature saves performance engineers from writing boilerplate code and provides a stable interface for the generated code, which is not the case if SHINE generates a single OpenCL kernel without host code.

3.4   Explicit Storage Folding for SHINE

Stencil computations are used in a wide range of domains. A stencil computation processes each element of an N-dimensional input array by accessing a fixed neighborhood pattern (the stencil) to produce an output element. When optimising stencils, it is important to leverage spatio-temporal locality [23]. For example, to compute the next output of a 1D stencil during iteration i of sequential execution, only the last m stencil inputs are required. Instead of naively storing all stencil inputs in a temporary storage T, storage folding optimisations only store the last m inputs, improving memory usage. We consider two storage folding optimisations:

- *Circular buffering* stores the last m temporary results in memory, using modulo indexing: $T[j]$ is stored in $M[j \mod m]$ with $j \in [i; i + m - 1]$.

- *Register rotation* stores the last m temporary results in registers, rotating them between computation iterations: $T[i], \ldots, T[i + m - 1]$ is stored in registers $r_0, \ldots, r_{m-1}$.

Circular buffering is supported by most image processing specific compilers, including Halide. However, register rotation is not supported by Halide schedules.[4]

This section enables explicit storage folding in RISE by adding the `circularBuffer` and `rotateValues` patterns, along with 2 additional ones. As with kernel execution in the last section, introducing these patterns makes storage folding controllable and explorable by rewriting in the domain-extensible SHINE compiler. Chapter 4 will introduce storage folding patterns during rewriting to generate faster code than Halide on a case study.

---

[4]`https://github.com/halide/Halide/issues/2905`

3.4.1   Adding Rɪsᴇ Patterns for Storage Folding

To explicitly encode storage folding decisions in functional programs, we introduce two new functional patterns, called **circularBuffer** and **rotateValues**, and with following types:

```
circularBuffer : (a: addr) → (alloc m: nat) →
                 (s → t) → (n + m - 1).s → n.m.t
rotateValues   : (a: addr) → (m: nat) →
                 (t → t) → (n + m - 1).t → n.m.t
```

Both patterns have similar types, as they derive from **slide** m 1 : (n + m - 1).t → n.m. t, a sliding window pattern that was introduced to support stencils in Lɪꜰᴛ [16]. Both patterns take an address space a as parameter, controlling where temporary memory will be allocated.

    circularBuffer takes an additional `alloc` parameter, allowing to allocate memory for more than m values (e.g. using a power of two makes modulo indexing cheaper). It takes an s → t function specifying how to load values into the circular buffer, potentially performing computations.

    rotateValues takes a t → t function specifying how to write values into memory. This function must be an identity function as it will be used to rotate values between iterations. We talk about value rotation instead of register rotation because the use of registers is not guaranteed. Regular variables are used in the generated OpenCL code, and the OpenCL compiler may decide to allocate registers for them if the address space is **private**.

    Both patterns implement a finite *stream*, where a sequence of values is made available one at a time, sequentially. We add two other stream based functional patterns, with following types:

```
iterateStream : (s → t) → n.s → n.t
mapStream     : (s → t) → n.s → n.t
```

    Like **mapSeq**, both patterns represent concrete implementations of **map** and have exactly the same type. However, while **mapSeq** operates on arrays, **mapStream** operates on streams, and **iterateStream** transforms streams into arrays. The lack of stream type in Rɪsᴇ is discussed in Section 3.4.3.

    Two simple examples using the new storage folding patterns in Rɪsᴇ are shown in Figure 3.7. Both examples compute a simple stencil summing windows of 3 input values, corresponding to the high-level program **slide** 3 1 ▷ **map** (**reduce** + 0). The code that we want to generate from the two example Rɪsᴇ programs is depicted in Figure 3.8.

```
1  rotateValues private 3 (λx. x) ▷
2  iterateStream (reduceSeq + 0)
```

```
1  circularBuffer private 3 3 (λx. x) ▷
2  iterateStream (reduceSeq + 0)
```

Figure 3.7: Example RISE programs using storage folding patterns.

```
1  int tmp[3];
2  for (int i = 0; i < 2; i += 1) {
3    tmp[i] = input[i];
4  }
5
6  for (int i = 0; i < n-2; i += 1) {
7    tmp[2] = input[2 + i];
8
9    int acc = 0;
10   for (int j = 0; j < 3; j += 1) {
11     acc += tmp[j];
12   }
13
14   output[i] = acc;
15   tmp[0] = tmp[1];
16   tmp[1] = tmp[2];
17 }
```

```
1  int tmp[3];
2  for (int i = 0; i < 2; i += 1) {
3    tmp[i] = input[i];
4  }
5
6  for (int i = 0; i < n-2; i += 1) {
7    tmp[(2 + i) % 3] = input[2 + i];
8
9    int acc = 0;
10   for (int j = 0; j < 3; j += 1) {
11     acc += tmp[(i + j) % 3];
12   }
13
14   output[i] = acc;
15 }
```

Figure 3.8: Example OpenCL programs corresponding to the RISE programs of Figure 3.7.

3.4.2   Adding a New Stream Translation to DPIA

To generate code for storage folding patterns, we extend DPIA with a new translation from functional to imperative. The *stream translation* $\mathcal{S}$ produces a command which reads from a functional read expression e and calls a stream continuation function sk to continue the translation as required (Listing 3.21). The stream continuation function is called by providing a function of type (i: **nat**) → (**exp**[dt, **read**] → **comm**) → **comm**, that should stream the i-th value by calling the inner continuation function of type **exp**[dt, **read**] → **comm**.

```
𝒜(e: exp[dt, write], out: acc[dt]): comm
𝒞(e: exp[dt, read], k: exp[dt, read] → comm): comm
𝒮(e: exp[n.dt, read],
   sk: ((i: nat) → (exp[dt, read] → comm) → comm) → comm): comm
```

Listing 3.21: Types of translations, including the new stream translation

The acceptor translation of **iterateStream** triggers a stream translation (Listing 3.22). This makes sense because the pattern writes to an array output by reading from a stream input. A selection of stream translations is shown in Listing 3.23. It excludes the stream translation for **circularBuffer** which is similar to the one for **rotateValues**.

```
𝒜(iterateStream(n, dt1, dt2, f, array), out) =
  𝒮(λ(next : (i: nat) → (exp[dt1, read] → comm) → comm).
  for(n, Λi. next i (λ(x : exp[dt1, read]).
    𝒜(f x, out @ i))))
```

Listing 3.22: The acceptor translation of **iterateStream** calls $\mathcal{S}$

```
1  𝒮(mapStream(n, dt1, dt2, f, input), sk) =
2    𝒮(input, λ(nextInput : (i: nat) → (exp[dt1, read] → comm) → comm).
3      sk(Λi. λ(k : exp[dt2, read] → comm).
4        nextInput i (λ(x : exp[dt1, read]). 𝒞(f x, k))
5      ))
6  𝒮(rotateValues(a, n, m, dt, wrdt, input), sk) =
7    𝒮(input, λ(nextInput : (i: nat) → (exp[dt1, read] → comm) → comm).
8      new(a, m.dt, λrs.
9        for(m - 1, Λi.
10          nextInput i (λ(x : exp[dt, read]). 𝒜(wrdt x, rs.wr @ i)));
11        sk(Λi. λ(k : exp[m.dt, read] → comm).
12          nextInput (i + m - 1) (λ(x : exp[dt, read]).
13            𝒜(wrdt x, rs.wr @ (m - 1)));
14          k(rs.rd);
15          for(m - 1, Λi. 𝒜(wrdt (rs.rd @ (i + 1)), rs.wr @ i))
16        )))
```

Listing 3.23: Selection of stream translations

For **mapStream**, the stream translation of the input is first called in line 2, giving access to nextInput. In line 3, the stream continuation sk is called to define how to stream the i-the output value. The continuation k should be called to return this output value. Finally in line 4, nextInput is called to consume the next input value x, which is used to produce the next output value f x. $\mathcal{C}(f\ x,\ k)$ is used to translate f x to imperative.

For **rotateValues**, the stream translation is a bit more involved. A temporary buffer is allocated to hold window values in line 8. The first $m - 1$ window values are stored in the buffer in lines 9-10, before sk allows streaming values. To stream an output value, which is a window of m values:

- The m-th window value, or $i + m$-th input value, is loaded in lines 12-13.

- The window of m values is returned in line 14, by calling the continuation k.

- The values are rotated in the buffer to prepare for the next iteration in line 15.

To write values to the buffer, acceptor translations such as $\mathcal{A}(\text{wrdt x, rs.wr @ i})$ are called, since wrdt x is a functional program that needs to be translated to imperative.

It is also possible to transform any array into a stream. If there is no specialised stream translation available, we fallback on a generic one, shown in Listing 3.24.

```
𝒮(e : exp[n.dt, read], sk) =
  sk(Λi. λ(k : exp[dt, read] → comm). 𝒞(e @ i, k))
```

Listing 3.24: Generic stream translation used as a fallback

Using this newly defined stream translation, the code generated for Figure 3.7 corresponds to the desired code from Figure 3.8, modulo small syntactic details. It is interesting to note that no new imperative patterns are required to support storage folding.

3.4.3   Limitations of Storage Folding in Shine

The proposed storage folding implementation has two notable limitations.

Stream Type   The concept of finite streams is introduced, but is not reflected into the type system. As a result, it is possible to construct invalid programs that would try to use sequential streams as data-parallel arrays, for example:

```
circularBuffer private 3 3 (λx. x) ▷ mapGlobal (λx. x)
```

Code generation will fail for such programs, because stream-returning patterns have no 𝒜/𝒞 translations. This is good because buggy code will not be generated, but this is bad because well-typed programs may be invalid. Future work may look into adding a dedicated stream type to make invalid programs ill-typed.

Stencil Stride   Only stencil strides of 1 are supported (i.e. the 1 in `slide` m 1). Future work may look into adding support for arbitrary strides, and whether it would be useful.

3.4.4   Related Work

We mentioned that Halide schedules support circular buffering, but not register rotation. There are many compilers that perform storage folding, but are domain-agnostic or domain-specific.

Storage Folding in Compilers for Stream Processing   Programs that need to process potentially infinite streams of values with finite buffers make extensive use of optimisations comparable to storage folding. Optimising compilers for stream programs are an active field of research [146, 147, 148, 149, 150, 151, 152]. Circular buffering and related optimisations were studied for Synchronous Data Flow, one way to model stream programs, decades ago [153].

Storage Folding in Compilers for Stencil Computations   There exists many domain-specific compilers for stencil computations [154, 155, 156]. Storage folding optimisations are sometimes viewed as *streaming* optimisations in this domain [157].

Storage Folding in Compilers for High-Level Synthesis   Compilers that synthesise hardware often achieve storage folding through *shift-registers*, also known as *line buffers* [48, 158].

Some domain-extensible compilers were extended with storage folding optimisations independently during this thesis.

Storage Folding in Domain-Extensible Compilers   AnyHLS [159] synthesises FPGA designs, but differs from Shine by following the partial evaluation approach of AnyDSL [17]. Shir also targets FPGAs, introducing shift-registers through a `SlideStm` pattern [40] that is closely related to our **rotateValues** pattern. Shir differs from Shine as it focuses on greedy optimisation and emits code in VHDL, a hardware description language. A `MapSeqSlide` pattern was added to Lift for 2.5D tiling [160]. `MapSeqSlide` is less flexible than our patterns, as it is a monolithic equivalent to combining **rotateValues** with **iterateStream**.

3.4.5 Summary

This section adds Rise patterns to enable explicit storage folding, notably **circularBuffer** and **rotateValues**. To generate the desired imperative code, a stream translation $\mathcal{S}$ is added to DPIA on top of the existing $\mathcal{A}$ and $\mathcal{C}$ translations. Chapter 4 will demonstrate how introducing storage folding patterns via rewriting leads to generating faster code than Halide on a case study.

3.5 Conclusion

This chapter first introduced the Rise language and its Shine compiler [4, 106], both resulting from collaboration and heavily inspired by the domain-extensible Lift compiler. Shine rewrites functional programs before generating imperative code, and differs from Lift as it aims to address the controllability challenge by exploring trade-offs between automation and control of rewrite rule applications.

The novel contribution of this chapter is the design and implementation of three important code generation features. These features are crucial when using Shine to generate faster code than Halide on an image processing case study via controlled rewriting (Chapter 4), and similarly fast code as TVM on a linear algebra case study via semi-automated rewriting (Chapter 5).

- We contribute a synchronisation *barrier insertion* algorithm (DPIA$_{BI}$) that does not need to be modified when extending RISE patterns, contrasting with the barrier elimination algorithm of LIFT [107]. DPIA$_{BI}$ transforms the intermediate imperative DPIA programs. The correctness and efficiency of barrier insertion is evaluated on 38 unit tests and 10 benchmarks, mostly taken from prior LIFT work. We identify 6 differences in the code generated by SHINE and LIFT, and observe that our algorithm fixes bugs in 13 unit tests and 1 benchmark, where LIFT generates incorrect barriers (Table 3.2). There is only 1 benchmark where SHINE inserts a barrier that LIFT eliminates, and we provide a clear pathway to improve our algorithm to generate more efficient barriers than LIFT on all 48 unit tests and benchmarks.

While barriers are implicit in RISE, and thus not controllable by rewriting, the next two features add new low-level RISE patterns. Low-level patterns make implementation choices explicit in RISE, and thus controllable during rewriting.

- We add the **oclRun** RISE pattern to enable explicit *kernel execution*, where the value of an expression is computed by launching an OpenCL kernel. This requires modifying SHINE to generate imperative code for multiple OpenCL kernels, as well as the necessary host code to launch them. With this feature, 1K lines of handwritten host code are replaced with 1.2K lines of automatically generated code on a relatively simple design space exploration case study (Tables 3.3 and 3.4).

- We add the **circularBuffer** and **rotateValues** RISE patterns to enable explicit *storage folding* for temporary arrays. SHINE is modified to generate the desired imperative code when using these patterns, by adding a new stream translation to DPIA. Chapter 4 relies on storage folding to generate high performance code.

This chapter focused on code generation, at the bottom of the compilation stack. The following two chapters will focus on rewriting, moving up in the compilation stack (Figure 3.1).

I love rewriting because that is where and how you discover the story. It's like you have this skeleton, and you get to put flesh on it and hair and clothes and really wonderful jewelry.

— Caroline Leavitt

Chapter 4

Beyond Halide Scheduling with Controlled Rewriting

In Chapter 2, we saw that LIFT addresses the *extensibility challenge* using an extensible rewrite system [16], but does not address the *controllability challenge* as it fully automates optimisation. Halide addresses the *controllability challenge* through *schedules* [9], but does not address the *extensibility challenge* as it is domain-specific.

This chapter demonstrates how both extensibility and controllability are combined in SHINE to generate high-performance code. We optimise the Harris corner detection [30] – a standard image processing pipeline already encountered in Section 3.3.4 – on 4 ARM CPUs. Our results on Cortex A53 show that SHINE generates $1.32\times$ faster code than Halide (Figure 4.1). By contrast, LIFT is missing important optimisations and generates $3.12\times$ slower code than Halide.

Section 4.1 explains how optimisation is controlled by defining ELEVATE *rewriting strategies* [3, 161], and clarifies the scope of extensibility in SHINE. Section 4.2 introduces the Harris corner detection case study and how the corresponding algorithm is represented in RISE. Section 4.3 describes how 6 well-known optimisations are applied by composing rewrite rules with ELEVATE. Section 4.4 presents an experimental evaluation and Section 4.5 concludes.
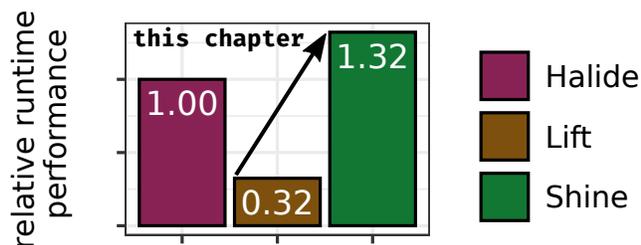

Figure 4.1: LIFT performs poorly compared to an expert Halide schedule when optimising the Harris corner detection image processing pipeline for the Cortex A53 ARM CPU. By combining domain-extensibility with controlled rewriting, SHINE outperforms Halide by $1.3\times$. Extracted from the results of Section 4.4 for an image resolution of 1536×2560 pixels.
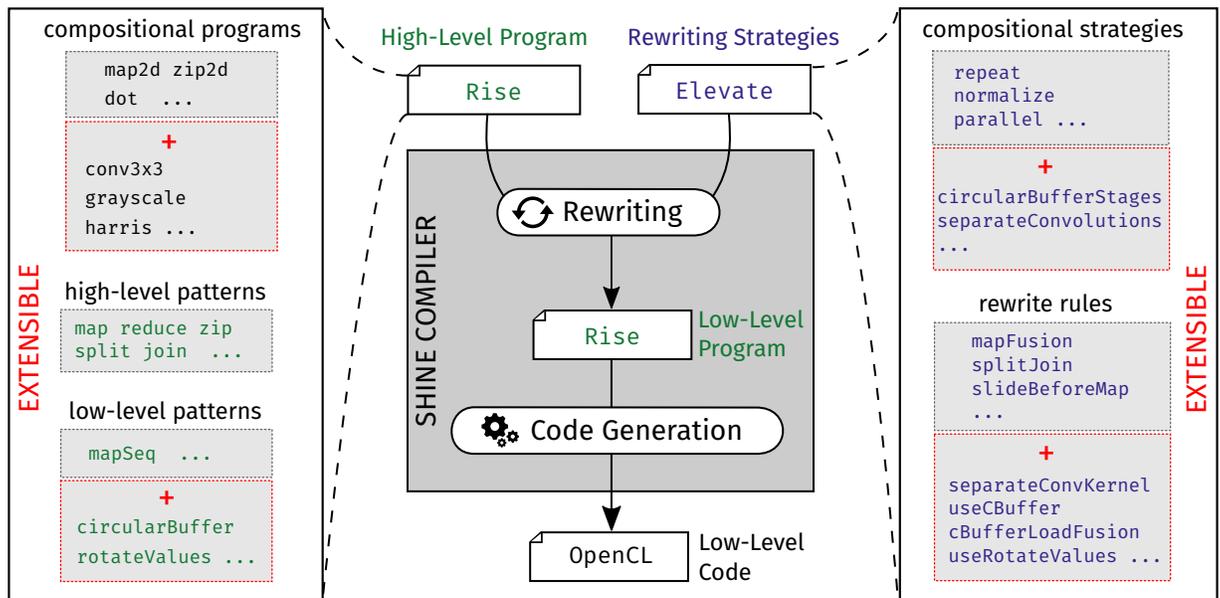
Figure 4.2: SHINE with control of optimisations: computations are composed using extensible patterns in RISE; optimisations are composed using extensible rewrite rules in ELEVATE.

4.1 Domain-Extensibility with Control of Optimisations

This chapter focuses on combining domain-extensibility with control of optimisations. As discussed in Chapter 2, our aim is to enable performance engineers to take control over the optimisation process to achieve their performance goals instead of forcing them to bypass the compiler and resort to manual optimisation.

Control of Optimisations   Control is enabled in SHINE by the use of ELEVATE rewriting strategies, as in Figure 4.2. High-level RISE programs describe computations by composing extensible high-level patterns. Optimisations are encoded by applying rewrite rules, leading to a low-level program from which imperative code such as OpenCL is generated. The novelty is that *rewriting strategies* are defined in a companion ELEVATE language, enabling performance engineers to describe optimisations as compositions of rewrite rules (top right of Figure 4.2).

ELEVATE is the work of Bastian Hagedorn [3, 161]. I contributed to its early design and demonstrated that the ideas generalise to image processing [1]. The separation between RISE programs and ELEVATE strategies resembles the separation between algorithms and schedules in domain-specific compilers such as Halide. There, a *schedule* describes the optimisations to apply to an *algorithm* that defines the functional behavior of the computation. This fine grained control of optimisations allows performance engineers to steer the compiler to generate highly efficient code that would have been hard to reach automatically. While Halide schedules are implemented as a set of ad-hoc, predefined APIs exposed by compiler writers, ELEVATE strategies enable performance engineers to define their own program transformations in an extensible and composable way.

Domain-Extensibility   The domain-extensibility of Shine is pictured in red on the sides of Figure 4.2. This domain-extensibility has restricted scope as Rise remains an array language: the extension mechanisms are patterns, rewrite rules, as well as rewriting strategies.

High-level patterns are added, such as `slide` that creates a sliding window and enables defining stencil computations, previously added in [16]. High-level rewrite rules are added, such as `separateConvKernel` that encodes the separability of a convolution kernel. Lower-level extensions are also possible, and may be target-specific. Low-level implementation patterns are added, such as `circularBuffer` or `rotateValues`, which also requires implementing code generation for them as discussed in Chapter 3. Low-level patterns typically come with rewrite rules such as `useCBuffer` or `useRotateValues` enabling their introduction during rewriting, as we will see later.

Abstractions are built on top of patterns and rewrite rules by composition. Compositional programs may be defined to abstract over patterns, such as `stencil2d` that is defined in Section 4.2 by composing `map`, `slide` and `transpose`. Compositional Elevate rewriting strategies may be defined to abstract over rewrite rules, such as `separateConvolutions` that composes `separateConvKernel` with other generic rewrite rules.

Although domain-specific extensions are critical to achieve high performance, we strive to minimise extensions and maximise reuse across algorithms and targets. It is possible to create a rich space of optimisations by composing simple, specialised rewrite rules with generic ones. For example, of the 74 rewrite rules applied to the Harris corner detection in this chapter, 2 rewrite rules are inherited from the lambda calculus, 49 rewrite rules express generic properties of high-level patterns, 19 rewrite rules manipulate low-level patterns (4 are for storage folding), and 4 rewrite rules require domain-specific knowledge (for separability and vectorisation).

Reproducing Well-Known Optimisations   This chapter defines Elevate strategies that reproduce well-known domain-specific optimisations when applied to a given Rise program. The primary goal is to demonstrate the existence of a rewrite sequence leading to the desired optimised program, which could be found automatically. Chapter 5 will explore trade-offs between control and automation of optimisations.

Next, we showcase Elevate strategies by example before giving a language overview.

4.1.1   Elevate Strategies by Example

To introduce Elevate strategies, we pick up the dot product example from Section 3.1.1. Recall that the high-level `dot` program (Listing 3.1) may be rewritten into the low-level `dotSeq` program (Listing 3.3) by applying the `reduceMapFusion` rewrite rule (Listing 3.2, repeated in Listing 4.1). However, many other rewrites are possible, and Section 3.1.1 did not discuss how to decide which rewrite rule to apply where. Defining an Elevate strategy is one way to make this decision explicit.

```
rule reduceMapFusion = map f ▷ reduce g init
                    ↦ reduceSeq (λacc x. g acc (f x)) init
```

Listing 4.1: reduceMapFusion rewrite rule.

For example, we define the lowerDot strategy to rewrite the dot program:

```
strategy lowerDot = topDown(reduceMapFusion)
```

lowerDot uses a **topDown** *traversal* combinator to apply the reduceMapFusion rewrite rule once, following a depth-first top-down traversal of the RISE Abstract Syntax Tree (AST). We can visualise the traversal steps on the dot program:[1]

```
1  zip a b ▷ map (λx. (fst x) × (snd x)) ▷ reduce + 0
2  zip a b ▷ map (λx. (fst x) × (snd x)) ▷ reduce + 0
3  zip a b ▷ map (λx. (fst x) × (snd x)) ▷ reduce + 0
4  zip a b ▷ map (λx. (fst x) × (snd x)) ▷ reduce + 0
5  zip a b ▷ map (λx. (fst x) × (snd x)) ▷ reduce + 0
6  zip a b ▷ map (λx. (fst x) × (snd x)) ▷ reduce + 0
7  zip a b ▷ map (λx. (fst x) × (snd x)) ▷ reduce + 0
8  zip a b ▷ reduceSeq (λacc x. acc + (fst x) × (snd x)) 0
```

After failing to apply reduceMapFusion six times (red rectangles), the traversal succeeds when visiting **map** .. ▷ **reduce** + 0 (green rectangle). The left hand side of the rewrite rule is then replaced with its right hand side, producing the low-level dotSeq program.

4.1.2 ELEVATE Language Overview

This subsection gives Section 4.1.2 an overview of the ELEVATE strategy language. A more detailed description is available in [3]. ELEVATE is heavily inspired by earlier works on strategy languages for term rewriting systems [70, 162]. Like RISE, ELEVATE is embedded in Scala.[2]

ELEVATE strategies are modeled as functions transforming programs from RISE, or other languages [163]. The type of a strategy is parameterised by P, the type of the rewritten program:

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

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

A **RewriteResult** is an applicative error monad [164, 165]. Applied to a program, a strategy may succeed and return the transformed program in **Success**, or alternatively fail and return the unsuccessful strategy in **Failure**.

---

[1]for simplicity, we pretend that ▷ is an AST node, even though it is syntactic sugar for function application
[2]the implementation of ELEVATE is open-source: https://github.com/elevate-lang/elevate

The simplest examples of strategies are **id** that always succeeds by returning the input program unchanged, and **fail** that always fails by returning itself:

```
strategy id = λp. Success(p)
strategy fail = λp. Failure(fail)
```

In this thesis we use a convenient **strategy** syntax, different from what we would write in the Scala embedding:

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

Rewrite Rules as Strategies   In ELEVATE, rewrite rules are modelled as strategies. For example, reduceMapFusion succeeds when the input program matches its left-hand side, returning its instantiated right-hand side. The convenient **rule** syntax used in Listing 4.1 allows defining rules independently from the concept of ELEVATE strategy. Separating rules from strategies is also useful because rules need to be trusted or verified, while strategies are correct by composition. Rules defined using the **rule** syntax have a corresponding definition as ELEVATE strategies in Scala. For example, reduceMapFusion may be defined by pattern matching over the AST of RISE programs:

```
def reduceMapFusion: Strategy[Rise] = {
  case app(app(app(reduce, g), init), app(app(map, f), in)) =>
    Success(app(app(app(reduceSeq, fun(acc => fun(x =>
      app(app(g, acc), app(f, x)))), init), in)))
  case _ => Failure(reduceMapFusion)
}
```

Strategy Combinators   *Strategy combinators* enable defining strategies as compositions of other strategies. The sequential combinator (**;**) composes two strategies by performing the second one on the transformed program from the first strategy. It may be defined in terms of the standard monadic bind combinator. The **<+** combinator (called *left choice*) composes two strategies by performing the second only if the first strategy fails. It may be defined in terms of the standard monadic mplus combinator. More strategy combinators can be defined in terms of these two basic ones, as shown in Listing 4.2. The **try** combinator tries to perform a given strategy, and does nothing if the strategy fails, returning the input program unmodified. **repeat** performs a strategy repeatedly until it fails.

```
strategy try(s) = s <+ id
strategy repeat(s) = λp. try(s; repeat(s))(p)
```

Listing 4.2: Selection of ELEVATE combinator definitions

Traversals   *Traversals* are strategy combinators controlling the AST location at which other strategies are applied. We have already seen **topDown** that traverses the program using a depth-first top-down AST traversal, performing the given strategy at the first possible location. Traversals like **topDown** or **bottomUp** are defined recursively in terms of more basic traversal steps called one, some and all that need to be provided for a given program type P. More traversals are defined in [3], including RISE-specific traversals.

Normal Forms   *Normal forms* are often desired when rewriting programs, such as the standard lambda calculus $\beta\eta$ normal form [166] or fusion normal forms [167]. Later in this chapter, a custom reduceFusedForm is used, combining reduction rules such as β and η reductions with fusion rules such as map fusion. To enforce normal forms, the **normalize** combinator applies a given strategy somewhere in the program repeatedly, until it cannot be applied anymore:

```
strategy normalize(s) = repeat(topDown(s))
```

After performing **normalize**(s), we know that s can no longer be applied to any location in the transformed program. Termination of normal forms, and ELEVATE strategies in general, is the performance engineer's responsibility.

4.2   The Harris Corner Detection Case Study

The image processing domain is a good candidate for a case study. First, the industrial-strength domain-specific compiler Halide offers a challenging comparison point. Halide also supports controlling optimisation through schedules, enabling comparison to the control of ELEVATE strategies. Second, image processing pipelines present unique optimisation challenges that were not explored in prior LIFT work. To reach high-performance, SHINE needs to be extended.

The Harris corner (and edge) detector [30] is a well established image processing pipeline that we use as a case study in this chapter. The following subsections first describe how the Harris corner detection is defined in Halide before describing how to represent the Harris corner detection in RISE.

4.2.1   The Harris Corner Detection Algorithm in Halide

While many algorithmic variations of the Harris corner detector exist, we use the algorithm found in the Halide repository as our reference (Listing 4.3). The Harris image processing pipeline is defined by inter-dependent functions. Each function is defined using an indexed notation and goes from coordinates – belonging to infinite integer domains – to value. This variant does not include padding for the stencil borders, and instead the output image is slightly smaller than the input image. Image bounds are implicit in the Halide algorithm, they are inferred using interval arithmetic, proceeding from output to input.

```
1  Var x, y, c;
2  Func gray, Iy, Ix, Ixx, Iyy, Ixy, Sxx, Syy, Sxy, det, trace;
3
4  gray(x, y) = (0.299f * input(x, y, 0) +
5                0.587f * input(x, y, 1) +
6                0.114f * input(x, y, 2));
7
8  Iy(x, y) = gray(x - 1, y - 1) * (-1.0f / 12) + gray(x - 1, y + 1) * (1.0f / 12) +
9             gray(x, y - 1) * (-2.0f / 12) + gray(x, y + 1) * (2.0f / 12) +
10            gray(x + 1, y - 1) * (-1.0f / 12) + gray(x + 1, y + 1) * (1.0f / 12);
11
12 Ix(x, y) = gray(x - 1, y - 1) * (-1.0f / 12) + gray(x + 1, y - 1) * (1.0f / 12) +
13            gray(x - 1, y) * (-2.0f / 12) + gray(x + 1, y) * (2.0f / 12) +
14            gray(x - 1, y + 1) * (-1.0f / 12) + gray(x + 1, y + 1) * (1.0f / 12);
15
16 Ixx(x, y) = Ix(x, y) * Ix(x, y);
17 Iyy(x, y) = Iy(x, y) * Iy(x, y);
18 Ixy(x, y) = Ix(x, y) * Iy(x, y);
19
20 Expr sum3x3(Func f, Var x, Var y) {
21   return f(x - 1, y - 1) + f(x - 1, y) + f(x - 1, y + 1) +
22          f(x, y - 1) + f(x, y) + f(x, y + 1) +
23          f(x + 1, y - 1) + f(x + 1, y) + f(x + 1, y + 1);
24 }
25
26 Sxx(x, y) = sum3x3(Ixx, x, y);
27 Syy(x, y) = sum3x3(Iyy, x, y);
28 Sxy(x, y) = sum3x3(Ixy, x, y);
29
30 det(x, y) = Sxx(x, y) * Syy(x, y) - Sxy(x, y) * Sxy(x, y);
31 trace(x, y) = Sxx(x, y) + Syy(x, y);
32 output(x, y) = det(x, y) - 0.04f * trace(x, y) * trace(x, y);
```

Listing 4.3: Harris operator algorithm in Halide, from the official Github repository (https://
github.com/halide/Halide/blob/c2b6da28843a36e53b1c2cf9fd6fa390afeb5896/apps/harris/harris_generator.cpp#L19-L61).
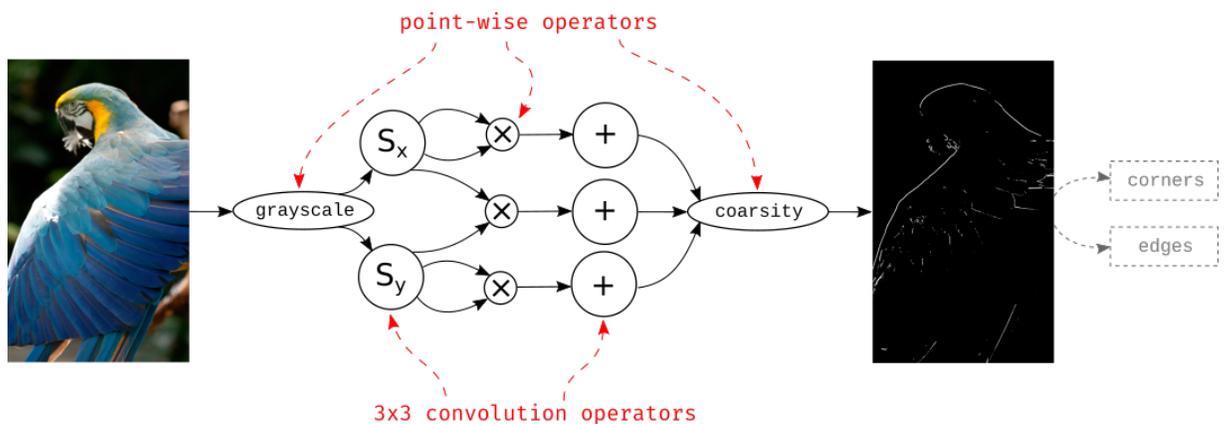

Figure 4.3: Harris corner detection computation flow, and example image from the Halide
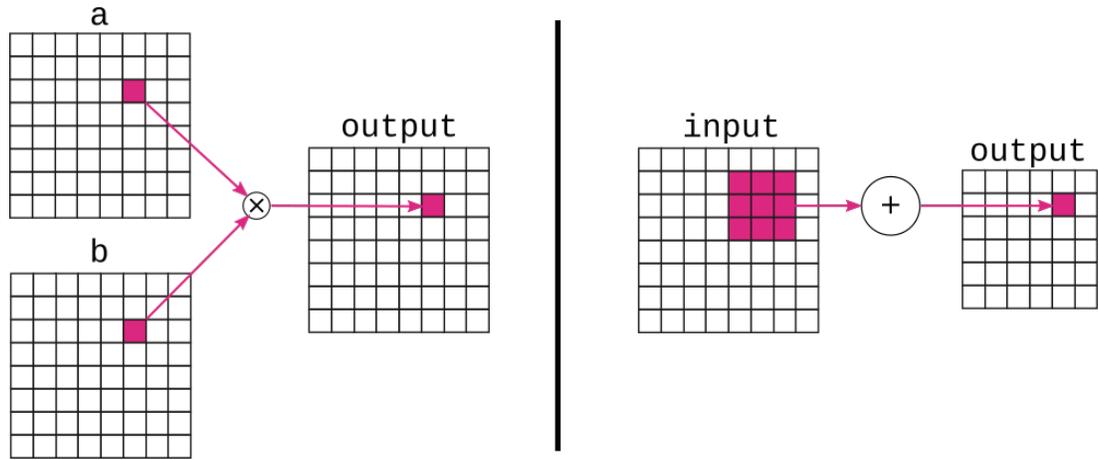repository (https://github.com/halide/Halide/blob/c2b6da28843a36e53b1c2cf9fd6fa390afeb5896/apps/images/rgb.png).

Figure 4.4: Example of pointwise operator (left) and convolution operator (right).

Computation Flow   Figure 4.3 visualises the Harris operator computation flow, showing computation nodes and data dependency edges. Given an image on the left, point-to-point operators (grayscale, multiplications $\times$, coarsity) and 3×3 convolutions (sobel operators S_x and S_y, sums $+$) are combined to detect corners and edges highlighted in the output on the right. The output of the S_x computation is the `Ix` image and the output of the coarsity computation is the `output` image. The det and trace images from the Halide definition are not explicitly visible in this graph, they are collapsed into the coarsity computation.

Example Operators   Figure 4.4 gives visual examples of pointwise and convolution operators. On the left, the pointwise operator $\times$ multiplies each pixel of a with a pixel of b to produce a pixel of output. On the right, the convolution operators $+$ sums each 3×3 neighbourhood of input pixels to produce a pixel of output. To avoid reading outside of the image, the output of the convolution (6×6) is smaller than its input (8×8).

4.2.2   Representing the Harris Corner Detection in RISE

To represent the Harris corner detection as a high-level RISE program, we proceed by composition. We first look at how to define pointwise operators before looking at convolution operators. Once we have these building blocks, we can easily assemble them into the complete Harris operator according to the computation flow from Figure 4.3.

Representing Pointwise Operators   To represent pointwise operators in the high-level RISE language, we do not require any image-specific patterns (Listing 4.4).

Where indices are used to access values in Halide, **map** is used to access array values in RISE. Where values from multiple inputs are freely combined in Halide, **zip** is used to combine arrays in RISE. Finally, using **transpose** in RISE is similar to swapping indices in Halide. As done in LIFT[15, 16], 2D mapping and zipping is defined by composition (Lines 1 and 4).

In Line 7, we define `grayscale` by moving the outer array dimension corresponding to the RGB color channels to the inside where it is consumed by a dot product. In Lines 11 and 14, we define $\times_{2D}$ and `coarsity` by combining pixels of input arrays with `zip2d` before processing each combination with `map2d`.

Note that we use **let** x = v ... as syntactic sugar for (λx. ...) v.

```
1  def map2d (f: s → t): n.m.s → n.m.t =
2    map (map f)
3
4  def zip2d (a: n.m.s) (b: n.m.t): n.m.(s × t) =
5    zip a b ▷ map (λp. zip (fst p) (snd p))
6
7  def grayscale (RGB: 3.n.m.f32): n.m.f32 =
8    RGB ▷ transpose ▷ map transpose ▷
9    map2d (dot [0.299  0.587  0.114])
10
11 def ×₂D (a: n.m.f32) (b: n.m.f32): n.m.f32 =
12   zip2d a b ▷ map2d ×
13
14 def coarsity (Sₓₓ: n.m.f32) (Sₓᵧ: n.m.f32) (Sᵧᵧ: n.m.f32) (κ: f32): n.m.f32 =
15   zip2d Sₓₓ (zip2d Sₓᵧ Sᵧᵧ) ▷
16   map2d (λp.
17     let (sₓₓ, (sₓᵧ, sᵧᵧ)) = p
18     let det = sₓₓ × sᵧᵧ - sₓᵧ × sₓᵧ
19     let trace = sₓₓ + sᵧᵧ
20     det - κ × trace × trace)
```

Listing 4.4: High-level pointwise operators in RISE.

Representing Convolution Operators Where neighbor values are accessed by indexing in Halide, they are collected into arrays using **slide** in RISE (Listing 4.5). The **slide** pattern is one-dimensional but can be composed to create multi-dimensional sliding windows [16], `slide2d` is an example shown in Line 1. Two-dimensional stencil operators are then constructed by first creating neighborhoods with `slide2d` before processing them using `map2d` as shown in Line 4.

In Line 7, we define the more specific `conv3x3` in terms of `stencil2d`, since convolutions are stencils. For each neighbourhood, convolution weights are combined with neighbourhood values using `dot` (**join** weights)(**join** w). From there, defining S_x and S_y only requires providing the convolution weights (Lines 10 and 12). Finally in Line 14, $+_{3\times3}$ is defined by summing neighbourhood values using **reduce** + ∘ (**join** w).

```
1  def slide2d (n_sz: nat) (n_sp: nat) (m_sz: nat) (m_sp): nat =
2    map (slide n_sz n_sp) ▷ slide m_sz m_sp ▷ map transpose
3
4  def stencil2d (f: N.M.s → t): (n + N − 1).(m + M − 1).s → n.m.t =
5    slide2d N 1 M 1 ▷ map2d f
6
7  def conv3x3 (weights: 3.3.f32): (n + 2).(m + 2).f32 → n.m.f32 =
8    stencil2d (λ(w : 3.3.f32). dot (join weights) (join w))
9
```

$$
10 \quad \textbf{def} \ S_x \ = \ \texttt{conv3x3} \ (\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \times \tfrac{1}{12})
$$

```
11
```

$$
12 \quad \textbf{def} \ S_y \ = \ \texttt{conv3x3} \ (\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \times \tfrac{1}{12})
$$

```
13
14  def +_{3×3} = stencil2d (λ(w : 3.3.f32). reduce + 0 (join w))
```

Listing 4.5: High-level stencil operators in RISE.

Representing the Entire Pipeline Listing 4.6 defines the entire Harris image detection pipeline by composing all of the previously defined building blocks. Note that the final RISE program only contains generic high-level patterns and basic language constructs - no image-specific internal representation is required. All program abstractions such as map2d or slide2d can be eliminated by inlining their definition. In the RISE program, array bounds are inferred as part of type inference and may be explicitly annotated as seen in previous listings. Here type inference will check that the RGB input is of type 3.(n+4).(m+4).f32 and the smaller output of type n.m.f32.

```
1  def harris (RGB: 3.(n + 4).(m + 4).f32): n.m.f32 =
2    let I = grayscale RGB
3    let I_x  = S_x  I
4    let I_y  = S_y  I
5    let I_xx = ×_{2D} I_x  I_x
6    let I_xy = ×_{2D} I_x  I_y
7    let I_yy = ×_{2D} I_y  I_y
8    let S_xx = +_{3×3} I_xx
9    let S_xy = +_{3×3} I_xy
10   let S_yy = +_{3×3} I_yy
11   coarsity S_xx S_xy S_yy 0.04
```

Listing 4.6: High-level Harris operator in RISE.

4.3 Optimising the Harris Corner Detection with ELEVATE

We now study how to optimise the RISE Harris corner detection. As a composition of point-wise and stencil operators, the Harris operator is more complex than its individual parts, which exposes more optimisation opportunities. Although our case study focuses on the Harris operator, the optimisations that we study are generalisable and applicable to other image processing pipelines that compose pointwise and stencil operators.

The LIFT project has been extended to express stencils and overlapped tiling [16], but lacks crucial optimisations for image processing [23], such as operator fusion or circular buffering, that are supported by Halide. This leads to poor performance, Figure 4.1 showed an example where LIFT generates $3.12\times$ slower code than Halide. This section uses an optimised Halide schedule of the Harris operator as reference to demonstrate how ELEVATE is used to perform equivalent and additional optimisations of RISE programs.

Listing 4.7 shows the Halide schedule describing the optimisations applied to the Harris operator. The schedule applies multi-threading, vectorisation and describes how the stages interact by storing images in intermediate buffers. Halide makes some implicit optimisation decisions appropriate for image processing pipelines, such as using circular buffers.

```
const int vec = natural_vector_size<float>();
output.split(y, y, yi, 32).parallel(y)
  .vectorize(x, vec);
gray.store_at(output, y).compute_at(output, yi)
  .vectorize(x, vec);
Ix.store_at(output, y).compute_at(output, yi)
  .vectorize(x, vec);
Iy.store_at(output, y).compute_at(output, yi)
  .vectorize(x, vec);
Ix.compute_with(Iy, x);
```

Listing 4.7: Optimised Harris operator schedule from the Halide repository (https://github.com/halide/Halide/blob/c2b6da28843a36e53b1c2cf9fd6fa390afeb5896/apps/harris/harris_generator.cpp#L86-L90).

Figure 4.5 visualises the computation with these optimisations applied. The upper part of Figure 4.5 shows the input image on the left, where three color channels are combined by *grayscale*. Grayscale lines are stored in a buffer to be processed by the sobel operators (S_x and S_y). The resulting buffers are then multiplied ($\times$), summed ($+$) and *coarsity* is applied to compute the final output. *Operator fusion* is applied to the computational flow (Figure 4.3) so that only two intermediate buffers are used. *Multi-threading* is exploited by parallelising the y dimension and computing chunks of output lines in parallel (thread$_0$ and thread$_1$ annotations on the right). *Circular buffers* are used for the intermediate results. Each thread stores three lines in the buffer I. These lines are used to compute two lines and store them in buffers I_x and I_y. Similarly, three lines of both I_x and I_y are used to compute one line of the output.
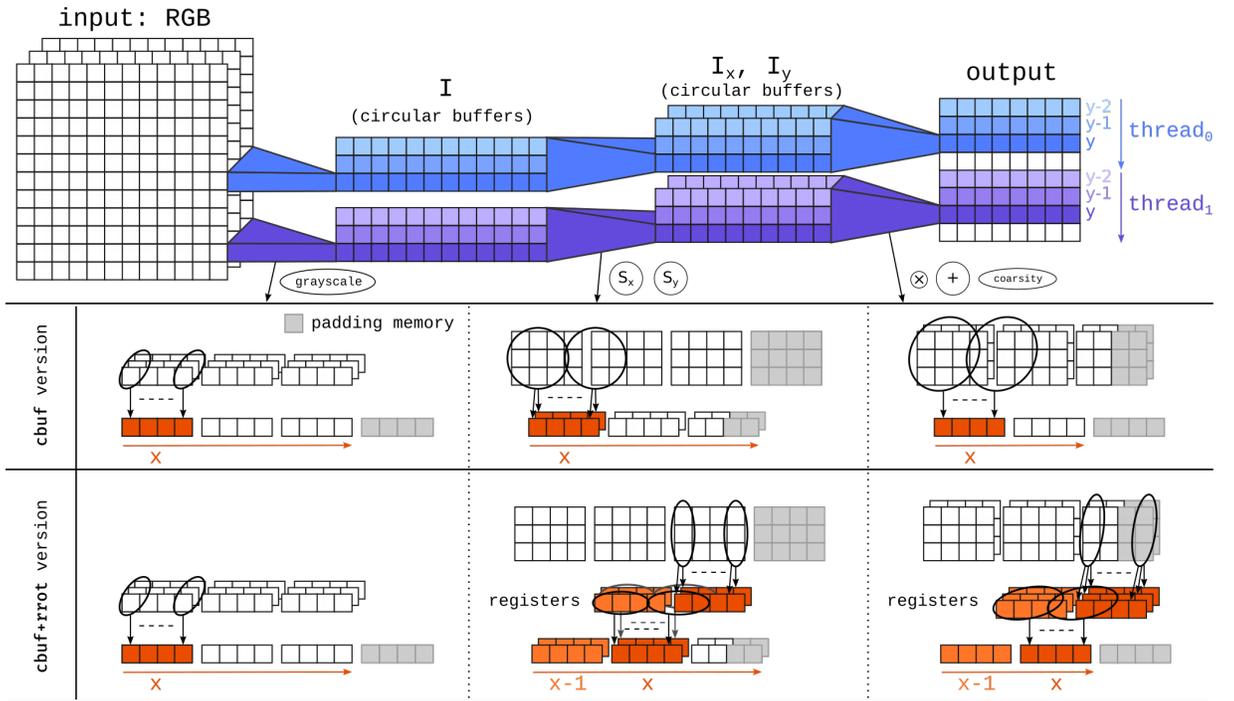
Figure 4.5: Overview of the optimisations applied on the Harris corner detector.

The lower part of Figure 4.5 shows two different ways to optimise the computations of individual image lines. The `cbuf` version is what Halide does: it uses *vectorisation* to process lines one vector at a time. The `cbuf+rrot` version below is not supported by Halide at the time of writing: it also uses *vectorisation* but further incorporates *convolution separation*, enabling *register rotation* as described in [23]. This is shown in the center and right of the bottom row, where the two-dimensional reductions are decomposed in a vertical reduction followed by a horizontal reduction. Temporary vector registers are rotated to hold the last vertical reductions that are used for a horizontal reduction.

The following subsections first show how to replicate the optimisations described by the Halide schedule with extensible Elevate strategies. This already goes beyond the capabilities of the Lift compiler. Then, they show how to go beyond the optimisations that Halide performs by incorporating the additional convolution separation and register rotation optimisations.

4.3.1   Reproducing the Halide Optimisations with Elevate

Listing 4.8 shows the Elevate rewriting strategy that reproduces the optimisations from the reference Halide schedule through a sequential composition of smaller strategies. In the following, we discuss the purpose of each smaller strategy and give intuitions on how to define them in Elevate by composing many individual rewrite rule applications.

```
1  strategy cbufVersion =
2    fuseOperators;
3    splitPipeline(32); parallel;
4    vectorizeReductions(vec);
5    harrisIxWithIy;
6    circularBufferStages;
7    sequentialLines;
8    usePrivateMemory; unrollReductions
```

Listing 4.8: ELEVATE strategy using circular buffering for the Harris operator.

Operator Fusion   The reference Halide schedule specifies which temporary values should be stored in memory using **store_at** directives. Otherwise operators are fused by default, storing temporary results in registers instead of memory. This transformation is more complex than loop fusion, which is why LIFT fails to apply it using its simple **map**-fusion rules. In, ELEVATE we define the fuseOperators strategy (all strategy definitions can be found in our artefact) transforming the Harris program (Listing 4.6) into a pipeline over image lines:

```
map grayLine  ▷ slide 3 1 ▷
map sobelLine ▷ slide 3 1 ▷ map coarsityLine
```

Where grayLine is a function computing a grayscale line, sobelLine a function computing a line of sobel convolutions, and coarsityLine a function computing a line of output (with multiplications, sums and coarsity fused) as shown in Figure 4.5.

Multi-threading   To take advantage of thread-level parallelism, the Halide schedule splits the output into parallel chunks of 32 lines: output.**split**(y, y, yi, 32).**parallel**(y). The ELEVATE strategy splitPipeline(32);parallel has the same effect, producing a program that slides over $32 + 4$ lines of input with step 32 to compute chunks of size 32 in parallel:

```
slide (32+4) 32 ▷ mapGlobal (
  map grayLine  ▷ slide 3 1 ▷
  map sobelLine ▷ slide 3 1 ▷
  map coarsityLine
) ▷ join
```

Parallelism is achieved with the low-level **mapGlobal** pattern that applies the nested function in parallel across global threads. The strategy itself starts by splitting the last map in the pipeline with the splitJoin rewrite rule. Then, it propagates this split to the rest of the pipeline by normalising it with various movement rules. Finally, all possible map fusions are applied in the pipeline. All the involved rules are in Listing 4.9.

```
1  rule splitJoin(p: nat) =
2    map f ↦ split p ▷ map (map f) ▷ join
3
4  rule slideAfterSplit =
5        slide n m ▷ split p
6    ↦ slide (p+n-m) p ▷ map (slide n m)
7
8  rule slideBeforeMap =
9    map f ▷ slide n m ↦ slide n m ▷ map (map f)
10
11 rule slideBeforeSlide =
12       slide n 1 ▷ slide m k
13   ↦ slide (m+n-1) k ▷ map (slide n 1)
14
15 rule mapFusion = map f ▷ map h ↦ map (f ▷ h)
16
17 rule useMapGlobal = map f ↦ mapGlobal f
```

Listing 4.9: Rules involved in the multi-threading optimisation.
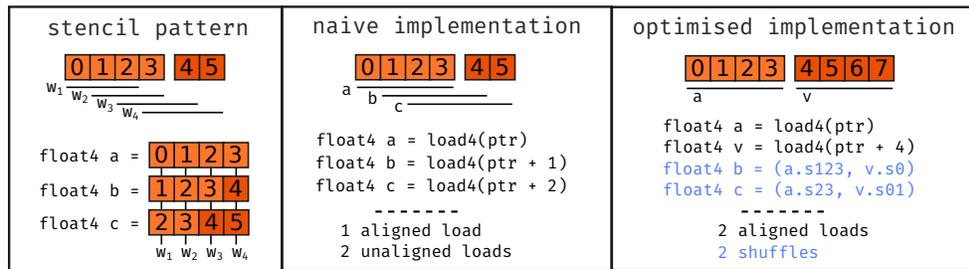
Figure 4.6: Example of memory loads for a vectorised 1D stencil of size 3. The code is written in pseudo OpenCL syntax.

Vectorisation   SIMD parallelism (Single Instruction, Multiple Data) is achieved with vector instructions such as NEON instructions on ARM processors. In the Halide schedule, multiple `.vectorize`(x, vec) directives enable this optimisation at multiple spots.

The `vectorizeReductions`(vec) strategy has a similar effect, vectorising all reductions of a program. To illustrate how the strategy works, we consider a sub-expression found in the Harris operator:

```
map (reduce + 0) ▷ map f
```

It is vectorised by interpreting the input as a two dimensional array of vectors using **asVector** and computing on vectorised data before going back to scalars using **asScalar**:

```
transpose ▷ map (asVector v) ▷ transpose
  ▷ map (reduce (mapVec +) (vectorFromScalar 0))
  ▷ map (mapVec f) ▷ asScalar
```

Where **mapVec** vectorises a scalar function. Currently, **mapVec** can only be used on functions that consists of easy to vectorise operators such as addition, multiplication, or constants.

```
1  strategy vectorize(v: nat) =
2    startVectorization(v);
3    normalize(vectorizeBeforeMap <+ vectorizeBeforeMapReduce)
4
5  rule startVectorization(v: nat) =
6    a: n × v.s ↦ a ▷ asVector v ▷ asScalar
7
8  rule vectorizeBeforeMap =
9    map f ▷ asVector v ↦ asVector v ▷ map (mapVec f)
10
11 rule vectorizeBeforeMapReduce =
12     map (reduce f init) ▷ asVector v
13 ↦ transpose ▷ map (asVector v) ▷ transpose ▷
14     map (reduce (mapVec f) (vectorFromScalar init))
```

Listing 4.10: Strategy and rules involved in vectorisation

The desired vectorisation can be defined with an ELEVATE strategy composing simpler rewrite rules as shown in Listing 4.10. In practice, arrays are often not multiples of the vector width. There are different ways to handle this, but in this case study we round inputs, outputs and temporaries up to a multiple of the vector width – an option that Halide also provides.

When vectorising stencils the computations are performed on the w_i components of three vector values, as shown in the left of Figure 4.6. The inputs of vectorised stencils are not aligned in memory and can be loaded in different ways. The naive implementation performs three loads, two of which are not aligned at a vector boundary, as shown in the middle of Figure 4.6. The optimised implementation, used by RISE, only performs two vector loads followed by vector shuffle instructions, as shown in the right of Figure 4.6.

Circular Buffering   Circular buffers leverage both the spatial locality of stencils and the temporal locality of sequential execution: only the last m intermediate results need to be stored in memory, and modulo indexing is used: $T[i]$ can be stored in $M[i \mod m]$. With transparently managed caches, this reduces memory usage and delays cache overflow.

With Halide, `.store_at`(output, y).`compute_at`(output, yi) implicitly triggers the use of circular buffers for the introduced temporary. When combined with the previous multi-threading optimisation, a separate set of circular buffers is used inside each parallel chunk – where execution is still sequential – as shown in Figure 4.5.

The ELEVATE strategy `circularBufferStages` has the same effect, producing a program with the shape:

```
slide (32+4) 32 ▷ mapGlobal (
  circularBuffer global 3 3 grayLine  ▷
  circularBuffer global 3 3 sobelLine ▷
  iterateStream coarsityLine
) ▷ join
```

```
1  rule useCBuffer(a: addr) =
2    slide m 1 ↦ circularBuffer a m m (λx. x)
3
4  rule cBufferLoadFusion =
5      circularBuffer a alloc m load (map f in)
6    ↦ circularBuffer a alloc m (λx. load (f x)) in
7
8  rule useIterateStream = map f ↦ iterateStream f
```

Listing 4.11: Rewrite rules involved in circular buffering

The **circularBuffer** pattern is a new low-level pattern that we added to RISE in Section 3.4. Given an input array, the **circularBuffer** pattern returns an array of sliding windows similar to the **slide** pattern, but the last m values have been loaded into the circular buffer. The **iterateStream** pattern is used to read sequentially from the circular buffer. circularBufferStages works by rewriting **slide** into the **circularBuffer** pattern, fusing **circularBuffer** and **map**, and introducing the **iterateStream** pattern using the rewrites rules of Listing 4.11.

Other Optimisations A couple of other optimisations are encoded as ELEVATE strategies. The harrisIxWithIy strategy emulates the Ix.**compute_with**(Iy, x) directive from Halide, fusing the loops computing these two intermediate results. The sequentialLines strategy makes individual line computations sequential, usePrivateMemory stores various temporaries in private memory, and unrollReductions unrolls reduction loops.

These transformations are not mentioned in the Halide schedule, the two first ones happen implicitly, while reductions have already been unrolled in the algorithm definition. All these optimisations are already well supported as rewrites by LIFT but have not been encoded as rewriting strategies before.

4.3.2 Expressing Optimisations beyond Halide with ELEVATE

The previous section demonstrates how to express all of the optimisations applied by the Halide schedule from Listing 4.7, using ELEVATE strategies. These optimisations are already beyond the reach of the LIFT compiler and its automatic search. This section further demonstrates how the extensibility of ELEVATE is leveraged to apply optimisations that are not implemented by Halide, further reducing runtime as shown in Section 4.4.

Listing 4.12 shows an ELEVATE strategy that additionally incorporates convolution separation and register rotation optimisations, on top of the previous optimisations from Section 4.3.1. These two optimisations are orthogonal to multi-threading and circular buffering in this case study, as they operate on a different dimension. Separating the convolution is necessary to enable register rotation, and is not expressible in Halide without manually changing the algorithm. Register rotation is recognised as a worthwhile optimisation by the Halide developers,

but is not yet supported by Halide (https://github.com/halide/Halide/issues/2905).

Implementing register rotation in Halide would require non trivial extensions including changes to the scheduling API, resulting in significant work. We discuss here how both optimisations are applied by defining an ELEVATE strategy outside of the SHINE compiler, leveraging the simple **rotateValues** pattern added in Section 3.4.

```
1  strategy cbuf+rrotVersion =
2    fuseOperators;
3    splitPipeline(32); parallel;
4    separateConvolutions;
5    vectorizeReductions(vec);
6    harrisIxWithIy;
7    circularBufferStages;
8    rotateValuesAndConsumeLines;
9    usePrivateMemory; unrollReductions
```

Listing 4.12: ELEVATE strategy applying convolution separation and register rotation to the Harris operator. These optimisations are not available in Halide. Changes compared to Listing 4.8 are highlighted in pink.

Convolution Separation   The two-dimensional sobel and sum convolutions in the Harris detector are separable into two one-dimensional convolutions following the observation that the convolution kernel matrix is separable into a column and row vector:[3]

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$

This decomposition, sometimes called convolution separation, is used to reduce both memory accesses and arithmetic complexity, but is not possible for arbitrary convolutions as it depends on the weights involved. Convolution separation is often manually applied [168, 169, 170, 171].

With ELEVATE, such a domain-specific, or even convolution-specific optimisation can be defined outside of the SHINE compiler. This is valuable because incorporating this capability does not require re-engineering the entire compilation stack, something that is challenging in domain-specific compilers as discussed in Section 2.2.2. In RISE, 2D convolutions typically operate on a vertical neighborhood of lines (nbhV). Multiple 2D neighborhoods are created (nbh2d) with **slide** and **transpose** before performing a dot product between the 2D weights and each neighborhood:

```
nbhV ▷ map (slide 3 1) ▷ transpose ▷ map (λnbh2d.
  dot (join weights2d) (join nbh2d))
```

---

[3]the same principles apply to both column-row and row-column decompositions

```
1  rule separateConvKernel(weights2d, wV, wH) =
2      dot (join weights2d) (join nbh)
3    ↦ nbh ▷ transpose ▷ map (dot wV) ▷ dot wH
4
5  strategy pushSeparation(separate) =
6    topDown(separate); reducedFissionedForm;
7    topDown(mapSlideAfterTranspose);
8    reducedFusedForm; reducedFissionedForm;
9    normalize(slideAfterMapMapF)
10
11 rule mapSlideAfterTranspose =
12     map (slide n m) ▷ transpose
13   ↦ transpose ▷ slide n m ▷ map transpose
14
15 rule slideAfterMapMapF =
16   slide n m ▷ map (map f) ↦ map f ▷ slide n m
```

Listing 4.13: Strategy and rules to separate a convolution through one dimension.

The 2D weights are separated into vertical ones (wV) and horizontal ones (wH), each being used to perform a 1D convolution:

```
nbhV ▷ transpose ▷ map (dot wV)
     ▷ slide 3 1 ▷ map (dot wH)
```

The strategy pushSeparation(separateConvKernel(weights2d, wV, wH)) realises this transformation (Listing 4.13). Given explicit weights, the rewrite rule separateConvKernel encodes the decomposition on the dot product. Then, the pushSeparation strategy uses generic rewrite rules to "push" the dot product decomposition across the surrounding dimensions. separateConvolutions in Listing 4.12 uses these components to separate the sobel and sum convolutions of the Harris operator.

Register Rotation   As seen in Section 3.4, register rotation is a storage folding optimisation, just like circular buffering. In the bottom of Figure 4.5, convolution separation is combined with register rotation and vectorisation: vectors of vertical reductions are rotated while computing vectors of horizontal reductions.

If we start from a convolution that was separated as described above:

```
map (dot wV) ▷ slide 3 1 ▷ map (dot wH)
```

The **rotateValues** pattern from Section 3.4 may be introduced in stead of the second **slide**. As the rotation produces a sequential stream, we must use **iterateStream** afterwards:

```
map (dot wV) ▷ rotateValues private 3 (λx. x) ▷ iterateStream (dot wH)
```

Given an input array, **rotateValues** returns an array of sliding windows: the last m values that have been stored in registers. Values are rotated while the array is read sequentially.

The Elevate strategy `rotateValuesAndConsume` shown in Listing 4.14 performs this program transformation. A similar strategy is used in Listing 4.12 for the Harris operator.

```
1  strategy rotateValuesAndConsume = topDown(useRotateValues(private));
2                                    topDown(useIterateStream)
3
4  rule useRotateValues(a: addr) = slide m 1 ↦ rotateValues a m (λx. x)
```

Listing 4.14: Strategy and rules involved in register rotation.

4.4 Evaluation of Runtime Performance

The previous section discussed how well-known image processing pipeline optimisations are expressed in a composable and extensible way as Elevate rewriting strategies. The expressed optimisations go beyond what both Lift and Halide support.

This section investigates whether we truly achieve competitive performance, and the performance impact of the additional optimisations. We report a systematic runtime performance comparison between the Shine, Halide and Lift compilers, the Halide compiler, the Lift compiler. Additionally, we compare to the popular OpenCV image processing library, providing an additional reference point to validate the benefits of whole-program optimising compilers.

4.4.1 Experimental Setup

Experiments are conducted on two computers with ARM big.LITTLE configuration as these mobile CPUs are often used in image processing applications. We use an Odroid XU4 board with a 4-core Cortex A7 and a 4-core Cortex A15 as well as an Odroid N2 board with a 2-core Cortex A53 and a 4-core Cortex A73.

The clock frequencies are set to 1.5Ghz for the XU4, and 1.8Ghz for the N2. Our compiler implementation generates OpenCL kernels that are executed using the POCL [172] open source implementation of OpenCL that is built on top of LLVM. We use POCL 1.3 with LLVM 8 on the XU4 and POCL 1.5 with LLVM 10 on the N2.

The OpenCL kernels generated with Shine (Appendix A) are compared against OpenCV, the OpenCL kernels generated with the Lift implementation from [16], and the binaries generated by Halide (commit c2b6da2 `https://tinyurl.com/rr7awsr`). We use OpenCV 4.3 with NEON vector support enabled. For Shine we use the two Elevate strategies discussed in Section 4.3, resuling in two different versions called cbuf and cbuf+rrot. For Halide we use the reference optimised schedule from Listing 4.7. Neither the Shine-generated OpenCL code nor the Halide schedule is specialised for each individual processor, but the final assembly will be respectively specialised by the OpenCL implementation and the Halide compiler.
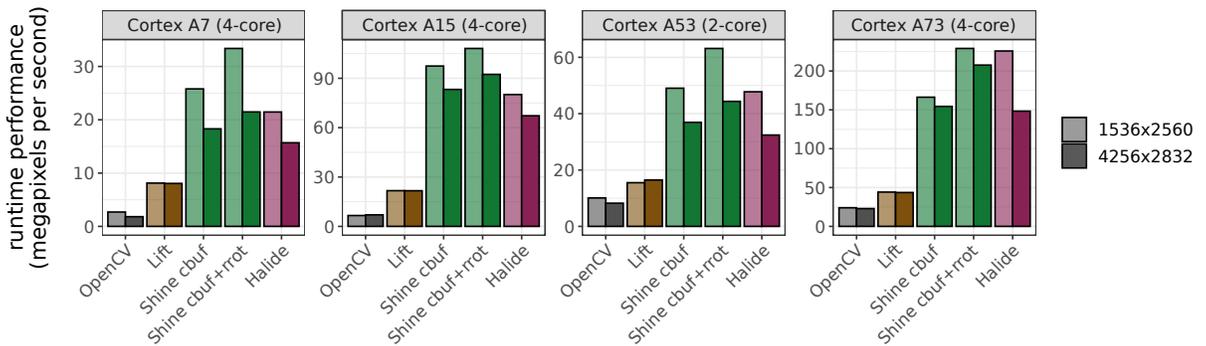
Figure 4.7: Runtime performance of the Harris corner detection for 4 processors (top legend), 5 implementations (bottom legend) and 2 image resolutions (right legend).

We report the median runtime of 30 executions, which gives reasonably stable results. To measure the runtime of the OpenCL kernels, we use the OpenCL profiling API. For Halide we use C++'s `std::chrono` clocks as it is done with other benchmarks in the Halide repository.

Two input images are used, one with a resolution of 1536×2560 pixels, and one of 4256×2832 pixels. The first one is taken from the Halide repository, and was shown in Figure 4.3. The second image is taken from the PolyMage repository.[4] We verify that the outputs of the different Harris operator implementations are consistent by computing the Mean-Squared Error and PSNR (Peak Signal-to-Noise Ratio) with the reference output from Halide. The recorded PSNR is always above 170 decibels, indicating a very strong similarity.

4.4.2   Performance Results

Figure 4.7 shows the runtime performance results. All 3 compilers – Lift, Shine and Halide – outperform OpenCV on all 4 processors and 2 images although OpenCV describes itself as a highly optimised library. Shine cbuf+rrot outperforms OpenCV by up to $16\times$ with a geomean of $9.48\times$. This highlights the performance benefits brought by whole-program optimisations.

Shine cbuf+rrot outperforms Lift by up to $4.5\times$ with a geomean speedup of $3.87\times$. This is expected because prior Lift work focuses on individual stencil computations and lacks optimisations for image processing pipelines: notably operator fusion and circular buffering.

Shine cbuf is on par with the Halide reference with a geomean speedup of $1.02\times$. This demonstrate that Shine's domain-extensible design is capable of achieving the same performance as Halide, a highly optimised domain-specifc compiler. While the applied coarse-grain optimisations are the same, small differences in the generated code remain and the resulting performance depends on fine-grain code generation details down to assembly which are out of the scope of this chapter. At worst, Shine cbuf achieves $0.7\times$ Halide's performance on the A73 for the 1536×2560 image. At best, Shine cbuf achieves $1.2\times$ Halide's performance on the A15 for the 4256×2832 image.

---

[4] https://bitbucket.org/udayb/polymage/src/446f75628c2272651ea002160f50d95f7fbf4b93/images/venice_wikimedia.jpg

With convolution separation and register rotation, Shine cbuf+rrot performs much better than without with a geomean speedup of $1.24\times$ over Shine cbuf and of $1.27\times$ over Halide. At best, Shine cbuf+rrot achieves $1.4\times$ Halide's performance on A7 for the 1536×2560 image. This shows that register rotation is an optimisation worth considering even though it was not implemented in Halide at the time of writing. Moreover, the Shine cbuf+rrot results demonstrate that a domain-extensible compiler can outperform even a state-of-the-art domain-specific compiler like Halide. This is achieved by using extensibility to add optimisations that are not built into existing domain-specific compilers.

4.4.3   Artifact

The source code used to produce the performance results from this chapter is available as a peer-reviewed artifact, publicly available on GitHub.[5]

Artifact Evaluation   The artifact supplements the paper *Towards a Domain-Extensible Compiler: Optimizing an Image Processing Pipeline on Mobile CPUs* [1], presented at the *International Symposium on Code Generation and Optimization* (CGO) in 2021. The artifact was evaluated and received the following ACM badges: Artifacts Available, Artifacts Evaluated, and Results Reproduced. The main goals for artifact evaluation was to use the provided Shine compiler to regenerate the OpenCL kernels used in the experimental evaluation and to reproduce the performance results seen in Figure 4.1 and Figure 4.7. For the same processors (or similar enough), we expect the results to show similar performance trends as observed in Section 4.4.2.

Contents   The artifact contains the Rise, Shine and Elevate Scala implementations, the Halide compiler, the Lift-generated OpenCL kernels, as well as the benchmarking and plotting programs. The two input images are provided, and the expected workflow output is CSV and PDF files corresponding to Figure 4.1 and Figure 4.7.

Requirements   We recommend using an X86 Linux for the host with at least 2GB of disk space available; and Linux targets with OpenCL support and at least 20MB of disk space available (dependencies excluded). The software dependencies are listed in the README and we provide an Ubuntu Focal Fossa (20.04 LTS) Dockerfile for convenience.

Reproducing the results reported in Figure 4.1 and Figure 4.7 requires access to ARM Cortex A7, A15, A53 and A73 processors (we used Odroid XU4 and Odroid N2 boards). Other OpenCL-enabled processors can be used, but may have different performance behavior. The benchmarks can be run on different processors by writing a small YAML configuration file as long as the benchmark dependencies are available.

---

[5]`https://github.com/rise-lang/2021-CGO-artifact`

Workflow   Preparing and completing the artifact workflow should take between 2 hours and 1 day, approximately:

1. Install host dependencies
2. Clone repository on the host
3. Generate binaries and OpenCL kernels for each target
4. Configure each target
5. Run benchmarks over ssh for each target
6. Plot figures

More detailed instructions are provided in the README.

4.5 Conclusion

This chapter demonstrates how extensibility and controllability are exploited in Shine to generate high-performance code. Using the Harris corner detection as a case study, our runtime results on four mobile ARM multi-core CPUs and two different image resolutions show that Shine outperforms OpenCV library code by up to $16\times$ (geomean of $9.48\times$), outperforms the similarly designed Lift compiler by up to $4.5\times$ (geomean of $3.87\times$) and performs up to $1.4\times$ (geomean of $1.27\times$) better than the domain-specific compiler Halide (Figure 4.7).

Optimisations are controlled using Elevate rewriting strategies. Elevate allows us to reproduce the effect of an optimised Halide schedule applying operator fusion, multi-threading, vectorisation and circular buffering optimisations. Further, Elevate allows us to go beyond what is possible with Halide schedules, incorporating additional convolution separation and register rotation optimisations. Shine is extended with new specialised optimisations by adding Rise patterns (Chapter 3), rewrite rules and Elevate rewriting strategies.

All of the presented optimisations are well-known, and have been studied on the Harris operator before in [23], where they are performed manually. In general, stencil optimisations such as overlapped tiling [173, 174, 175, 176] are well-studied and useful beyond image processing. Similarly, we believe that the presented patterns and rewrite rules are re-usable beyond image processing and across hardware targets, although this remains to be demonstrated.

Automated heuristics and explorations are not always desirable or even feasible as they lack user control, may result in poor performance, and may be too time consuming. This justifies controlling optimisations with rewriting strategies. However, Elevate strategies are not easy to write, can be over-detailed and program-specific. The strategies defined to apply the 6 optimisations consist of more than 600 lines of code defining 57 helper strategies. To perform all 6 optimisations, thousands of rewrite steps are applied. The strategies are specialised to our Harris case study and would be challenging to generalise and reuse across image processing pipelines. The next chapter discusses a novel practical tradeoff between precise control of optimisations (as in this chapter) and full automation of optimisations (as in Lift).

Every sketch goes through a rewrite stage where a group of writers sits around a table and pitches more jokes and ideas for the piece.

— Vanessa Bayer

Chapter 5

Sketch-Guided Equality Saturation

The previous chapter combined the control of rewriting strategies with the domain-extensibility of SHINE to achieve 6 case study optimisations, and hence generate faster code than Halide schedules that do not support 2 of the optimisations. However, rewriting strategies are not easy to write. Deciding when to apply which rewrite is hard: the so-called *phase ordering problem* (Section 2.3).

Equality saturation [85, 86] is an automated optimisation technique mitigating the phase ordering problem by automatically exploring many possible ways to apply rewrites. It relies on an efficient representation of many equivalent programs in an e-graph data structure. Unfortunately, automatically discovering the RISE optimisations applied using rewriting strategies as in [3] or Chapter 4 using equality saturation is prohibitively expensive (Section 5.6).

This chapter proposes a practical tradeoff between the control of ELEVATE strategies, and the automation of techniques such as equality saturation. Section 5.1 provides background on equality saturation. Section 5.2 explains our motivations for semi-automatic optimisation. Section 5.3 introduces *sketch-guided equality saturation*, a semi-automatic technique that allows performance engineers to guide rewriting by providing *sketches*: program patterns that leave details unspecified. Section 5.4 explains inefficiencies in the naive encoding of the lambda calculus for equality saturation [86], and explores new techniques to efficiently encode a polymorphically typed lambda calculus such as RISE. Two systematic evaluations of RISE applications are then conducted. Section 5.5 demonstrates that our lambda calculus encoding reduces the runtime and memory consumption of equality saturation by orders of magnitude when optimising a binomial filter. Section 5.6 evaluates sketch-guided equality saturation by reproducing seven realistic optimisations of matrix multiplication from [3]. Even with efficient lambda calculus encoding, unguided equality saturation can locate only the two simplest optimisations, the remaining five are undiscovered even with an hour of compilation time and 60GB of RAM. By specifying three or fewer sketch guides, all seven optimisations are found in seconds of compilation time, using under 1GB of RAM, and generating high performance code. Section 5.7 concludes.

5.1   Background on Equality Saturation

Equality saturation [85, 86] is a technique for efficiently implementing rewrite-driven compiler optimisations without committing to a single rewrite choice. Many successful applications of equality saturation sparked from the recent egg library [86]: optimising linear algebra [177], shrinking 3D CAD (Computer-Aided Design) models [178], optimising deep learning programs [179, 180], vectorising digital signal processing code [181], inferring rewrite rules [182].

   We now demonstrate how equality saturation mitigates the phase ordering problem with a rewriting example where greedily reducing a cost function is not sufficient to find the optimal program.

5.1.1   Getting Stuck in a Local Optimum with Greedy Rewriting

Rewriting is often used to fuse operators and avoid writing intermediate results to memory, for example:

$$(map\ (map\ f)) \circ (transpose \circ (map\ (map\ g))) \qquad\qquad (A)$$

$$\longmapsto^{*}$$

$$(map\ (map\ (f \circ g))) \circ transpose \qquad\qquad (B)$$

The initial term (A) applies function g to each element of a two-dimensional matrix (using two nested maps), transposes the result, and then applies function f to each element. The optimised term (B) avoids storing an intermediate matrix in memory and transposes the input before applying g and f to each element. Applying the following rewrite rules in the correct order is sufficient to perform this optimisation:

$$transpose \circ map\ (map\ a) \longleftrightarrow map\ (map\ a) \circ transpose \qquad\qquad (5.1)$$

$$a \circ (b \circ c) \longleftrightarrow (a \circ b) \circ c \qquad\qquad (5.2)$$

$$map\ a \circ map\ b \longleftrightarrow map\ (a \circ b) \qquad\qquad (5.3)$$

   Rule (5.1) states that transposing a two-dimensional array before or after applying a function to the elements is equivalent. Rule (5.2) states that function composition is associative. Finally, rule (5.3) is the rewrite rule for map fusion. In this example, minimising the term size results in maximising fused maps and is, therefore, a good cost model.

   If we greedily apply rewrite rules that lower term size, we will only apply rule (5.3) as this is the only rule that reduces term size. However, rule (5.3) cannot be directly applied to term (A): it is a local optimum. The only way to reduce term size further is to first apply the other rewrite rules, which may or may not pay off depending on future rewrites.
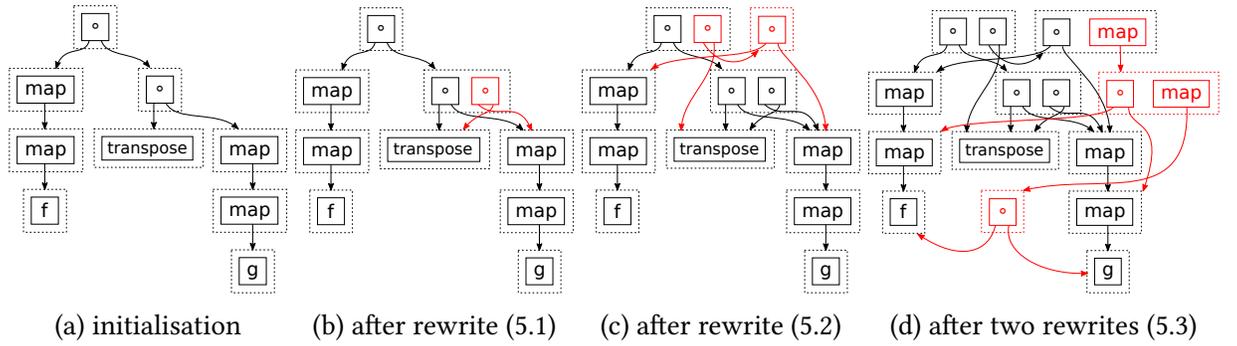
(a) initialisation (b) after rewrite (5.1) (c) after rewrite (5.2) (d) after two rewrites (5.3)

Figure 5.1: Growing an e-graph for the term $(map \ (map \ f)) \circ (transpose \circ (map \ (map \ g)))$. An e-graph is a set of e-classes themselves containing equivalent e-nodes. The dashed boxes are e-classes, and the solid boxes are e-nodes. New e-nodes and e-classes are shown in red.

5.1.2 Exploring Past the Local Optimum with Equality Saturation

The first phase of equality saturation consists of exploring many possible rewrites. An e-graph (equivalence graph) is used to efficiently represent and rewrite a set of equivalent programs, intuitively:

- An *e-graph* is a set of equivalence classes (e-classes) representing all terms represented by its e-classes.

- An *e-class* is a set of equivalent nodes (e-nodes) representing all terms represented by its e-nodes.

- An *e-node* $F(e_1, .., e_n)$ is an n-ary function symbol (F) from the term language, associated with child e-classes (e_i). It represents all terms $F(t_1, .., t_n)$ where each term t_i is represented by the e-class e_i. In this example, symbols are map, $transpose$, $\circ$, f, g.

To start the exploration phase, an e-graph representing the initial term (A) is constructed (Figure 5.1a). The e-graph is then iteratively grown by applying rewrite rules non-destructively (Figures 5.1b to 5.1d). On each equality saturation iteration, all possible rewrites are explored in a breadth-first manner. This contrasts from standard term rewriting where a single possible rewrite is selected in a depth-first manner, requiring careful ordering of rewrite rule applications. For the sake of simplicity, we only apply a handful of rewrite rules per iteration in Figure 5.1. Rewrite rule applications are considered even if they do not lower cost, which avoids getting stuck in local optima. When applying a rewrite rule, the equality between its matched left-hand side and its instantiated right-hand side is recorded in the e-graph. This also contrasts from standard term rewriting that destructively replaces the matched left-hand side with the instantiated right-hand side, producing a new term from the initial one.

The exploration phase terminates, and rewrite rules stop being applied, when a fixed point is reached (saturation), or when another stopping criteria is reached (e.g. timeout or achieved goal). If saturation is reached, it means that all possible rewrites have been explored.
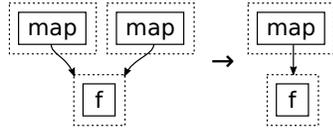
Figure 5.3: Smallest term size computed during extraction, shown for each e-class in the top-right corner in blue. Where it differs from its e-class value, the smallest term size of e-nodes is also shown. The e-class and e-nodes of the smallest term (B) are shown in green.

Figure 5.2: The congruence invariant simplifies the e-graph on the left by merging two identical e-nodes for $map\ f$ into a single e-node as shown on the right.

Crucially for efficiency, an e-graph is far more compact than a naive set of terms, as equivalent sub-terms are shared. E-graphs can represent exponentially many terms in polynomial space, and even infinitely many terms in the presence of cycles [86]. To maximise sharing, a *congruence invariant* is maintained: intuitively identical e-nodes should not be in different e-classes (Figure 5.2). Later we will see that even extensive sharing does not necessarily prevent e-graph sizes from exploding.

5.1.3   Extracting a Global Optimum with Equality Saturation

The second and final phase of equality saturation consists of extracting the best term from the e-graph according to a cost function, e.g. the one with smallest term size. The extracted term is a global optimum if saturation was reached in the exploration phase.

The *extraction* procedure can be a relatively simple bottom-up e-graph traversal if the cost function is local [183]. Non-local cost functions require more complex extraction procedures [177, 184].

A *local cost function* c can be defined as a function of a term language symbol F and the costs of its children, i.e. c has signature $c(F(k_1 : K, .., k_n : K)) : K$ with costs of type K. Term size is a local cost function:

$$\text{termSize}(F(k_1, .., k_n)) = 1 + \sum_i k_i$$

The term sizes computed during a bottom-up extraction procedure are shown in Figure 5.3. The figure reveals that there is a smaller term (B) of size 7 in the same e-class as the original term (A) of size 9 (top left in Figure 5.3). Therefore, (B) is extracted as the optimised term.

5.2 Motivation for Semi-Automatic Optimisation

This section motivates the need for a semi-automatic optimisation technique, expanding on Section 2.3. Controlling thousands of rewrite rule applications with ELEVATE rewriting strategies as in [3] and Chapter 4 is tedious. Instead, we would like to optimise RISE programs *without having to control individual rewrite rule applications* (Section 5.2.1).

To automate rewrite rule application, globally exploring all possible rewrites with equality saturation is unfeasible, as the search space is too big. Novel ways to scale equality saturation are required, and semi-automation is an appealing trade-off (Section 5.2.2).

```
1  map (λaRow.                 |  for m:
2    map (λbCol.               |    for n:
3      dot aRow bCol)          |      for k:
4      (transpose b)) a        |        ...
```

$$\longmapsto^*$$

```
1  join (map (map join) (map transpose
2    map                           |  for m / 32:
3      (map λx2.                   |    for n / 32:
4        reduceSeq (λx3. λx4.      |      for k / 4:
5          reduceSeq λx5. λx6.     |        for 4:
6          map                     |          for 32:
7            (map (λx7.            |            for 32:
8            (fst x7) + (fst (snd x7)) ×
9            (snd (snd x7)))
10           (map (λx7. zip (fst x7) (snd x7))
11           (zip x5 x6)))
12         (transpose (map transpose
13           (snd (unzip (map unzip map (λx5.
14           zip (fst x5) (snd x5))
15           (zip x3 x4)))))))
16         (generate (λx3. generate (λx4. 0)))
17         transpose (map transpose x2))
18     (map (map (map (map (split 4))))
19       (map transpose
20         (map (map (λx2. map (map (zip x2)
21         (split 32 (transpose b)))))
22         split 32 a)))))
```

Figure 5.4: Applying a blocking optimisation to matrix multiplication via rewriting in RISE. In the initial program (top), a dot product is computed between each row of a (aRow) and column of b (bCol). In the final program (bottom), a blocking optimisation has been applied. Loops characteristic of the optimisation are shown on the right of the **|** symbols, and are not part of the RISE program. The remaining program regions reshape input arrays, initialise arrays, compute with scalars, and reshape output arrays.

```
1  strategy blocking =
2    baseline;
3    tileMaps(32,32)   @ outermost(mapNest(2));;
4    fissionReduceMap  @ outermost(appliedReduce);;
5    tileReduce(4)     @ innermost(appliedReduce);;
6    reorder(List(1,2,5,6,3,4))
```

Listing 5.1: ELEVATE strategy for matrix multiplication blocking [3].

5.2.1   Controlling Many Rewrite Steps with Strategies is Tedious

Section 4.1.1 introduced ELEVATE strategies with a trivial dot product example. A simple strategy was defined to apply a single rewrite rule using a top-down AST traversal:

```
strategy lowerDot = topDown(reduceMapFusion)
```

To achieve more complex RISE optimisations leading to high performance, significantly more complex ELEVATE strategies are defined. The strategies defined in Chapter 4 to apply the 6 Harris corner detection optimisations consist of more than 600 lines of code defining 57 helper strategies. Similarly, 36 rewriting strategies were defined in 200 lines of code to apply 7 matrix multiplication optimisations in [3]. In both case studies, thousands of rewrite rules are applied by the strategies to achieve the optimisations (including backtracked rule applications). The ELEVATE strategy authors estimate having spent between two and five person-weeks developing the strategies for each case study.

We now focus on the limitations of a particular ELEVATE strategy defined in [3] to apply a blocking optimisation to the RISE matrix multiplication (Listing 5.1). Blocking (or tiling) is a common loop optimisation that improves data locality, and therefore memory usage [185, 186]. The ELEVATE strategy has for effect to rewrite the RISE program at the top of Figure 5.4 to the one at the bottom. Reading the strategy from top to bottom, a baseline strategy is applied, rewriting the program into a particular normal form as well as applying a single reduceMapFusion rule. The tileMaps strategy is applied to the outermost nest of 2 maps, creating 32×32 blocks (or tiles). Note that while ; is the straightforward sequential combinator, ;; is a sequential combinator that also enforces a particular normal form before continuing. The fissionReduceMap rewrite rule is applied at the outermost reduction pattern. The tileReduce rewrite rule is applied to the innermost reduction pattern, creating another block of 4 Line 5. Finally, the reorder strategy is applied, reordering nested map and reduce patterns to create 4×32×32 blocks and hence produce the loop nest at the bottom of Figure 5.4.

ELEVATE enables the development of abstractions that help write concise strategies (Section 4.1.2), such as tileMaps, reorder, @, outermost and innermost in this example. Unfortunately, these abstractions are often program specific and complex to implement. Table 5.1 shows how many lines of code and internal strategies are defined for tileMaps and reorder, as well as their limited applicability. Both strategies are defined recursively.

Strategy	LOC	IS	Limited Applicability
`tileMaps`	32	6	only works for perfect **map** nests, e.g. **map** f, **map** (**map** f), **map** (**map** (**map** f)), etc.
`reorder`	43	8	not capable of reordering arbitrary loop nests; depends on recursively lifting a **reduce** pattern outside of a **map** pattern which is hard-coded for specific cases.

Table 5.1: Main strategies backing Listing 5.1, their lines of code (LOC), internal strategies (IS), and limited applicability.

Overall, developing generic ELEVATE strategies is difficult because small syntactic differences in RISE programs require adjustments to the rewrite sequence. While ELEVATE empowers performance engineers to manually control the rewrite process, it also delegates them the problem of ordering thousands of rewrites in order to achieve their goals (9K rewrite rules are applied when executing Listing 5.1). To mitigate the phase ordering problem, automation techniques such as equality saturation are compelling.

5.2.2 Semi-Automating Many Rewrite Steps with Equality Saturation

Equality saturation (Section 5.1) has many successful applications, but its applicability remains limited by scaling issues. As the e-graph grows, iterations become slower and require more memory. The growth rate is aggravated by some combinations of useful rewrite rules, such as associativity and commutativity, that generate an exponential number of equivalent permutations [177, 178, 86]. This makes exploring long rewrite sequences inherently hard, as the breadth-first exploration of all possible rewrites leads to an exponential increase of the e-graph size, despite its compact representation of terms. In many applications, expecting to reach saturation, and therefore to find optimal solutions according to the cost function, is unrealistic. We encounter these issues when attempting to optimise matrix multiplication in RISE using equality saturation, as we discuss in Section 5.6.

One way to reduce e-graph growth is to limit the number of rules applied [177, 86], but this risks not finding optimisations that require an omitted rule. An alternative is to use an external solver to speculatively add equivalences [178], but this requires the identification of sub-tasks that can benefit from being delegated. It is also possible to trade-off between the exploitation of greedy rewriting and the exploration of equality saturation [187], but this requires a good enough heuristic cost function to make local decisions.

This chapter proposes another approach, sketch-guiding, that factors unfeasible equality saturations into a sequence of smaller, and hence feasible, equality saturations (Section 5.3). The appeal of sketch-guiding is that it still allows performance engineers to control the rewrite process, but saves them from ordering thousands of rewrites. Additionally, an efficient encoding of the λ calculus is introduced to reduce the sizes of the e-graphs produced when optimising RISE programs (Section 5.4).

5.3   Sketch-Guided Equality Saturation

This section introduces *sketch-guided equality saturation*, a novel semi-automated process offering a trade-off between manually defined rewriting strategies and automated equality saturation. The performance engineer guides multiple equality saturations by specifying a sequence of sketches that describe how a program should evolve during optimisation. In this process, the intent is to break down complex optimisations into a sequence of simpler rewrites, each sufficiently simple to be found by equality saturation. Intuitively, a rewrite is *complex* if achieving it requires applying many inter-dependent rewrite rules. While rewriting strategies require ordering many rewrite rules, sketch-guided equality saturation enables ordering few sketches instead. For example, instead of specifying how to apply thousands of rules to achieve matrix multiplication blocking, specifying just two sketches suffices.

5.3.1   The Intuition for Sketches

When designing optimisations, performance engineers often visualise the desired *shape* of the optimised program. The effect of rewriting strategies or schedules is often explained with program snippets [1, 137, 10, 28, 188, 26, 29]. Indeed, this is how we have explained the loop nest blocking optimisation in Figure 5.4, as well as many optimisations in Chapter 4.

Our *key new insight* is that explanatory program snippets can be formalised as sketches and used to guide an optimisation search. *Sketches* are program patterns that capture program shape intuitions. The guided optimisation search is still based on semantic preserving rules, allowing to provide sketches that leave details unspecified without sacrificing correctness.

We illustrate by presenting sketches for matrix multiplication blocking. Listing 5.2 shows a sketch for the unoptimised *baseline* goal, specifying the desired program structure as two nested **map** patterns and a nested **reduce**, with innermost addition and multiplication operations. The formal definitions of **containsMap**, **containsReduceSeq** and **containsAddMul** are in Section 5.3.2. The comments on the right show the equivalent nested **for** loops, using the same intuition as in Figure 5.4. Listing 5.3 shows a sketch for the *blocking* goal, corresponding to the optimised program where the "loop nests" have been split and reordered to chunk the iteration space into blocks of $4 \times 32 \times 32$, processed by the three innermost **for** loops.

Searching for the *blocking* goal can be made more tractable by specifying intermediate *sketch guides*, and Listing 5.4 is an example. This sketch guide describes a program shape where the **map** and **reduce** patterns have been split but not yet reordered.

ELEVATE strategies, as in Listing 5.1, are detailed *imperative* specifications of how to rewrite the program. In contrast, a sketch is a *declarative* specification of the optimisation goal, and equality saturation is used to search for a sequence of rewrites to achieve that goal. A sequence of sketches (e.g., Listing 5.4 followed by Listing 5.3) may be used to achieve a desired optimisation when the equality saturation search with a single sketch as a goal does not succeed.

```
1   containsMap(m,              |   for m:
2    containsMap(n,             |    for n:
3     containsReduceSeq(k,      |     for k:
4      containsAddMul)))        |        .. + .. × ..
```

Listing 5.2: A sketch for the *baseline* matrix multiplication goal

```
1   containsMap(m / 32,            |   for m / 32:
2    containsMap(n / 32,           |    for n / 32:
3     containsReduceSeq(k / 4,     |     for k / 4:
4      containsReduceSeq(4,        |      for 4:
5       containsMap(32,            |       for 32:
6        containsMap(32,           |        for 32:
7         containsAddMul))))))     |           .. + .. × ..
```

Listing 5.3: A sketch for the *blocking* matrix multiplication goal

```
1   containsMap(m / 32,            |   for m / 32:
2    containsMap(32,               |    for 32:
3     containsMap(n / 32,          |     for n / 32:
4      containsMap(32,             |      for 32:
5       containsReduceSeq(k / 4,   |       for k / 4:
6        containsReduceSeq(4,      |        for 4:
7         containsAddMul))))))     |           .. + .. × ..
```

Listing 5.4: A sketch guide specifying how to split loops for *blocking*

5.3.2 Defining Sketches

Sketches are specified in a SKETCHBASIC language with just four constructors. The syntax of SKETCHBASIC and the set of terms that the constructors represent are defined in Figure 5.5. A sketch s represents a set of terms $\mathcal{R}[\![s]\!]$, such that $\mathcal{R}[\![s]\!] \subset T$ where T denotes all terms in the language we rewrite. We say that any $t \in \mathcal{R}[\![s]\!]$ satisfies sketch s.

The $?$ sketch is the least precise, representing all terms in the language. The $F(s_1, .., s_n)$ sketch represents all terms that match a specific n-ary function symbol F from the term language, and whose n children satisfy sketches s_i. The *contains*(s) sketch represents all terms containing a term that satisfies sketch s: the greatest solution to the recursive $\mathcal{R}[\![contains(s)]\!]$ equation. Finally, the $s_1 \vee s_2$ sketch represents terms satisfying either s_1 or s_2.

When rewriting terms in a typed language, sketches may be annotated with a type sketch $(s :: s_t)$ constraining the type of terms. If $\mathcal{R}[\![s_t]\!]$ denotes the set of terms satisfying the type sketch s_t, then $\mathcal{R}[\![s :: s_t]\!] = \mathcal{R}[\![s]\!] \cap \mathcal{R}[\![s_t]\!]$. The grammar of type sketches depends on the language we rewrite. We elide type sketches from our definition of SKETCHBASIC as they are just a convenience to define sketches over the language T', the explicitly typed version of T where types would be embedded in terms.

$$S ::= \; ? \mid F(S, .., S) \mid contains(S) \mid S \vee S$$

$$\mathcal{R}[\![?]\!] = T = \{F(t_1, .., t_n)\}$$
$$\mathcal{R}[\![F(s_1, .., s_n)]\!] = \{F(t_1, .., t_n) \mid t_i \in \mathcal{R}[\![s_i]\!]\}$$
$$\mathcal{R}[\![contains(s)]\!] = \mathcal{R}[\![s]\!] \cup \{F(t_1, .., t_n) \mid \exists t_i \in \mathcal{R}[\![contains(s)]\!]\}$$
$$\mathcal{R}[\![s_1 \vee s_2]\!] = \mathcal{R}[\![s_1]\!] \cup \mathcal{R}[\![s_2]\!]$$

Figure 5.5: Grammar of SKETCHBASIC (top) and terms represented by SKETCHBASIC (bottom).

```
1  def containsMap(n: NatSketch, f: Sketch): Sketch =
2    contains((map :: ?t → n.?dt → ?t) f)
3  def containsReduceSeq(n: NatSketch, f: Sketch): Sketch =
4    contains((reduceSeq :: ?t → ?t → n.?dt → ?t) f)
5  def containsAddMul: Sketch =
6    contains(? + contains(×))
```

Listing 5.5: Some sketch abstractions used in this chapter.

Sketch Abstractions   Sketch abstractions are defined by combining generic constructs from SKETCHBASIC with type annotations from the term language. To illustrate, Listing 5.5 shows some sketch abstractions used for our RISE matrix multiplication case study. Recall that $\rightarrow$ is a function type, and n.dt an array type of n elements of domain type dt. The type annotations restrict the iteration domains of **map** and **reduceSeq**: the input arrays must have type n.?dt, and therefore such patterns will iterate over n elements.

Sketch Precision   Writing a useful sketch to guide an optimisation search requires striking a balance between being too precise and too vague.

An overly precise sketch may exclude valid optimised programs with a slightly different structure. Replacing **containsAddMul** with **contains**(λx. x + (**fst** x) × (**snd** x)) in the blocking sketch (Listing 5.3) would prevent the search from finding any suitable program in the experiment of Section 5.6.

An overly vague sketch may lead to finding undesirable programs. Removing the array sizes from the splitting sketch guide (Listing 5.4) would result in finding a program with the following undesired loop nest in the experiment of Section 5.6:

```
1  for m:
2   for n / 32 / 32:
3    for 32:
4     for 32:
5      for k / 4:
6       for 4:
7        ..
```
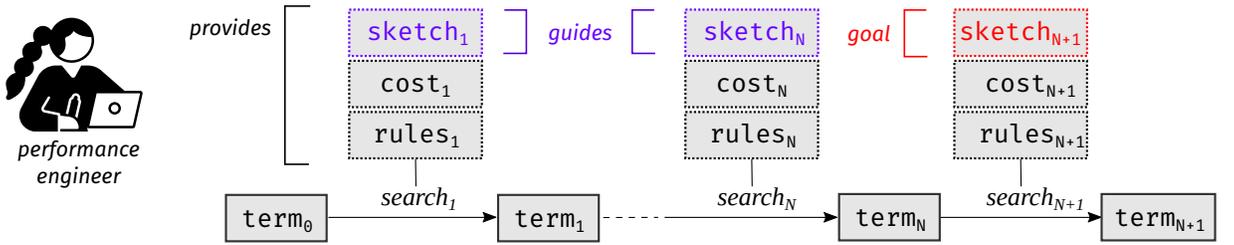
Figure 5.6: Sketch-Guided Equality Saturation. The performance engineer provides N intermediate sketch guides and 1 final goal sketch. Starting from the input term, $N+1$ consecutive equality saturation searches attempt to find a term satisfying each sketch, using the associated cost models and sets of rules (Figure 5.7).
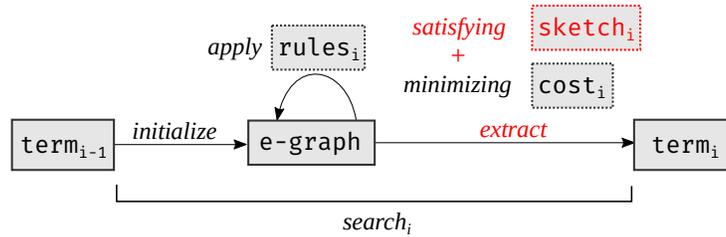

Figure 5.7: Sketch-Satisfying Equality Saturation implementing each $search_i$. Changes made to standard equality saturation are highlighted in red.

This balance also interacts with the set of rewrite rules used, since programs that may be found by the search are $\mathcal{R}[\![s]\!] \cap \mathcal{E}_{rules}[\![t]\!]$ where $\mathcal{E}_{rules}[\![t]\!]$ represents the set of terms that can be discovered to be equivalent to the initial term t according to the given $rules$. This means that using a more restricted set of rules generally enables specifying less precise sketches. How to best select rules and write effective sketches are topics for future work (Chapter 6).

5.3.3   Sketch-Guided Equality Saturation

While sketches have previously been used as a starting point for program synthesis [189], our work uses sketches in a novel way, as intermediate goals (guides) for program optimisation.

The process of guiding equality saturation with a sequence of sketches is illustrated in Figure 5.6. The performance engineer provides a sequence of N intermediate sketch guides and a final goal sketch: $sketch_1, .., sketch_{N+1}$. Successive equality saturation searches are performed to find equivalent terms that satisfy each sketch in the sequence. As each sketch may be satisfied by many terms, the performance engineer must also provide a sequence of cost models $cost_1, .., cost_{N+1}$ to select the term to be used as the starting point for the next search. Sets of rewrite rules ($rules_1, .., rules_{N+1}$) are provided to grow the e-graph in each search. The cost model and set of rules may be identical for many or all of the searches, but we show in Section 5.6 how restricting the set of rules can reduce search runtime. Figure 5.7 shows how each search is performed, and how these searches differ from standard equality saturation.

```
1  def SGES(term, params): Option[Term] =
2    if params.isEmpty
3    then Some(term)
4    else search(term, params.head)
5          .and_then(λt. SGES(t, params.tail))
6
7  def search(term, param): Option[Term] =
8    (sketch, cost, rules) = param
9    g = create empty e-graph
10   normTerm = normalize(term)
11     using a configurable normal form
12   e = g.add(normTerm)
13   grow g using rules until found(g, e, sketch)
14   if found(g, e, sketch) then
15     Some(extract(g, e, sketch, cost))
16   else
17     None
```

Listing 5.6: Sketch-Guided Equality Saturation Algorithm

The pseudo-code for the sketch-guided equality saturation algorithm is shown in Listing 5.6. The entry point is the SGES function (line 1) that takes a term and a sequence of sketches, cost models and rewrite rules (params). It repeatedly searches (line 4) for each sketch using the associated cost model and rewrite rules, and outputs a term if found, otherwise nothing. At the beginning of each search, we may normalise the input term (line 10) to apply destructive rewrites that are always desired before starting a purely additive equality saturation. For our matrix multiplication running example we use a $\beta\eta$ normal form. The extract function (line 15) is used to extract a term from the e-graph that satisfies the specified sketch while minimising the specified cost model, and we describe it in the next subsection. In this paper, we terminate equality saturation as soon as a program satisfying the current sketch is found, whether or not the cost could be further improved by a longer search. This is because we give more value to satisfying the sketch than to minimising the cost. Other applications of sketch-guided equality saturation could use different stopping criteria. The found function (line 14) is used to stop growing the e-graph by checking whether extract would succeed.

Note that sketch-guided equality saturation does not provide any guarantee of optimality or completeness of the search. Without sketch-guidance, reaching saturation is unfeasible in many use cases (Section 5.2.2), and terminating equality saturation before saturation also provides no such guarantees.

Not finding a term satisfying a given sketch in the given resource budget may be for one of two reasons. The search may be too difficult, and the performance engineer may fix the issue by providing additional or better sketch guides. It may be impossible to construct an equivalent term satisfying the sketch with the given set of rules, and the performance engineer may fix the issue by fixing an error in the sketch or providing missing rewrite rules.

5.3.4 Sketch-Satisfying Extraction

To `extract` the best program that satisfies a SKETCHBASIC sketch s from an e-graph g we define a helper function $E(c, s, g)$, where c is a cost function that must be monotonic and local. With costs of type K, c is local if it can be defined as a function of a term language symbol F and the cost of its children, i.e. it has signature $c(F(k_1 : K, .., k_n : K)) : K$ The helper E returns a map from e-classes to optional tuple values of type *Option[(K, Term)]*. `extract` uses E to return a term if possible, failing otherwise:

$$\text{extract}(g, e, s, c) = t \quad \text{if} \quad E(c, s, g)[e] = \textit{Some } (\_, t)$$

We write $E(c, s, g)[e]$ for indexing into the map returned by E. E is memoized for efficiency, and recursively defined over the 4 SKETCHBASIC cases as follows.

Case 1: $E(c, ?, g)$. This case is equivalent to extracting the programs minimising c from the e-graph (Section 5.1.3). Such an extraction procedure can be implemented using an e-class analysis [86]. An *e-class analysis* propagates analysis data of type D in a bottom-up fashion, and can be used for extraction when the cost function is local. An e-class analysis is defined by providing two functions: one to *make* the analysis data from an n-ary symbol F combined with the data d_i of its child e-classes; and one to *merge* the analysis data of e-nodes in the same e-class. The domain of the analysis data together with the *merge* operation should form a semilattice.

$$\textit{make}(F(d_1 : D, .., d_n : D)) : D$$

$$\textit{merge}(d_1 : D, d_2 : D) : D$$

We implement $E(c, ?, g)$ as an e-class analysis with data type $D = \textit{Option}[(K, \textit{Term})]$ and the following *make* and *merge* functions. *make* computes the best cost of an e-node and the corresponding best term, based on children analysis data, if available. *merge* returns the analysis data with best cost, if available.

$$\textit{make}(F(d_1, .., d_n)) = \begin{cases} \textit{Some} \left(\begin{array}{c} c(F(k_1, .., k_n)), \\ F(t_1, .., t_n) \end{array} \right) & \forall i.\ d_i = \textit{Some } (k_i, t_i) \\ \textit{None} & \text{otherwise} \end{cases}$$

$$\textit{merge}(d_1, d_2) = \begin{cases} \text{if } k_1 \leq k_2 \text{ then } d_1 \text{ else } d_2 & \forall i.\ d_i = \textit{Some } (k_i, \_) \\ d_i & \exists i.\ d_i = \textit{Some } (k_i, \_) \\ \textit{None} & \text{otherwise} \end{cases}$$

Case 2: $E(c, \boldsymbol{F(s_1, .., s_n)}, g)$. We consider each e-class e containing $F(e_1, .., e_n)$ e-nodes and the terms that should be extracted for each child e-class e_i:

$$E(c, F(s_1, .., s_n), g)[e] = \begin{cases} \textit{Some} \begin{pmatrix} c(F(k_1, .., k_n)), \\ F(t_1, .., t_n) \end{pmatrix} & \forall i.\ E(c, s_i, g)[e_i] = \textit{Some}\ (k_i, t_i) \\ \textit{None} & \text{otherwise} \end{cases}$$

Case 3: $E(c, \boldsymbol{contains(s_2)}, g)$. We use another e-class analysis and initialise the analysis data to $E(c, s_2, g)$ corresponding to the base case where $\mathcal{R}[\![s_2]\!] \subset \mathcal{R}[\![contains(s_2)]\!]$. To *merge* the analysis data, we do the same as for $s = ?$. To *make* the analysis data we consider all terms that would contain terms from s_2 and keep the best by folding them using *merge*:

$$make(F(d_1, .., d_n)) = \textit{foldl merge None}$$

$$\left\{ \textit{Some} \begin{pmatrix} c(F(k_1, .., k_j, .., k_n)), \\ F(t_1, .., t_j, .., t_n) \end{pmatrix} \middle| \begin{array}{c} i \neq j, \\ E(c, ?, g)[e_i] = \textit{Some}\ (k_i, t_i), \\ d_j = \textit{Some}\ (k_j, t_j) \end{array} \right\}$$

Case 4: $E(c, \boldsymbol{s_1 \vee s_2}, g)$. We *merge* the results from s_1 and s_2:

$$E(c, s_1 \vee s_2, g)[e] = \textit{merge}(E(c, s_1, g)[e], E(c, s_2, g)[e])$$

5.4 Efficient Equality Saturation for the Lambda Calculus

The egg library [86] implements a partial evaluator for the lambda calculus using a naive lambda calculus encoding based on explicit substitution. Although conceptually simple, this encoding is not efficient enough to optimise RISE programs (Section 5.5). Glenside [180] proposes *access patterns* to increase efficiency by avoiding the need for binding structures when representing tensor programs. However, using this technique would require re-designing the RISE language, including its patterns and rewrite rules.

This section instead explores how to efficiently encode a polymorphically typed lambda calculus such as RISE, for the purposes of equality saturation. A set of design choices are realised for the RISE language in the new RISEGG implementation that is heavily inspired by the egg library [86]. Our optimised encoding reduces the runtime and memory consumption of equality saturation over lambda terms by orders of magnitude (Section 5.5).

First, applying equality saturation to lambda calculus terms requires encoding them as terms of shape $F(t_1, .., t_n)$. A naive encoding is shown in Table 5.2. Lambda abstraction is encoded as a unary symbol, lambda application as a binary symbol, and variables as constant symbols. Variable names are not modeled directly as terms, but as symbol metadata: 'lam x', 'lam y', 'var x' and 'var y' are all treated as distinct symbols.

λ calculus	F	$t_1, .., t_n$
λx. e	lam x	e
e$_1$ e$_2$	app	e$_1$, e$_2$
x	var x	

Table 5.2: Naive encoding of λ calculus terms as $F(t_1, .., t_n)$ terms for equality saturation.

$$(\lambda x.\ b)\ e \longmapsto b[e/x] \qquad\qquad (\beta\text{-reduction})$$
$$\lambda x.\ f\ x \longmapsto f \qquad\qquad \text{if } x \text{ not free in } f \quad (\eta\text{-reduction})$$
$$map\ f\ (map\ g\ arg) \longmapsto map\ (\lambda x.\ f\ (g\ x))\ arg \qquad\qquad (\text{map-fusion})$$
$$map\ (\lambda x.\ f\ gx) \longmapsto \lambda y.\ map\ f\ (map\ (\lambda x.\ gx)\ y) \quad \text{if } x \text{ not free in } f \quad (\text{map-fission})$$

Figure 5.8: Rise rewrite rules using substitution, name bindings (lambda abstractions), and freshness predicates.

Second, applying equality saturation to lambda calculus terms requires the efficient support of standard operations and rewrites. Figure 5.8 shows the standard rules of β-reduction and η-reduction that use substitution, name bindings and freshness predicates. The other two rules encode map-fusion and map-fission, and are interesting because they introduce new name bindings on their right-hand-side.

The following subsections first discuss how to efficiently implement substitution, name bindings and freshness predicates. Then, they discuss how polymorphic types are added and how external, user-friendly rewrite rules are compiled into internal rewrite rules.

5.4.1 Substitution

The β-reduction rewrite rule requires substituting $b[e/x]$. Standard term substitution cannot be used directly during equality saturation, as the b and e pattern variables are not matched by terms, but by e-classes. A simple way to address this is to use *explicit substitution* as in egg's lambda calculus example [86]. A syntactic constructor is added to represent substitution, with rewrite rules to encode its small-step behavior:

$$(a\ b)[e/v] \longmapsto (a[e/v]\ b[e/v])$$
$$v[e/v] \longmapsto e$$

Unfortunately, explicit substitution adds all intermediate substitution steps to the e-graph, quickly exploding its size. Section 5.5 shows that this is a major problem in practice, making relatively simple rewrites involving map-fusion and map-fission unfeasible. To avoid adding intermediate substitution steps to the e-graph, we propose *extraction-based substitution* that works as follows.
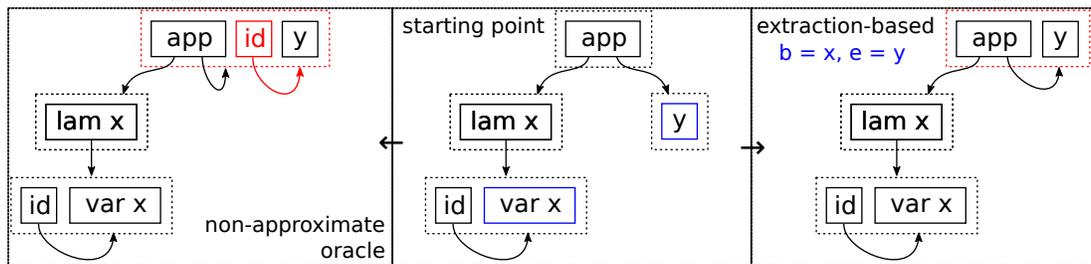
Figure 5.9: Example of β-reduction with extraction-based substitution (right). The initial e-graph (middle) represents $(\lambda x.\ id\ x)\ y$. After extraction-based β-reduction, the e-graph does not represent $id\ y$ because x has been extracted for b in the rewrite rule; ignoring $id\ x$.

1. extract a term for each e-class involved in the substitution (i.e b and e);

2. perform standard term substitution;

3. add the resulting term to the e-graph.

 Section 5.5 demonstrates that extraction-based substitution is far more efficient than explicit substitution. Extraction-based substitution is, however, an approximation as it computes the substitution for a subset of the terms represented by b and e, and ignores the rest. Figure 5.9 shows an example. The initial e-graph is in the middle, the e-graph after a non-approximate oracle substitution is on the left, and the e-graph after extraction-based substitution with $b = x$ and $e = y$ is on the right. This particular choice results in an e-graph lacking the $id\ y$ program.

 In practice, we have not observed the approximation to be an issue when optimising RISE programs (Sections 5.5 and 5.6), and believe that two main reasons account for this. First, the substitution is computed on each equality saturation iteration, where different terms may be extracted, increasing coverage of the set of terms represented by b and e. Second, many of the ignored equivalences are recovered either by e-graph congruence, or by applying further rewrite rules. Future work may investigate alternative substitution implementations to balance efficiency with non-approximation.

5.4.2 Name Bindings

During equality saturation, inappropriate handling of name bindings easily leads to serious efficiency issues. Consider rewrite rules like map-fusion that create a new lambda abstraction on their right-hand side. What name should be introduced when they are applied? In standard term rewriting, generating a fresh name using a global counter (aka. gensym) is a common solution [190]. But if a new name is generated each time the rewrite rule is applied, the e-graph is quickly burdened with many α-equivalent terms[1].

---

[1]Two λ terms are α-equivalent if one can be made equivalent to the other simply by renaming variables.

Fewer α-equivalent terms are introduced if fresh names are generated as a function of the matched e-class identifiers. However as the e-graph grows and e-classes are merged, e-class identifiers change, and α-equivalent terms are still generated and duplicated in the e-graph.

De Bruijn indices [191] are a standard technique for representing lambda calculus terms without naming the bound variables, and avoid the need for α conversions. If De Bruijn indices enable two α-equivalent terms to be structurally equivalent, the standard e-graph congruence invariant prevents their duplication, by ensuring that equivalent e-nodes are not allocated to different e-classes. Hence we translate our terms and rewrite rules to use De Bruijn indices instead of names, and achieve significant efficiency gains (Section 5.5).

The following paragraphs detail various aspects of using De Bruijn indices, and discuss the alternative choice of avoiding name bindings entirely using combinators.

True Equality Modulo α-renaming While De Bruijn indices give a significant performance improvement, they do not provide equality modulo α-renaming for sub-terms. Consider $f\ (\lambda x.\ f) = \%0\ (\lambda.\ \%1)$, where $\%i$ are De Bruijn indices. Although $\%0$ and $\%1$ are structurally different, they both correspond to the same variable f. In practice, we have not observed this to be a significant issue when optimising RISE programs, but it does require care when comparing sub-terms that have a different number of surrounding lambdas. Future work may investigate alternatives to De Bruijn indices, for example through hashing modulo α-renaming [192], nominal rewriting techniques [193], or hierarchical abstract syntax graphs [194].

Translating Name-based Rules into Index-based Rules Using De Bruijn indices means that rewrite rules must manipulate terms with De Bruijn indices. Thankfully, more user-friendly name-based rewrite rules can be automatically translated to the index-based rules used internally [195]. An example demonstrating this is given in Section 5.4.5.

Explicit or Extraction-based Substitution Both explicit substitution and extraction-based substitution are compatible with De Bruijn indices, and for explicit substitution we use the λs calculus [196].

Shifting De Bruijn Indices De Bruijn indices must be shifted when a term is used with different surrounding lambdas (example in Section 5.4.5). As for substitution, shifting can be implemented with explicit rewrite rules, or with *extraction-based index shifting*:

1. extract a term from the e-class whose indices need shifting;

2. perform index shifting on the term;

3. add the resulting term to the e-graph.

In RISEGG we use extraction-based index shifting when extraction-based substitution is used.

Avoiding Name Bindings using Combinators   It is also possible to avoid name bindings entirely [180]. For example, it is possible to introduce a function composition combinator '∘' as in Section 5.1, greatly simplifying the map-fusion and map-fission rules:

$$f \ (g \ x) \longmapsto (f \circ g) \ x \qquad\qquad\qquad\qquad (\circ\text{-intro})$$

$$map \ f \circ map \ g \longmapsto map \ (f \circ g) \qquad\qquad (\text{map-fusion}_2)$$

$$map \ (f \circ g) \longmapsto map \ f \circ map \ g \qquad\qquad (\text{map-fission}_2)$$

However, this approach has its own downsides. Associativity rules are required, which increases the growth rate of the e-graph [86]. Only using a left-/right-most associativity rule reduces growth, but requires other rewrite rules to take this convention into account, making their definition more difficult and their matching more expensive. In general, matching modulo associativity or commutativity are algorithmically hard problems [197].

The function composition ∘ combinator on its own is also not sufficient to remove the need for name bindings. At one extreme, combinatory logic could be used as any lambda calculus term can be represented, replacing function abstraction by a limited set of combinators. However, translating a lambda calculus term into combinatory logic results in a term of size $O(n^3)$ in the worst case, where n is the size of the initial term [198]. Translating existing rewrite systems to combinatory logic would be challenging in itself.

5.4.3   Freshness Predicates

Handling predicates is not trivial in equality saturation. The η-reduction rewrite rule has the side condition "if x not free in f", but in an e-graph f is an e-class and not a term.

The predicate could be precisely handled by filtering f into $f' = \{t \mid t \in f, x \text{ not free in } t\}$, and using f' on the right-hand-side of the rule. However, current e-graph and equality saturation frameworks do not allow discriminating between different e-class subsets in such a way due to their goal of maximal sharing.

The design of RISEGG makes the engineering trade-off to only apply the η-reduction rewrite rule if $\forall t \in f. x$ not free in t, following egg's lambda calculus example [86]. Advantages are that this predicate is efficient to compute using an e-class analysis, and that there is no need to discriminate between different e-class subsets. The disadvantage is that it is an approximation that ignores some valid terms.

Figure 5.10 shows an example where η-reduction is not applied. In practice, we have not observed the approximation to be an issue, e.g. for the results presented in Sections 5.5 and 5.6.
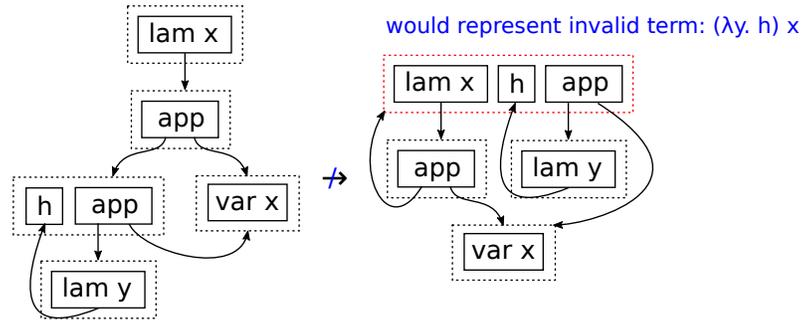
Figure 5.10: Example where η-reduction is not applied. The initial e-graph represents $\lambda x.\ h\ x$, but η-reduction is not triggered because $h = (\lambda y.\ h)\ x$ where x is free. Using $\exists$ instead of $\forall$ in our predicate, we would obtain the e-graph on the right which represents h, but also invalid terms such as $(\lambda y.\ h)\ x$ where x is no longer bound.

5.4.4 Adding Polymorphic Types

A key consideration is how to add types to the e-graph, as typed lambda calculi are pervasive and lay the foundations of almost all functional languages.

More specifically, we look at how polymorphic types interact with e-classes. If types can be computed in a bottom-up fashion, an e-class analysis can be used, similar to how the size and shape of tensors is computed in [177]. However, if polymorphic types are monomorphised, their instantiation is context-dependent and cannot be computed bottom-up. For example, consider the terms $(\lambda x.\ x)\ (0 : \text{i32})$ and $(\lambda x.\ x)\ (0.0 : \text{f32})$. In RISE, the identify function is monomorphised and has two different type instantiations that should live in different e-classes: $\lambda x.\ x : \text{i32} \to \text{i32}$ and $\lambda x.\ x : \text{f32} \to \text{f32}$. Hence, in RISEGG instantiated types are embedded in the e-graph instead of computed by an analysis. Each e-class is associated with a type that all of its e-nodes must satisfy.

In an e-graph there is a tension between sharing and the availability of contextual information in a given e-class. Type instantiation prevents sharing, as the same polymorphic expression produces a different e-class at each type. However, instantiation provides additional contextual information, as each e-class is associated with a precise monotype.

Hash-consing Types Since types are duplicated many times in the e-graph, and since structural type equality is often required, we hash-cons types for efficiency [199], representing each unique type with a unique identifier and leveraging structural sharing. More specifically for RISE, we use separate hash-conses for the different type kinds such as **nat** and **data**, to clearly separate their identifiers in the implementation. For **nat**, hash-consing is combined with arithmetic simplification, e.g. $x \times 0$ will be given the identifier of 0. For **addr**, hash-consing is not necessary since there are only constant addresses (e.g. **global**) and no address constructors to hash-cons.

Alternatively, types could be stored as e-graph terms to provide equational reasoning at the type level.

5.4.5 Compiling user-defined rewrite rules

To avoid explicit typing or explicit use of De Bruijn indices in user-defined rewrite rules, user-friendly name-based and partially typed rewrite rules are compiled internally as required.

Types are inferred with RISE type inference: both sides of rewrite rules can be seen as terms where free variables correspond to pattern variables. After inferring the types on the left-hand-side, we check that the right-hand-side is well-typed for any well-typed left-hand-side match. When applied, typed rewrite rules match (deconstruct) types with their left-hand-side, and construct types on their right-hand-side. Type annotations can be used to constrain the inferred types.

Bound variables are replaced with their corresponding De Bruijn index, and indices are shifted as required for terms with differing numbers of surrounding lambdas. We illustrate with examples.

Example 1: η-reduction

$$\lambda x.\ f\ x \quad \longmapsto \quad f \qquad\qquad \text{if } x \text{ not free in } f$$

RISEGG translates this rule into:

$$(\lambda.f\ (\%0:\ t_0)):\ t_0 \to t_1 \quad \longmapsto \quad (\varphi_1^{-1}f):\ t_0 \to t_1 \qquad \text{if } \%0 \text{ not free in } f$$

The transformed rule uses a De Bruijn index $\%0$ for the bound variable, and pattern variables otherwise: f, t_0 and t_1. It provides index shifting through $\varphi_1^{-1}f$ that shifts all indices ≥ 1 by -1, because a surrounding λ has been removed. Some types are matched on the left-hand-side (t_0 and t_1), and used to construct types on the right-hand-side ($t_0 \to t_1$). Some types, like the type of f, are not matched on the left-hand-side as RISEGG avoids matching on redundant type information to some extent, assuming that the terms matched are well-typed.

Example 2: η-abstraction

$$f:\ t_0 \to t_1 \quad \longmapsto \quad \lambda x.\ f\ x$$

η-abstraction illustrates how type annotations may be used, and are sometimes required. RISEGG translates this rule into:

$$f:\ t_0 \to t_1 \quad \longmapsto \quad (\lambda.\ (((\varphi_0^1 f):\ t_0 \to t_1)\ (\%0:\ t_0)):\ t_1):\ t_0 \to t_1$$

A type error occurs if the type of f is not annotated as in $f:\ t_0 \to t_1$ on the left-hand-side, since the right-hand-side requires it to be a function type.
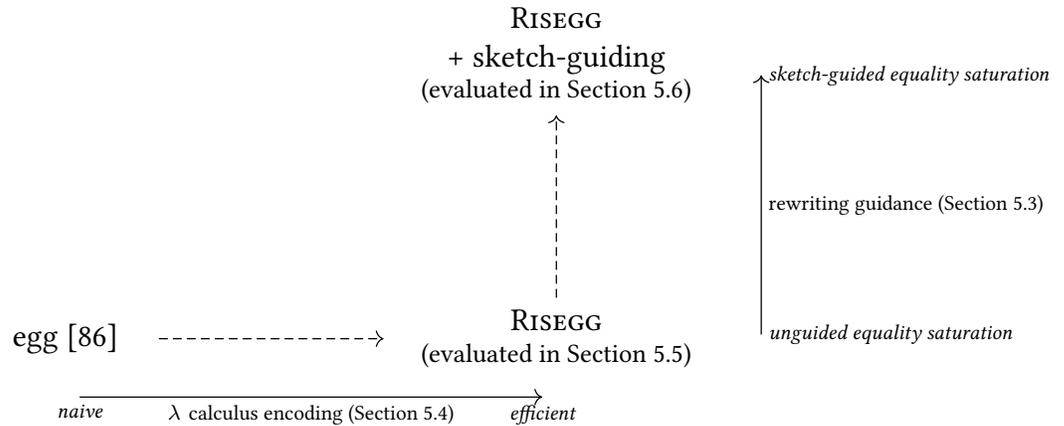
RISEGG
+ sketch-guiding
(evaluated in Section 5.6)

sketch-guided equality saturation

rewriting guidance (Section 5.3)

egg [86] -------------> RISEGG
(evaluated in Section 5.5)

unguided equality saturation

naive λ calculus encoding (Section 5.4)   *efficient*

Figure 5.11: We first evaluate our efficient λ calculus encoding before evaluating sketch-guided equality saturation using this encoding.

5.5 Evaluation of Lambda Calculus Encoding

Figure 5.11 illustrates our process to evaluate our two proposed techniques for scaling equality saturation to the RISE optimisations previously applied using ELEVATE rewriting strategies. Starting from the naive lambda calculus encoding used in egg's example [86], we first investigate the effectiveness of the new encoding from Section 5.4, implemented in RISEGG, for unguided equality saturation (this section). The naive lambda calculus encoding uses explicit substitution and variable names. The efficient encoding uses extraction-based substitution to avoid intermediate substitution steps, and De Bruijn indices to avoid duplicating α-equivalent terms. Thereafter, we adopt the new lambda calculus encoding in RISEGG, and compare unguided and sketch-guided equality saturation to reproduce seven realistic matrix multiplication optimisations previously applied with ELEVATE rewriting strategies in [3] (Section 5.6).

This section evaluates the efficiency of equality saturation for the lambda calculus by attempting to discover three rewrite goals using four combinations of the substitution and name binding techniques outlined in Section 5.4. *Discovering a rewrite goal* means that it is feasible to grow an e-graph starting from the initial program until the goal program is represented in the e-class of the initial program. In other words, it is feasible to discover that the initial program is equal to the goal program.

5.5.1 Experimental Setup

The alternate encodings are realised in an early prototype of RISEGG.[2] This prototype is implemented in Rust using the egg library, and re-implements an untyped subset of RISE (originally implemented in Scala) which is sufficient for quick prototyping.

---

[2]https://github.com/Bastacyclop/egg-rise

To measure search runtime, we use egg's built-in mechanisms, falling back to the GNU time utility in case of an out-of-memory exception. Maximum memory residency is measured with the GNU time utility. The experiments are run on a laptop with an AMD Ryzen 5 PRO 2500U processor, and we limit the available RAM to 2 GB. The results are reported from a single run since we dot not care about small variations but rather about orders of magnitude.

Rewrite Rule Scheduling   By default, the egg library uses a `BackoffScheduler` preventing specific rules from being applied too often, and reducing e-graph growth in the presence of "explosive" rules such as associativity and commutativity. Our experience with Rise optimisation is that using the `BackoffScheduler` is counterproductive as the desired optimisation depends on some explosive rules. For this reason, and to make result analysis easier, Risegg does not use a rewrite rule scheduler.

Rewrite Goals   We compare the lambda calculus encodings using three rewrite goals with increasing complexity.

The *reduction* rewrite goal in Figure 5.12 is based on a unit test from egg's lambda calculus example, and simply uses η-reduction and β-reduction rules to normalise a term. The lambda calculus examples from egg are relatively simple, as the rewrite rules involved do not introduce new names on their right-hand side and in most cases do not increase term size: the e-graph size does not grow explosively.

The *fission* rewrite goal in Figure 5.13 adds the use of map-fusion and map-fission rewrite rules to perform a **map** fission that is more complex than a single map-fission rule, as it goes through 3 chained functions. The map-fusion and map-fission rewrite rules introduce new name bindings on their right-hand-side, and interact with each other as well as β-reduction to create many possibilities: the e-graph size starts to explode.

The *binomial* rewrite goal from Figure 5.14 corresponds to a convolution separation optimisation as seen in Chapter 4, where it was used to reduce both memory accesses and arithmetic complexity. A binomial filter is an essential component of many image processing pipelines, where it reduces noise or detail. For example, it is sometimes used as part of the Harris corner detection instead of the $3{\times}3$ '+' convolution used in Chapter 4. The purpose of the rewrite is to separate the 2D convolution into two 1D convolutions according to the well-known convolution kernel equation:

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$$

The *binomial* goal adds the use of 6 more rewrite rules: 3 rules for **slide** interactions, 1 rule for the **transpose** (**transpose** x) identity, and 2 rules for dot product decompositions. The total 10 rules have many possible interactions, aggravating e-graph growth rate.

The author personally contributed to [1] by demonstrating how to achieve this optimisation via an ELEVATE strategy applying a sequence of 30 rewrite rules, including 17 η/β-reductions. Although more complex than the *reduction* and *fission* rewrite goals, the *binomial* rewrite goal is still relatively simple compared to complete Harris corner detection and matrix multiplication optimisations. We suggest that unguided equality saturation should at least scale to the *binomial* goal and its relatively low complexity, to be useful in practice for RISE.

```
1  (λcomp.
2   (λadd1.
3    comp add1 (comp add1 (comp add1 (comp add1 (comp add1 (comp add1 add1))))))
4   ) (λy. y + 1)
5  ) (λf. λg. λx. f (g x))
```

$$\mapsto^*$$

```
1  λx. ((((((x + 1)+ 1) + 1) + 1) + 1) + 1) + 1
```

Figure 5.12: ***reduction* rewrite goal.** The initial program creates a `comp` combinator for function composition and uses it to compose `add1` with itself 7 times. All uses of `comp` and `add1` are β-reduced in the final program, which simply applies `+ 1` to its input value 7 times.

```
1  map (λx. f₅ (f₄ (f₃ (f₂ (f₁ x)))))
```

$$\mapsto^*$$

```
1  λy. map (λx. f₅ (f₄ (f₃ x))) (map (λx. (f₂ (f₁ x))) y)
```

Figure 5.13: ***fission* rewrite goal.** The initial program successively applies f_1 to f_5 inside a **map** pattern. The final program first applies f_1 and f_2 inside one **map** pattern, before applying f_3 to f_5 inside another **map** pattern.

```
1  map (map λnbh. dot (join weights2d) (join nbh))
2   (map transpose (slide 3 1 (map (slide 3 1) input)))
```

$$\mapsto^*$$

```
1  map (λnbhL. map (λnbhH. dot weightsH nbhH)
2   (slide 3 1 (map (λnbhV. dot weightsV nbhV) transpose nbhL)))
3   (slide 3 1 input)
```

Figure 5.14: ***binomial* rewrite goal.** The initial RISE program iterates over 2D neighborhoods (nbh). A `dot` product is computed between the 2D weights and each 2D neighborhood. The final program iterates over two 1D neighborhoods combined with 1D weights instead.

λ calculus encoding		goal	found?	runtime	RAM	rules	e-graph size	
extraction?	De Bruijn?						e-nodes	e-classes
✗	✗	*reduction*	✓	0.02s	3 MB	0.5K	0.5K	0.1K
✗	✗	*fission*	✗	16s	>2000 MB			
✗	✗	*binomial*	✗	15s	>2000 MB			
✗	✓	*reduction*	✓	0.2s	35 MB	28K	53K	25K
✗	✓	*fission*	✓	0.3s	36 MB	39K	21K	10K
✗	✓	*binomial*	✗	30s	>2000 MB			
✓	✗	*reduction*	✓	0.004s	3 MB	0.1K	0.3K	0.2K
✓	✗	*fission*	✓	0.006s	3 MB	0.2K	1K	0.7K
✓	✗	*binomial*	✗	20s	>2000 MB			
✓	✓	*reduction*	✓	0.002s	3 MB	0.1K	0.2K	0.1K
✓	✓	*fission*	✓	0.006s	3 MB	0.6K	0.6K	0.3K
✓	✓	*binomial*	✓	0.1s	8 MB	5K	3K	1K

Table 5.3: Evaluating the efficiency of lambda calculus encoding techniques on three rewrite goals. Combining extraction-based substitution and De Bruijn indices minimises runtime and memory consumption (green background), and is the only encoding that finds the *binomial* rewrite goal.

5.5.2   Performance Results

Table 5.3 compares the performance of equality saturation using different combinations of substitution (explicit/extraction-based) and name binding (named/De Bruijn) techniques. It reports whether the goal is found, the search runtime, memory consumption, number of applied rewrite rules, and e-graph size. The simple *reduction* goal is found by all combinations, although explicit substitution with De Bruijn indices is less efficient with 25K rewrite rules, 25K e-classes, and occupying 35 MB. The *fission* goal is not found if explicit substitution is used with named variables, exhausting the 2 GB memory and showing that this encoding is particularly inefficient.

The *binomial* goal is only found by combining extraction-based substitution and De Bruijn indices. With this encoding, all three rewrite goals are found by applying less than 5K rewrite rules, producing e-graphs with fewer than 3K e-nodes and 1K e-classes, and occupying less than 8 MB of memory. For all three rewrite goals, this combination provides the fastest searches and most compact e-graphs, often by orders of magnitude.

We conclude that *combining extraction-based substitution and De Bruijn indices gives an efficient encoding of lambda calculus for equality saturation*, and adopt this encoding in RISEGG. This experiment also demonstrates that we can discover relatively simple RISE rewrite goals using unguided equality saturation. In the following experiment, we seek to build upon this success to discover significantly more complex RISE rewrite goals using sketch-guided equality saturation.

5.6   Evaluation of Sketch Guidance

This section compares unguided equality saturation to the new sketch-guided equality satura-
tion to achieve complex optimisation goals. In the evaluation, *both* equality saturation tech-
niques use the efficient lambda calculus encoding from Section 5.5. We evaluate the 7 matrix
multiplication optimisations described in the TVM [10] manual[3] and reproduced in [3] using
ELEVATE strategies. Matrix multiplication is selected as the case study as it allows us to com-
pare against published ELEVATE strategies that specify optimisations equivalent to TVM sched-
ules [3]. The TVM schedules were not automatically discovered, but written by performance
engineers, similar to the Halide schedule discussed in Chapter 4 for Harris corner detection.
The ELEVATE strategies [3] express the optimisations performed by TVM as compositions of
rewrites and achieve the same high performance as TVM. The code generated by SHINE is at
worst $0.29\times$ slower and on average $1.14\times$ faster than the code produced by TVM on Intel core
i5-4670K [3]. The optimisations are typical compiler optimisations, including loop blocking,
loop permutation, vectorisation, and multi-threading.

In this evaluation, we compare how much runtime and memory are required for unguided
equality saturation and for sketch-guided equality saturation. For both guided and unguided
equality saturation the optimisation goal is specified as a sketch, that acts as the stopping
criteria. This is less restrictive than the searches for a goal program in the previous subsection
as the sketch goal may be satisfied by many programs.

We validate that the result of each optimisation goal is high performance code as follows.
When a program satisfying the optimisation goal sketch is found, we check that the generated
C code is equivalent, modulo variable names, to the code obtained using ELEVATE strategies
(Appendix B).

5.6.1   Experimental Setup

The full version of RISEGG[4] is implemented in Scala, allowing it to leverage the existing RISE
codebase. We decided to reimplement the features of the egg library in Scala instead of deal-
ing with Rust-Scala interoperability. The standard Java utilities are used for measurements:
`System.nanoTime()` to measure search runtime, and the `Runtime` API to approximate max-
imum heap memory residency with regular sampling.

The experiments are performed on two platforms. For ELEVATE strategies and our sketch-
guided equality saturation, we use a less powerful AMD Ryzen 5 PRO 2500U with 4 GB of RAM
available to the JVM. For unguided equality saturation, we use a more powerful Intel Xeon E5-
2640 v2 with 60 GB of RAM available to the JVM. The results are reported from a single run
since we dot not care about small variations but rather about orders of magnitude.

---

[3]`https://tvm.apache.org/docs/how_to/optimize_operators/opt_gemm.html`
[4]`https://github.com/rise-lang/shine/tree/sges/src/main/scala/rise/eqsat`

Optimisation Goals   Each optimisation goal incrementally adds more optimisations. Appendix B shows the corresponding C code generated by SHINE.

The *baseline* optimisation goal uses 3 straightforward nested loops to perform the matrix multiplication. The *blocking* optimisation goal adds a blocking (or tiling) optimisation for improved data locality, resulting in 6 nested loops where the 3 innermost ones process $4 \times 32 \times 32$ blocks. The *vectorisation* optimisation goal adds parallelism by vectorising the innermost loop over 32 elements. The *loop-perm* optimisation goal changes the order of the 6 nested loops, for improved data locality. The *array-packing* optimisation goal adds intermediate storage for the transposed b matrix, improving memory access patterns. The *cache-blocks* optimisation goal unrolls the inner reduction loop. The *parallel* optimisation goal adds parallelism by multi-threading the outermost loop.

5.6.2   Runtime and Memory Consumption of (Un)Guided Search

Unguided Equality Saturation   Table 5.4 shows the runtime and memory consumption required to find the optimisation goals with unguided equality saturation. The search terminates when the sketch describing the optimisation goal is found in the e-graph.

The 5 most complex optimisation goals are not found before exhausting the 60 GB of available memory. Only the *baseline* and *blocking* goals are found, and the search for *blocking* requires more than 1h and about 35 GB of RAM. Millions of rewrite rules are applied, and the e-graph contains millions of e-nodes and e-classes. More complex optimisations involve more rewrite rules, creating a richer space of equivalent programs but exhausting memory faster. As examples, *vectorisation* and *loop-perm* use vectorisation rules, while *array-packing*, *cache-blocks*, and *parallel* use rules for optimising memory storage.

Sketch-Guided Equality Saturation   Table 5.5 shows the runtime and memory consumption for sketch-guided equality saturation, where sketch guides are used to break a single equality saturation search into multiple.

All optimisations are found in less than 10s, using less than 0.5 GB of RAM. Interestingly, the number of rewrite rules applied by sketch-guided equality saturation is in the same order of magnitude as for the manual ELEVATE strategies reported in [3]. On one hand, equality saturation applies more rules than necessary because of its explorative nature. On the other hand, ELEVATE strategies apply more rules than necessary because they re-apply the same rule to the same sub-expression and do not necessarily orchestrate the shortest possible rewrite path. The e-graphs contain no more than 10^4 e-nodes and e-classes, two orders of magnitude less than the 10^6 required for *blocking* without sketch-guidance.

goal	found?	runtime	RAM	rules	e-nodes	e-classes
baseline	✓	0.5s	0.02 GB	2	51	49
blocking	✓	>1h	35 GB	5M	4M	2M
vectorisation	✗	>1h	>60 GB			
loop-perm	✗	>1h	>60 GB			
array-packing	✗	35mn	>60 GB			
cache-blocks	✗	35mn	>60 GB			
parallel	✗	35mn	>60 GB			

Table 5.4: Runtime and memory consumption for **unguided equality saturation** with efficient lambda calculus encoding. Only the *baseline* and *blocking* optimisation goals are found, with other optimisations exceeding 60 GB.

goal	sketch guides	found?	runtime	RAM	rules	e-nodes	e-classes
baseline	0	✓	0.5s	0.02 GB	2	51	49
blocking	1	✓	7s	0.3 GB	11K	11K	7K
vectorisation	2	✓	7s	0.4 GB	11K	11K	7K
loop-perm	2	✓	4s	0.3 GB	6K	10K	7K
array-packing	3	✓	5s	0.4 GB	9K	10K	7K
cache-blocks	3	✓	5s	0.5 GB	9K	10K	7K
parallel	3	✓	5s	0.4 GB	9K	10K	7K

Table 5.5: Runtime and memory consumption for **sketch-guided equality saturation** with efficient lambda calculus encoding. All optimisations are found in seconds using less than 0.5 GB of memory, and requiring at most 3 sketch guides.

5.6.3   E-Graph Evolution in (Un)Guided Search

Figure 5.15 plots the growth of the e-graphs during unguided and sketch-guided searches for the *blocking* and *parallel* optimisation goals from Tables 5.4 and 5.5. The e-graphs produced by unguided equality saturation grow exponentially with each search iteration. The e-graph contains millions of e-nodes and e-classes after applying millions of rules within a small number of iterations (less than 10). Such rapid growth limits the scalability of unguided search, for example in the 7th iteration of the *parallel* search the e-graph exhausts 60GB memory.

While the e-graphs produced with sketch-guidance typically also grow exponentially with each iteration, sketches are satisfied within a small number of iterations thanks to an appropriate selection of sketch guides. The number of rewrites and the maximum e-graph size is *three orders of magnitudes smaller* than for unguided search: no more than 11K in our example searches. Once a program satisfying a sketch guide is found, a new search is started for the next sketch using that program, growing a fresh e-graph. Hence sketch-guidance enables scaling to more complex RISE optimisations, such as *parallel*. Conceptually there is no limit on the complexity of the optimisations that may be searched for, as optimisations may be factored into as many sketch-guided searches as necessary.

The search for the final *parallel* sketch goal shows linear rather than exponential growth, as the rewrite rules selected for the search have little interaction.

(a) unguided equality saturation
blocking (found: ✓)

(b) unguided equality saturation
parallel (found: ✗)

(c) sketch-guided equality saturation
blocking (found: ✓)

(d) sketch-guided equality saturation
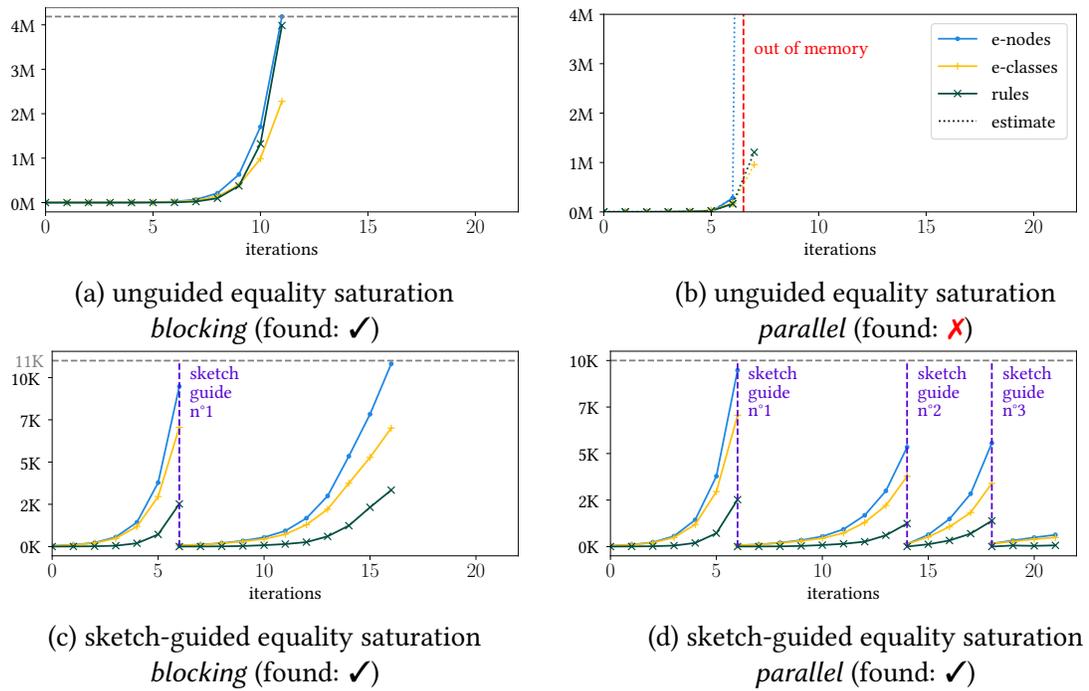*parallel* (found: ✓)

Figure 5.15: The evolution of the e-graph, and the number of rewrite rules applied, during searches for two optimisation goals. Sketch guides are depicted with purple vertical lines. Note that the scale of the y-axes for unguided graphs (a) and (b) is millions, while for guided graphs (c) and (d) it is thousands.

5.6.4 Provided Guidance

Sketches Guiding the Search Table 5.6 shows how each optimisation goal is achieved by logical steps, each corresponding to a sketch describing the program after the step is applied. It transpires that the *split* sketch in Listing 5.4 is a useful first guide for all goals. While the sketch sizes range from 7 to 12, programs are of size 90 to 124, showing that a sketches elides around 90% of the program. Even when 4 sketches must be written, the total sketch size is still small: the largest total being 38. Appendix C contains all handwritten sketches as well as examples of discovered RISE programs. Intricate program aspects never need to be specified in the sketches, for example array reshaping patterns such as **split**, **join** and **transpose**.

goal	sketch guides	sketch goal	sketch sizes	program size
blocking	*split*	*reorder$_1$*	7	90
vectorisation	*split + reorder$_1$*	*lower$_1$*	7	124
loop-perm	*split + reorder$_2$*	*lower$_2$*	7	104
array-packing	*split + reorder$_2$ + store*	*lower$_3$*	7-12	121
cache-blocks	*split + reorder$_2$ + store*	*lower$_4$*	7-12	121
parallel	*split + reorder$_2$ + store*	*lower$_5$*	7-12	121

Table 5.6: Decomposition of each optimisation goal into logical steps. A sketch is defined for each logical step. In this table, sketch size counts operators such as **containsMap**, program size counts operators such as **map**, lambdas, variables and constants: not λ applications.

Choice of Rules and Cost Model   Besides the sketches, performance engineers also specify the rules used in each search and a cost model. For the *split* sketch, 8 rules explain how to split `map` and `reduce`. The *reorder* sketches require 9 rules that swap various nestings of `map` and `reduce`. The *store* sketch requires 4 rules and the *lower* sketches 10 rules including map-fusion, 6 rules for vectorisation, 1 rule for loop unrolling and 1 rule for loop parallelisation. If we naively use all rules for the blocking search, the search runtime increases by about $25\times$, still finding the goal in minutes but showing the importance of selecting a small set of rules.

We use a simple cost model that minimises weighted term size. For example, we may give a penalty to using `mapPar`, to avoid implicit multi-threading if it is not explicit in the sketch. Rules and cost models may be reused and packaged into libraries for recurring logical steps.

5.7   Conclusion

This chapter contributes *sketch-guided equality saturation*, a semi-automated optimisation technique that offers a practical trade-off between the painstaking control of rewriting strategies, and the automated, but often unsuccessful, equality saturation. Performance engineers guide rewriting by describing how a program should evolve using a sequence of sketches, factoring an infeasible equality saturation search into a sequence of feasible equality saturation searches.

Sketch-guiding leverages the observation that program optimisations are often explained with potentially incomplete program snippets, as in Figure 5.4 and Chapter 4. Sketch-guiding enables performance engineers to focus on what the optimised program should look like, rather than on individual program transformation steps.

We demonstrate that sketch-guiding enables seven complex optimisations of matrix multiplication to be applied within seconds in the RISE functional language, using less than 1 GB of RAM (Table 5.5), using no more than three sketch guides, each 10 times smaller than the complete program (Table 5.6). By contrast, traditional unguided equality saturation cannot discover the five most complex optimisations even with an hour of runtime and 60 GB of RAM (Table 5.4). For each optimisation, the generated code is identical to the high-performance code generated by ordering thousands of rewrites via the definition of 36 ELEVATE rewriting strategies in 200 lines of code. The generated code is at worst $0.29\times$ slower and on average $1.14\times$ faster than the code produced by the state-of-the-art TVM compiler on Intel core i5-4670K [3].

This chapter also explores engineering design choices to effectively encode a polymorphically typed lambda calculi like RISE for equality saturation. The key innovations are extraction-based substitution and representing identifiers as De Bruijn indices. Combining the techniques reduces the runtime and memory consumption of equality saturation over lambda terms by orders of magnitude, and is necessary to enable unguided equality saturation to discover even relatively simple RISE optimisation goals, such as convolution separation (Table 5.3).

There is no real ending. It's just the place where you stop the story.

— Frank Herbert

Chapter 6

Discussion

6.1 Summary

Optimising programs is challenging, even for skilled performance engineers. Modern compilers targeting heterogeneous architectures face two major challenges. First, domain-specific compilers such as Halide for image processing and TVM for machine learning are difficult to extend with the new optimisations required by new algorithms and hardware. Second, automatic optimisation is often unable to achieve the required performance, and performance engineers often fall back to painstaking manual optimisation.

To mitigate these challenges, this thesis shows the potential of the novel SHINE compiler to achieve domain-extensibility, controllable automation, and generate high performance code. Domain-extensibility facilitates adaptation to new algorithms and hardware. Controllable automation enables performance engineers to gradually take control of the optimisation process, with the assistance of the compiler. In SHINE, optimisations are applied by rewriting functional programs in the RISE array language, before generating imperative code.

This thesis makes the following research contributions:

1. **Enhancing Code Generation in a Domain-Extensible Compiler** (Chapter 3).
   Three important code generation features are added to SHINE, enabling the generation of high-performance code in Chapters 4 and 5.

   - We contribute a synchronisation *barrier insertion* algorithm that does not need to be modified when extending RISE patterns, as opposed to the barrier elimination algorithm of LIFT [107]. The correctness and efficiency of barrier insertion is evaluated on 38 unit tests and 10 benchmarks, mostly taken from prior LIFT work. We identify 6 differences in the code generated by SHINE and LIFT, and observe that our algorithm fixes bugs in 13 unit tests and 1 benchmark, where LIFT generates incorrect barriers (Table 3.2). There is only 1 benchmark where SHINE inserts a barrier that LIFT eliminates, and we provide a clear pathway to improve our algorithm

to generate more efficient barriers than Lift on all unit tests and benchmarks.

While barrier insertion is implicit and not controllable by rewriting, the next two features add new Rise patterns in order to expose implementation choices to be controlled during rewriting, allowing design space exploration.

- We add the `oclRun` Rise pattern to represent *kernel execution* explicitly, computing the value of an expression by launching an OpenCL kernel. This requires modifying the Shine compiler to generate imperative code for multiple OpenCL kernels, as well as the necessary host code to launch them. With this feature, 1K lines of handwritten host code are replaced with 1.2K lines of automatically generated code on a relatively simple design space exploration case study (Tables 3.3 and 3.4).

- We add the `circularBuffer` and `rotateValues` Rise patterns to enable explicit *storage folding* for temporary arrays. This requires modifying Shine to generate the desired imperative code when using these patterns. Chapter 4 relies on storage folding to generate high performance code.

2. **Going Beyond Halide Scheduling with Controlled Rewriting** [1] (Chapter 4).
   Domain-extensibility is combined with controlled rewriting to optimise a standard image processing pipeline: the Harris corner detection [30]. Optimisations are controlled in Shine using Elevate rewriting strategies [3] that compose rewrite rules. First, Elevate allows us to reproduce the effect of an optimised Halide schedule applying 4 standard optimisations: operator fusion, multi-threading, vectorisation and circular buffering. Second, Elevate allows us to apply 2 additional optimisations that are not supported by Halide schedules: convolution separation and register rotation. Circular buffering and convolution separation in particular leverage the low-level Rise patterns exposed in Chapter 3, that are introduced via controlled applications of rewrite rules.

   Our results on four mobile ARM multi-core CPUs and two different image resolutions show that, with these 6 optimisations, Shine generates code (Appendix A) up to $16\times$ (geomean of $9.48\times$) faster than OpenCV library code, up to $4.5\times$ (geomean of $3.87\times$) faster than the similarly designed Lift compiler, and up to $1.4\times$ (geomean of $1.27\times$) faster than Halide (Figure 4.7).

   However, we also observe that controlling rewriting with Elevate strategies is tedious. The strategies defined to apply the 6 optimisations are specialised to our Harris case study, and consist of more than 600 lines of code defining 57 helper strategies. To perform all 6 optimisations, thousands of rewrite steps are applied. It is unclear how to generalise the strategies for reuse across diverse image processing pipelines. This motivates the following chapter that aims to lower performance engineer effort through semi-automatic optimisation.

3. **Proposing a Novel Semi-Automatic Optimisation Technique** [2] (Chapter 5).

   A new semi-automatic optimisation technique called *sketch-guided equality saturation* is developed, offering a practical trade-off between the painstaking control of rewriting strategies, and the automated, but often unsuccessful, equality saturation. Sketch-guiding allows performance engineers to guide program rewriting by specifying rewrite goals as *sketches*: program patterns that leave details unspecified. Sketch-guiding leverages the observation that program optimisations are often explained with incomplete program snippets, as in Figure 5.4 and Section 4.3. Sketch-guiding enables performance engineers to focus on what the optimised program should look like, rather than on individual program transformation steps.

   Chapter 5 evaluates sketch-guided equality saturation by applying 7 realistic optimisations of matrix multiplication in the RISE language. Unguided equality saturation alone does not scale to the 5 most complex optimisations, even given an hour and 60GB of RAM (Table 5.4). With the guidance of at most 3 sketch guides, each 10 times smaller than the complete program (Table 5.6, Appendix C), the compiler applies the optimisations in seconds using less than 1GB of RAM (Table 5.5). For each optimisation, the generated code (Appendix B) is identical to the high-performance code generated by manually ordering thousands of rewrites via the definition of 36 rewriting strategies in 200 lines of code. The runtime performance of this code on Intel core i5-4670K is at worst $0.71\times$ and on average $1.14\times$ that of the code produced by the state-of-the-art TVM compiler [3].

   In addition, Chapter 5 demonstrates how to efficiently encode a polymorphically typed lambda calculus such as RISE for equality saturation. The key innovations are extraction-based substitution and representing identifiers as De Bruijn indices. Combining the techniques reduces the runtime and memory consumption of equality saturation over lambda terms by orders of magnitude, and is necessary for unguided equality saturation to discover even relatively simple RISE optimisation goals, such as convolution separation (Table 5.3).

Overall, this thesis demonstrates how extensible rewriting systems are a powerful approach to build domain-extensible compilers with controllable automation of optimisations, that generate high-performance code. We envision a future were compilers adapt to the rapid pace of change in algorithms and hardware in collaboration with performance engineers. As new algorithms and hardware are developed, performance engineers will be able to extend compilers with simple, specialised transformations expressed as rewrite rules, to create and explore complex optimisation spaces. Depending on performance requirements and engineering budgets, performance engineers and compilers will cooperate to explore complex optimisation spaces, for example via rewriting strategies or sketch-guidance.

6.2   Limitations

Code Generation   The intention is that functional program rewriting should be extensible and controllable, and imperative code generation should be reusable and predictable. The benefit is that main optimisation choices are encoded in RISE via rewriting, without worrying about imperative details or side effects. The drawback is that imperative code generation is not easily extensible and controllable by performance engineers. First, adding new imperative patterns to the intermediate DPIA language requires modifying internal compiler code, e.g. for barrier insertion (Section 3.2). Second, exposing an implementation choice to rewriting requires encoding that choice into the functional RISE language, e.g. as done with `rotateValues` (Section 3.4).

Memory Re-use   In SHINE, memory is allocated in a simplistic way when translating functional DPIA to imperative DPIA. Therefore, there is potential for memory re-use or in-place computation which is not exploited. For example, there is currently no way to compute `mapSeq` `f  x` in-place by overriding the memory for the array x.

Arithmetic Expressions   In SHINE, unrestricted symbolic arithmetic expressions are used to represent array sizes and indices. Type inference relies on symbolic unification, and code generation on symbolic simplification of arithmetic expressions. Therefore, type inference success and code generation quality are impacted by the best-effort heuristics used to solve non-linear integer arithmetic formulas in SHINE, an undecidable problem in general [200].

Custom Types   The design of SHINE does not facilitate adding new types to RISE. For example, adding a stream type for storage folding patterns (Section 3.4) would require modifying internal compiler code, such as the code for type inference.

Property Reasoning   When rewriting reductions, it is necessary to reason about associativity and commutativity of the reduction operator. Currently, SHINE does not abstract well over such program properties. For example, the operator of a `reduce` pattern and the `add` function are considered associative and commutative, but there is no mechanism to assess whether an arbitrary function is associative or commutative. Similarly, the layout and alignment of arrays in memory matters for the Harris corner detection case study (Chapter 4). Currently, the desired layout is achieved via careful control using ELEVATE strategies. Achieving the desired layout using sketch-guiding would require the ability to constrain memory layouts in sketches.

ELEVATE Strategies   The ELEVATE strategies developed to optimise the Harris corner detection (Chapter 4) are specialised to that case study. By contrast, the Halide scheduling primitives are more generic, and are reused across diverse image processing pipelines.

Sketch-Guided Equality Saturation   Sketch-guided equality saturation is a promising technique, but it remains limited. Currently, there is no clear methodology to guide performance engineers in using this technique. The strength of sketch-guiding is to allow a declarative specification of the optimisation goal, however in some cases imperative specifications of how to rewrite the program like rewriting strategies might be more suitable. The expressivity of sketches could be improved, for example we would not know how to support a $\wedge$ SKETCHBA-SIC constructor. Some limitations are inherited from equality saturation, as it is used as the search method. For example, lambda calculus support remains unsatisfactory as our proposed encoding relies on approximations. Follow-up work is discussed in the next section.

User Study   This thesis is motivated by the need to reduce performance engineering efforts, but does not claim to quantify this effort. While the need for compiler extensibility and controllability is motivated by common knowledge and prior work (Chapter 2), no user study was conducted. Conducting user studies could give insights into the needs of performance engineers, and inform the design of future semi-automatic compilers for best impact.

6.3   Ongoing & Future Work

Case Studies   This thesis focuses on two non-trivial case studies, rather than many trivial case studies. The Harris corner detection is a standard benchmark for image processing pipeline optimisation (Chapter 4). Matrix multiplication is a standard benchmark for linear algebra optimisation (Chapter 5). Achieving competitive performance compared to state-of-the-art domain-specific compilers like Halide and TVM on these case studies is a promising first step. To demonstrate broader applicability of the techniques explored in this thesis, future work should consider more algorithms and more heterogenous hardware architectures.

The author already developed high-level RISE programs for more diverse image processing pipelines from the Halide benchmark suite (camera pipeline, local laplacian, multiscale interpolation, unsharp mask), as well as deep neural networks subgraphs (e.g. fusing matrix multiplication with activation functions). The challenge is now to optimise them via rewriting. We expect such case studies to stress test the SHINE compiler design, and to inspire novel ideas, just like they inspired the author to develop sketch-guided equality saturation.

The author also reproduced some of the Harris corner detection optimisations from Chapter 4 using sketch-guided equality saturation. Sketches closely resembling the program snippets from Section 4.3.1 were written for the outcome of operator fusion, `harrisIxWithIy`, multi-threading, circular buffering, `sequentialLines` and `unrollReductions`. Except for operator fusion, where the search is currently failing, sketch-guiding successfully finds programs as desired. To fully reproduce the SHINE cbuf version of Harris corner detection using sketch-guiding, only the operator fusion and vectorisation transformation steps are missing.

Optimisation Assistant   ELEVATE rewriting strategies and sketch-guided equality saturation are tools for performance engineers to optimise programs via rewriting. For increased productivity, future work may integrate such rewriting tools into an interactive, rewrite-based optimisation assistant. Rewriting strategies and sketch-guiding can be combined into hybrid solutions: it is possible to expose sketch-guided equality saturation as a strategy, i.e. a P → `RewriteResult`[P] function. Such an assistant could serve as a testbed for many other rewriting tools: LIFT-style stochastic search, SPIRAL-style hardware-aware search, cost estimation feedback, rewrite rule or even sketch synthesis, etc. Multiple interactive optimisation assistants were recently developed independently, showing that this research direction is valued by the community. Roly-poly supports performance engineers when developing Halide schedules [26]. DLOOPT is an optimisation assistant for TVM [201]. OptiTrust allows developing high-performance C code via series of source-to-source transformations [202].

Sketch-Guided Rewriting   This thesis only scratches the surface of what is possible with the novel idea of sketch-guided rewriting. Future work could:

1. Develop a methodology for performance engineers to come up with appropriate sketches and sets of rewrite rules. We imagine that performance engineers would start by writing their sketch goal, and include all potentially useful rewrite rules for the search. Then, if the search is unsuccessful, performance engineers can incrementally write sketch guides to make the search easier, but also to debug the search. For example, if tiling fails but intermediate splitting succeeds, then the problem lies in the reordering search.

2. Investigate how to compute the intersection between the set of programs represented by a sketch, and the set of programs represented by an e-graph. This could be used to recover optimality and search completeness guarantees for use cases where reaching saturation is feasible throughout the entire sketch-guided equality saturation process.

3. Attempt to remove the need for intermediate sketch goals by synthesizing or inferring them. For example, given a tiling sketch (splitting + reordering), could an intermediate splitting sketch be automatically inferred? Recent related work constructs a search space by generating program sketches [203].

4. Develop reusable transformation libraries. For example, would it be possible to provide a complete set of rewrite rules for splitting and reordering the loops of any RISE program? How would such libraries interact with domain-extensibility and the addition of new RISE patterns?

5. Investigate how to design useful generic sketches, that may be used for multiple programs. An example use-case would be to eliminate a certain pattern from any given program, by introducing a `notContains` sketch construct.

6. Combine sketch-guiding with other search methods (e.g. polyhedral modeling, reinforcement learning, stochastic search, greedy search, beam search). This could be used to trade-off between exploration and exploitation of the search space [187], and to provide optimised search strategies for specific transformations.

7. Compare sketch-guiding to program synthesis tools like Sketch [189] or Rosette [204]. Is incremental sketch-guiding beneficial on standard program synthesis benchmarks compared to traditional counterexample guided inductive synthesis? Is writing a program to start rewriting from easier than writing a program specification to check against?

We hope that the community will be inspired to apply sketch-guided equality saturation, or sketch-guiding, to more diverse applications. To this end, the author has already released a Rust library implementing sketch-guided equality saturation on top of the egg library, it can be used for any term language compatible with egg (`https://github.com/Bastacyclop/egg-sketches`).

Domain-Extensible Compilers   We advocate that domain-extensible compilers should be developed to cope with the rapid pace of change in algorithms and hardware, and contribute to this relatively recent research direction. More work is required to establish domain-extensible compilers as a viable, production-ready alternative to domain-specific compilers. A sensible next step would be to investigate compiling frontend languages through SHINE (Figure 3.1), for example by translating Halide algorithms into RISE programs, or by translating Fortran code as done in TyTraCL [38, 205]. With the aim to build the next generation of domain-extensible compilers, we are currently exploring multiple potential collaborations.

We are engaging with the authors of the recent Exo language [13], independently developed to help performance engineers write, optimise, and target high-performance computing kernels onto new hardware accelerators. *Exocompilation* is about externalising target-specific code generation support and optimisation policies to user-level code, and is rooted in similar motivations as domain-extensibility and controllable automation. In [13], performance engineers can add support for custom imperative instructions, and control when they are introduced using a scheduling language. Exo inherits ideas from both Halide schedules and ELEVATE strategies, its scheduling language is implemented by composing rewrite rules over imperative programs.

We are engaging with the authors of AnyDSL [17], that are currently working on the next generation of their IR: Thorin 2. To provide additional flexibility, Thorin 2 will be based on the calculus of constructions [206], which is also the basis of proof assistants like Coq [207]. Of particular interest is the possibility to embed the RISE language into Thorin 2, and to bring some of our program rewriting techniques into this ecosystem, where they could be combined with powerful partial evaluation techniques.

We are engaging with users and developers of MLIR [208] (the Multi-Layer Intermediate Representation). MLIR is a novel approach for building reusable and extensible compiler infrastructures that was independently developed during this thesis. MLIR allows compiler writers to define their own specialised *dialects* that can interact with other MLIR dialects. Martin Lücke has already embedded a subset of Rise as an MLIR dialect [209], allowing to interact with a higher-level TensorFlow machine learning dialect, and a lower-level polyhedral affine dialect. There is also interest in bringing our program rewriting techniques into the MLIR ecosystem, so that they can be applied to any dialect.

Formal Verification   This thesis uses rewrite rules that must be semantics preserving, but makes no attempt at formally verifying rewrite rule correctness. Steuwer's thesis [45] and Qin's work[1], mentioned in [3], provide proofs for multiple rewrite rules, however we use many additional rules. In Shine, rewrite rules are treated as axioms, and trusted to be semantics preserving without being verified. Code generation is also not formally verified in Shine.

Future work could attempt to formally verify all or parts of the Shine compiler. During this thesis, a paper was independently written on formally verified rewriting for tensor program optimisation [63]. We expect combining domain-extensibility, controllable automation and formal verification to be highly rewarding, but also highly challenging. For example, formally verifying the patterns and rewrite rules added to Rise & Shine by performance engineers would provide extension safety. In Exo [13], performance engineers can introduce their own instructions by pairing them with a semantic model, allowing to automatically check for semantic preservation of code replacements using effect analyses and SMT solving.

The connection between theorem proving and program optimisation is intriguing, and may lead to promising research. While rewriting strategies specify how to transform a program, proof tactics [210] specify how to transform a proof state. Just as program sketches specify partial programs and can guide optimisation, proof sketches specify partial proofs and can guide theorem proving [211, 212]. Although equality saturation exploits e-graphs for program optimisation [85], e-graphs were originally designed for efficient congruence closure in theorem provers [213, 214].

Numerical Analysis   This thesis treats floating points as real numbers, ignoring accuracy problems coming from rounding errors [215]. Similarly, underflow and overflow problems are ignored for integers. Future work may incorporate numerical analyses [216, 183, 217] in Shine to reason about the accuracy of number representations.

---

[1]`https://github.com/XYUnknown/individual-project`

Appendix A

Optimised Harris Corner Detection

This appendix contains the OpenCL programs generated by SHINE after rewriting the RISE program for Harris corner detection using the two ELEVATE strategies from Listings 4.8 and 4.12.

To keep the code readable and compact, we may apply cosmetic changes to the code such as renaming variables and removing unnecessary parentheses, brackets, or space. For example, we may rewrite the code from Listing A.1 into the code from Listing A.2.

```
1  {
2    float x145;
3    x145 = 0.0f;
4    /* comment */
5    x145 = (x145 + 1.0f);
6    vstore(x145, (&(x234[((((((((2 * n0) * n1) + (4 * i1)) + (8 * n1)) + ((32 * gid) * n1)) + (i0 *
       ↪   n1))])))));
7  }
```

Listing A.1: Code sample before cosmetic changes.

```
1  float x2 = 0.0f;
2  x2 += 1.0f;
3  vstore(x2, &x1[2*n0*n1 + 4*i1 + 8*n1 + 32*gid*n1 + i0*n1]);
```

Listing A.2: Code sample after cosmetic changes.

A.1 Comparable to Halide Reference

The OpenCL code generated by SHINE after rewriting the RISE Harris corner detection from Listing 4.6 (Section 4.3) using the cbuf ELEVATE strategy from Listing 4.8 (Section 4.3).

```
1  struct Record_float4_float4 {
2    float4 a;
3    float4 b;
4  };
5
6  __kernel
7  void harris(global float* restrict output, int n0, int n1, const global float* restrict x0,
8              global float* restrict t1, global float* restrict t2, global float* restrict t3) {
9    for (int gid = get_global_id(0);(gid < (n0 / 32));gid = (gid + get_global_size(0))) {
```

```
10      for (int i0 = 0;(i0 < 2);i0 = (1 + i0)) {
11        for (int i1 = 0;(i1 < (n1 / 4));i1 = (1 + i1)) {
12          float4 t4 = (float4)(0.0f);
13          t4 = t4 + (float4)(0.299f) * vload4(0, &x0[4*i1 + 32*gid*n1 + i0*n1]);
14          t4 = t4 + (float4)(0.587f) * vload4(0, &x0[4*i1 + 4*n1 + 32*gid*n1 + i0*n1 + n0*n1]);
15          t4 = t4 + (float4)(0.114f) * vload4(0, &x0[2*n0*n1 + 4*i1 + 8*n1 + 32*gid*n1 + i0*n1]);
16          vstore4(t4, 0, &t3[3*n1*get_global_id(0) + 4*i0 + 4*i1 + 12*get_global_id(0) + i0*n1]);
17        }
18      }
19
20      for (int i2 = 0;(i2 < 2);i2 = (1 + i2)) {
21        for (int i3 = 0;(i3 < (n1 / 4));i3 = (1 + i3)) {
22          float4 t4 = (float4)(0.0f);
23          t4 = t4 + ((float4)(0.299f) * vload4(0, &x0[((((2 * n1) + (4 * i3)) + ((32 * gid) * n1)) + (i2 * n1))]));
24          t4 = t4 + ((float4)(0.587f) * vload4(0, &x0[(((((4 * i3) + (6 * n1)) + ((32 * gid) * n1)) + (i2 * n1)) + (n0 * n1))]));
25          t4 = t4 + ((float4)(0.114f) * vload4(0, &x0[(((2 * n0) * n1) + (4 * i3) + (10 * n1) + ((32 * gid) * n1) + (i2 * n1))]));
26          vstore4(t4, 0, &t3[3*n1*get_global_id(0) + 4*i3 + 4*((2 + i2) % 3) + 12*get_global_id(0) + n1*((2 + i2) % 3)]);
27        }
28
29        for (int i4 = 0;(i4 < (n1 / 4));i4 = (1 + i4)) {
30          float4 t5[6];
31          t5[0] = vload4(0, &t3[((3 * n1) * get_global_id(0)) + (4 * i2) + (4 * i4) + (12 * get_global_id(0)) + (i2 * n1)]);
32          t5[1] = vload4(0, &t3[(4 + (3 * n1 * get_global_id(0))) + (4 * i2) + (4 * i4) + (12 * get_global_id(0)) + (i2 * n1)]);
33          t5[2] = vload4(0, &t3[4 + n1 + 3*n1*get_global_id(0) + 4*i2 + 4*i4 + 12*get_global_id(0) + i2*n1]);
34          t5[3] = vload4(0, &t3[8 + n1 + 3*n1*get_global_id(0) + 4*i2 + 4*i4 + 12*get_global_id(0) + i2*n1]);
35          t5[4] = vload4(0, &t3[3*n1*get_global_id(0) + 4*i4 + 4*((2 + i2) % 3) + 12*get_global_id(0) + n1*((2 + i2) % 3)]);
36          t5[5] = vload4(0, &t3[4 + 3*n1*get_global_id(0) + 4*i4 + 4*((2 + i2) % 3) + 12*get_global_id(0) + n1*((2 + i2) % 3)]);
37
38          float4 t6 = (float4)(0.0f);
39          t6 = (t6 + ((float4)-0.083333336f * (float4)(t5[0].s0, t5[0].s1, t5[0].s2, t5[0].s3)));
40          t6 = (t6 + ((float4)0.0f * (float4)(t5[0].s1, t5[0].s2, t5[0].s3, t5[1].s0)));
41          t6 = (t6 + ((float4)0.083333336f * (float4)(t5[0].s2, t5[0].s3, t5[1].s0, t5[1].s1)));
42          t6 = (t6 + ((float4)-0.16666667f * (float4)(t5[2].s0, t5[2].s1, t5[2].s2, t5[2].s3)));
43          t6 = (t6 + ((float4)0.0f * (float4)(t5[2].s1, t5[2].s2, t5[2].s3, t5[3].s0)));
44          t6 = (t6 + ((float4)0.16666667f * (float4)(t5[2].s2, t5[2].s3, t5[3].s0, t5[3].s1)));
45          t6 = (t6 + ((float4)-0.083333336f * (float4)(t5[4].s0, t5[4].s1, t5[4].s2, t5[4].s3)));
46          t6 = (t6 + ((float4)0.0f * (float4)(t5[4].s1, t5[4].s2, t5[4].s3, t5[5].s0)));
47          t6 = (t6 + ((float4)0.083333336f * (float4)(t5[4].s2, t5[4].s3, t5[5].s0, t5[5].s1)));
48          vstore4(t6, 0, &t2[((((((3 * n1) * get_global_id(0)) + (4 * i2)) + (4 * i4)) + (12 * get_global_id(0))) + (i2 * n1))]);
49
50          float4 t7 = (float4)(0.0f);
51          t7 = (t7 + ((float4)-0.083333336f * (float4)(t5[0].s0, t5[0].s1, t5[0].s2, t5[0].s3)));
52          t7 = (t7 + ((float4)-0.16666667f * (float4)(t5[0].s1, t5[0].s2, t5[0].s3, t5[1].s0)));
53          t7 = (t7 + ((float4)-0.083333336f * (float4)(t5[0].s2, t5[0].s3, t5[1].s0, t5[1].s1)));
54          t7 = (t7 + ((float4)0.0f * (float4)(t5[2].s0, t5[2].s1, t5[2].s2, t5[2].s3)));
55          t7 = (t7 + ((float4)0.0f * (float4)(t5[2].s1, t5[2].s2, t5[2].s3, t5[3].s0)));
56          t7 = (t7 + ((float4)0.0f * (float4)(t5[2].s2, t5[2].s3, t5[3].s0, t5[3].s1)));
57          t7 = (t7 + ((float4)0.083333336f * (float4)(t5[4].s0, t5[4].s1, t5[4].s2, t5[4].s3)));
58          t7 = (t7 + ((float4)0.16666667f * (float4)(t5[4].s1, t5[4].s2, t5[4].s3, t5[5].s0)));
59          t7 = (t7 + ((float4)0.083333336f * (float4)(t5[4].s2, t5[4].s3, t5[5].s0, t5[5].s1)));
60          vstore4(t7, 0, &t1[((((((3 * n1) * get_global_id(0)) + (4 * i2)) + (4 * i4)) + (12 * get_global_id(0))) + (i2 * n1))]);
61        }
62      }
63
64      for (int i5 = 0;(i5 < 32);i5 = (1 + i5)) {
65        for (int i6 = 0;(i6 < (n1 / 4));i6 = (1 + i6)) {
66          float4 t8 = (float4)(0.0f);
67          t8 = (t8 + ((float4)(0.299f) * vload4(0, &x0[4*i6 + 4*n1 + 32*gid*n1 + i5*n1])));
68          t8 = (t8 + ((float4)(0.587f) * vload4(0, &x0[4*i6 + 8*n1 + 32*gid*n1 + i5*n1 + n0*n1])));
69          t8 = (t8 + ((float4)(0.114f) * vload4(0, &x0[2*n0*n1 + 4*i6 + 12*n1 + 32*gid*n1 + i5*n1])));
70          vstore4(t8, 0, &t3[3*n1*get_global_id(0) + 4*i6 + 4*((1 + i5) % 3) + 12*get_global_id(0) + n1*((1 + i5) % 3)]);
71        }
72
73        for (int i7 = 0;(i7 < (n1 / 4));i7 = (1 + i7)) {
74          float4 t5[6];
75          t5[0] = vload4(0, &t3[3*n1*get_global_id(0) + 4*i7 + 4*((2 + i5) % 3) + 12*get_global_id(0) + n1*((2 + i5) % 3)]);
76          t5[1] = vload4(0, &t3[4 + 3*n1*get_global_id(0) + 4*i7 + 4*((2 + i5) % 3) + 12*get_global_id(0) + n1*((2 + i5) % 3)]);
77          t5[2] = vload4(0, &t3[3*n1*get_global_id(0) + 4*i7 + 4*(i5 % 3) + 12*get_global_id(0) + n1*(i5 % 3)]);
78          t5[3] = vload4(0, &t3[4 + 3*n1*get_global_id(0) + 4*i7 + 4*(i5 % 3) + 12*get_global_id(0) + n1*(i5 % 3)]);
79          t5[4] = vload4(0, &t3[3*n1*get_global_id(0) + 4*i7 + 4*((1 + i5) % 3) + 12*get_global_id(0) + n1*((1 + i5) % 3)]);
80          t5[5] = vload4(0, &t3[4 + 3*n1*get_global_id(0) + 4*i7 + 4*((1 + i5) % 3) + 12*get_global_id(0) + n1*((1 + i5) % 3)]);
81
82          float4 t6 = (float4)(0.0f);
83          t6 = (t6 + ((float4)-0.083333336f * (float4)(t5[0].s0, t5[0].s1, t5[0].s2, t5[0].s3)));
84          t6 = (t6 + ((float4)0.0f * (float4)(t5[0].s1, t5[0].s2, t5[0].s3, t5[1].s0)));
85          t6 = (t6 + ((float4)0.083333336f * (float4)(t5[0].s2, t5[0].s3, t5[1].s0, t5[1].s1)));
86          t6 = (t6 + ((float4)-0.16666667f * (float4)(t5[2].s0, t5[2].s1, t5[2].s2, t5[2].s3)));
87          t6 = (t6 + ((float4)0.0f * (float4)(t5[2].s1, t5[2].s2, t5[2].s3, t5[3].s0)));
88          t6 = (t6 + ((float4)0.16666667f * (float4)(t5[2].s2, t5[2].s3, t5[3].s0, t5[3].s1)));
```

```
89      t6 = (t6 + ((float4)-0.083333336f * (float4)(t5[4].s0, t5[4].s1, t5[4].s2, t5[4].s3)));
90      t6 = (t6 + ((float4)0.0f * (float4)(t5[4].s1, t5[4].s2, t5[4].s3, t5[5].s0)));
91      t6 = (t6 + ((float4)0.083333336f * (float4)(t5[4].s2, t5[4].s3, t5[5].s0, t5[5].s1)));
92      vstore4(t6, 0, &t2[3*n1*get_global_id(0) + 4*i7 + 4*((2 + i5) % 3) + 12*get_global_id(0) + n1*((2 + i5) % 3)]);
93
94      float4 t7 = (float4)(0.0f);
95      t7 = (t7 + ((float4)-0.083333336f * (float4)(t5[0].s0, t5[0].s1, t5[0].s2, t5[0].s3)));
96      t7 = (t7 + ((float4)-0.16666667f * (float4)(t5[0].s1, t5[0].s2, t5[0].s3, t5[1].s0)));
97      t7 = (t7 + ((float4)-0.083333336f * (float4)(t5[0].s2, t5[0].s3, t5[1].s0, t5[1].s1)));
98      t7 = (t7 + ((float4)0.0f * (float4)(t5[2].s0, t5[2].s1, t5[2].s2, t5[2].s3)));
99      t7 = (t7 + ((float4)0.0f * (float4)(t5[2].s1, t5[2].s2, t5[2].s3, t5[3].s0)));
100     t7 = (t7 + ((float4)0.0f * (float4)(t5[2].s2, t5[2].s3, t5[3].s0, t5[3].s1)));
101     t7 = (t7 + ((float4)0.083333336f * (float4)(t5[4].s0, t5[4].s1, t5[4].s2, t5[4].s3)));
102     t7 = (t7 + ((float4)0.16666667f * (float4)(t5[4].s1, t5[4].s2, t5[4].s3, t5[5].s0)));
103     t7 = (t7 + ((float4)0.083333336f * (float4)(t5[4].s2, t5[4].s3, t5[5].s0, t5[5].s1)));
104     vstore4(t7, 0, &t1[3*n1*get_global_id(0) + 4*i7 + 4*((2 + i5) % 3) + 12*get_global_id(0) + n1*((2 + i5) % 3)]);
105   }
106
107   for (int i8 = 0;(i8 < (n1 / 4));i8 = (1 + i8)) {
108     struct Record_float4_float4 t9[6];
109     t9[0].a = vload4(0, &t2[3*n1*get_global_id(0) + 4*i8 + 4*(i5 % 3) + 12*get_global_id(0) + n1*(i5 % 3)]);
110     t9[0].b = vload4(0, &t1[3*n1*get_global_id(0) + 4*i8 + 4*(i5 % 3) + 12*get_global_id(0) + n1*(i5 % 3)]);
111     t9[1].a = vload4(0, &t2[4 + 3*n1*get_global_id(0) + 4*i8 + 4*(i5 % 3) + 12*get_global_id(0) + n1*(i5 % 3)]);
112     t9[1].b = vload4(0, &t1[4 + 3*n1*get_global_id(0) + 4*i8 + 4*(i5 % 3) + 12*get_global_id(0) + n1*(i5 % 3)]);
113     t9[2].a = vload4(0, &t2[3*n1*get_global_id(0) + 4*i8 + 4*((1+i5) % 3) + 12*get_global_id(0) + n1*((1+i5) % 3)]);
114     t9[2].b = vload4(0, &t1[3*n1*get_global_id(0) + 4*i8 + 4*((1+i5) % 3) + 12*get_global_id(0) + n1*((1+i5) % 3)]);
115     t9[3].a = vload4(0, &t2[4 + 3*n1*get_global_id(0) + 4*i8 + 4*((1+i5) % 3) + 12*get_global_id(0) + n1*((1+i5) % 3)]);
116     t9[3].b = vload4(0, &t1[4 + 3*n1*get_global_id(0) + 4*i8 + 4*((1+i5) % 3) + 12*get_global_id(0) + n1*((1+i5) % 3)]);
117     t9[4].a = vload4(0, &t2[3*n1*get_global_id(0) + 4*i8 + 4*((2+i5) % 3) + 12*get_global_id(0) + n1*((2+i5) % 3)]);
118     t9[4].b = vload4(0, &t1[3*n1*get_global_id(0) + 4*i8 + 4*((2+i5) % 3) + 12*get_global_id(0) + n1*((2+i5) % 3)]);
119     t9[5].a = vload4(0, &t2[4 + 3*n1*get_global_id(0) + 4*i8 + 4*((2+i5) % 3) + 12*get_global_id(0) + n1*((2+i5) % 3)]);
120     t9[5].b = vload4(0, &t1[4 + 3*n1*get_global_id(0) + 4*i8 + 4*((2+i5) % 3) + 12*get_global_id(0) + n1*((2+i5) % 3)]);
121
122     float4 t10 = (float4)(0.0f);
123     t10 += (float4)(t9[0].a.s0,t9[0].a.s1,t9[0].a.s2,t9[0].a.s3) * (float4)(t9[0].b.s0,t9[0].b.s1,t9[0].b.s2,t9[0].b.s3);
124     t10 += (float4)(t9[0].a.s1,t9[0].a.s2,t9[0].a.s3,t9[1].a.s0) * (float4)(t9[0].b.s1,t9[0].b.s2,t9[0].b.s3,t9[1].b.s0);
125     t10 += (float4)(t9[0].a.s2,t9[0].a.s3,t9[1].a.s0,t9[1].a.s1) * (float4)(t9[0].b.s2,t9[0].b.s3,t9[1].b.s0,t9[1].b.s1);
126     t10 += (float4)(t9[2].a.s0,t9[2].a.s1,t9[2].a.s2,t9[2].a.s3) * (float4)(t9[2].b.s0,t9[2].b.s1,t9[2].b.s2,t9[2].b.s3);
127     t10 += (float4)(t9[2].a.s1,t9[2].a.s2,t9[2].a.s3,t9[3].a.s0) * (float4)(t9[2].b.s1,t9[2].b.s2,t9[2].b.s3,t9[3].b.s0);
128     t10 += (float4)(t9[2].a.s2,t9[2].a.s3,t9[3].a.s0,t9[3].a.s1) * (float4)(t9[2].b.s2,t9[2].b.s3,t9[3].b.s0,t9[3].b.s1);
129     t10 += (float4)(t9[4].a.s0,t9[4].a.s1,t9[4].a.s2,t9[4].a.s3) * (float4)(t9[4].b.s0,t9[4].b.s1,t9[4].b.s2,t9[4].b.s3);
130     t10 += (float4)(t9[4].a.s1,t9[4].a.s2,t9[4].a.s3,t9[5].a.s0) * (float4)(t9[4].b.s1,t9[4].b.s2,t9[4].b.s3,t9[5].b.s0);
131     t10 += (float4)(t9[4].a.s2,t9[4].a.s3,t9[5].a.s0,t9[5].a.s1) * (float4)(t9[4].b.s2,t9[4].b.s3,t9[5].b.s0,t9[5].b.s1);
132
133     float4 t12 = (float4)(0.0f);
134     t12 += (float4)(t9[0].b.s0,t9[0].b.s1,t9[0].b.s2,t9[0].b.s3) * (float4)(t9[0].b.s0,t9[0].b.s1,t9[0].b.s2,t9[0].b.s3);
135     t12 += (float4)(t9[0].b.s1,t9[0].b.s2,t9[0].b.s3,t9[1].b.s0) * (float4)(t9[0].b.s1,t9[0].b.s2,t9[0].b.s3,t9[1].b.s0);
136     t12 += (float4)(t9[0].b.s2,t9[0].b.s3,t9[1].b.s0,t9[1].b.s1) * (float4)(t9[0].b.s2,t9[0].b.s3,t9[1].b.s0,t9[1].b.s1);
137     t12 += (float4)(t9[2].b.s0,t9[2].b.s1,t9[2].b.s2,t9[2].b.s3) * (float4)(t9[2].b.s0,t9[2].b.s1,t9[2].b.s2,t9[2].b.s3);
138     t12 += (float4)(t9[2].b.s1,t9[2].b.s2,t9[2].b.s3,t9[3].b.s0) * (float4)(t9[2].b.s1,t9[2].b.s2,t9[2].b.s3,t9[3].b.s0);
139     t12 += (float4)(t9[2].b.s2,t9[2].b.s3,t9[3].b.s0,t9[3].b.s1) * (float4)(t9[2].b.s2,t9[2].b.s3,t9[3].b.s0,t9[3].b.s1);
140     t12 += (float4)(t9[4].b.s0,t9[4].b.s1,t9[4].b.s2,t9[4].b.s3) * (float4)(t9[4].b.s0,t9[4].b.s1,t9[4].b.s2,t9[4].b.s3);
141     t12 += (float4)(t9[4].b.s1,t9[4].b.s2,t9[4].b.s3,t9[5].b.s0) * (float4)(t9[4].b.s1,t9[4].b.s2,t9[4].b.s3,t9[5].b.s0);
142     t12 += (float4)(t9[4].b.s2,t9[4].b.s3,t9[5].b.s0,t9[5].b.s1) * (float4)(t9[4].b.s2,t9[4].b.s3,t9[5].b.s0,t9[5].b.s1);
143
144     float4 t14 = (float4)(0.0f);
145     t14 += (float4)(t9[0].a.s0,t9[0].a.s1,t9[0].a.s2,t9[0].a.s3) * (float4)(t9[0].a.s0,t9[0].a.s1,t9[0].a.s2,t9[0].a.s3);
146     t14 += (float4)(t9[0].a.s1,t9[0].a.s2,t9[0].a.s3,t9[1].a.s0) * (float4)(t9[0].a.s1,t9[0].a.s2,t9[0].a.s3,t9[1].a.s0);
147     t14 += (float4)(t9[0].a.s2,t9[0].a.s3,t9[1].a.s0,t9[1].a.s1) * (float4)(t9[0].a.s2,t9[0].a.s3,t9[1].a.s0,t9[1].a.s1);
148     t14 += (float4)(t9[2].a.s0,t9[2].a.s1,t9[2].a.s2,t9[2].a.s3) * (float4)(t9[2].a.s0,t9[2].a.s1,t9[2].a.s2,t9[2].a.s3);
149     t14 += (float4)(t9[2].a.s1,t9[2].a.s2,t9[2].a.s3,t9[3].a.s0) * (float4)(t9[2].a.s1,t9[2].a.s2,t9[2].a.s3,t9[3].a.s0);
150     t14 += (float4)(t9[2].a.s2,t9[2].a.s3,t9[3].a.s0,t9[3].a.s1) * (float4)(t9[2].a.s2,t9[2].a.s3,t9[3].a.s0,t9[3].a.s1);
151     t14 += (float4)(t9[4].a.s0,t9[4].a.s1,t9[4].a.s2,t9[4].a.s3) * (float4)(t9[4].a.s0,t9[4].a.s1,t9[4].a.s2,t9[4].a.s3);
152     t14 += (float4)(t9[4].a.s1,t9[4].a.s2,t9[4].a.s3,t9[5].a.s0) * (float4)(t9[4].a.s1,t9[4].a.s2,t9[4].a.s3,t9[5].a.s0);
153     t14 += (float4)(t9[4].a.s2,t9[4].a.s3,t9[5].a.s0,t9[5].a.s1) * (float4)(t9[4].a.s2,t9[4].a.s3,t9[5].a.s0,t9[5].a.s1);
154
155     vstore4(t14*t12 - t10*t10 - (float4)(0.04f) * (t14 + t12) * (t14 + t12), 0, &output[4*i8 + 32*gid*n1 + i5*n1]);
156   }
157   }
158  }
159 }
```

Listing A.3: OpenCL code generated by SHINE for the cbuf Harris corner detection (Listing 4.8).

A.2   Beyond Halide Reference

The OpenCL code generated by Shine after rewriting the Rise Harris corner detection from Listing 4.6 (Section 4.3) using the cbuf+rrot Elevate strategy from Listing 4.12 (Section 4.3).

```
1   struct Record_float4_float4 {
2     float4 a;
3     float4 b;
4   };
5
6   struct Record_float4__float4_float4_ {
7     float4 a;
8     struct Record_float4_float4 b;
9   };
10
11  __kernel
12  void harris(global float* restrict output, int n0, int n1, const global float* restrict x0,
13             global float* restrict t1, global float* restrict t2, global float* restrict t3){
14    for (int gid = get_global_id(0);(gid < (n0 / 32));gid = (gid + get_global_size(0))) {
15      for (int i0 = 0;(i0 < 2);i0 = (1 + i0)) {
16        for (int i1 = 0;(i1 < (n1 / 4));i1 = (1 + i1)) {
17          float4 t4 = (float4)(0.0f);
18          t4 = (t4 + ((float4)(0.299f) * vload4(0, &x0[4*i1 + 32*gid*n1 + i0*n1))]);
19          t4 = (t4 + ((float4)(0.587f) * vload4(0, &x0[4*i1 + 4*n1 + 32*gid*n1 + i0*n1])));
20          t4 = (t4 + ((float4)(0.114f) * vload4(0, &x0[2*n0*n1 + 4*i1 + 8*n1 + 32*gid*n1 + i0*n1])));
21          vstore4(t4, 0, &t3[3*n1*get_global_id(0) + 4*i0 + 4*i1 + 12*get_global_id(0) + i0*n1]);
22        }
23      }
24
25      for (int i2 = 0;(i2 < 2);i2 = (1 + i2)) {
26        for (int i3 = 0;(i3 < (n1 / 4));i3 = (1 + i3)) {
27          float4 t4 = (float4)(0.0f);
28          t4 = (t4 + ((float4)(0.299f) * vload4(0, &x0[2*n1 + 4*i3 + 32*gid*n1 + i2*n1])));
29          t4 = (t4 + ((float4)(0.587f) * vload4(0, &x0[4*i3 + 6*n1 + 32*gid*n1 + i2*n1 + n0*n1])));
30          t4 = (t4 + ((float4)(0.114f) * vload4(0, &x0[2*n0*n1 + 4*i3 + 10*n1 + 32*gid*n1 + i2*n1])));
31          vstore4(t4, 0, &t3[3*n1*get_global_id(0) + 4*i3 + 4*((2 + i2) % 3) + 12*get_global_id(0) + n1*((2 + i2) % 3)]);
32        }
33
34        struct Record_float4_float4 t16[2];
35
36        float4 t5 = (float4)(0.0f);
37        t5 += (float4)(1.0f) * vload4(0, &t3[3*n1*get_global_id(0) + 4*i2 + 12*get_global_id(0) + i2*n1]);
38        t5 += (float4)(2.0f) * vload4(0, &t3[4 + n1 + 3*n1*get_global_id(0) + 4*i2 + 12*get_global_id(0) + i2*n1]);
39        t5 += (float4)(1.0f) * vload4(0, &t3[3*n1*get_global_id(0) + 4*((2+i2) % 3) + 12*get_global_id(0) + n1*((2+i2) % 3)]);
40        t16[0].a = t5;
41
42        float4 t8 = (float4)(0.0f);
43        t8 += (float4)(-1.0f) * vload4(0, &t3[3*n1*get_global_id(0) + 4*i2 + 12*get_global_id(0) + i2*n1]);
44        t8 += (float4)(0.0f) * vload4(0, &t3[4 + n1 + 3*n1*get_global_id(0) + 4*i2 + 12*get_global_id(0) + i2*n1]);
45        t8 += (float4)(1.0f) * vload4(0, &t3[3*n1*get_global_id(0) + 4*((2+i2) % 3) + 12*get_global_id(0) + n1*((2+i2) % 3)]);
46        t16[0].b = t8;
47
48        for (int i4 = 0;(i4 < (n1 / 4));i4 = (1 + i4)) {
49          float4 t9 = (float4)(0.0f);
50          t9 += (float4)(1.0f) * vload4(0, &t3[4 + 3*n1*get_global_id(0) + 4*i2 + 4*i4 + 12*get_global_id(0) + i2*n1]);
51          t9 += (float4)(2.0f) * vload4(0, &t3[8 + n1 + 3*n1 * get_global_id(0) + 4*i2 + 4*i4 + 12*get_global_id(0) + i2*n1]);
52          t9 += (float4)(1.0f) * vload4(0, &t3[4 + 3*n1*get_global_id(0) + 4*i4 + 4*((2+i2) % 3) + 12*get_global_id(0) + n1*((2+
                 ↪ i2) % 3)]);
53          t16[1].a = t9;
54
55          float4 t12 = (float4)(0.0f);
56          t12 += (float4)(-1.0f) * vload4(0, &t3[4 + 3*n1*get_global_id(0) + 4*i2 + 4*i4 + 12*get_global_id(0) + i2*n1]);
57          t12 += (float4)(0.0f) * vload4(0, &t3[8 + n1 + 3*n1*get_global_id(0) + 4*i2 + 4*i4 + 12*get_global_id(0) + i2*n1]);
58          t12 += (float4)(1.0f) * vload4(0, &t3[4 + 3*n1*get_global_id(0) + 4*i4 + 4*((2+i2) % 3) + 12*get_global_id(0) + n1*((2+
                 ↪ i2) % 3)]);
59          t16[1].b = t12;
60
61          float4 t13 = (float4)(0.0f);
62          t13 = (t13 + ((float4)(-0.083333336f) * (float4)(t16[0].a.s0, t16[0].a.s1, t16[0].a.s2, t16[0].a.s3)));
63          t13 = (t13 + ((float4)(0.0f) * (float4)(t16[0].a.s1, t16[0].a.s2, t16[0].a.s3, t16[1].a.s0)));
64          t13 = (t13 + ((float4)(0.083333336f) * (float4)(t16[0].a.s2, t16[0].a.s3, t16[1].a.s0, t16[1].a.s1)));
65          vstore4(t13, 0, &t2[3*n1*get_global_id(0) + 4*i2 + 4*i4 + 12*get_global_id(0) + i2*n1]);
66
67          float4 t14 = (float4)(0.0f);
68          t14 = (t14 + ((float4)(0.083333336f) * (float4)(t16[0].b.s0, t16[0].b.s1, t16[0].b.s2, t16[0].b.s3)));
69          t14 = (t14 + ((float4)(0.16666667f) * (float4)(t16[0].b.s1, t16[0].b.s2, t16[0].b.s3, t16[1].b.s0)));
```

```
70      t14 = (t14 + ((float4)(0.083333336f) * (float4)(t16[0].b.s2, t16[0].b.s3, t16[1].b.s0, t16[1].b.s1)));
71      vstore4(t14, 0, &t1[3*n1*get_global_id(0) + 4*i2 + 4*i4 + 12*get_global_id(0) + i2*n1]);
72
73      t16[0].a = t16[1].a;
74      t16[0].b = t16[1].b;
75    }
76  }
77
78  for (int i5 = 0;(i5 < 32);i5 = (1 + i5)) {
79    for (int i6 = 0;(i6 < (n1 / 4));i6 = (1 + i6)) {
80      float4 t15 = (float4)(0.0f);
81      t15 = (t15 + ((float4)(0.299f) * vload4(0, &x0[4*i6 + 4*n1 + 32*gid*n1 + i5*n1])));
82      t15 = (t15 + ((float4)(0.587f) * vload4(0, &x0[4*i6 + 8*n1 + 32*gid*n1 + i5*n1 + n0*n1])));
83      t15 = (t15 + ((float4)(0.114f) * vload4(0, &x0[2*n0*n1 + 4*i6 + 12*n1 + 32*gid*n1 + i5*n1])));
84      vstore4(t15, 0, &t3[3*n1*get_global_id(0) + 4*i6 + 4*((1 + i5) % 3) + 12*get_global_id(0) + n1*((1 + i5) % 3)]);
85    }
86
87    struct Record_float4_float4 t16[2];
88
89    float4 t5 = (float4)(0.0f);
90    t5 += (float4)(1.0f) * vload4(0, &t3[3*n1*get_global_id(0) + 4*((2+i5) % 3) + 12*get_global_id(0) + n1*((2+i5) % 3)]);
91    t5 += (float4)(2.0f) * vload4(0, &t3[3*n1*get_global_id(0) + 4*(i5 % 3) + 12*get_global_id(0) + n1*(i5 % 3)]);
92    t5 += (float4)(1.0f) * vload4(0, &t3[3*n1*get_global_id(0) + 4*((1+i5) % 3) + 12*get_global_id(0) + n1*((1+i5) % 3)]);
93    t16[0].a = t5;
94
95    float4 t18 = (float4)(0.0f);
96    t18 += (float4)(-1.0f) * vload4(0, &t3[3*n1*get_global_id(0) + 4*((2+i5) % 3) + 12*get_global_id(0) + n1*((2+i5) % 3)]);
97    t18 += (float4)(0.0f) * vload4(0, &t3[3*n1*get_global_id(0) + 4*(i5 % 3) + 12*get_global_id(0) + n1*(i5 % 3)]);
98    t18 += (float4)(1.0f) * vload4(0, &t3[3*n1*get_global_id(0) + 4*((1+i5) % 3) + 12*get_global_id(0) + n1*((1+i5) % 3)]);
99    t16[0].b = t18;
100
101   for (int i7 = 0;(i7 < (n1 / 4));i7 = (1 + i7)) {
102     float4 t9 = (float4)(0.0f);
103     t9 += (float4)(1.0f) * vload4(0, &t3[4 + 3*n1*get_global_id(0) + 4*i7 + 4*((2 + i5) % 3) + 12*get_global_id(0) + n1*((2
          ↪  + i5) % 3)]);
104     t9 += (float4)(2.0f) * vload4(0, &t3[4 + 3*n1*get_global_id(0) + 4*i7 + 4*(i5 % 3) + 12*get_global_id(0) + n1*(i5 % 3)
          ↪  ]);
105     t9 += (float4)(1.0f) * vload4(0, &t3[4 + 3*n1*get_global_id(0) + 4*i7 + 4*((1 + i5) % 3) + 12*get_global_id(0) + n1*((1
          ↪  + i5) % 3)]);
106     t16[1].a = t9;
107
108     float4 t20 = (float4)(0.0f);
109     t20 = (t20 + ((float4)(-1.0f) * vload4(0, &t3[4 + 3*n1*get_global_id(0) + 4*i7 + 4*((2 + i5) % 3) + 12*get_global_id(0)
          ↪  + n1*((2 + i5) % 3)])));
110     t20 = (t20 + ((float4)(0.0f) * vload4(0, &t3[4 + 3*n1*get_global_id(0) + 4*i7 + 4*(i5 % 3) + 12*get_global_id(0) + n1*(
          ↪  i5 % 3)])));
111     t20 = (t20 + ((float4)(1.0f) * vload4(0, &t3[4 + 3*n1*get_global_id(0) + 4*i7 + 4*((1 + i5) % 3) + 12*get_global_id(0)
          ↪  + n1*((1 + i5) % 3)])));
112     t16[1].b = t20;
113
114     float4 t13 = (float4)(0.0f);
115     t13 = (t13 + ((float4)(-0.083333336f) * (float4)(t16[0].a.s0, t16[0].a.s1, t16[0].a.s2, t16[0].a.s3)));
116     t13 = (t13 + ((float4)(0.0f) * (float4)(t16[0].a.s1, t16[0].a.s2, t16[0].a.s3, t16[1].a.s0)));
117     t13 = (t13 + ((float4)(0.083333336f) * (float4)(t16[0].a.s2, t16[0].a.s3, t16[1].a.s0, t16[1].a.s1)));
118     vstore4(t13, 0, &t2[3*n1*get_global_id(0) + 4*i7 + 4*((2 + i5) % 3) + 12*get_global_id(0) + n1*((2 + i5) % 3)]);
119
120     float4 t14 = (float4)(0.0f);
121     t14 = (t14 + ((float4)(0.083333336f) * (float4)(t16[0].b.s0, t16[0].b.s1, t16[0].b.s2, t16[0].b.s3)));
122     t14 = (t14 + ((float4)(0.16666667f) * (float4)(t16[0].b.s1, t16[0].b.s2, t16[0].b.s3, t16[1].b.s0)));
123     t14 = (t14 + ((float4)(0.083333336f) * (float4)(t16[0].b.s2, t16[0].b.s3, t16[1].b.s0, t16[1].b.s1)));
124     vstore4(t14, 0, &t1[3*n1*get_global_id(0) + 4*i7 + 4*((2 + i5) % 3) + 12*get_global_id(0) + n1*((2 + i5) % 3)]);
125
126     t16[0].a = t16[1].a;
127     t16[0].b = t16[1].b;
128   }
129
130   struct Record_float4__float4_float4_ t21[2];
131
132   float4 t33 = (float4)(0.0f);
133   t33 = (t33 + (vload4(0, &t2[3*n1*get_global_id(0) + 4*(i5 % 3) + 12*get_global_id(0) + n1*(i5 % 3)]) * vload4(0, &t2[3*n1
          ↪  *get_global_id(0) + 4*(i5 % 3) + 12*get_global_id(0) + n1*(i5 % 3)]);
134   t33 = (t33 + (vload4(0, &t2[3*n1*get_global_id(0) + 4*((1 + i5) % 3) + 12*get_global_id(0) + n1*((1 + i5) % 3)]) * vload4
          ↪  (0, &t2[3*n1*get_global_id(0) + 4*((1 + i5) % 3) + 12*get_global_id(0) + n1*((1 + i5) % 3)])));
135   t33 = (t33 + (vload4(0, &t2[3*n1*get_global_id(0) + 4*((2 + i5) % 3) + 12*get_global_id(0) + n1*((2 + i5) % 3)]) * vload4
          ↪  (0, &t2[3*n1*get_global_id(0) + 4*((2 + i5) % 3) + 12*get_global_id(0) + n1*((2 + i5) % 3)])));
136   t21[0].a = t33;
137
138   float4 t22 = (float4)(0.0f);
139   t22 = (t22 + (vload4(0, &t2[3*n1*get_global_id(0) + 4*(i5 % 3) + 12*get_global_id(0) + n1*(i5 % 3)]) * vload4(0, &t1[3*n1
```

```
            ↪ *get_global_id(0) + 4*(i5 % 3) + 12*get_global_id(0) + n1*(i5 % 3)])));
140     t22 = (t22 + (vload4(0, &t2[3*n1*get_global_id(0) + 4*((1 + i5) % 3) + 12*get_global_id(0) + n1*((1 + i5) % 3)]) * vload4
            ↪ (0, &t1[3*n1*get_global_id(0) + 4*((1 + i5) % 3) + 12*get_global_id(0) + n1*((1 + i5) % 3)])));
141     t22 = (t22 + (vload4(0, &t2[3*n1*get_global_id(0) + 4*((2 + i5) % 3) + 12*get_global_id(0) + n1*((2 + i5) % 3)]) * vload4
            ↪ (0, &t1[3*n1*get_global_id(0) + 4*((2 + i5) % 3) + 12*get_global_id(0) + n1*((2 + i5) % 3)])));
142     t21[0].b.a = t22;

144     float4 t23 = (float4)(0.0f);
145     t23 = (t23 + (vload4(0, &t1[3*n1*get_global_id(0) + 4*(i5 % 3) + 12*get_global_id(0) + n1*(i5 % 3)]) * vload4(0, &t1[3*n1
            ↪ *get_global_id(0) + 4*(i5 % 3) + 12*get_global_id(0) + n1*(i5 % 3)])));
146     t23 = (t23 + (vload4(0, &t1[3*n1*get_global_id(0) + 4*((1 + i5) % 3) + 12*get_global_id(0) + n1*((1 + i5) % 3)]) * vload4
            ↪ (0, &t1[3*n1*get_global_id(0) + 4*((1 + i5) % 3) + 12*get_global_id(0) + n1*((1 + i5) % 3)])));
147     t23 = (t23 + (vload4(0, &t1[3*n1*get_global_id(0) + 4*((2 + i5) % 3) + 12*get_global_id(0) + n1*((2 + i5) % 3)]) * vload4
            ↪ (0, &t1[3*n1*get_global_id(0) + 4*((2 + i5) % 3) + 12*get_global_id(0) + n1*((2 + i5) % 3)])));
148     t21[0].b.b = t23;

150     for (int i8 = 0;(i8 < (n1 / 4));i8 = (1 + i8)) {
151       float4 t24 = (float4)(0.0f);
152       t24 += vload4(0, &t2[4 + 3*n1*get_global_id(0) + 4*i8 + 4*(i5 % 3) + 12*get_global_id(0) + n1*(i5 % 3)]) * vload4(0, &
              ↪ t2[4 + 3*n1*get_global_id(0) + 4*i8 + 4*(i5 % 3) + 12*get_global_id(0) + n1*(i5 % 3)]);
153       t24 += vload4(0, &t2[4 + 3*n1*get_global_id(0) + 4*i8 + 4*((1+i5) % 3) + 12*get_global_id(0) + n1*((1+i5) % 3)]) *
              ↪ vload4(0, &t2[4 + 3*n1*get_global_id(0) + 4*i8 + 4*((1+i5) % 3) + 12*get_global_id(0) + n1*((1+i5) % 3)]);
154       t24 += vload4(0, &t2[4 + 3*n1*get_global_id(0) + 4*i8 + 4*((2+i5) % 3) + 12*get_global_id(0) + n1*((2+i5) % 3)]) *
              ↪ vload4(0, &t2[4 + 3*n1*get_global_id(0) + 4*i8 + 4*((2+i5) % 3) + 12*get_global_id(0) + n1*((2+i5) % 3)]);
155       t21[1].a = t24;

157       float4 t25 = (float4)(0.0f);
158       t25 += vload4(0, &t2[4 + 3*n1*get_global_id(0) + 4*i8 + 4*(i5 % 3) + 12*get_global_id(0) + n1*(i5 % 3)]) * vload4(0, &
              ↪ t1[4 + 3*n1*get_global_id(0) + 4*i8 + 4*(i5 % 3) + 12*get_global_id(0) + n1*(i5 % 3)]);
159       t25 += vload4(0, &t2[4 + 3*n1*get_global_id(0) + 4*i8 + 4*((1+i5) % 3) + 12*get_global_id(0) + n1*((1+i5) % 3)]) *
              ↪ vload4(0, &t1[4 + 3*n1*get_global_id(0) + 4*i8 + 4*((1+i5) % 3) + 12*get_global_id(0) + n1*((1+i5) % 3)]);
160       t25 += vload4(0, &t2[4 + 3*n1*get_global_id(0) + 4*i8 + 4*((2+i5) % 3) + 12*get_global_id(0) + n1*((2+i5) % 3)]) *
              ↪ vload4(0, &t1[4 + 3*n1*get_global_id(0) + 4*i8 + 4*((2+i5) % 3) + 12*get_global_id(0) + n1*((2+i5) % 3)]);
161       t21[1].b.a = t25;

163       float4 t26 = (float4)(0.0f);
164       t26 += vload4(0, &t1[4 + 3*n1*get_global_id(0) + 4*i8 + 4*(i5 % 3) + 12*get_global_id(0) + n1*(i5 % 3)]) * vload4(0, &
              ↪ t1[4 + 3*n1*get_global_id(0) + 4*i8 + 4*(i5 % 3) + 12*get_global_id(0) + n1*(i5 % 3)]);
165       t26 += vload4(0, &t1[4 + 3*n1*get_global_id(0) + 4*i8 + 4*((1+i5) % 3) + 12*get_global_id(0) + n1*((1+i5) % 3)]) *
              ↪ vload4(0, &t1[4 + 3*n1*get_global_id(0) + 4*i8 + 4*((1+i5) % 3) + 12*get_global_id(0) + n1*((1+i5) % 3)]);
166       t26 += vload4(0, &t1[4 + 3*n1*get_global_id(0) + 4*i8 + 4*((2+i5) % 3) + 12*get_global_id(0) + n1*((2+i5) % 3)]) *
              ↪ vload4(0, &t1[4 + 3*n1*get_global_id(0) + 4*i8 + 4*((2+i5) % 3) + 12*get_global_id(0) + n1*((2+i5) % 3)]);
167       t21[1].b.b = t26;

169       float4 t27 = (float4)(0.0f);
170       t27 = (t27 + (float4)(t21[0].b.a.s0, t21[0].b.a.s1, t21[0].b.a.s2, t21[0].b.a.s3));
171       t27 = (t27 + (float4)(t21[0].b.a.s1, t21[0].b.a.s2, t21[0].b.a.s3, t21[1].b.a.s0));
172       t27 = (t27 + (float4)(t21[0].b.a.s2, t21[0].b.a.s3, t21[1].b.a.s0, t21[1].b.a.s1));

174       float4 t29 = (float4)(0.0f);
175       t29 = (t29 + (float4)(t21[0].b.b.s0, t21[0].b.b.s1, t21[0].b.b.s2, t21[0].b.b.s3));
176       t29 = (t29 + (float4)(t21[0].b.b.s1, t21[0].b.b.s2, t21[0].b.b.s3, t21[1].b.b.s0));
177       t29 = (t29 + (float4)(t21[0].b.b.s2, t21[0].b.b.s3, t21[1].b.b.s0, t21[1].b.b.s1));

179       float4 t31 = (float4)(0.0f);
180       t31 = (t31 + (float4)(t21[0].a.s0, t21[0].a.s1, t21[0].a.s2, t21[0].a.s3));
181       t31 = (t31 + (float4)(t21[0].a.s1, t21[0].a.s2, t21[0].a.s3, t21[1].a.s0));
182       t31 = (t31 + (float4)(t21[0].a.s2, t21[0].a.s3, t21[1].a.s0, t21[1].a.s1));

184       vstore4(t31*t29 - t27*t27 - (float4)(0.04f) * (t31 + t29) * (t31 + t29), 0, &output[4*i8 + 32*gid*n1 + i5*n1]);

186       t21[0].a = t21[1].a;
187       t21[0].b.a = t21[1].b.a;
188       t21[0].b.b = t21[1].b.b;
189     }
190   }
191   }
192 }
```

Listing A.4: OpenCL code generated by SHINE for the cbuf+rrot Harris corner detection (Listing 4.12).

Appendix B

Optimised Matrix Multiplication

This appendix contains the C programs generated by SHINE after rewriting the RISE program for matrix multiplication using the 7 ELEVATE strategies from [3]. We apply trivial cosmetic changes to keep the code readable and compact, as in Appendix A.

The same programs are generated using sketch-guided equality saturation in Section 5.6, modulo variable names. The handwritten sketches from Table 5.6 of Section 5.6 are included in Appendix C.

```
1  void baseline(float* output, float* x0, float* x1) {
2    for (int i0 = 0;(i0 < 1024);i0 = (1 + i0)) {
3      for (int i1 = 0;(i1 < 1024);i1 = (1 + i1)) {
4        float t1 = 0.0f;
5        for (int i2 = 0;(i2 < 1024);i2 = (1 + i2)) {
6          t1 += x0[(i2 + (1024 * i0))] *
7                x1[(i1 + (1024 * i2))];
8        }
9
10       output[(i1 + (1024 * i0))] = t1;
11     }
12   }
13 }
```

Listing B.1: C code generated by SHINE for the *baseline* mat-mul in Section 5.6.

```
1  void blocking(float* output, float* x0, float* x1) {
2    for (int i0 = 0;(i0 < 32);i0 = (1 + i0)) {
3      for (int i1 = 0;(i1 < 32);i1 = (1 + i1)) {
4        float t1[1024];
5        for (int i2 = 0;(i2 < 32);i2 = (1 + i2)) {
6          for (int i3 = 0;(i3 < 32);i3 = (1 + i3)) {
7            t1[(i3 + (32 * i2))] = 0.0f;
8          }
9        }
10
11       for (int i4 = 0;(i4 < 256);i4 = (1 + i4)) {
12         float t2[1024];
13         for (int i5 = 0;(i5 < 32);i5 = (1 + i5)) {
14           for (int i6 = 0;(i6 < 32);i6 = (1 + i6)) {
15             t2[(i6 + (32 * i5))] = t1[(i6 + (32 * i5))];
16           }
17         }
18
19         for (int i7 = 0;(i7 < 4);i7 = (1 + i7)) {
20           for (int i8 = 0;(i8 < 32);i8 = (1 + i8)) {
21             for (int i9 = 0;(i9 < 32);i9 = (1 + i9)) {
22               t2[i9 + 32*i8] += x0[i7 + 4*i4 + 1024*i8 + 32768*i0] *
23                                 x1[i9 + 32*i1 + 1024*i7 + 4096*i4];
24             }
25           }
26         }
27
28         for (int i10 = 0;(i10 < 32);i10 = (1 + i10)) {
29           for (int i11 = 0;(i11 < 32);i11 = (1 + i11)) {
30             t1[(i11 + (32 * i10))] = t2[(i11 + (32 * i10))];
31           }
32         }
33       }
34
35       for (int i12 = 0;(i12 < 32);i12 = (1 + i12)) {
36         for (int i13 = 0;(i13 < 32);i13 = (1 + i13)) {
37           output[(((i13 + (32 * i1)) + (1024 * i12)) + (32768 * i0))] = t1[(i13 + (32 * i12))];
38         }
39       }
40     }
41   }
42 }
```

Listing B.2: C code generated by SHINE for the *blocking* mat-mul in Section 5.6.

```
1  void vectorization(float* output, float* x0, float* x1) {
2    for (int i0 = 0;(i0 < 32);i0 = (1 + i0)) {
3      for (int i1 = 0;(i1 < 32);i1 = (1 + i1)) {
4        float t1[1024];
5        for (int i2 = 0;(i2 < 32);i2 = (1 + i2)) {
6          for (int i3 = 0;(i3 < 32);i3 = (1 + i3)) {
7            t1[(i3 + (32 * i2))] = 0.0f;
8          }
9        }
10
11       for (int i4 = 0;(i4 < 256);i4 = (1 + i4)) {
12         float t2[1024];
13         for (int i5 = 0;(i5 < 32);i5 = (1 + i5)) {
14           for (int i6 = 0;(i6 < 32);i6 = (1 + i6)) {
15             t2[(i6 + (32 * i5))] = t1[(i6 + (32 * i5))];
16           }
17         }
18
19         for (int i7 = 0;(i7 < 4);i7 = (1 + i7)) {
20           for (int i8 = 0;(i8 < 32);i8 = (1 + i8)) {
21             #pragma omp simd
22             for (int i9 = 0;(i9 < 32);i9 = (1 + i9)) {
23               t2[i9 + 32*i8] += x0[i7 + 4*i4 + 1024*i8 + 32768*i0] *
24                                 x1[i9 + 32*i1 + 1024*i7 + 4096*i4];
25             }
26           }
27         }
28
29         for (int i10 = 0;(i10 < 32);i10 = (1 + i10)) {
30           for (int i11 = 0;(i11 < 32);i11 = (1 + i11)) {
31             t1[(i11 + (32 * i10))] = t2[(i11 + (32 * i10))];
32           }
33         }
34       }
35
36       for (int i12 = 0;(i12 < 32);i12 = (1 + i12)) {
37         for (int i13 = 0;(i13 < 32);i13 = (1 + i13)) {
38           output[(((i13 + (32 * i1)) + (1024 * i12)) + (32768 * i0))] = t1[(i13 + (32 * i12))];
39         }
40       }
41     }
42   }
43 }
```

Listing B.3: C code generated by SHINE for the *vectorisation* mat-mul in Section 5.6.

```c
1  void loop_permutation(float* output, float* x0, float* x1) {
2    for (int i0 = 0;(i0 < 32);i0 = (1 + i0)) {
3      for (int i1 = 0;(i1 < 32);i1 = (1 + i1)) {
4        float t1[1024];
5        for (int i2 = 0;(i2 < 32);i2 = (1 + i2)) {
6          for (int i3 = 0;(i3 < 32);i3 = (1 + i3)) {
7            t1[(i3 + (32 * i2))] = 0.0f;
8          }
9        }
10
11       for (int i4 = 0;(i4 < 256);i4 = (1 + i4)) {
12         for (int i5 = 0;(i5 < 32);i5 = (1 + i5)) {
13           float t2[32];
14           for (int i6 = 0;(i6 < 32);i6 = (1 + i6)) {
15             t2[i6] = t1[(i6 + (32 * i5))];
16           }
17
18           for (int i7 = 0;(i7 < 4);i7 = (1 + i7)) {
19             #pragma omp simd
20             for (int i8 = 0;(i8 < 32);i8 = (1 + i8)) {
21               t2[i8] += x0[i7 + 4*i4 + 1024*i5 + 32768*i0] *
22                         x1[i8 + 32*i1 + 1024*i7 + 4096*i4];
23             }
24           }
25
26           for (int i9 = 0;(i9 < 32);i9 = (1 + i9)) {
27             t1[(i9 + (32 * i5))] = t2[i9];
28           }
29         }
30       }
31
32       for (int i10 = 0;(i10 < 32);i10 = (1 + i10)) {
33         for (int i11 = 0;(i11 < 32);i11 = (1 + i11)) {
34           output[(((i11 + (32 * i1)) + (1024 * i10)) + (32768 * i0))] = t1[(i11 + (32 * i10))];
35         }
36       }
37     }
38   }
39 }
```

Listing B.4: C code generated by SHINE for the *loop-perm* mat-mul in Section 5.6.

```
1  void array_packing(float* output, float* x0, float* x1) {
2    float t1[1048576];
3    #pragma omp parallel for
4    for (int i0 = 0;(i0 < 32);i0 = (1 + i0)) {
5      for (int i1 = 0;(i1 < 1024);i1 = (1 + i1)) {
6        #pragma omp simd
7        for (int i2 = 0;(i2 < 32);i2 = (1 + i2)) {
8          t1[((i2 + (32 * i1)) + (32768 * i0))] = x1[((i2 + (32 * i0)) + (1024 * i1))];
9        }
10       }
11     }
12
13     for (int i3 = 0;(i3 < 32);i3 = (1 + i3)) {
14       for (int i4 = 0;(i4 < 32);i4 = (1 + i4)) {
15         float t2[1024];
16         for (int i5 = 0;(i5 < 32);i5 = (1 + i5)) {
17           for (int i6 = 0;(i6 < 32);i6 = (1 + i6)) {
18             t2[(i6 + (32 * i5))] = 0.0f;
19           }
20         }
21
22         for (int i7 = 0;(i7 < 256);i7 = (1 + i7)) {
23           for (int i8 = 0;(i8 < 32);i8 = (1 + i8)) {
24             float t3[32];
25             for (int i9 = 0;(i9 < 32);i9 = (1 + i9)) {
26               t3[i9] = t2[(i9 + (32 * i8))];
27             }
28
29             for (int i10 = 0;(i10 < 4);i10 = (1 + i10)) {
30               #pragma omp simd
31               for (int i11 = 0;(i11 < 32);i11 = (1 + i11)) {
32                 t3[i11] += x0[i10 + 4*i7 + 1024*i8 + 32768*i3] *
33                            t1[i11 + 32*i10 + 128*i7 + 32768*i4];
34               }
35             }
36
37             for (int i12 = 0;(i12 < 32);i12 = (1 + i12)) {
38               t2[(i12 + (32 * i8))] = t3[i12];
39             }
40           }
41         }
42
43         for (int i13 = 0;(i13 < 32);i13 = (1 + i13)) {
44           for (int i14 = 0;(i14 < 32);i14 = (1 + i14)) {
45             output[(((i14 + (32 * i4)) + (1024 * i13)) + (32768 * i3))] = t2[(i14 + (32 * i13))];
46           }
47         }
48       }
49     }
50   }
```

Listing B.5: C code generated by SHINE for the *array-packing* mat-mul in Section 5.6.

```
1  void cache_blocks(float* output, float* x0, float* x1) {
2    float t1[1048576];
3    #pragma omp parallel for
4    for (int i0 = 0;(i0 < 32);i0 = (1 + i0)) {
5      for (int i1 = 0;(i1 < 1024);i1 = (1 + i1)) {
6        #pragma omp simd
7        for (int i2 = 0;(i2 < 32);i2 = (1 + i2)) {
8          t1[((i2 + (32 * i1)) + (32768 * i0))] = x1[((i2 + (32 * i0)) + (1024 * i1))];
9        }
10       }
11     }
12     for (int i3 = 0;(i3 < 32);i3 = (1 + i3)) {
13       for (int i4 = 0;(i4 < 32);i4 = (1 + i4)) {
14         float t2[1024];
15         for (int i5 = 0;(i5 < 32);i5 = (1 + i5)) {
16           for (int i6 = 0;(i6 < 32);i6 = (1 + i6)) {
17             t2[(i6 + (32 * i5))] = 0.0f;
18           }
19         }
20         for (int i7 = 0;(i7 < 256);i7 = (1 + i7)) {
21           for (int i8 = 0;(i8 < 32);i8 = (1 + i8)) {
22             float t3[32];
23             for (int i9 = 0;(i9 < 32);i9 = (1 + i9)) {
24               t3[i9] = t2[(i9 + (32 * i8))];
25             }
26
27             #pragma omp simd
28             for (int i11 = 0;(i11 < 32);i11 = (1 + i11)) {
29               t3[i11] += x0[4*i7 + 1024*i8 + 32768*i3] * t1[i11 + 128*i7 + 32768*i4];
30             }
31             #pragma omp simd
32             for (int i11 = 0;(i11 < 32);i11 = (1 + i11)) {
33               t3[i11] += x0[1 + 4*i7 + 1024*i8 + 32768*i3] * t1[32 + i11 + 128*i7 + 32768*i4];
34             }
35             #pragma omp simd
36             for (int i11 = 0;(i11 < 32);i11 = (1 + i11)) {
37               t3[i11] += x0[2 + 4*i7 + 1024*i8 + 32768*i3] * t1[64 + i11 + 128*i7 + 32768*i4];
38             }
39             #pragma omp simd
40             for (int i11 = 0;(i11 < 32);i11 = (1 + i11)) {
41               t3[i11] += x0[3 + 4*i7 + 1024*i8 + 32768*i3] * t1[96 + i11 + 128*i7 + 32768*i4];
42             }
43
44             for (int i12 = 0;(i12 < 32);i12 = (1 + i12)) {
45               t2[(i12 + (32 * i8))] = t3[i12];
46             }
47           }
48         }
49         for (int i13 = 0;(i13 < 32);i13 = (1 + i13)) {
50           for (int i14 = 0;(i14 < 32);i14 = (1 + i14)) {
51             output[(((i14 + (32 * i4)) + (1024 * i13)) + (32768 * i3))] = t2[(i14 + (32 * i13))];
52           }
53         }
54 } } }
```

Listing B.6: C code generated by SHINE for the *cache-blocks* mat-mul in Section 5.6.

```c
1  void parallel(float* output, float* x0, float* x1) {
2    float t1[1048576];
3    #pragma omp parallel for
4    for (int i0 = 0;(i0 < 32);i0 = (1 + i0)) {
5      for (int i1 = 0;(i1 < 1024);i1 = (1 + i1)) {
6        #pragma omp simd
7        for (int i3 = 0;(i3 < 32);i3 = (1 + i3)) {
8          t1[((i3 + (32 * i1)) + (32768 * i0))] = x1[((i3 + (32 * i0)) + (1024 * i1))];
9        }
10       }
11     }
12   #pragma omp parallel for
13   for (int i4 = 0;(i4 < 32);i4 = (1 + i4)) {
14     for (int i5 = 0;(i5 < 32);i5 = (1 + i5)) {
15       float t2[1024];
16       for (int i6 = 0;(i6 < 32);i6 = (1 + i6)) {
17         for (int i_50146 = 0;(i_50146 < 32);i_50146 = (1 + i_50146)) {
18           t2[(i_50146 + (32 * i6))] = 0.0f;
19         }
20       }
21       for (int i7 = 0;(i7 < 256);i7 = (1 + i7)) {
22         for (int i8 = 0;(i8 < 32);i8 = (1 + i8)) {
23           float t3[32];
24           for (int i9 = 0;(i9 < 32);i9 = (1 + i9)) {
25             t3[i9] = t2[(i9 + (32 * i8))];
26           }
27
28           #pragma omp simd
29           for (int i11 = 0;(i11 < 32);i11 = (1 + i11)) {
30             t3[i11] += x0[4*i7 + 1024*i8 + 32768*i4] * t1[i11 + 128*i7 + 32768*i5];
31           }
32           #pragma omp simd
33           for (int i11 = 0;(i11 < 32);i11 = (1 + i11)) {
34             t3[i11] += x0[1 + 4*i7 + 1024*i8 + 32768*i4] * t1[32 + i11 + 128*i7 + 32768*i5];
35           }
36           #pragma omp simd
37           for (int i11 = 0;(i11 < 32);i11 = (1 + i11)) {
38             t3[i11] += x0[2 + 4*i7 + 1024*i8 + 32768*i4] * t1[64 + i11 + 128*i7 + 32768*i5];
39           }
40           #pragma omp simd
41           for (int i11 = 0;(i11 < 32);i11 = (1 + i11)) {
42             t3[i11] += x0[3 + 4*i7 + 1024*i8 + 32768*i4] * t1[96 + i11 + 128*i7 + 32768*i5];
43           }
44
45           for (int i12 = 0;(i12 < 32);i12 = (1 + i12)) {
46             t2[(i12 + (32 * i8))] = t3[i12];
47           }
48         }
49     , }
50       for (int i13 = 0;(i13 < 32);i13 = (1 + i13)) {
51         for (int i14 = 0;(i14 < 32);i14 = (1 + i14)) {
52           output[(((i14 + (32 * i5)) + (1024 * i13)) + (32768 * i4))] = t2[(i14 + (32 * i13))];
53         }
54       }
55 } } }
```

Listing B.7: C code generated by SHINE for the *parallel* mat-mul in Section 5.6.

Appendix C

Handwritten Sketches and Selection of Discovered Programs

This appendix contains the handwritten sketches from Table 5.6 of Section 5.6, used to guide matrix multiplication optimisation (Appendix C.1). It also contains the Rise programs found by sketch-guided equality saturation for the *parallel* optimisation goal in Section 5.6 (Appendix C.2).

C.1   Matrix Multiplication Sketches

The handwritten sketches from Table 5.6 of Section 5.6.

```
1  containsMap(m,
2   containsMap(n,
3    containsReduceSeq(k,
4     containsAddMul)))
```

Listing C.1: A sketch for the *baseline* goal (Listing 5.2).

```
1  containsMap(m / 32,
2   containsMap(32,
3    containsMap(n / 32,
4     containsMap(32,
5      containsReduceSeq(k / 4,
6       containsReduceSeq(4,
7        containsAddMul))))))
```

Listing C.2: *split* sketch specifying how to split loops for all 7 goals (Listing 5.4).

```
1  containsMap(m / 32,
2   containsMap(n / 32,
3    containsReduceSeq(k / 4,
4     containsReduceSeq(4,
5      containsMap(32,
6       containsMap(32,
7        containsAddMul))))))
```

Listing C.3: *reorder*$_1$ sketch for the *blocking* and *vectorisation* goals (Listing 5.3).

```
1  containsMap(m / 32,
2   containsMap(n / 32,
3    containsReduceSeq(k / 4,
4     containsMap(32,
5      containsReduceSeq(4,
6       containsMap(32,
7        containsAddMul))))))
```

Listing C.4: *reorder*$_2$ sketch for the *loop-perm*, *array-packing*, *cache-blocks*, and *parallel* goals.

```
1  containsMap(m / 32,
2   containsMap(n / 32,
3    containsReduceSeq(k / 4,
4     containsReduceSeq(4,
5      containsMap(32,
6       containsMap(1,
7        containsAddMulVec))))))
```

Listing C.5: *lower*$_1$ sketch goal for the *vectorisation* goal.

```
1  containsMap(m / 32,
2   containsMap(n / 32,
3    containsReduceSeq(k / 4,
4     containsMap(32,
5      containsReduceSeq(4,
6       containsMap(1,
7        containsAddMulVec))))))
```

Listing C.6: *lower*$_2$ sketch goal for the *loop-perm* goal.

```
1  containsMap(m / 32,
2   containsMap(n / 32,
3    containsReduceSeq(k / 4,
4     containsMap(32,
5      containsReduceSeq(4,
6       containsMap(32,
7        containsAddMul))))),
8   containsToMem(n.k.f32,
9    containsMap(n / 32,
10     containsMap(k,
11      containsMap(32.f32, ?))))))
```

Listing C.7: *store* sketch for the *array-packing*, *cache-blocks*, and *parallel* goals.

```
1  containsMap(m / 32,
2   containsMap(n / 32,
3    containsReduceSeq(k / 4,
4     containsMap(32,
5      containsReduceSeq(4,
6       containsMap(1,
7        containsAddMulVec))))),
8   containsToMem(n.k.f32,
9    containsMap(n / 32,
10     containsMap(k,
11      containsMap(1.<32>f32, ?))))))
```

Listing C.8: *lower₃* sketch goal for the *array-packing* goal.

```
1  containsMap(m / 32,
2   containsMap(n / 32,
3    containsReduceSeq(k / 4,
4     containsMap(32,
5      containsReduceSeqUnroll(4,
6       containsMap(1,
7        containsAddMulVec))))),
8   containsToMem(n.k.f32,
9    containsMapPar(n / 32,
10     containsMap(k,
11      containsMap(1.<32>f32, ?))))))
```

Listing C.9: *lower₄* sketch goal for the *cache-blocks* goal.

```
1  containsMapPar(m / 32,
2   containsMap(n / 32,
3    containsReduceSeq(k / 4,
4     containsMap(32,
5      containsReduceSeqUnroll(4,
6       containsMap(1,
7        containsAddMulVec)))))),
8   containsToMem(n.k.f32,
9    containsMapPar(n / 32,
10    containsMap(k,
11     containsMap(1.<32>f32, ?)))))))
```

Listing C.10: *lower*$_5$ sketch goal for the *parallel* goal.

C.2 RISE Programs for the *parallel* Matrix Multiplication

This section shows the RISE programs that are found using sketch-guided equality saturation for the *parallel* matrix multiplication optimisation goal in Section 5.6.

The initial program is shown in Listing C.11. The intermediate programs are shown in Listings C.12 to C.14, and each satisfy a corresponding sketch guide. The final program satisfying the sketch goal is shown in Listing C.15.

Additionally, a final automatic transformation is applied to obtain a valid low-level program that can be translated through DPIA, i.e. implying valid read-write annotations (Section 3.1.3). Sequential loops and memory copies are inserted where required, and let expressions are hoisted as much as possible, resulting in Listing C.16.

```
1  Λn0:nat. Λn1:nat. Λn2:nat. λx0. λx1.
2  map (λx2.
3   map (λx3.
4    reduce add 0 (map (λx4. (mul (fst x4) (snd x4))) (zip x2 x3)))
5    (transpose x1))
6   x0
```

Listing C.11: Initial RISE program for matrix multiplication.

```
1  Λn0:nat. Λn1:nat. Λn2:nat. λx0. λx1.
2  join (
3   map (
4    map (λx2.
5     join (
6      map (
7       map (λx3.
8        reduceSeq (λx4. λx5.
9         add x4 (reduceSeq (λx6. λx7. add x6 (mul (fst x7) (snd x7))) 0 x5))
10        0
11        (split 4 (zip x2 x3))))
12       (split 32 (transpose x1)))))
13  (split 32 x0))
```

Listing C.12: RISE program satisfying the *split* sketch guide.

```
1  Λn0:nat. Λn1:nat. Λn2:nat. λx0. λx1.
2  join (map (map join) (map transpose (
3   map (
4    map (λx2.
5     reduceSeq (λx3. λx4.
6      map (λx5.
7       (λx6. reduceSeq (λx7. λx8.
8        map (λx9. add (fst x9) (mul (fst (snd x9)) (snd (snd x9))))
9         (zip x7 x8))
10        (fst x6)
11        (transpose (snd x6)))
12       (unzip (zip (fst x5) (snd x5))))
13      (zip x3 x4))
14     (generate (λx3. generate (λx4. 0)))
15     (transpose x2)))
16   (map transpose (map (map (λx2.
17    map transpose (map (map (λx3.
18     split 4 (zip x2 x3))) (split 32 (transpose x1)))))
19    (split 32 x0))))))
```

Listing C.13: RISE program satisfying the *reorder*$_2$ sketch guide.

```
1  Λn0:nat. Λn1:nat. Λn2:nat. λx0. λx1.
2  join (map (map join) (map transpose (
3   map (
4    map (λx2.
5     reduceSeq (λx3. λx4.
6      map (λx5.
7       reduceSeq (λx6. λx7.
8        map (λx8.
9         add (fst x8) (mul (fst (snd x8)) (snd (snd x8))))
10        (zip x6 x7))
11       (fst (unzip (zip (fst x5) (snd x5))))
12       (transpose (snd (unzip (zip (fst x5) (snd x5))))))
13      (zip x3 x4))
14     (generate (λx3. generate (λx4. 0)))
15     (transpose x2)))
16   (map transpose (map (map (λx2.
17    map transpose (map (map (λx3. split 4 (zip x2 x3)))
18     (split 32 (let (toMem (
19      join (map transpose (map (map (map (λx3. x3)))
20       (map transpose (split 32 (transpose x1)))))))
21      (λx3. x3))))))
22   (split 32 x0))))))
```

Listing C.14: RISE program satisfying the *store* sketch guide.

```
1  Λn0:nat. Λn1:nat. Λn2:nat. λx0. λx1.
2  join (mapPar (λx2.
3    map join (transpose (map (λx3.
4      reduceSeq (λx4. λx5.
5       map (λx6.
6        reduceSeqUnroll (λx7. λx8.
7          (λx9. asScalar (map (λx10.
8           add (fst x10) (mul (fst (snd x10)) (snd (snd x10))))
9           (zip (asVector 32 (fst (unzip x9)))
10           (zip (asVector 32 (fst (unzip (snd (unzip x9)))))
11            (asVector 32 (snd (unzip (snd (unzip x9)))))))))))
12         (zip x7 x8))
13         (fst (unzip (zip (fst x6) (snd x6))))
14         (transpose (snd (unzip (zip (fst x6) (snd x6))))))))
15       (zip x4 x5))
16      (generate (λx4. generate (λx5. 0)))
17      (transpose x3))
18    (transpose x2))))
19   (map
20    (map (λx2. map transpose
21      (map (map (λx3. split 4 (zip x2 x3)))
22       (split 32 (let (toMem (join (
23         mapPar (λx3.
24          transpose (map (λx4.
25           asScalar (map (λx5. x5) (asVector 32 x4)))
26           (transpose x3)))
27          (split 32 (transpose x1)))))
28         (λx3. x3))))))
29     (split 32 x0)))
```

Listing C.15: RISE program satisfying the *lower*$_4$ sketch goal.

```
1  Λn0:nat. n1:nat. Λn2:nat. λx0. λx1.
2  let (toMem (
3   join (mapPar (λx18.
4    transpose (mapSeq (λx19.
5     asScalar (mapSeq (λx20. x20) (asVector 32 x19)))
6    (transpose x18)))
7   (split 32 (transpose x1)))))
8  (λx21. join (
9   mapPar (λx2.
10   map join (transpose (mapSeq (λx3.
11    mapSeq (mapSeq (λx4. x4)) (reduceSeq (λx5. λx6.
12     mapSeq (λx7. mapSeq (λx8. x8) (
13      reduceSeqUnroll (λx9. λx10.
14       asScalar (mapSeq (λx11.
15        add (fst x11) (mul (fst (snd x11)) (snd (snd x11))))
16        (zip (asVector 32 (fst (unzip (zip x9 x10))))
17         (zip (asVector 32 (fst (unzip (snd (unzip (zip x9 x10))))))
18          (asVector 32 (snd (unzip (snd (unzip (zip x9 x10)))))))))))))))
19        (mapSeq (λx12. x12) (fst (unzip (zip (fst x7) (snd x7)))))
20        (transpose (snd (unzip (zip (fst x7) (snd x7))))))))
21       (zip x5 x6))
22      (mapSeq (mapSeq (λx13. x13)) (generate (λx14. generate (λx15. 0))))
23      (transpose x3)))
24     (transpose x2))))
25    (map (λe3839.
26     map (λx16.
27      map transpose (map (map (λx17. split 4 (zip x16 x17))) (split 32 x21)))
28      e3839)
29     (split 32 x0))))
```

Listing C.16: RISE program after final lowering.

Bibliography

[1]   Thomas Kœhler and Michel Steuwer. "Towards a Domain-Extensible Compiler: Optimizing an Image Processing Pipeline on Mobile CPUs". In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2021, pp. 27–38.

[2]   Thomas Kœhler, Phil Trinder, and Michel Steuwer. *Sketch-Guided Equality Saturation: Scaling Equality Saturation to Complex Optimizations in Languages with Bindings.* 2021. arXiv: `2111.13040` `[cs.PL]`.

[3]   Bastian Hagedorn et al. "Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies". In: *Proc. ACM Program. Lang.* 4.ICFP (2020), 92:1–92:29. DOI: `10.1145/3408974`.

[4]   Michel Steuwer et al. *RISE & Shine: Language-Oriented Compiler Design.* 2022. arXiv: `2201.03611` `[cs.PL]`.

[5]   Jackson Woodruff et al. *Rewriting History: Repurposing Domain-Specific Accelerators with Rewrite Exploration.* under submission. 2022.

[6]   Roy Schwartz et al. "Green AI". In: *Commun. ACM* 63.12 (Nov. 2020), pp. 54–63. ISSN: 0001-0782. DOI: `10.1145/3381831`. URL: `https://doi.org/10.1145/3381831`.

[7]   Loïc Lannelongue, Jason Grealey, and Michael Inouye. "Green Algorithms: Quantifying the carbon emissions of computation". In: *CoRR* abs/2007.07610 (2020). arXiv: `2007.07610`. URL: `https://arxiv.org/abs/2007.07610`.

[8]   Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society. 2004, p. 75.

[9]   Jonathan Ragan-Kelley et al. "Decoupling algorithms from schedules for easy
        optimization of image processing pipelines".
        In: *ACM Trans. Graph.* 31.4 (2012), 32:1–32:12. DOI: `10.1145/2185520.2185528`.

[10]   Tianqi Chen et al.
        "{TVM}: An automated end-to-end optimizing compiler for deep learning". In: *13th*
        {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18).
        2018, pp. 578–594.

[11]   Paul Barham and Michael Isard. "Machine learning systems are stuck in a rut".
        In: *Proceedings of the Workshop on Hot Topics in Operating Systems.* 2019, pp. 177–183.

[12]   Li Du and Yuan Du. "Hardware accelerator design for machine learning".
        In: *Machine Learning-Advanced Techniques and Emerging Applications* (2017), pp. 1–14.

[13]   Yuka Ikarashi et al.
        "Exocompilation for Productive Programming of Hardware Accelerators".
        In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming*
        Language Design and Implementation. PLDI 2022.
        San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 703–718.
        ISBN: 9781450392655. DOI: `10.1145/3519939.3523446`.
        URL: `https://doi.org/10.1145/3519939.3523446`.

[14]   Hassan Chafi et al. "A domain-specific approach to heterogeneous parallelism".
        In: *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of*
        Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011. 2011,
        pp. 35–46. DOI: `10.1145/1941553.1941561`.

[15]   Michel Steuwer et al. "Generating Performance Portable Code using Rewrite Rules:
        From High-Level Functional Expressions to High-Performance OpenCL Code".
        In: *International Conference on Functional Programming (ICFP)* (2015).

[16]   Bastian Hagedorn et al. "High Performance Stencil Code Generation with Lift".
        In: *International Symposium on Code Generation and Optimization (CGO).* 2018.

[17]   Roland Leißa et al. "AnyDSL: A Partial Evaluation Framework for Programming
        High-performance Libraries".
        In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018), 119:1–119:30. ISSN: 2475-1421.
        DOI: `10.1145/3276489`.

[18]   Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula.
        "PolyMage: Automatic Optimization for Image Processing Pipelines".
        In: *SIGARCH Comput. Archit. News* 43.1 (Mar. 2015), pp. 429–443. ISSN: 0163-5964.
        DOI: `10.1145/2786763.2694364`.

[19]   Saeed Maleki et al. "An evaluation of vectorizing compilers".
       In: *2011 International Conference on Parallel Architectures and Compilation Techniques.*
       IEEE. 2011, pp. 372–382.

[20]   D. Parello et al. "Towards a Systematic, Pragmatic and Architecture-Aware Program
       Optimization Process for Complex Processors".
       In: *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing.* 2004,
       pp. 15–15. DOI: `10.1109/SC.2004.61`.

[21]   Craig Chambers. "Staged compilation". In: *ACM SIGPLAN Notices* 37.3 (2002), pp. 1–8.

[22]   Jarkko Niittylahti, Juha Lemmetti, and Juhana Helovuo.
       "High-performance implementation of wavelet algorithms on a standard PC".
       In: *Microprocessors and Microsystems* 26.4 (2002), pp. 173–179.

[23]   L. Lacassagne et al.
       High Level Transforms for SIMD and low-level computer vision algorithms. 2014.
       URL: `http://www-soc.lip6.fr/~lacas/Publications/WPMVP14.pdf`
       (visited on 07/18/2017).

[24]   Florian Lemaitre, Benjamin Couturier, and Lionel Lacassagne.
       "Cholesky factorization on SIMD multi-core architectures".
       In: *Journal of Systems Architecture* 79 (2017), pp. 1–15.

[25]   Bastian Hagedorn et al.
       "Fireiron: A Data-Movement-Aware Scheduling Language for GPUs".
       In: *Proceedings of the ACM International Conference on Parallel Architectures and
       Compilation Techniques.* PACT '20.
       Virtual Event, GA, USA: Association for Computing Machinery, 2020, pp. 71–82.
       ISBN: 9781450380751. DOI: `10.1145/3410463.3414632`.
       URL: `https://doi.org/10.1145/3410463.3414632`.

[26]   Yuka Ikarashi et al. "Guided Optimization for Image Processing Pipelines". In: *2021
       IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC).*
       IEEE. 2021, pp. 1–5.

[27]   Ravi Teja Mullapudi et al.
       "Automatically Scheduling Halide Image Processing Pipelines".
       In: *ACM Trans. Graph.* 35.4 (July 2016), 83:1–83:11. ISSN: 0730-0301.
       DOI: `10.1145/2897824.2925952`.

[28]   LUKE ANDERSON et al.
       "Learning to Optimize Halide with Tree Search and Random Programs". In: (2019).

[29] Luke Anderson et al.
"Efficient automatic scheduling of imaging and vision pipelines for the GPU".
In: *Proceedings of the ACM on Programming Languages* 5.OOPSLA (2021), pp. 1–28.

[30] Christopher G. Harris and Mike Stephens. "A Combined Corner and Edge Detector".
In: *Proceedings of the Alvey Vision Conference, AVC 1988, Manchester, UK, September,*
1988. 1988, pp. 1–6. DOI: `10.5244/C.2.23`.

[31] M. Horowitz and W. Dally. "How scaling will change processor architecture".
In: *2004 IEEE International Solid-State Circuits Conference (IEEE Cat. No.04CH37519).*
Feb. 2004, 132–133 Vol.1. DOI: `10.1109/ISSCC.2004.1332629`.

[32] Mark D Hill and Michael R Marty. "Amdahl's law in the multicore era".
In: *Computer* 41.7 (2008), pp. 33–38.

[33] H. Esmaeilzadeh et al. "Dark silicon and the end of multicore scaling".
In: *2011 38th Annual International Symposium on Computer Architecture (ISCA).*
June 2011, pp. 365–376.

[34] Mohamed Zahran.
"Heterogeneous Computing: Here to Stay: Hardware and Software Perspectives".
In: *Queue* 14.6 (Dec. 2016), pp. 31–42. ISSN: 1542-7730.
DOI: `10.1145/3028687.3038873`.
URL: `https://doi.org/10.1145/3028687.3038873`.

[35] Suejb Memeti et al. "Benchmarking OpenCL, OpenACC, OpenMP, and CUDA:
Programming Productivity, Performance, and Energy Consumption".
In: *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling*
for Cloud Computing. ARMS-CC '17.
Washington, DC, USA: Association for Computing Machinery, 2017, pp. 1–6.
ISBN: 9781450351164. DOI: `10.1145/3110355.3110356`.
URL: `https://doi.org/10.1145/3110355.3110356`.

[36] Pedro Fonseca et al.
"A study of the internal and external effects of concurrency bugs".
In: *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN).*
2010, pp. 221–230. DOI: `10.1109/DSN.2010.5544315`.

[37] David Luebke et al. "GPGPU: General Purpose Computation on Graphics Hardware".
In: *ACM SIGGRAPH 2004 Course Notes.* SIGGRAPH '04. Los Angeles, CA: ACM, 2004.
DOI: `10.1145/1103900.1103933`.

[38] Syed Waqar Nabi and Wim Vanderbauwhede. "Using type transformations to generate program variants for FPGA design space exploration".
In: *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*.
2015, pp. 1–6. DOI: `10.1109/ReConFig.2015.7393365`.

[39] Lukas Siefke et al.
"Systematically extending a high-level code generator with support for tensor cores".
In: *Proceedings of the 14th Workshop on General Purpose Processing Using GPU*. 2022,
pp. 1–6.

[40] Christof Schlaak, Tzung-Han Juang, and Christophe Dubach.
"Memory-Aware Functional IR for Higher-Level Synthesis of Accelerators".
In: *ACM Trans. Archit. Code Optim.* 19.2 (Jan. 2022). ISSN: 1544-3566.
DOI: `10.1145/3501768`. URL: `https://doi.org/10.1145/3501768`.

[41] Bo-Yuan Huang et al. *Specialized Accelerators and Compiler Flows: Replacing Accelerator APIs with a Formal Software/Hardware Interface.* 2022.
DOI: `10.48550/ARXIV.2203.00218`.
URL: `https://arxiv.org/abs/2203.00218`.

[42] Christopher Monroe and Jungsang Kim. "Scaling the ion trap quantum processor".
In: *Science* 339.6124 (2013), pp. 1164–1169.

[43] Mike Davies et al.
"Loihi: A neuromorphic manycore processor with on-chip learning".
In: *Ieee Micro* 38.1 (2018), pp. 82–99.

[44] Yiwei Xie et al. "Programmable optical processor chips: toward photonic RF filters with DSP-level flexibility and MHz-band selectivity".
In: *Nanophotonics* 7.2 (2018), pp. 421–454.

[45] Michel Steuwer.
"Improving programmability and performance portability on many-core processors".
PhD thesis. 2015.

[46] Aaftab Munshi. "The opencl specification".
In: *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE. 2009, pp. 1–314.

[47] John E Stone, David Gohara, and Guochun Shi.
"OpenCL: A parallel programming standard for heterogeneous computing systems".
In: *Computing in science & engineering* 12.3 (2010), p. 66.

[48] James Hegarty et al.
"Darkroom: compiling high-level image processing code into hardware pipelines".
In: *ACM Trans. Graph.* 33.4 (2014), 144:1–144:11.
DOI: `10.1145/2601097.2601174`.

[49] F. Franchetti et al. "SPIRAL: Extreme Performance Portability".
In: *Proceedings of the IEEE* 106.11 (Nov. 2018), pp. 1935–1968. ISSN: 0018-9219.
DOI: `10.1109/JPROC.2018.2873289`.

[50] Henrik Barthels and Paolo Bientinesi.
"Linnea: Compiling Linear Algebra Expressions to High-Performance Code".
In: *Proceedings of the 8th International Workshop on Parallel Symbolic Computation.*
Kaiserslautern, Germany, July 2017.

[51] Tobias Gysi et al. "STELLA: A Domain-Specific Tool for Structured Grid Methods in
Weather and Climate Models". In: *Proceedings of the International Conference for High
Performance Computing, Networking, Storage and Analysis.* SC '15.
Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450337236.
DOI: `10.1145/2807591.2807627`.
URL: `https://doi.org/10.1145/2807591.2807627`.

[52] Florian Rathgeber et al.
"Firedrake: automating the finite element method by composing abstractions".
In: *ACM Transactions on Mathematical Software (TOMS)* 43.3 (2016), pp. 1–27.

[53] Manuel M T Chakravarty et al.
"Accelerating Haskell array codes with multicore GPUs".
In: *DAMP '11: The 6th workshop on Declarative Aspects of Multicore Programming.*
ACM, Jan. 2011.

[54] Clemens Grelck and Sven-Bodo Scholz.
"SAC—a functional array language for efficient multi-threaded execution".
In: *International Journal of Parallel Programming* 34.4 (2006), pp. 383–427.

[55] Alexander Collins et al. "NOVA: A functional language for data parallelism".
In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and
Compilers for Array Programming.* 2014, pp. 8–13.

[56] Troels Henriksen et al. "Futhark: Purely Functional GPU-programming with Nested
Parallelism and In-place Array Updates". In: *Proceedings of the 38th ACM SIGPLAN
Conference on Programming Language Design and Implementation.* PLDI 2017.
Barcelona, Spain: ACM, 2017, pp. 556–571. ISBN: 978-1-4503-4988-8.
DOI: `10.1145/3062341.3062354`.

[57] Adam Paszke et al. "Getting to the point. index sets and parallelism-preserving
autodiff for pointful array programming". In: *arXiv preprint arXiv:2104.05372* (2021).

[58] Paolo Bientinesi et al.
"Tensor Computations: Applications and Optimization (Dagstuhl Seminar 20111)".
In: *Dagstuhl Reports* 10.3 (2020). Ed. by Paolo Bientinesi et al., pp. 58–70.
ISSN: 2192-5283. DOI: `10.4230/DagRep.10.3.58`.
URL: `https://drops.dagstuhl.de/opus/volltexte/2020/13430`.

[59] Fredrik Kjolstad et al. "The tensor algebra compiler".
In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (2017), pp. 1–29.

[60] Murray Cole. "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal
parallel programming". In: *Parallel Computing* 30.3 (2004), pp. 389–406.
ISSN: 0167-8191. DOI: `https://doi.org/10.1016/j.parco.2003.12.002`.

[61] Christopher Edward Cummins. "Deep learning for compilers". In: (2020).

[62] Shoumik Palkar et al.
"Weld: Rethinking the Interface Between Data-Intensive Applications".
In: *CoRR* abs/1709.06416 (2017). arXiv: `1709.06416`.
URL: `http://arxiv.org/abs/1709.06416`.

[63] Amanda Liu et al.
"Verified Tensor-Program Optimization via High-Level Scheduling Rewrites".
In: *Proc. ACM Program. Lang.* 6.POPL (Jan. 2022). DOI: `10.1145/3498717`.
URL: `https://doi.org/10.1145/3498717`.

[64] Lennart CL Kats and Eelco Visser. "The Spoofax language workbench: rules for
declarative specification of languages and IDEs".
In: *Proceedings of the ACM international conference on Object oriented programming
systems languages and applications*. 2010, pp. 444–463.

[65] Arvind K Sujeeth et al.
"Composition and reuse with compiled domain-specific languages".
In: *European Conference on Object-Oriented Programming*. Springer. 2013, pp. 52–78.

[66] Tiark Rompf et al. "Optimizing Data Structures in High-level Programs: New
Directions for Extensible Compilers Based on Staging". In: *Proceedings of the 40th
Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.
POPL '13. ISBN: 978-1-4503-1832-7. DOI: `10.1145/2429069.2429128`.

[67] Sam Tobin-Hochstadt et al. "Languages as Libraries".
In: *SIGPLAN Not.* 46.6 (June 2011), pp. 132–141. ISSN: 0362-1340.
DOI: `10.1145/1993316.1993514`.
URL: `https://doi.org/10.1145/1993316.1993514`.

[68] Toomas Remmelg et al.
"Performance portable GPU code generation for matrix multiplication".
In: *GPGPU@PPoPP*. ACM, 2016, pp. 22–31.

[69] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare.
"Playing by the rules: rewriting as a practical optimisation technique in GHC".
In: *Haskell workshop*. Vol. 1. 2001, pp. 203–233.

[70] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach.
"Building Program Optimizers with Rewriting Strategies". In: *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*. ICFP '98.
Baltimore, Maryland, USA: ACM, 1998, pp. 13–26. ISBN: 1-58113-024-4.
DOI: `10.1145/289423.289425`.

[71] A. Granicz and J. Hickey. "Phobos: a front-end approach to extensible compilers".
In: *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the.* 2003. DOI: `10.1109/HICSS.2003.1174890`.

[72] Markus Puschel et al. "SPIRAL: Code generation for DSP transforms".
In: *Proceedings of the IEEE* 93.2 (2005), pp. 232–275.

[73] Trevor L McDonell et al. "Optimising purely functional GPU programs".
In: *ACM SIGPLAN Notices* 48.9 (2013), pp. 49–60.

[74] Naums Mogers et al. "Mapping parallelism in a functional IR through constraint satisfaction: a case study on convolution for mobile GPUs". In: *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction.* 2022, pp. 218–230.

[75] Cristian Urlea. "Optimal program variant generation for hybrid manycore systems".
PhD thesis. University of Glasgow, 2021.

[76] Ulysse Beaugnon et al. "Optimization space pruning without regrets".
In: *CC 2017-26th International Conference on Compiler Construction.* ACM Press. 2017, pp. 34–44.

[77] Tobias Gysi, Tobias Grosser, and Torsten Hoefler. "Absinthe: Learning an analytical performance model to fuse and tile stencil codes in one shot". In: *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT).* IEEE. 2019, pp. 370–382.

[78] Syed Waqar Nabi and Wim Vanderbauwhede.
"FPGA design space exploration for scientific HPC applications using a fast and accurate cost model based on roofline analysis".
In: *Journal of Parallel and Distributed Computing* 133 (2019), pp. 407–419.
ISSN: 0743-7315. DOI: `https://doi.org/10.1016/j.jpdc.2017.05.014`.

URL: `https://www.sciencedirect.com/science/article/pii/S074373151730165X`.

[79]   Di Wang, David M Kahn, and Jan Hoffmann.
       "Raising expectations: automating expected cost analysis with types".
       In: *Proceedings of the ACM on Programming Languages* 4.ICFP (2020), pp. 1–31.

[80]   Dejice Jacob, Phil Trinder, and Jeremy Singer. "Pricing Python parallelism: a dynamic
       language cost model for heterogeneous platforms". In: *Proceedings of the 16th ACM
       SIGPLAN International Symposium on Dynamic Languages*. 2020, pp. 29–42.

[81]   Riyadh Baghdadi et al.
       "A Deep Learning Based Cost Model for Automatic Code Optimization".
       In: *Proceedings of Machine Learning and Systems*.
       Ed. by A. Smola, A. Dimakis, and I. Stoica. Vol. 3. 2021, pp. 181–193.
       URL: `https://proceedings.mlsys.org/paper/2021/file/3def184ad8f4755ff269862ea77393dd-Paper.pdf`.

[82]   Alok Mishra et al. "COMPOFF: A Compiler Cost model using Machine Learning to
       predict the Cost of OpenMP Offloading". In: *2022 IEEE International Parallel and
       Distributed Processing Symposium Workshops (IPDPSW)*. 2022, pp. 391–400.
       DOI: `10.1109/IPDPSW55747.2022.00074`.

[83]   Toru Kisuki et al. "Iterative compilation in program optimization".
       In: *Proc. CPC'10 (Compilers for Parallel Computers)*. Citeseer. 2000, pp. 35–44.

[84]   Keith D Cooper, Devika Subramanian, and Linda Torczon.
       "Adaptive optimizing compilers for the 21st century".
       In: *The Journal of Supercomputing* 23.1 (2002), pp. 7–22.

[85]   Ross Tate et al. "Equality saturation: a new approach to optimization".
       In: *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of
       programming languages*. 2009, pp. 264–276.

[86]   Max Willsey et al. "Egg: Fast and extensible equality saturation".
       In: *Proceedings of the ACM on Programming Languages* 5.POPL (2021), pp. 1–29.

[87]   Shimon Y Nof. "Automation: What it means to us around the world".
       In: *Springer handbook of automation*. Springer, 2009, pp. 13–52.

[88]   Cédric Bastoul. "Code Generation in the Polyhedral Model Is Easier Than You Think".
       In: *13th International Conference on Parallel Architectures and Compilation Techniques
       (PACT 2004), 29 September - 3 October 2004, Antibes Juan-les-Pins, France*. 2004,
       pp. 7–16. DOI: `10.1109/PACT.2004.10018`.

[89] Albert Cohen et al.
"Facilitating the Search for Compositions of Program Transformations".
In: *Proceedings of the 19th Annual International Conference on Supercomputing*. ICS '05.
Cambridge, Massachusetts: ACM, 2005, pp. 151–160. ISBN: 1595931678.
DOI: `10.1145/1088149.1088169`.

[90] Sylvain Girbal et al. "Semi-automatic composition of loop transformations for deep
parallelism and memory hierarchies".
In: *International Journal of Parallel Programming* 34.3 (2006), pp. 261–317.

[91] Chun Chen, Jacqueline Chame, and Mary Hall.
CHiLL: A framework for composing high-level loop transformations. Tech. rep.
Citeseer, 2008.

[92] Andreas Klöckner.
"Loo.Py: Transformation-Based Code Generation for GPUs and CPUs".
In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and
Compilers for Array Programming*. ARRAY'14.
Edinburgh, United Kingdom: Association for Computing Machinery, 2014, pp. 82–87.
ISBN: 9781450329378. DOI: `10.1145/2627373.2627387`.
URL: `https://doi.org/10.1145/2627373.2627387`.

[93] Riyadh Baghdadi et al.
"Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code".
In: *CoRR* abs/1804.10694 (2018). arXiv: `1804.10694`.
URL: `http://arxiv.org/abs/1804.10694`.

[94] Claude Kirchner. "Strategic rewriting".
In: *Electronic Notes in Theoretical Computer Science* 124.2 (2005), pp. 3–9.

[95] Lianmin Zheng et al.
"Ansor: Generating {High-Performance} Tensor Programs for Deep Learning".
In: *14th USENIX symposium on operating systems design and implementation (OSDI 20)*.
2020, pp. 863–879.

[96] Chao Gao et al. "Bansor: Improving Tensor Program Auto-Scheduling with Bandit
Based Reinforcement Learning".
In: *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)*.
2021, pp. 273–278. DOI: `10.1109/ICTAI52525.2021.00045`.

[97] Qing Yi et al. "POET: Parameterized Optimizations for Empirical Tuning".
In: *2007 IEEE International Parallel and Distributed Processing Symposium*. 2007,
pp. 1–8. DOI: `10.1109/IPDPS.2007.370637`.

[98] Malik Khan et al. "A Script-Based Autotuning Compiler System to Generate High-Performance CUDA Code". In: *ACM Trans. Archit. Code Optim.* 9.4 (Jan. 2013). ISSN: 1544-3566. DOI: `10.1145/2400682.2400690`. URL: `https://doi.org/10.1145/2400682.2400690`.

[99] Uday Bondhugula et al. "A practical automatic polyhedral parallelizer and locality optimizer". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 2008, pp. 101–113.

[100] Sven Verdoolaege et al. "Polyhedral Parallel Code Generation for CUDA". In: *ACM Trans. Archit. Code Optim.* 9.4 (Jan. 2013). ISSN: 1544-3566. DOI: `10.1145/2400682.2400713`. URL: `https://doi.org/10.1145/2400682.2400713`.

[101] Abhinav Jangda and Uday Bondhugula. "An Effective Fusion and Tile Size Model for PolyMage". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 42.3 (2020), pp. 1–27.

[102] Lénaïc Bagnères et al. "Opening Polyhedral Compiler's Black Box". In: *Proceedings of the 2016 International Symposium on Code Generation and Optimization.* CGO '16. Barcelona, Spain: Association for Computing Machinery, 2016, pp. 128–138. ISBN: 9781450337786. DOI: `10.1145/2854038.2854048`. URL: `https://doi.org/10.1145/2854038.2854048`.

[103] Oleksandr Zinenko, Lorenzo Chelini, and Tobias Grosser. *Declarative Transformations in the Polyhedral Model.* Research Report RR-9243. Inria ; ENS Paris - Ecole Normale Supérieure de Paris ; ETH Zurich ; TU Delft ; IBM Zürich, Dec. 2018. URL: `https://hal.inria.fr/hal-01965599`.

[104] Lorenzo Chelini et al. "Declarative Loop Tactics for Domain-Specific Optimization". In: *ACM Trans. Archit. Code Optim.* 16.4 (Dec. 2019). ISSN: 1544-3566. DOI: `10.1145/3372266`.

[105] Jun Shirako, Louis-Noël Pouchet, and Vivek Sarkar. "Oil and Water Can Mix: An Integration of Polyhedral and AST-Based Transformations". In: *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* 2014, pp. 287–298. DOI: `10.1109/SC.2014.29`.

[106] Robert Atkey et al. "Strategy Preserving Compilation for Parallel Functional Code". In: *CoRR* abs/1710.08332 (2017). arXiv: `1710.08332`.

[107] 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.

[108]   Henk P Barendregt. "Lambda calculi with types". In: (1992).

[109]   Jeffrey Dean and Sanjay Ghemawat.
        "MapReduce: simplified data processing on large clusters".
        In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[110]   Michael McCool, James Reinders, and Arch Robison.
        Structured Parallel Programming: Patterns for Efficient Computation. 1st.
        San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[111]   John C Reynolds. "The Essence of Algol". In: *ALGOL-like Languages.* Springer, 1997,
        pp. 67–88.

[112]   Guy E Blelloch. *Vector models for data-parallel computing.* Vol. 2.
        MIT press Cambridge, 1990.

[113]   Vadim Maslov.
        "Delinearization: An efficient way to break multiloop dependence equations".
        In: *ACM Sigplan Notices* 27.7 (1992), pp. 152–161.

[114]   Robert H. B. Netzer and Barton P. Miller.
        "What Are Race Conditions? Some Issues and Formalizations".
        In: *ACM Lett. Program. Lang. Syst.* 1.1 (Mar. 1992), pp. 74–88. ISSN: 1057-4514.
        DOI: `10.1145/130616.130623`.
        URL: `https://doi.org/10.1145/130616.130623`.

[115]   Toomas Remmelg. "Automatic performance optimisation of parallel programs for
        GPUs via rewrite rules". In: (2019).

[116]   Sreekaanth S Isloor and T Anthony Marsland. "The Deadlock Problem: An Overview."
        In: *Computer* 13.9 (1980), pp. 58–78.

[117]   Michael F. P. Oboyle, L Kervella, and Francois Bodin.
        "Synchronization minimization in a SPMD execution model".
        In: *Journal of parallel and distributed computing* 29.2 (1995), pp. 196–210.

[118]   Sungju Lee et al. "Considering Barrier Synchronization Overhead in Parallelizing
        Cryptographic Algorithms".
        In: *2011 International Conference on Information Science and Applications.* 2011,
        pp. 1–4. DOI: `10.1109/ICISA.2011.5772401`.

[119]   Ang Li et al. "Warp-Consolidation: A Novel Execution Model for GPUs".
        In: *Proceedings of the 2018 International Conference on Supercomputing.* ICS '18.
        Beijing, China: Association for Computing Machinery, 2018, pp. 53–64.
        ISBN: 9781450357838. DOI: `10.1145/3205289.3205294`.
        URL: `https://doi.org/10.1145/3205289.3205294`.

[120]   CLEMENS GRELCK.
"Shared memory multiprocessor support for functional array processing in SAC".
In: *Journal of Functional Programming* 15.3 (2005), pp. 353–401.
DOI: `10.1017/S0956796805005538`.

[121]   Dong Chen et al. "Automatic Mapping Single-Device OpenCL Program to
Heterogeneous Multi-device Platform". In: *2013 IEEE 10th International Conference on
High Performance Computing and Communications & 2013 IEEE International
Conference on Embedded and Ubiquitous Computing*. 2013, pp. 135–142.
DOI: `10.1109/HPCC.and.EUC.2013.28`.

[122]   Alain Darte and Robert Schreiber.
"A Linear-Time Algorithm for Optimal Barrier Placement". In: *Proceedings of the
Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.
PPoPP '05. Chicago, IL, USA: Association for Computing Machinery, 2005, pp. 26–35.
ISBN: 1595930809. DOI: `10.1145/1065944.1065949`.
URL: `https://doi.org/10.1145/1065944.1065949`.

[123]   Allen Leung et al. "A Mapping Path for Multi-GPGPU Accelerated Computers from a
Portable High Level Programming Abstraction". In: *Proceedings of the 3rd Workshop on
General-Purpose Computation on Graphics Processing Units*. GPGPU-3.
Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 2010,
pp. 51–61. ISBN: 9781605589350. DOI: `10.1145/1735688.1735698`.
URL: `https://doi.org/10.1145/1735688.1735698`.

[124]   Muthu Manikandan Baskaran, Jj Ramanujam, and P Sadayappan.
"Automatic C-to-CUDA code generation for affine programs".
In: *International Conference on Compiler Construction*. Springer. 2010, pp. 244–263.

[125]   Jun Shirako, Akihiro Hayashi, and Vivek Sarkar. "Optimized Two-Level
Parallelization for GPU Accelerators Using the Polyhedral Model".
In: *Proceedings of the 26th International Conference on Compiler Construction*. CC 2017.
Austin, TX, USA: Association for Computing Machinery, 2017, pp. 22–33.
ISBN: 9781450352338. DOI: `10.1145/3033019.3033022`.
URL: `https://doi.org/10.1145/3033019.3033022`.

[126]   Chau-Wen Tseng. "Compiler Optimizations for Eliminating Barrier Synchronization".
In: *SIGPLAN Not.* 30.8 (1995), pp. 144–155. ISSN: 0362-1340.
DOI: `10.1145/209937.209952`.
URL: `https://doi.org/10.1145/209937.209952`.

[127]   M. O'Boyle and E. Stohr. "Compile time barrier synchronization minimization".
In: *IEEE Transactions on Parallel and Distributed Systems* 13.6 (2002), pp. 529–543.
DOI: `10.1109/TPDS.2002.1011394`.

[128] Harenome Razanajato, Cidric Bastoul, and Vincent Loechner.
"Lifting Barriers Using Parallel Polyhedral Regions".
In: *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*.
2017, pp. 338–347. DOI: `10.1109/HiPC.2017.00046`.

[129] Philippe Tillet, H. T. Kung, and David Cox. "Triton: An Intermediate Language and
Compiler for Tiled Neural Network Computations". In: *Proceedings of the 3rd ACM
SIGPLAN International Workshop on Machine Learning and Programming Languages*.
New York, NY, USA: Association for Computing Machinery, 2019, pp. 10–19.
ISBN: 9781450367196. URL: `https://doi.org/10.1145/3315508.3329973`.

[130] Malek Ben Romdhane. "Extending the capabilities of Tiramisu".
PhD thesis. Massachusetts Institute of Technology, 2018.

[131] Pratik Fegade et al. "Cortex: A Compiler for Recursive Deep Learning Models".
In: *Proceedings of Machine Learning and Systems*.
Ed. by A. Smola, A. Dimakis, and I. Stoica. Vol. 3. 2021, pp. 38–54.
URL: `https://proceedings.mlsys.org/paper/2021/file/`
`182be0c5cdcd5072bb1864cdee4d3d6e-Paper.pdf`.

[132] Mathias Bourgoin, Emmanuel Chailloux, and Jean-Luc Lamotte.
"Efficient Abstractions for GPGPU Programming".
In: *Int. J. Parallel Program.* 42.4 (Aug. 2014), pp. 583–600. ISSN: 0885-7458.
DOI: `10.1007/s10766-013-0261-x`.

[133] Johan Enmyren and Christoph W. Kessler.
"SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems".
In: *Proceedings of the Fourth International Workshop on High-level Parallel
Programming and Applications*. HLPP '10. Baltimore, Maryland, USA: ACM, 2010,
pp. 5–14. ISBN: 978-1-4503-0254-8. DOI: `10.1145/1863482.1863487`.

[134] M. Steuwer, P. Kegel, and S. Gorlatch.
"SkelCL - A Portable Skeleton Library for High-Level GPU Programming".
In: *2011 IEEE International Symposium on Parallel and Distributed Processing
Workshops and Phd Forum*. May 2011, pp. 1176–1182.
DOI: `10.1109/IPDPS.2011.269`.

[135] Mihai Budiu, Joel Galenson, and Gordon D. Plotkin. "The Compiler Forest".
In: *Programming Languages and Systems*.
Ed. by Matthias Felleisen and Philippa Gardner.
Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 21–40.
ISBN: 978-3-642-37036-6.

[136] Thomas Johnsson. "Lambda lifting: Transforming programs to recursive equations".
In: *Conference on Functional programming languages and computer architecture.*
Springer. 1985, pp. 190–203.

[137] Jonathan Ragan-Kelley et al. "Halide: A Language and Compiler for Optimizing
Parallelism, Locality, and Recomputation in Image Processing Pipelines".
In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language
Design and Implementation.* PLDI '13.
Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 519–530.
ISBN: 9781450320146. DOI: `10.1145/2491956.2462176`.
URL: `https://doi.org/10.1145/2491956.2462176`.

[138] Chuntao Hong et al.
"MapCG: Writing Parallel Program Portable between CPU and GPU".
In: *Proceedings of the 19th International Conference on Parallel Architectures and
Compilation Techniques.* PACT '10.
Vienna, Austria: Association for Computing Machinery, 2010, pp. 217–226.
ISBN: 9781450301787. DOI: `10.1145/1854273.1854303`.
URL: `https://doi.org/10.1145/1854273.1854303`.

[139] Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz.
"Breaking the GPU Programming Barrier with the Auto-Parallelising SAC Compiler".
In: *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming.*
DAMP '11. Austin, Texas, USA: Association for Computing Machinery, 2011,
pp. 15–24. ISBN: 9781450304863. DOI: `10.1145/1926354.1926359`.
URL: `https://doi.org/10.1145/1926354.1926359`.

[140] Hans-Nikolai Vießmann and Sven-Bodo Scholz.
"Effective Host-GPU Memory Management Through Code Generation".
In: *IFL 2020: Proceedings of the 32nd Symposium on Implementation and Application of
Functional Languages.* 2020, pp. 138–149.

[141] Dave Cunningham, Rajesh Bordawekar, and Vijay Saraswat.
"GPU Programming in a High Level Language: Compiling X10 to CUDA".
In: *Proceedings of the 2011 ACM SIGPLAN X10 Workshop.* X10 '11.
San Jose, California: Association for Computing Machinery, 2011.
ISBN: 9781450307703. DOI: `10.1145/2212736.2212744`.
URL: `https://doi.org/10.1145/2212736.2212744`.

[142] Christophe Dubach et al. "Compiling a High-Level Language for GPUs: (Via Language
Support for Architectures and Compilers)". In: *Proceedings of the 33rd ACM SIGPLAN
Conference on Programming Language Design and Implementation.* PLDI '12.
Beijing, China: Association for Computing Machinery, 2012, pp. 1–12.

ISBN: 9781450312059. DOI: `10.1145/2254064.2254066`.
URL: `https://doi.org/10.1145/2254064.2254066`.

[143]   João Bispo, Luís Reis, and João M. P. Cardoso.
"C and OpenCL Generation from MATLAB".
In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. SAC '15.
Salamanca, Spain: Association for Computing Machinery, 2015, pp. 1315–1320.
ISBN: 9781450331968. DOI: `10.1145/2695664.2695911`.
URL: `https://doi.org/10.1145/2695664.2695911`.

[144]   Luís Reis, João Bispo, and João M. P. Cardoso.
"Compiler Techniques for Efficient MATLAB to OpenCL Code Generation".
In: *Proceedings of the 5th International Workshop on OpenCL*. IWOCL 2017.
Toronto, Canada: Association for Computing Machinery, 2017. ISBN: 9781450352147.
DOI: `10.1145/3078155.3078186`.
URL: `https://doi.org/10.1145/3078155.3078186`.

[145]   Larisa Stoltzfus et al. "Code Generation for Room Acoustics Simulations with
Complex Boundary Conditions".
In: *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
IEEE. 2021, pp. 485–496.

[146]   William Thies, Michal Karczmarek, and Saman Amarasinghe.
"StreamIt: A language for streaming applications".
In: *International Conference on Compiler Construction*. Springer. 2002, pp. 179–196.

[147]   Michael I Gordon, William Thies, and Saman Amarasinghe.
"Exploiting coarse-grained task, data, and pipeline parallelism in stream programs".
In: *ACM SIGPLAN Notices* 41.11 (2006), pp. 151–162.

[148]   Jeffrey Bosboom et al.
"StreamJIT: A commensal compiler for high-performance stream programming".
In: *ACM SIGPLAN Notices* 49.10 (2014), pp. 177–195.

[149]   Markus Aronsson, Emil Axelsson, and Mary Sheeran.
"Stream processing for embedded domain specific languages".
In: *Proceedings of the 26nd 2014 International Symposium on Implementation and
Application of Functional Languages*. 2014, pp. 1–12.

[150]   Kanat Tangwongsan et al. "General incremental sliding-window aggregation".
In: *Proceedings of the VLDB Endowment* 8.7 (2015), pp. 702–713.

[151]   Jakob Leben. "A programming language based on recurrence equations and
polyhedral compilation for stream processing". PhD thesis. 2019.

[152]   Nuno Miguel Nobre et al. "Bounded Stream Scheduling in Polyhedral OpenStream".
        In: *IMPACT 2020 - 10th International Workshop on Polyhedral Compilation Techniques*.
        Bologna, Italy, Jan. 2020. URL: `https://hal.inria.fr/hal-02441182`.

[153]   E.A. Lee and D.G. Messerschmitt. "Synchronous data flow".
        In: *Proceedings of the IEEE* 75.9 (1987), pp. 1235–1245.
        DOI: `10.1109/PROC.1987.13876`.

[154]   Protonu Basu et al. "Compiler-directed transformation for higher-order stencils".
        In: *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE. 2015,
        pp. 313–323.

[155]   Louis-Noel Pouchet et al.
        "Polyhedral-based data reuse optimization for configurable computing".
        In: *Proceedings of the ACM/SIGDA international symposium on Field programmable
        gate arrays*. 2013, pp. 29–38.

[156]   Kevin Stock et al. "A framework for enhancing data reuse via associative reordering".
        In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language
        Design and Implementation*. 2014, pp. 65–76.

[157]   Prashant Singh Rawat et al. "Domain-specific optimization and generation of
        high-performance GPU code for stencil computations".
        In: *Proceedings of the IEEE* 106.11 (2018), pp. 1902–1920.

[158]   Nitin Chugh et al.
        "A DSL compiler for accelerating image processing pipelines on FPGAs".
        In: *Proceedings of the 2016 International Conference on Parallel Architectures and
        Compilation*. 2016, pp. 327–338.

[159]   M Akif Özkan et al. "AnyHLS: High-level synthesis with partial evaluation".
        In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*
        39.11 (2020), pp. 3202–3214.

[160]   Larisa Stoltzfus et al.
        "Tiling optimizations for stencil computations using rewrite rules in lift". In: *ACM
        Transactions on Architecture and Code Optimization (TACO)* 16.4 (2019), pp. 1–25.

[161]   Bastian Hagedorn.
        "High-performance domain-specific compilation without domain-specific compilers".
        PhD thesis. University of Münster, Germany, 2020.
        URL: `http://d-nb.info/1217852468`.

[162]   Hélène Kirchner. "Rewriting strategies and strategic rewrite programs".
        In: *Logic, Rewriting, and Concurrency*. Springer, 2015, pp. 380–403.

[163]   Bastian Hagedorn et al. "A Language for Describing Optimization Strategies".
        In: *CoRR* abs/2002.02268 (2020). arXiv: `2002.02268`.
        URL: `https://arxiv.org/abs/2002.02268`.

[164]   Mark P Jones and Luc Duponcheel. *Composing monads.* Tech. rep. Technical Report
        YALEU/DCS/RR-1004, Department of Computer Science. Yale …, 1993.

[165]   Sheng Liang, Paul Hudak, and Mark Jones.
        "Monad transformers and modular interpreters". In: *Proceedings of the 22nd ACM
        SIGPLAN-SIGACT symposium on Principles of programming languages.* 1995,
        pp. 333–343.

[166]   Marc Bezem and Jan F Groote. *Typed Lambda Calculi and Applications: International
        Conference on Typed Lambda Calculi and Applications, TLCA'93, March 16-18, 1993,
        Utrecht, The Netherlands. Proceedings.* Vol. 664.
        Springer Science & Business Media, 1993.

[167]   Kento Emoto et al. "Domain-specific optimization strategy for skeleton programs".
        In: *European Conference on Parallel Processing.* Springer. 2007, pp. 705–714.

[168]   Patrenahalli M. Narendra. "A Separable Median Filter for Image Noise Smoothing".
        In: *IEEE Trans. Pattern Anal. Mach. Intell.* 3.1 (1981), pp. 20–29.
        DOI: `10.1109/TPAMI.1981.4767047`.

[169]   Jan-Mark Geusebroek, Arnold W. M. Smeulders, and Joost van de Weijer.
        "Fast anisotropic Gauss filtering".
        In: *IEEE Trans. Image Process.* 12.8 (2003), pp. 938–943.
        DOI: `10.1109/TIP.2003.812429`.

[170]   Vutipong Areekul et al.
        "Separable Gabor filter realization for fast fingerprint enhancement".
        In: *Proceedings of the 2005 International Conference on Image Processing, ICIP 2005,
        Genoa, Italy, September 11-14, 2005.* IEEE, 2005, pp. 253–256.
        DOI: `10.1109/ICIP.2005.1530376`.

[171]   Tuan Q. Pham and Lucas J. van Vliet.
        "Separable bilateral filtering for fast video preprocessing".
        In: *Proceedings of the 2005 IEEE International Conference on Multimedia and Expo,
        ICME 2005, July 6-9, 2005, Amsterdam, The Netherlands.* IEEE, 2005, pp. 454–457.
        DOI: `10.1109/ICME.2005.1521458`.

[172]   Pekka Jääskeläinen et al. "pocl: A Performance-Portable OpenCL Implementation".
        In: *Int. J. Parallel Program.* 43.5 (2015), pp. 752–785.
        DOI: `10.1007/s10766-014-0320-y`.

[173] Sriram Krishnamoorthy et al.
"Effective Automatic Parallelization of Stencil Computations". In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI '07. San Diego, California, USA: ACM, 2007, pp. 235–244. ISBN: 9781595936332. DOI: 10.1145/1250734.1250761.

[174] Kaushik Datta et al. "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures". In: *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2008, November 15-21, 2008, Austin, Texas, USA.* IEEE/ACM, 2008, p. 4. DOI: 10.1109/SC.2008.5222004.

[175] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. "High-Performance Code Generation for Stencil Computations on GPU Architectures". In: *Proceedings of the 26th ACM International Conference on Supercomputing.* ICS '12. San Servolo Island, Venice, Italy: ACM, 2012, pp. 311–320. ISBN: 9781450313162. DOI: 10.1145/2304576.2304619.

[176] Xing Zhou et al. "Hierarchical Overlapped Tiling". In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization.* CGO '12. San Jose, California: ACM, 2012, pp. 207–218. ISBN: 9781450312066. DOI: 10.1145/2259016.2259044.

[177] Yisu Remy Wang et al. "SPORES: sum-product optimization via relational equality saturation for large scale linear algebra". In: *arXiv preprint arXiv:2002.07951* (2020).

[178] Chandrakana Nandi et al. "Synthesizing structured CAD models with equality saturation and inverse transformations". In: *Proceedigngs of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation.* 2020, pp. 31–44.

[179] Yichen Yang et al. "Equality saturation for tensor graph superoptimization". In: *Proceedings of Machine Learning and Systems* 3 (2021).

[180] Gus Henry Smith et al.
"Pure tensor program rewriting via access patterns (representation pearl)". In: *arXiv preprint arXiv:2105.09377* (2021).

[181] Alexa VanHattum et al.
"Vectorization for Digital Signal Processors via Equality Saturation". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS '21. Virtual, USA: Association for Computing Machinery, 2021, pp. 874–886. ISBN: 9781450383172. DOI: 10.1145/3445814.3446707. URL: https://doi.org/10.1145/3445814.3446707.

[182] Chandrakana Nandi et al. "Rewrite Rule Inference Using Equality Saturation".
In: *Proc. ACM Program. Lang.* 5.OOPSLA (Oct. 2021). DOI: 10.1145/3485496.
URL: https://doi.org/10.1145/3485496.

[183] Pavel Panchekha et al.
"Automatically improving accuracy for floating point expressions".
In: *ACM SIGPLAN Notices* 50.6 (2015), pp. 1–11.

[184] Chenming Wu et al. "Carpentry compiler".
In: *ACM Transactions on Graphics (TOG)* 38.6 (2019), pp. 1–14.

[185] Kyle Gallivan et al.
"Impact of hierarchical memory systems on linear algebra algorithm design".
In: *The International Journal of Supercomputing Applications* 2.1 (1988), pp. 12–48.

[186] Michael Wolfe. "More iteration space tiling".
In: *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*. 1989, pp. 655–664.

[187] Smail Kourta et al. "Caviar: an e-graph based TRS for automatic code optimization".
In: *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*. 2022, pp. 54–64.

[188] Savvas Sioutas et al. "Schedule synthesis for halide pipelines on gpus".
In: *TACO* (2020).

[189] A Solar Lezama. "Program synthesis by sketching".
PhD thesis. PhD thesis, EECS Department, University of California, Berkeley, 2008.

[190] Lennart Augustsson, Mikael Rittri, and Dan Synek.
"Functional Pearl: On generating unique names".
In: *Journal of Functional Programming* 4.1 (1994), pp. 117–123.
DOI: 10.1017/S0956796800000988.

[191] N.G 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)* 75.5 (1972), pp. 381–392. ISSN: 1385-7258.

[192] Krzysztof Maziarz et al. "Hashing modulo alpha-equivalence".
In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021, pp. 960–973.

[193] Maribel Fernández and Murdoch J Gabbay. "Nominal rewriting".
In: *Information and Computation* 205.6 (2007), pp. 917–965.

[194] Dan R. Ghica. "Operational Semantics with Hierarchical Abstract Syntax Graphs".
In: *Electronic Proceedings in Theoretical Computer Science* 334 (Feb. 2021), pp. 1–10.
DOI: 10.4204/eptcs.334.1.
URL: https://doi.org/10.4204%2Feptcs.334.1.

[195] Eduardo Bonelli, Delia Kesner, and Alejandro Ríos.
"A de Bruijn notation for higher-order rewriting".
In: *International Conference on Rewriting Techniques and Applications*. Springer. 2000, pp. 62–79.

[196] Fairouz Kamareddine and Alejandro Ríos.
"A λ-calculus à la de Bruijn with explicit substitutions". In: *International Symposium on Programming Language Implementation and Logic Programming*. Springer. 1995, pp. 45–62.

[197] Dan Benanav, Deepak Kapur, and Paliath Narendran.
"Complexity of matching problems".
In: *Journal of symbolic computation* 3.1-2 (1987), pp. 203–216.

[198] Łukasz Lachowski et al. "On the complexity of the standard translation of lambda calculus into combinatory logic".
In: *Reports on Mathematical Logic* 53 (2018), pp. 19–42.

[199] Jean-Christophe Filliâtre and Sylvain Conchon. "Type-safe modular hash-consing".
In: *Proceedings of the 2006 Workshop on ML*. 2006, pp. 12–19.

[200] Gereon Kremer, Florian Corzilius, and Erika Ábrahám.
"A generalised branch-and-bound approach and its application in SAT modulo nonlinear integer arithmetic".
In: *International Workshop on Computer Algebra in Scientific Computing*. Springer. 2016, pp. 315–335.

[201] Yu-Sheng Hsieh and Yi-Ping You.
"DLOOPT: An Optimization Assistant on AutoTVM for Deep Learning Operators".
In: *Journal of Signal Processing Systems* (2022), pp. 1–23.

[202] Arthur Charguéraud et al.
"OptiTrust: an Interactive Framework for Source-to-Source Transformations".
In: (2022).

[203] Siyuan Feng et al.
TensorIR: An Abstraction for Automatic Tensorized Program Optimization. 2022.
DOI: `10.48550/ARXIV.2207.04296`.
URL: `https://arxiv.org/abs/2207.04296`.

[204] Emina Torlak and Rastislav Bodik. "Growing solver-aided languages with Rosette".
In: *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. 2013, pp. 135–152.

[205] Syed Waqar Nabi and Wim Vanderbauwhede.
"Automatic pipelining and vectorization of scientific code for FPGAs".
In: *International Journal of Reconfigurable Computing* 2019 (2019).

[206] Thierry Coquand and Gérard Huet. "The calculus of constructions".
PhD thesis. INRIA, 1986.

[207] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring.
"The coq proof assistant a tutorial". In: *Rapport Technique* 178 (1997).

[208] Chris Lattner et al. "MLIR: A compiler infrastructure for the end of Moore's law".
In: *arXiv preprint arXiv:2002.11054* (2020).

[209] Martin Lücke, Michel Steuwer, and Aaron Smith.
"Integrating a functional pattern-based IR into MLIR". In: *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction.* 2021, pp. 12–22.

[210] Michael J Gordon, Arthur J Milner, and Christopher P Wadsworth.
Edinburgh LCF: a mechanised logic of computation. Springer, 1979.

[211] Freek Wiedijk. "Formal proof sketches".
In: *International Workshop on Types for Proofs and Programs.* Springer. 2003,
pp. 378–393.

[212] Pierre Corbineau. "A declarative language for the Coq proof assistant".
In: *International Workshop on Types for Proofs and Programs.* Springer. 2007, pp. 69–84.

[213] Charles Gregory Nelson. *Techniques for program verification.*
Stanford University, 1980.

[214] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver".
In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer. 2008, pp. 337–340.

[215] David Goldberg.
"What every computer scientist should know about floating-point arithmetic".
In: *ACM computing surveys (CSUR)* 23.1 (1991), pp. 5–48.

[216] Eva Darulova and Viktor Kuncak. "Towards a compiler for reals". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 39.2 (2017), pp. 1–28.

[217] Heiko Becker et al.
"Verified Compilation and Optimization of Floating-Point Programs in CakeML".
In: *36th European Conference on Object-Oriented Programming (ECOOP 2022).*
Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2022.