THE UNIVERSITY
of EDINBURGH

Automatic Performance Optimisation of Parallel Programs for GPUs via Rewrite Rules

Toomas Remmelg

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2019

Abstract

Graphics Processing Units (GPUs) are now commonplace in computing systems and are the most successful parallel accelerators. Their performance is orders of magnitude higher than traditional Central Processing Units (CPUs) making them attractive for many application domains with high computational demands. However, achieving their full performance potential is extremely hard, even for experienced programmers, as it requires specialised software tailored for specific devices written in low-level languages such as OpenCL. Differences in device characteristics between manufacturers and even hardware generations often lead to large performance variations when different optimisations are applied. This inevitably leads to code that is not performance portable across different hardware.

This thesis demonstrates that achieving performance portability is possible using LIFT, a functional data-parallel language which allows programs to be expressed at a high-level in a hardware-agnostic way. The LIFT compiler is empowered to automatically explore the optimisation space using a set of well-defined rewrite rules to transform programs seamlessly between different high-level algorithmic forms before translating them to a low-level OpenCL-specific form.

The first contribution of this thesis is the development of techniques to compile functional LIFT programs that have optimisations explicitly encoded into efficient imperative OpenCL code. Producing efficient code is non-trivial as many performance sensitive details such as memory allocation, array accesses or synchronisation are not explicitly represented in the functional LIFT language. The thesis shows that the newly developed techniques are essential for achieving performance on par with manually optimised code for GPU programs with the exact same complex optimisations applied.

The second contribution of this thesis is the presentation of techniques that enable the LIFT compiler to perform complex optimisations that usually require from tens to hundreds of individual rule applications by grouping them as *macro-rules* that cut through the optimisation space. Using matrix multiplication as an example, starting from a single high-level program the compiler automatically generates highly optimised and specialised implementations for desktop and mobile GPUs with very different architectures achieving performance portability.

The final contribution of this thesis is the demonstration of how low-level and GPU-specific features are extracted directly from the high-level functional LIFT program, enabling building a statistical performance model that makes accurate predictions about the performance of differently optimised program variants. This performance model is then used to drastically speed up the time taken by the optimisation space exploration by ranking the different variants based on their predicted performance.

Overall, this thesis demonstrates that performance portability is achievable using LIFT.

Lay Summary

In recent years the devices originally developed to accelerate the creation of images for displaying on computer screens (Graphics Processing Units or GPUs) have found widespread adoption for performing other computational tasks. GPUs are found in virtually every computer system, almost all desktop PCs, mobile devices and tablets, as well as supercomputers all contain GPUs. GPUs are now being used to solve computationally intensive problems in domains such as biology, chemistry, physics, economics and machine learning.

GPU programming is very challenging and requires expert knowledge about the hardware details of GPUs to make efficient use of them. To make matters worse, programs written in existing GPU programming languages have large performance variations when being run on different GPU models. This is because of the differences between the designs of different manufacturers as well as rapidly evolving architectures in the pursuit of higher performance and better energy efficiency.

This thesis presents novel techniques to ease the programming of GPUs by using a high-level functional programming language that is capable of automatically creating different program variations using rewrite rules and exploring the options to choose suitable implementations for different GPU models. The results show that these techniques offer performance on par with highly-tuned libraries written by experts, while greatly simplifying the development process.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material in this thesis has been published in the following papers:

- Toomas Remmelg, Thibaut Lutz, Michel Steuwer, and Christophe Dubach. *Performance Portable GPU Code Generation for Matrix Multiplication.* In Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit (GPGPU '16).

- Michel Steuwer, Toomas Remmelg, and Christophe Dubach. *Matrix Multiplication Beyond Auto-Tuning: Rewrite-based GPU Code Generation.* In Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES '16).

- Michel Steuwer, Toomas Remmelg, and Christophe Dubach. *LIFT: A Functional Data-Parallel IR for High-Performance GPU Code Generation.* In Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO '17).

(*Toomas Remmelg*)

Table of Contents

Chapter 1

Introduction

The design of computer architectures is going through major changes as a result of the end of Dennard scaling [Denn 74], as transistors get smaller their power density no longer stays constant and the operating voltage can no longer be reduced. This results in an inability to increase clock-frequencies without increasing the overall power consumption. As Moore's law [Moor 65] is also coming to an end, the industry is turning to multi-core solutions and specialised hardware designs to further increase the performance of computer systems [Henn 19].

Programmers now have to share a much larger part of the burden of achieving performance, instead of being able to rely on constantly rising clock-frequencies and getting performance gains for free. They are now forced to write parallel programs to be able to harness the power of multi-core processors. Compared to sequential programming, the additional complexity of communication and synchronisation has to be correctly handled to avoid new types of problems, such as race conditions, deadlocks and non-deterministic behaviour.

In addition to multi-core CPUs, specialised hardware designs such as GPUs and other parallel accelerators are now commonplace in computing systems and available for performing general purpose computation. GPUs are found in virtually all desktop PCs, mobile devices, as well as supercomputers. Their computational performance is orders of magnitude higher than that of traditional CPUs making them attractive for many application domains with high computational demands. However, achieving the full performance potential of these hardware devices is extremely hard, even for experienced programmers, as it requires specialised kernels written in low-level languages such as OpenCL to take advantage of specific hardware features.

Optimising programs is crucial for achieving high performance and performance requirements are usually the key reasons for using parallel accelerators. Figure 1.1 shows how much performance there is to gain from optimising matrix multiplication for an AMD GPU. A naïve textbook implementation is the baseline shown as the leftmost bar. A vendor provided library implentation reaches 5x of the performance of the naïve implementation out-of-the-box or more than 7.5x after tuning the library to the specific GPU used by picking the best performing con-

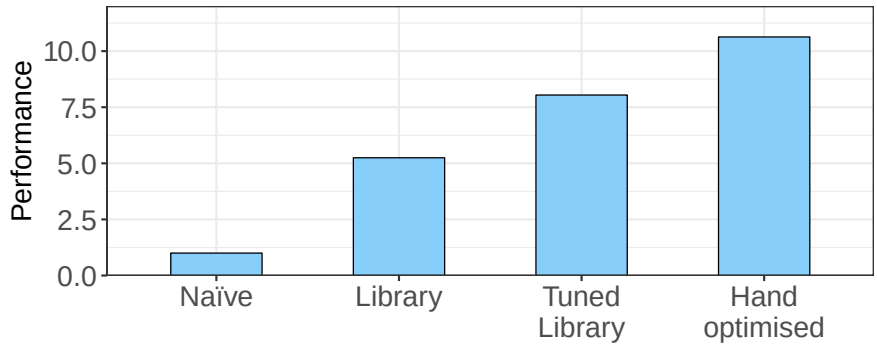


Figure 1.1: Performance comparison of matrix multiplication implementations on an AMD GPU. From Chapter 5.

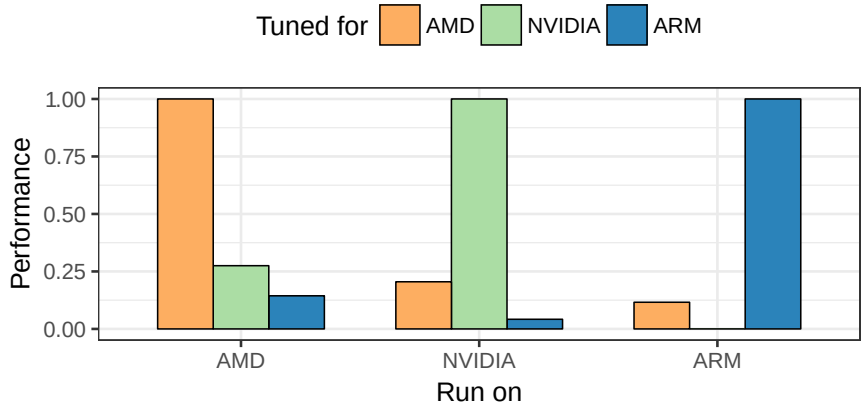


Figure 1.2: Relative performance of matrix multiplication implementations tuned for different GPUs being run on others. From Chapter 5.

figuration of implementation parameters. A carefully hand-crafted and optimised implementation for the specific GPU can in many cases be even faster than a tuned library implementation, in this case reaching more than 10x of the performance of the naïve implementation. Optimising programs, therefore, makes a huge difference in terms of the performance and is very important to pay attention to when programming GPUs.

Unfortunately, performance optimisations are not portable across different hardware devices. To achieve high-performance, programs are tailored to specific hardware devices with different characteristics resulting in large performance variations when different optimisations are applied. Figure 1.2 shows the performance of different implementations of matrix multiplication being run on different GPUs. In this experiment, an implementation running on a specific GPU but tuned for a different one reaches only a fraction of the performance. For example, the implementation tuned for an NVIDIA GPU running on an AMD GPU reaches only 27.5% of the available performance. All other cases in Figure 1.2 are even worse, such as the implementation optimised for an ARM GPU running on an NVIDIA GPU, achieving only 4.2% of the available performance. At worst, the implementation will not even run because

of different resource constraints like the implementation for an NVIDIA GPU on an ARM GPU. The fact that programs tuned for one device perform poorly on others is know as the performance portability problem.

The end of Dennard scaling and the end of Moore's law are accelerating innovation in hardware design and new hardware devices are being designed faster than ever. Therefore, problems with performance portability are becoming more acute. One major factor of new hardware designs is to provide high performance for domains, such as machine learning, computer vision and physics simulations, that require large amounts of computational power. To develop lasting software for these important domains, a way to achieve high performance without resorting to low-level non-performance portable solutions is highly desirable.

To tackle the performance portability problem, this thesis argues that a declarative, not imperative, high-level programming model is needed, allowing the underlying compiler to optimise for a wide set of hardware devices. The compiler is the key component responsible for transforming a high-level program into low-level code that performs well on different devices. To build such a compiler we need:

- A capability to automatically explore optimisation choices for specialising high-level programs to low-level programs that explicitly encode optimisation choices. This will enable users to write programs in a high-level hardware-agnostic way without committing to a particular implementation, as well as facilitate code generation from the explicit low-level representation.
- To be able to compile a low-level functional program with optimisations explicitly encoded into efficient low-level imperative code. This will solve the problem that while a functional language is useful for exploring different optimisation choices, the code that actually runs on the device must be imperative.
- A fast explorative optimisation process that can quickly evaluate which implementations of a program are worth considering for achieving high-performance. This will enable the optimisation process to be used in practice in a reasonable timeframe while exploring a large space of optimisation choices.

1.1 Contributions

This thesis addresses the needs identified above by making the following contributions to tackle performance portability by extending the high-level programming language LIFT first introduced in [Steu 15b].

- To address the need for translating from a low-level functional program to an imperative one, this thesis presents novel techniques to compile functional LIFT programs that en-

code optimisations explicitly into efficient OpenCL code. While all optimisation choices are encoded explicitly, producing efficient imperative code is still a non-trivial task as many performance sensitive implementation details, such as memory allocation, array accesses and synchronisation, are not explicitly represented in the functional LIFT programs. Chapter 4 shows that common but complex optimisations in GPU programming are expressible in low-level LIFT programs. Chapter 4 then introduces the newly developed techniques that are required to achieve performance on par with manually optimised code applying the exact same optimisations.

- To address the need for an exploration capability, the thesis presents techniques enabling LIFT to apply complex optimisations comprising of tens or hundreds of individual rule applications. This is achieved by adding additional rewrite rules and a capability to group them as provably correct *macro-rules* to cut through the optimisation space. Using matrix multiplication as an example, starting from a single high-level program the compiler automatically generates highly optimised and specialised implementations for desktop and mobile GPUs with very different architectures, achieving true performance portability. This work is described in Chapter 5.

- To address the need for a fast explorative optimisation process, a performance model is built and used for drastically speeding up the optimisation space exploration. To do this, low-level, GPU-specific features are extracted directly from the functional LIFT program and a statistical performance model is built using these features to accurately predict the performance of different program variants. This work is presented in Chapter 6.

The work in this thesis goes significantly beyond the original version of LIFT presented in [Steu 15b] and summarised in Chapter 2, Section 2.4. The shortcomings of the original LIFT approach and implementation that significantly limit its scalability and applicability for more complex applications are detailed in section 2.4.5. This thesis addresses these shortcomings and enables, for the first time, performance portability of complex applications, like matrix multiplication, in LIFT.

1.2 Thesis Outline

The rest of the thesis is organised as follows:

Chapter 2 presents the technical background needed to understand the thesis. It describes different GPU architectures used in this thesis; the OpenCL standard for general purpose programming across CPUs, GPUs and other devices; the high-level functional LIFT data-parallel

language that targets OpenCL; and finally, a summary of design space exploration and auto-tuning techniques.

Chapter 3 describes related work. It introduces and discusses prior work about high-level approaches for GPU programming using libraries and GPU specific languages; different compilers, compiler building frameworks and intermediate representations for GPUs; auto-tuning systems for parameter selection and techniques for exposing and choosing between different optimisation choices; and finally analytical and statistical performance modelling for GPUs.

Chapter 4 presents the first contributon by describing the compilation of low-level functional LIFT programs into low-level imperative OpenCL code and the optimisations that are applied during this process. It presents newly developed techniques for address space inference, memory allocation, array access generation, barrier elimination and the OpenCL code generation that are required to produce high-performance GPU code that is on par with manually written OpenCL code applying the same complex optimisations.

Chapter 5 presents the second contribution by showing how the optimisation process takes place. It shows how complex optimisations are represented in LIFT, the rewrite rules that are used as building blocks, the *macro-rules* built out of them and the process for exploring the optimisation space to achieve performance portability across different GPUs starting from a single high-level program.

Chapter 6 presents the third contrbution by describing building a performance model for speeding up the exploration of the optimisation space. It shows how different features about parallelism, memory, control flow and synchronisation are extracted directly from LIFT programs and how those features are used to build a performance model. It analyses the performance of the model and shows how it is used to drastically speed up the exploration of the optimisation space.

Chapter 7 concludes the thesis by summarising the contributions, analysing their limitations and discussing possible future extensions.

Chapter 2

Background

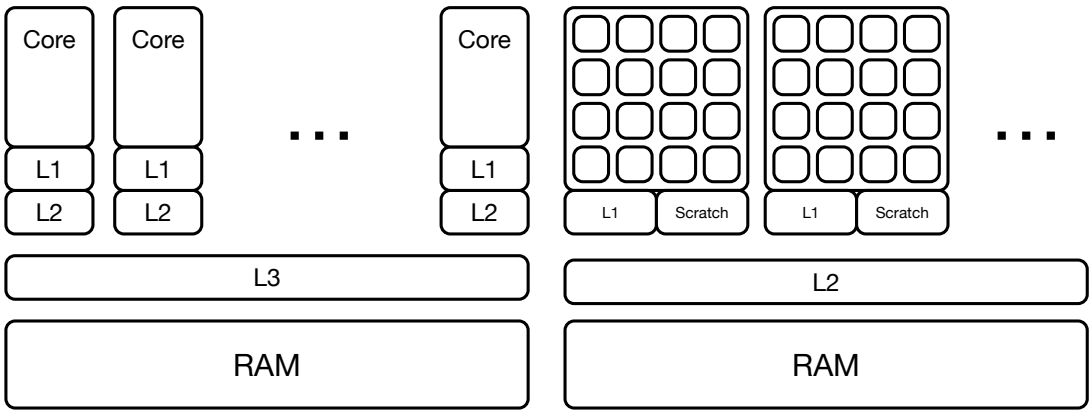
This chapter introduces the technical background required to understand the thesis. Section 2.1 and Section 2.2 describe the hardware that is used and optimised for in this thesis. A distincton between desktop and mobile GPUs is established as these have very different design and performance characteristics. Section 2.3 describes OpenCL, the low-level language and framework for programming and expressing the optimisations for the hardware. Section 2.4 describes LIFT, the high-level programming language that the optimisation process takes place in and OpenCL code is generated from. Section 2.5 describes different approaches of design space exploration for targeting different types of hardware and trying to achieve performance portability. Section 2.6 summarises the chapter.

2.1 Desktop GPU Architectures

While Graphics Processing Units (GPUs) were originally designed for rendering graphics, they are now also widely used for general purpose workloads. Compared to Central Processing Units (CPUs) they are designed radically differently and offer higher performance per watt, making them attractive for performing computationally intensive tasks.

Figure 2.1 shows a high-level comparison of CPU and GPU architectures. Both architectures are composed of cores that perform computation and memory for storing data. Random-access memory (RAM) is off chip memory for storing data used by the processor at runtime. A core is a processing unit executing instructions for a single thread. A cache is a small fast memory for frequently used data that is managed in hardware.

CPUs (Figure 2.1a) contain a small number of complex cores designed to extract instruction level parallelism and run a single thread of execution fast. CPU architectures focus on reducing the latency of operations using features such as several levels of large caches between the cores and off chip RAM (per core L1, L2 and a shared L3 in Figure 2.1a), pipelining, out-of-order execution and branch prediction.



(a) CPU. A small number of complex cores. (b) GPU. A large number of simple cores.

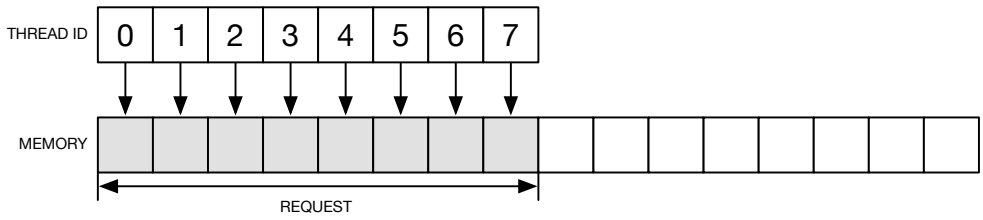
Figure 2.1: High-level comparison of CPU and GPU architectures

Desktop GPUs (Figure 2.1b) on the other hand are throughput oriented and focus on increasing the overall throughput rather than the latency of a single task. Desktop GPU architectures contain a large number of simple cores generally grouped into clusters, with cores in the same cluster sharing resources such as cache space, scratchpad memory space and registers. Scratchpad memory is a small fast addressable memory that serves as a programmer managed cache. The clusters of cores are connected to off chip RAM, usually via an L2 cache. GPU architectures rely on a large number of threads and thread level parallelism to hide memory latencies by swapping out threads that are waiting for memory requests. The rest of this section investigates the design of desktop GPU architectures in more depth.

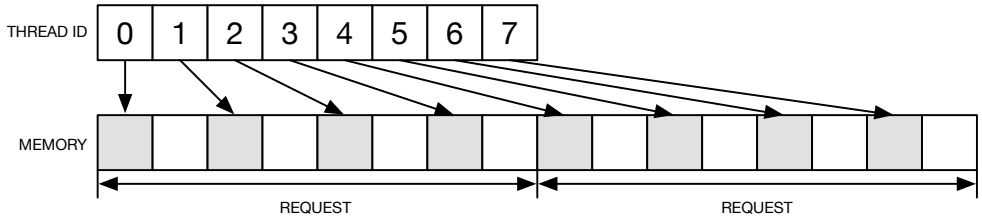
Execution Model On Desktop GPUs threads are scheduled for execution in groups that are called warps or wavefronts, usually of 32 or 64 threads. Every thread in a warp is executing the same instruction in a lock-step manner but on different data. This execution model is called *single instruction, multiple threads* (SIMT) and is similar to *single instruction, multiple data* (SIMD). In SIMT every data element is operated on by a different thread which is allowed to follow a different path of control flow, which is not possible in the SIMD model.

As a consequence of the lock-step execution threads in a warp cannot independently execute different paths of code. If some threads follow a different execution path, they will need to wait until the first path is finished before executing theirs. This means that threads within a warp should avoid divergent execution paths to avoid needing to pause their execution. Avoiding divergent execution within a warp is an important optimisation which affects how GPU software is implemented.

Memory Hierarchy The memory hierarchy on desktop GPUs consists of several layers of different types of memory. It includes main memory, an L2 cache shared by all cores, an L1



(a) Coalesced memory access combined into a single memory request.



(b) Strided memory access requiring several memory requests.

Figure 2.2: Memory access patterns. Each request will load an entire cache line.

cache, scratchpad memory and registers.

Accessing main memory is very expensive, taking several hundred clock cycles, but GPUs can optimise simultaneous memory accesses with certain patterns. This is achieved by *coalescing* memory requests by several threads into a single memory request to main memory. The best performance is achieved when consecutive threads access consecutive memory locations and all accesses of a warp fall within the same cache line as shown in Figure 2.2a. This means only a single memory request needs to be issued instead of 32 individual requests. Figure 2.2b shows a strided memory access pattern, where several loads need to be issued. In extreme cases, only a single element of a cache line might be used, while the whole cache line will be loaded from memory, effectively wasting available memory bandwidth. Coalescing is one of the most important optimisations and not using these access patterns means underutilising the memory bandwidth and possibly severely hurting performance.

In addition to caches, which are comparably small compared to the ones on CPUs and shared between a much larger number of threads, desktop GPUs feature small and fast scratchpad memories shared among cores. These memories serve as programmer managed caches. A group of collaborating threads can cooperatively copy data into the scratchpad memory where it can then be reused by all of them. The scratchpad memory is not nearly as sensitive to memory access patterns as the main memory and it is also beneficial to use it for accesses that are not coalesced. Finally, since the scratchpad memory is shared among threads, it can be used for communication with other threads in the same cluster of cores.

Compared to CPUs, the register files on a desktop GPU are much larger to enable the execution of a large number of threads. Using too many registers per thread limits the total amount of threads that can run concurrently.

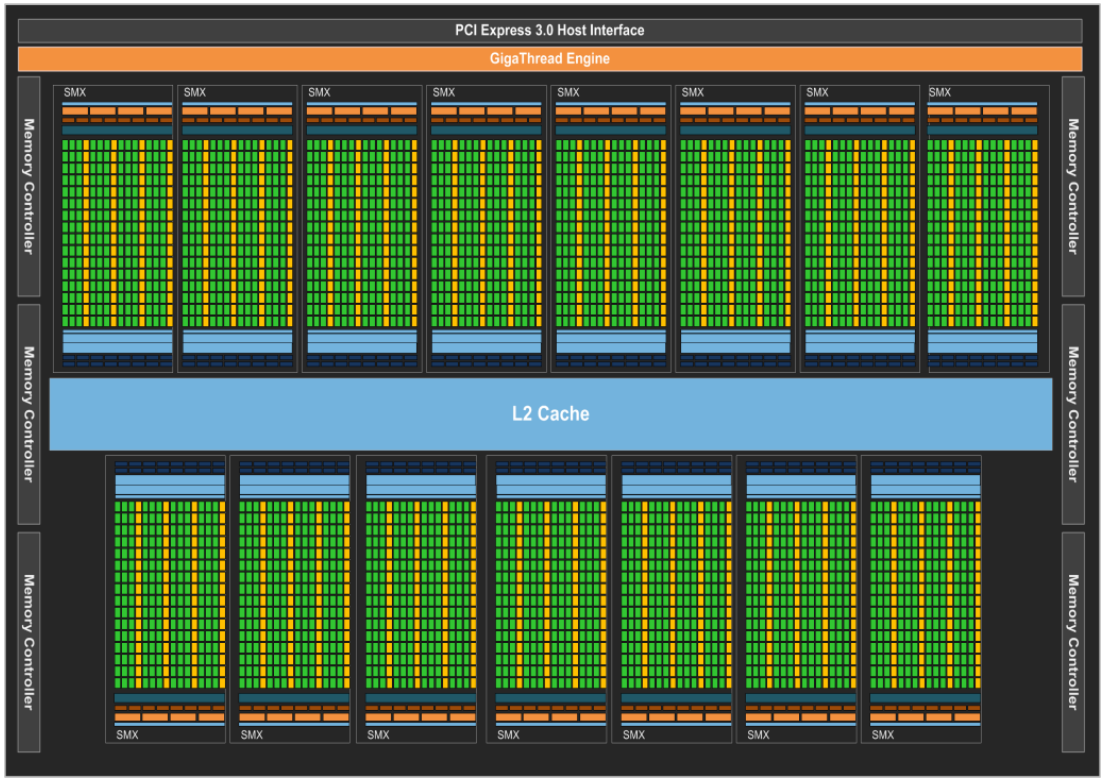


Figure 2.3: The NVIDIA Kepler GK110 architecture. From [NVID 14].

Next, two concrete desktop GPU architectures that are used in this thesis will be discussed.

2.1.1 NVIDIA Kepler GPU Architecture

The NVIDIA Kepler [NVID 14] micro-architecture was designed for efficiency, performance and programmability. It is NVIDIA's third general purpose GPU architcture after Tesla and Fermi. Figure 2.3 presents an overview of the architecture. An implementation has up to 15 clusters of cores called streaming multiprocessors (SMX in Figure 2.3) and a 1.5 MB dedicated L2 cache shared by all SMXs.

An SMX schedules threads in warps of 32 threads each, with up to 64 warps or 2,048 threads in total per SMX. Each SMX contains four warp schedulers and eight instruction dispatch units. In each cycle, up to four warps and two independent instructions from each warp can be selected to be dispatched.

Each Kepler SMX contains 192 single-precision CUDA cores, 64 double-precision arithmetic units, 32 special function units and 32 load/store units. Each CUDA core has a pipelined floating-point and integer unit. Each SMX contains a register file of 65,536 32-bit registers with up to 255 registers allowed per thread. However, using 255 registers per thread limits the number of threads that can run concurrently on a streaming multiprocessor to only 256. If each thread uses more than 32 registers each, the number of threads that can be scheduled

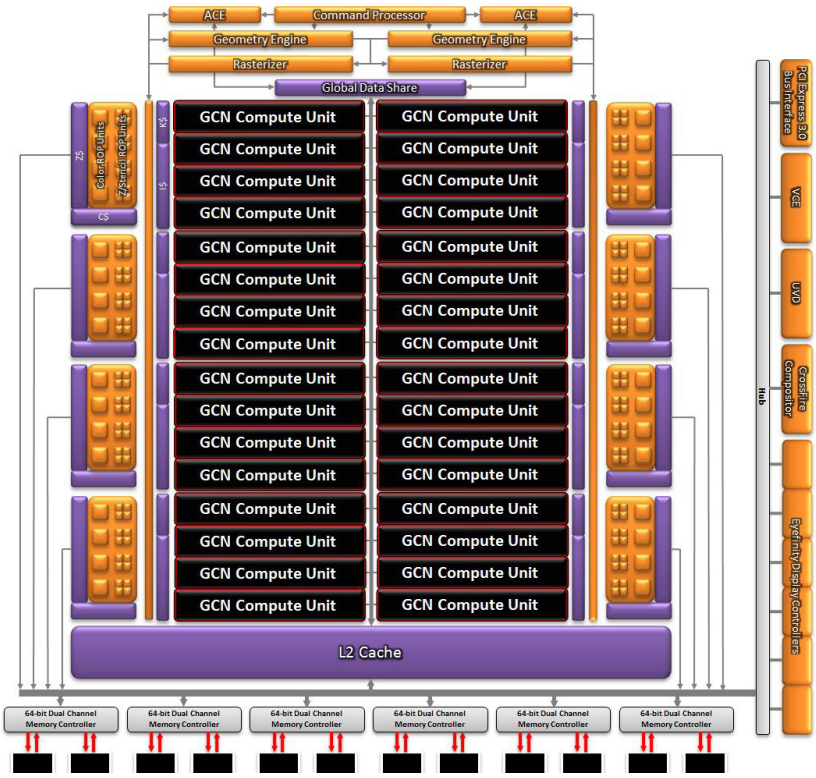


Figure 2.4: The AMD Radeon HD 7970 architecture. From [AMDI 12].

will start dropping. It is worth noting that for some applications, using more registers and less parallelism is beneficial, while for others it is the exact opposite. Each SMX has a 48KB read-only data cache and 64KB memory that is divided between a scratchpad memory and L1 cache (configurable with 48KB for one and 16KB for the other). That is only a few dozens of bytes per thread compared to tens of KBs of L1 cache space per thread on a CPU.

2.1.2 AMD Graphics Core Next GPU Architecture

Graphics Core Next (GCN) [AMDI 12] was designed to be more suitable for general purpose computation than the earlier TeraScale architecture. Figure 2.4 presents an overview of the architecture. It is a SIMT architecture like the NVIDIA Kepler architecture. The basic building block of a CGN GPU is a GCN Compute Unit (CU). A typical number of compute units is 32, like shown in Figure 2.4. Like on NVIDIA Kepler, the L2 cache is shared by all compute units. The size of the L2 cache is 768 KB or 1MB for the AMD GPUs used in this thesis.

Each CU contains 4 SIMD units, each of which simultaneously executes a single instruction across 16 threads. Each SIMD unit manages the execution of up to 10 wavefronts of 64 threads each. There is a 64KB register file per SIMD unit. Each CU also has a scalar unit for operating on values common to all threads, such as control flow instructions. There are 16KB L1 read-write cache and a 64KB scratchpad memory per CU. To achieve memory coalescing,

a quarter of a wavefront (16 threads) must be all be accessing consecutive memory locations. The emphasis is on finding parallel wavefronts to execute rather than independent operations from a single wavefront.

2.2 ARM Mali Midgard Mobile GPU Architecture

As mobile GPUs have stricter limitations in terms of space and energy budget they are designed differently from desktop GPUs. Mobile GPUs feature less parallelism than desktop GPUs but still more than traditional CPUs. As a result, different optimisations are crucial for achieving high performance than on desktop GPUs. Optimisation techniques for the ARM Mali mobile GPU, are discussed in the ARM documentation [ArmL 13] as well as in [Gras 14] and [Gron 14]. These are not aligned and quite often contradict the advice given by AMD or NVIDIA for their GPUs. This leads to a large performance portability gap when executing kernels optimised for a desktop GPU on the Mali GPU, or vice versa.

Vectorisation is one of the most important optimisations given the SIMD architecture of the Mali GPU. Arithmetic operations performed on vector values are executed by the hardware SIMD units. Performing vectorised memory operations reduces the number of load and store instructions issued and helps to better utilise the memory bandwidth.

While vectorisation is a crucial optimisation for the Mali GPU, it is usually not beneficial on AMD and NVIDIA desktop GPUs, as they do not have hardware arithmetic vector units. Therefore, code optimised for AMD and NVIDIA desktop GPUs will most likely make no use of vector data types and perform poorly on mobile Mali GPUs.

Register pressure is an extremely important topic on the Mali GPU, as the number of registers is small and it influences the amount of threads managed by the hardware. Therefore, reducing the number of registers used increases the amount of active threads which helps to hide memory latencies and keeps the cores busy. While register pressure is also important on desktop GPUs, there are more registers available and the degree of thread level parallelism degrades more gracefully than on Mali.

There exist some optimisations for AMD and NVIDIA GPUs which are not beneficial on the Mali GPU. While using scratchpad memory is crucial for good performance on desktop GPUs, the Mali does not have a dedicated scratchpad memory. Any attempt use to scratchpad memory in programs will result in using main memory instead.

Memory accesses to the main memory are coalesced on AMD and NVIDIA if all threads in the same execution batch access consecutive memory locations. Optimising code for coalesced accesses is hugely beneficial on these architectures, whereas on Mali this might increase cache misses and memory accesses should be vectorised instead.

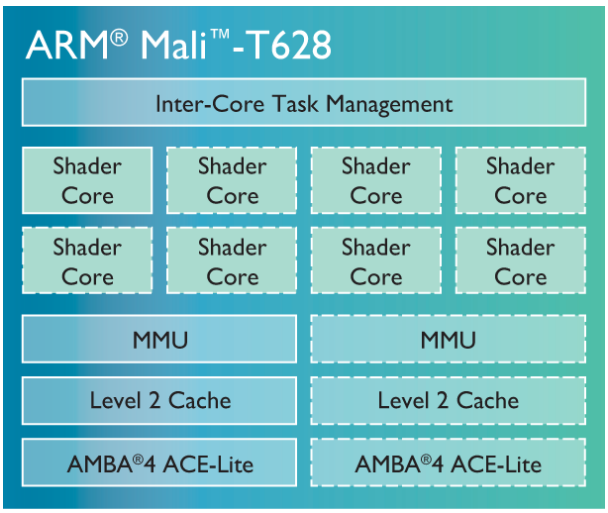


Figure 2.5: Architecture diagram of the ARM Mali-T628 GPU. From [ArmL 19].

2.2.1 ARM Mali-T628 GPU

The Mali-T628 GPU is a mobile GPU implementing ARMs second generation Midgard micro-architecture. A high-level overview is shown in Figure 2.5. Vendors who integrate a Mali-T628 GPU into their system on a chip can choose the number of shader cores which can vary between 1 and 8. Cores are organised into groups of up to four. There is an L2 cache per core group which is shared by all of the shader cores in the group. The size of this is configured by the vendor, but is typically 32KB per shader core.

Each core has two arithmetic pipelines, each of which processes 128-bits of data at a time using SIMD operations. A single core can simultaneously manage up to 256 threads in hardware, depending on the amount of registers required by each thread. A kernel using more than 4 128-bit registers reduces the number of simultaneously executed work items from 256 to 128. If the kernel uses more than 8 registers, the amount of threads halves again. This large number of threads is used to hide memory latencies, as stalled threads waiting for memory can be overtaken by other threads. There are two 16KB L1 data caches per shader core; one for texture access and one for generic memory access.

We have seen that there are significant differences between desktop and mobile GPUs, requiring different optimisation strategies to achieve performance. Next, the OpenCL low-level programming model will be discussed. OpenCL allows people to program and manually optimise software for these GPUs.

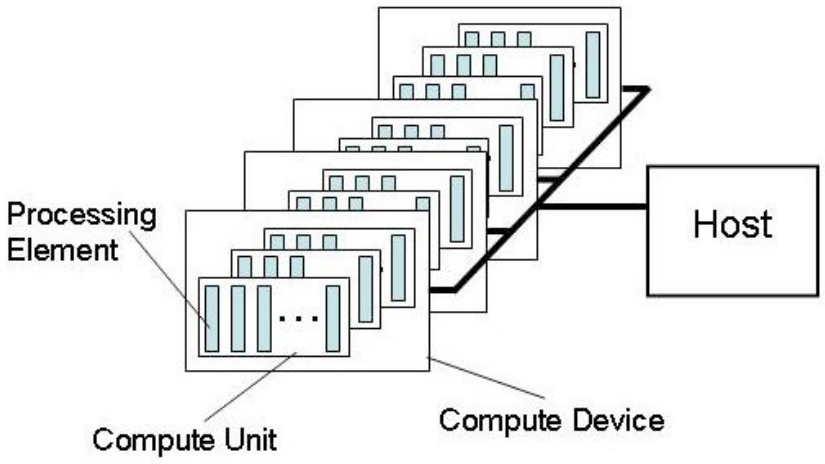


Figure 2.6: The OpenCL platform model. From [Khro 12].

2.3 OpenCL

OpenCL is a framework and language for programming heterogeneous systems consisting of different types of devices such as Central Processing Units (CPUs), Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), Digital Signal Processors (DSPs) and other accelerators in a portable manner. [Khro 12] Version 1.2 of the specification will be described and it is the most widely implemented version by hardware vendors, including NVIDIA, AMD and ARM, to support their devices.

2.3.1 Platform Model

An OpenCL platform consists of a host device connected to one or more OpenCL devices. Each OpenCL compute device contains one or more compute units and each compute unit contains one or more processing elements. Figure 2.6 shows a visual representation of a host device on the right connected to multiple compute devices on the left. An OpenCL program runs on the host device and queues commands to execute computations on a compute device.

2.3.2 Execution Model

An OpenCL program is divided into two parts: a host program and device kernels. The host program is responsible for managing OpenCL devices, moving data between the host and device and enqueueing kernels for the devices to execute.

The host program needs to create a *context* for the devices on an OpenCL platform it wishes to use. In addition to devices, the context will contain kernels, program objects and memory objects. The host program then needs to create a *command-queue* for a device to coordinate the execution of kernels. The host will enqueue commands into the *command-queue* which will be

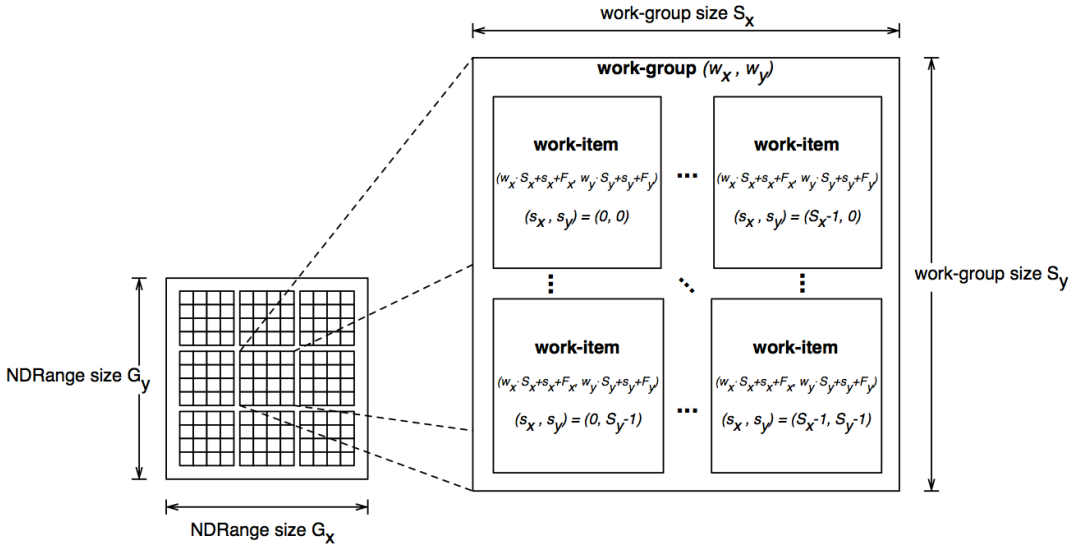


Figure 2.7: A two-dimensional `NDRange`. From [Khro 12].

scheduled for the device to execute. These commands include:

- kernel execution commands,
- memory commands to transfer data to, from or between memory objects,
- and synchronisation commands.

When the host enqueues a kernel it provides a N-dimensional index space (`NDRange`) configuration. The kernel is executed once for each point in this index space possibly in parallel. Each point in an `NDRange` is called a *work-item* and is generally mapped to a thread executing on a GPU. Each work-item is identified by a unique global-id within the `NDRange`.

Work-items are further organised into *work-groups*. Each work-group is identified by a unique work-group-id that has the same dimensionality as the `NDRange`. Each work-item also has a unique local-id within a work-group. Work-items can be uniquely identified by their global-id or a combination of their local-id and work-group-id.

An example of a two-dimensional `NDRange` is shown in Figure 2.7. The `NDRange` is (G_x, G_y), the work-group size is (S_x, S_y). The number of work-groups is $(W_x, W_y) = (G_x/S_x, G_y/S_y)$.

Work group indices are (w_x, y_y), work-item global indices are (g_x, g_y), and work-item local indices are (s_x, s_y). The work-item global indices can be calculated given the global work-group indices, number of work-groups and work-item local-indices as $(g_x, g_x) = (w_x \times S_x + s_x, w_y \times S_y + s_y)$. The work-group index for a work-item can be calculated given its global-id, local-id and work-group sizes as $(w_x, w_y) = ((g_x - s_x)/S_x, (g_y - s_y)/S_y)$

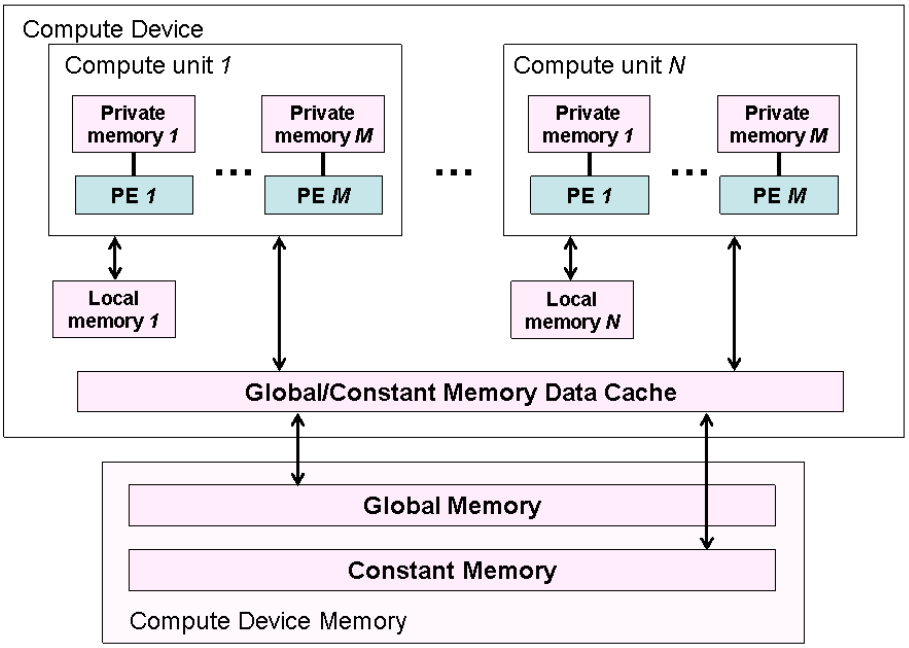


Figure 2.8: An OpenCL device architecture. From [Khro 12].

2.3.3 Memory Model

OpenCL provides four logically distinct memory regions on the device and Figure 2.8 shows how they relate to the OpenCL device abstraction.

- **Global Memory**: Accessible to all work-items in all work-groups. All locations are accessible to all work items. Typically mapped to off-chip RAM on a GPU.
- **Constant Memory**: Read-only global memory that remains constant during kernel execution. Typically cached in a dedicated constant cache on a GPU.
- **Local Memory**: Memory local to a work-group. It can be used to share data between work-items within the same work-group. It is not possible to access the local memory of another work-group. Typically mapped to fast on-chip scratchpad memory on a GPU.
- **Private Memory**: Memory private to a work-item. It is not possible to access data in another work-item's private memory. Typically mapped to registers on a GPU.

OpenCL uses a relaxed memory model and the memory visible to work-items is not guaranteed to be consistent across all work-items.

Synchronisation between work-items within a single work-groups is achieved using the `barrier` built-in function. Local memory is consistent across work-items within a single work-group at a `barrier` executed with a local memory fence. Global memory is consistent across

```
1  kernel void reduce(global float* input, global float* output,
2                     unsigned long N, local float* tmp) {
3    size_t l_id = get_local_id(0);
4    size_t g_id = get_global_id(0);
5
6    // Copy data from global memory to local memory
7    tmp[l_id] = (g_id < N) ? input[g_id] : 0;
8    barrier(CLK_LOCAL_MEM_FENCE);
9
10   // Do a reduction in local memory
11   for (size_t s = 1; s < get_local_size(0); s *= 2) {
12     if ((l_id % (2 * s)) == 0) {
13       tmp[l_id] += tmp[l_id + s];
14     }
15     barrier(CLK_LOCAL_MEM_FENCE);
16   }
17   // Write the result for this work-group to global memory
18   if (l_id == 0) {
19     output[get_group_id(0)] = tmp[0];
20   }
21 }
```

Listing 2.1: Parallel tree-based reduction in OpenCL C

work-items within a single work-group at a `barrier` executed with a global memory fence. There are no guarantees of memory consistency across work-groups.

2.3.4 OpenCL C Kernel Language

OpenCL kernels are written in the OpenCL C programming language. It is based on the C99 specification with some restrictions as well as extensions. Limitations of OpenCL C include the following: a kernel argument cannot be a pointer to a pointer, pointers to functions are not allowed, recursion and dynamic memory allocation are not supported. Extensions to C99 include address space qualifiers, vector and image data types, and built-in functions, including for parallelism and vectorisation. Kernels written in OpenCL C are compiled at runtime using the `clBuildProgram` function that will invoke the vendor compiler for a specific device.

Listing 2.1 shows an example of an OpenCL kernel performing a parallel reduction. `global` (line 1) and `local` (line 2) are examples of address space qualifiers and denote that those pointers point to values in the global and local address space. `get_global_id`, `get_local_id`, `get_group_id`, `get_local_size` and `barrier` are examples of built-in functions. The first three, `get_global_id`, `get_local_id` and `get_group_id` provide access to identifiers within the `NDRange`, `get_local_size` gives information about the size of the `NDRange` while `barrier` provides synchronisation and a memory fence within a work-group.

In line 7 all threads copy an element from global memory to local memory before syn-

```
1 kernel void patterns(global float* input, global float* output, unsigned long N) {
2   size_t g_id_0 = get_global_id(0);
3   size_t g_id_1 = get_global_id(1);
4   float tmp = input[g_id_1 * N + g_id_0]; // Coalesced read
5   output[g_id_0 * N + g_id_1] = tmp; // Non-coalesced write
6 }
```

Listing 2.2: Different memory access patterns in OpenCL C

```
1 kernel void add(global float* input1, global float* input2,
2                 global float* output) {
3   size_t g_id = get_global_id(0);
4   float4 tmp1 = vload4(g_id, input1); float4 tmp2 = vload4(g_id, input2);
5   float4 result = tmp1 + tmp2;
6   vstore4(result, g_id, output);
7 }
```

Listing 2.3: Vectorisation in OpenCL C

chronising in line 8. The `for` loop in line 11 performs the reduction with the number of active threads halving in every iteration. In each iteration every thread that still has work to do reads two elements from local memory, adds them, and writes them back to local memory in line 13. All threads synchronise at the end of every iteration in line 15. Finally, in line 19 the thread with id 0 in the work-group stores the result from local memory back to global memory.

Consecutive threads have consecutive global or local identifies in dimension 0 and achieving coalesced accesses therefore requires them to access consecutive memory locations. Listing 2.2 shows an example. The read from global memory in line 4 is coalesced. The write to global memory in line 5 is not coalesced, as consecutive threads write to non-consecutive locations.

Listing 2.3 shows how vectorisation is performed in OpenCL C. Line 4 loads two `float4` values from global memory using the `vload4` built-in function. A `float4` is a vector of 4 single-precision floating-point values. Similar types exist for 2, 3, 4, 8 and 16 wide vectors of all the basic numeric types. `vload4` and `vstore4` are built-in functions for loading and storing vector values. Line 5 performs a vectorised addition of the two values. Arithmetic, logical, relational, equality and other operators are defined for vector types as well as scalar types. Line 6 stores the result of the addition to global memory using the `vstore4` built-in function.

This section has introduced OpenCL, a low-level programming model for GPUs. It allows to manually express optimisations but is complex and error prone due to its low-level nature. Next, the higher-level GPU programming approach of LIFT is introduced.

2.4 LIFT

This section presents LIFT, a functional data-parallel high-level language based on parallel patterns first presented in [Steu 15b]. It raises the abstraction level and provides the opportunity to optimise programs using rewrite-rules. LIFT specifically targets OpenCL, although many concepts are more widely applicable.

2.4.1 Design Principles

High-level languages based on parallel patterns capture rich information about the algorithmic structure of programs. Take the computation of the dot product as an example:

```
dot(x, y) = reduce(0, +, map(×, zip(x, y)))
```

Here the `zip` pattern captures that arrays `x` and `y` are accessed pairwise. Furthermore, the `map` pattern allows the compiler to perform the multiplication in parallel, as well as the final summation, expressed using the `reduce` pattern.

The most important design goal is to preserve algorithmic information in the compiler for as long as possible. LIFT achieves this by expressing the OpenCL programming model functionally. One of the key advantages of the LIFT approach is that it is possible to decouple the problem of mapping and exploiting parallelism from the code generation process.

2.4.2 Language

LIFT expresses programs as compositions and nesting of functions which operate on arrays. The formal foundation of LIFT is lambda calculus which formalises the reasoning about functions, their composition, nesting and application.

Type System The type system of LIFT is a limited form a dependent type system and supports *scalar types*, *vector types*, *array types* and *tuple types*. A dependent type is a type whose definition depends on a value. Dependent type systems can be used for applications such as static elimination of array bounds checking. [Xi 98] In LIFT, array types depend on integer values that define their length and these types are known as vectors in the dependent typing community. Chapter 4 will show how the length information is used to generate efficient array accesses and improve the performance of the code as well as to detect whether barriers are taken by all threads to check for correctness.

Vector types are written as T_m for an m element vector of a scalar type T. Array types are written as $[T]_n$ for an array of n elements of type T. Array types can also be nested to represent multi-dimensional arrays and carry information about the length of each dimension. For instance, $[[T]_n]_m$ represents a two-dimensional array of m by n elements of type T. Tuple

types are written as $\langle T_1, T_2, \ldots \rangle$ with elements of types T_i. Functions are written as $(T_1, ...) \to U$ for a function taking arguments of types T_i and producing a value of type U.

Values of vector type are written as $\overrightarrow{x_1, x_2, \ldots}$ where x_i are the elements of the vector. Arrays are written as $[x_1, x_2, \ldots]$ where x_i are elements of the array, and tuples are written as $\langle x_1, x_2, \ldots \rangle$.

The following section introduces the predefined patterns used as building blocks to express programs. For each pattern the type as well as a definition of its high level semantics is given. Besides these patterns, LIFT also supports *user functions* written in a subset OpenCL C operating on scalar, vector or tuple values, which implement the application specific computations. Note that array arguments and, therefore, explicit indexing of memory are not supported.

2.4.2.1 General Patterns

Algorithmic Patterns LIFT supports five algorithmic patterns corresponding to implementations of the well known *map* and *reduce* patterns, the *identity* and the *iterate* primitive. These patterns directly affect how the computation is performed. The *reduce* pattern requires the binary function $\oplus$ to be associative and commutative to allow for a parallel implementation. The *reduceSeq* pattern is the sequential version of *reduce* and does not have the restrictions. The *iterate* pattern applies a function f m times by re-injecting the output of each iteration as the input of the next. The length of the output array is given by the function h of the number of iterations m, the input array length n and the change of the array length by a single iteration captured by the function g.

$$\textbf{map} : ((T \to U), [T]_n) \to [U]_n$$
$$\textbf{map}(f, \boxed{x_1 \mid x_2 \mid \cdots \mid x_n}) = \boxed{f(x_1) \mid f(x_2) \mid \cdots \mid f(x_n)}$$

$$\textbf{reduce} : (T, (T, T) \to T, [T]_n) \to [T]_1$$
$$\textbf{reduce}(z, \oplus, \boxed{x_1 \mid x_2 \mid \cdots \mid x_n}) = \boxed{z \oplus x_1 \oplus x_2 \cdots \oplus x_n}$$

$$\textbf{reduceSeq} : (U, (U, T) \to U, [T]_n) \to [U]_1$$
$$\textbf{reduceSeq}(z, \oplus, \boxed{x_1 \mid x_2 \mid \cdots \mid x_n}) = \boxed{(((z \oplus x_1) \oplus x_2) \cdots \oplus x_n)}$$

$$\textbf{id} : T \to T$$
$$\textbf{id}(x) = x$$

$$\textbf{iterate} : (m : int, [T]_k \to [T]_{g(k)}, [T]_n) \to [T]_{h(m,n,g)}$$
$$\textbf{iterate}(m, f, \boxed{x_1 \mid x_2 \mid \cdots \mid x_n}) = \underbrace{f(\cdots(f(}_{m \text{ times}}\boxed{x_1 \mid x_2 \mid \cdots \mid x_n}))\underbrace{)\cdots)}_{m \text{ times}}$$

Data Layout Patterns LIFT defines a set of patterns that do not perform any computation but simply reorganise the data layout. Since they perform no computation then no code is generated for these patterns but they change how the arrays are accessed. The first two patterns, *split* and *join*, add or remove a dimension from the input array.

$$\textbf{split} : (m : int, [T]_n) \rightarrow [[T]_m]_{n/m}$$

$$\textbf{split}(m, \boxed{x_1}\boxed{x_2}\boxed{\cdots}\boxed{\cdots}\boxed{\cdots}\boxed{\cdots}\boxed{\cdots}\boxed{\cdots}\boxed{\cdots}\boxed{x_n}) = \underbrace{\boxed{x_1}\boxed{x_2}\boxed{\cdots}\boxed{\cdots}}_{m}\boxed{\cdots}\boxed{\cdots}\boxed{\cdots}\boxed{x_n}$$

$$\textbf{join} : [[T]_m]_{n/m} \rightarrow [T]_n$$

$$\textbf{join}(\boxed{x_1}\boxed{x_2}\boxed{\cdots}\boxed{\cdots}\boxed{\cdots}\boxed{\cdots}\boxed{\cdots}\boxed{\cdots}\boxed{\cdots}\boxed{\cdots}\boxed{\cdots}\boxed{x_n}) = \boxed{x_1}\boxed{x_2}\boxed{\cdots}\boxed{\cdots}\boxed{\cdots}\boxed{\cdots}\boxed{\cdots}\boxed{\cdots}\boxed{\cdots}\boxed{x_n}$$

The *gather* and *scatter* patterns apply a permutation function f which remaps indices when reading from or writing to arrays respectively. Combined with *split* and *join*, for instance, these primitives can express matrix transposition: *transpose = split(nrows) ∘ gather(i → (i* mod *ncols) × nrows + i / ncols) ∘ join*. The ∘ symbol is used to denote sequential function composition, i. e. $(f \circ g)(x) = f(g(x))$.

$$\textbf{gather} : ((int \rightarrow int), [T]_n) \rightarrow [T]_n$$

$$\textbf{gather}(f, \boxed{x_{f(1)}}\boxed{x_{f(2)}}\boxed{\cdots}\boxed{x_{f(n)}}) = \boxed{x_1}\boxed{x_2}\boxed{\cdots}\boxed{x_n}$$

$$\textbf{scatter} : ((int \rightarrow int), [T]_n) \rightarrow [T]_n$$

$$\textbf{scatter}(f, \boxed{x_1}\boxed{x_2}\boxed{\cdots}\boxed{x_n}) = \boxed{x_{f(1)}}\boxed{x_{f(2)}}\boxed{\cdots}\boxed{x_{f(n)}}$$

The *zip* pattern is used to combine two arrays of elements into a single array of pairs while the *get* primitive projects a component of a tuple.

$$\textbf{zip} : ([T]_n, [U]_n) \rightarrow [\langle T, U \rangle]_n$$

$$\textbf{zip}(\boxed{x_1}\boxed{x_2}\boxed{\cdots}\boxed{x_n}, \boxed{y_1}\boxed{y_2}\boxed{\cdots}\boxed{y_n}) = \boxed{\langle x_1, y_1 \rangle}\boxed{\langle x_2, y_2 \rangle}\boxed{\cdots}\boxed{\langle x_n, y_n \rangle}$$

$$\textbf{get} : (i : int, \langle T_1, T_2, \ldots, T_n \rangle) \rightarrow T_i$$

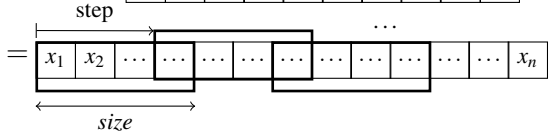
$$\textbf{get}(i, \langle x_1, x_2, \ldots, x_n \rangle) = x_i$$

The *slide* pattern applies a sliding window to the input data and is used to express stencil computations. For instance, *mapSeq(reduceSeq(0,+)) ∘ slide(3,1,input)* expresses a simple

3-point stencil. Multi-dimensional stencils are also expressible by composing several *slide* functions interleaved with transpositions.

$$\textbf{slide} : (size : int, step : int, [T]_n) \rightarrow [[T]_{size}]_{\frac{n-size+step}{step}}$$

$$\textbf{slide}(size, step, [x_1 \mid x_2 \mid \cdots \mid \cdots \mid \cdots \mid \cdots \mid \cdots \mid \cdots \mid \cdots \mid x_n])$$



Finally, the *pad* pattern adds $left$ and *right* elements at the beginning and end of the input array, respectively. There are two variants of *pad*. In the first, the extra elements are computed by *h*, and in the second, the extra elements are added by indexing into the input array using the value returned by *h*. The *pad* patten is used to express boundary condition in stencil codes. For example, the `clamp(i, n) = (i < 0) ? 0 : ((i >= n) ? n - 1 : i)` function is used to extend the array by repeating the elements at the beginning and end.

$$\textbf{pad} : (left : int, right : int, h : (i : int, length : int) \rightarrow int, [T]_n)$$
$$\rightarrow [T]_{n+left+right}$$
$$\textbf{pad} : (left, right, h, in : [x_1 \mid x_2 \mid \cdots \mid x_n])$$
$$= [x_{h(0-left,n)} \mid \cdots \mid x_{h(0)} \mid x_1 \mid x_2 \mid \cdots \mid x_n \mid x_{h(n+1,n)} \mid \cdots \mid x_{h(n+right,n)}]$$

2.4.2.2 OpenCL-specific Patterns

Parallel Patterns OpenCL provides a hierarchical organisation of parallelism where threads are grouped into *work groups* of *local threads* or a flat organisation with *global threads*. This hierarchy is represented with three patterns, *mapGlb*, *mapWrg* and *mapLcl*. A *mapLcl* must be nested inside of *mapWrg* to respect the OpenCL thread hierarchy.

$$\textbf{mapGlb}(i) : ((T \to U), [T]_n) \to [U]_n$$

$$\textbf{mapGlb}(i)(f, \boxed{x_1}\boxed{x_2}\boxed{\cdots}\boxed{x_n}) = \boxed{f(x_1)}\boxed{f(x_2)}\boxed{\cdots}\boxed{f(x_n)}$$

where i is $0, 1$ or 2

$$\textbf{mapWrg}(i) : ((T \to U), [T]_n) \to [U]_n$$

$$\textbf{mapWrg}(i)(f, \boxed{x_1}\boxed{x_2}\boxed{\cdots}\boxed{x_n}) = \boxed{f(x_1)}\boxed{f(x_2)}\boxed{\cdots}\boxed{f(x_n)}$$

where i is $0, 1$ or 2

$$\textbf{mapLcl}(i) : ((T \to U), [T]_n) \to [U]_n$$

$$\textbf{mapLcl}(i)(f, \boxed{x_1}\boxed{x_2}\boxed{\cdots}\boxed{x_n}) = \boxed{f(x_1)}\boxed{f(x_2)}\boxed{\cdots}\boxed{f(x_n)}$$

where i is $0, 1$ or 2

$$\textbf{mapSeq}(i) : ((T \to U), [T]_n) \to [U]_n$$

$$\textbf{mapSeq}(i)(f, \boxed{x_1}\boxed{x_2}\boxed{\cdots}\boxed{x_n}) = \boxed{f(x_1)}\boxed{f(x_2)}\boxed{\cdots}\boxed{f(x_n)}$$

OpenCL supports up to three thread dimensions, represented by the $0, 1, 2$. The semantic and type of these patterns is identical to *map*, except that f is applied in parallel. Finally, *mapSeq* applies f sequentially.

Vectorisation Patterns LIFT supports two primitives that transform data between scalar and vector types, and one pattern which applies vectorises a function.

$$\textbf{asVector} : (m : int, [T]_n) \to [T_m]_{n/m}$$

$$\textbf{asVector}(m, \boxed{x_1}\boxed{x_2}\boxed{\cdots}\boxed{\cdots}\boxed{\cdots}\boxed{\cdots}\boxed{\cdots}\boxed{\cdots}\boxed{\cdots}\boxed{x_n})$$

$$= \boxed{\overrightarrow{x_1, x_2, \ldots, x_m}}\boxed{\overrightarrow{\ldots, \ldots, \ldots, \ldots}}\boxed{\cdots}\boxed{\overrightarrow{\ldots, \ldots, \ldots, x_n}}$$

$$\textbf{asScalar} : [T_m]_{n/m} \to [T]_n$$

$$\textbf{asScalar}(\boxed{\overrightarrow{x_1, x_2, \ldots, x_m}}\boxed{\overrightarrow{\ldots, \ldots, \ldots, \ldots}}\boxed{\cdots}\boxed{\overrightarrow{\ldots, \ldots, \ldots, x_n}})$$

$$= \boxed{x_1}\boxed{x_2}\boxed{\cdots}\boxed{\cdots}\boxed{\cdots}\boxed{\cdots}\boxed{\cdots}\boxed{\cdots}\boxed{\cdots}\boxed{x_n}$$

$$\textbf{vectorize} : (n : int, f : (T_1, \ldots) \to U) \to ((T_n, \ldots) \to U_n)$$

For the *vectorise* pattern, the function f is transformed into a vectorised form during code generation. This transformation is straightforward for functions based on simple arithmetic

operations since OpenCL already defines vectorised forms for these operation. In the other more complicated cases, the code generator simply applies f to each scalar in the vector. Future work intends to incorporate existing research on vectorising more complex functions [Karr 11].

Address Space Patterns OpenCL distinguishes between *global*, *local* and *private* address spaces. LIFT offers three corresponding primitives which wrap a function and influence the address space used to store the output:

$$\textbf{toGlobal} : ((T,\ldots) \to U) \to ((T,\ldots) \to U)$$
$$\textbf{toLocal} : ((T,\ldots) \to U) \to ((T,\ldots) \to U)$$
$$\textbf{toPrivate} : ((T,\ldots) \to U) \to ((T,\ldots) \to U)$$

For example, a sequential copy of an array x into local memory is expressed as follows: $toLocal(mapSeq(id))(x)$. This design decouples the decision of *where* to store data (i. e. the address space) from the decision of *how* the data is produced (i. e. sequentially or in parallel). Like the generic data layout patterns, the address space patterns do not perform any computation and no code is generated for them, but they affect the array indices that are generated.

2.4.2.3 Example: Dot Product in LIFT

Listing 2.4 shows one possible low-level implementation of dot product expressed in LIFT. This program encodes explicitly how the computation should be performed and can be translated into OpenCL. The program is represented using a functional style, therefore, the program is read from right to left instead of the familiar left to right common in imperative programming. Furthermore, to simplify the notation the $\circ$ symbol is used to denote sequential function composition, i. e. $(f \circ g)(x) = f(g(x))$.

In the program of Listing 2.4 the input arrays `x` and `y` are combined using the zip pattern in line 13. The zipped array is then split into chunks of size 128 (line 13). A work group processes a single chunk using the *mapWrg* pattern (line 2) before combining the computed chunks using the join pattern (line 2). Inside of a work group three steps are performed to process a chunk of 128 elements: 1) the chunk is split further into pairs of two zipped elements in line 12, which are multiplied and added up before copying the computed result into local memory (lines 10 and 11); 2) two elements at a time are iteratively reduced in local memory (lines 6 to 9); 3) the computed result is copied back into global memory (line 4).

Note that the code shown here corresponds to a single OpenCL kernel which only computes a partial dot product. A second kernel is required to sum up all intermediate results.

```
1  partialDotProduct(x: [float]_N, y: [float]_N) =
2    join ∘ mapWrg(0)(
3      join ∘
4      toGlobal(mapLcl(0)(mapSeq(id))) ∘
5      split(1) ∘
6      iterate(6)( join ∘
7                  mapLcl(0)( toLocal(mapSeq(id)) ∘
8                             reduceSeq(0, add) ) ∘
9                  split(2) ) ∘
10     join ∘ mapLcl(0)( toLocal(mapSeq(id)) ∘
11                       reduceSeq(0, multAndSumUp) ) ∘
12     split(2) ) ∘
13   split(128, zip(x, y))
```

Listing 2.4: LIFT implementation of a partial dot product

2.4.3 LIFT Rewrite Rules

Rewrite rules transform the program into semantically equivalent forms. Rewriting for compiler optimisations is an established approach in the functional community and used, for example, in the Glasgow Haskell compiler [Peyt 01]. The high-level dot product program comprised of *map*, *reduce* and *zip* is rewritten into the low-level program in Listing 2.4. The left-hand side of the rule shows the pattern the rule can be applied to and the right-hand side shows the result after the transformation. This section will discuss the individual rewrite rules which are divided into algorithmic and OpenCL specific ones. These rules are proven correct in [Steu 15a] but in prior work on LIFT it remains unclear how they are applied to optimise practical applications. Chapter 5 will explore techniques on how these rewrite rules are applied to optimise matrix multiplication for 3 GPU architectures from different manufacturers.

2.4.3.1 Algorithmic Rules

Figure 2.9 shows algorithmic rules which represent different algorithmic choices that can be made during optimisation.

Iterate Decomposition Rule The rule in Figure 2.9a expresses the fact that the number of iterations of an *iterate* can be decomposed into several *iterate* patterns.

Split-Join Rule The split-join rule in Figure 2.9b turns a *map* into two nested *map*s. This allows, for example, mapping different parts of the computation to different levels of the OpenCL thread hierarchy.

```
iterate(i+j, f) ⇒ iterate(i, f) ∘ iterate(j, f)
```

(a) Iterate decomposition rule

```
map(f) ⇒ join ∘ map(map(f)) ∘ split(m)
```

(b) Split-join rule

```
reduce(z, f) ⇒ reduce(z, f) ∘ partialReduce(z, f)
partialReduce(z, f) ⇒ reduce(z, f)
partialReduce(z, f) ⇒ partialReduce(z, f) ∘ gather(p)
partialReduce(z, f) ⇒ iterate(i, partialReduce(z, f))
partialReduce(z, f) ⇒ join ∘ map(partialReduce(z, f)) ∘ split(m)
```

(c) Reduce rules

```
map(f) ∘ map(g) ⇒ map(f∘g)
reduceSeq(z, f) ∘ mapSeq(g) ⇒ reduceSeq(z, λ (acc, x) . f(acc, g(x)))
```

(d) Fusion rules

```
join ∘ split(m) ⇒ id
asScalar ∘ asVector(n) ⇒ id
```

(e) Cancellation rules

Figure 2.9: Algorithmic Rules

Reduce Rules The rules in Figure 2.9c show how a *reduce* can be performed in several steps, possibly partially iteratively or in parallel. The *partialReduce* primitive performs a partial reduce, i.e. an array of n elements is reduced to an array of m elements where $1 \leq m \leq n$.

Fusion Rules The fusion rules are shown in Figure 2.9d. The first rule fuses two consecutive *map* patterns into a single one. The second rule fuses a *mapSeq* followed by a *reduceSeq* by applying the function g on the fly while performing the reduction. The rules only applies to the sequential version of reduction as this version of reduction does not require associativity.

Cancellation Rules The rules in Figure 2.9e express the fact that a *split* followed by *join* or *asVector* followed by *asScalar* is equivalent to the identity. These rules are the only algorithmic rules that do not change how the computation is performed or how the code is generated.

2.4.3.2 OpenCL-specific Rules

Figure 2.10 shows OpenCL specific rules which map generic primitives to OpenCL specific ones.

Map Rules The rules in Figure 2.10a turn the generic *map* primitive into a sequential version or parallel versions targeting different levels of the OpenCL thread hierarchy.

Reduce Rule There is only one low-level rule for *reduce*. It turns the generic *reduce* into a sequential version and is shown in Figure 2.10a.

Vectorisation Rule Figure 2.10c shows the rule to vectorise the data worked on and the function applied by a *map* primitive.

Memory Rules The rule in Figure 2.10d allow programs to use different levels of the OpenCL memory hierarchy. They make the function f store its result into the specified address space. These rules do not change the computation but only affect the address space used and the index used to access the memory.

2.4.4 LIFT Implementation

This section introduces the implementation of LIFT. One of the key features of LIFT is that it preserves a functional representation of the program all the way through.

```
map(f) ⇒ mapGlb(i)(f) | mapWrg(i)(f) | mapLcl(i)(f) | mapSeq(f)
```

(a) Map rules

```
reduce(z, f) ⇒ reduceSeq(z, f)
```

(b) Reduce rule

```
map(f) ⇒ asScalar ∘ map(vectorize(f)) ∘ asVector(n)
```

(c) Vectorisation rule

```
f ⇒ toGlobal(f) | toLocal(f) | toPrivate(f)
```

(d) Memory rules

Figure 2.10: OpenCL-specific rules

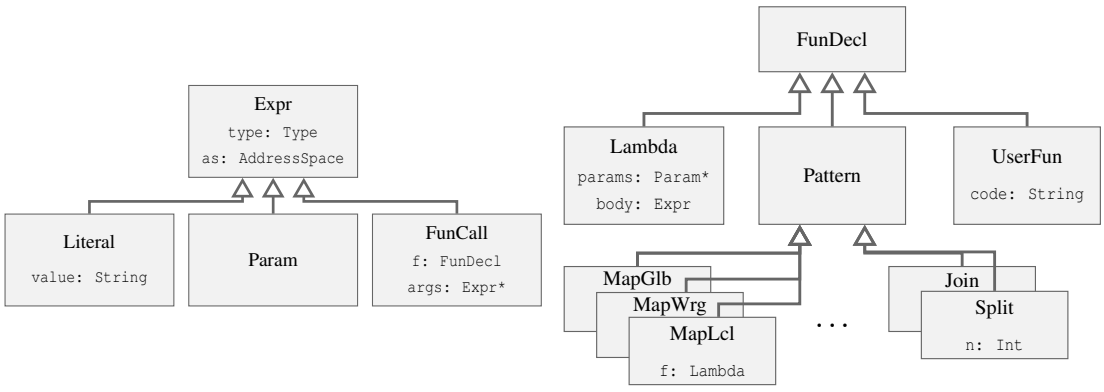


Figure 2.11: Class diagram of the LIFT implementation. From [Steu 17].

2.4.4.1 Organisation of classes

LIFT is implemented as an embedded language in Scala. Programs are represented as graphs where nodes are implemented as *objects*. The use of a graph-based representation avoids the problem of performing extensive renaming when transforming functional programs [Leis 15]. The class diagram of LIFT in Figure 2.11 shows two main classes: *expressions* (`Expr`) and *function declarations* (`FunDecl`).

Expressions represent values and have a type associated with them. Expressions are either literals, parameters or function calls. Literals represent compile time known constants such as `3.4f`, or arrays or tuples. Parameters are used inside functions and their values are the arguments of a function call. Finally, function calls connect a function to be called (a `FunDecl`) with its arguments (`Exprs`).

Function Declarations correspond to either a lambda, a predefined pattern or a user function. Lambdas are anonymous function declarations with parameters and a body which is evaluated when the lambda is called. A pattern is a built-in function such as *map* or *reduce*. The `UserFun` corresponds to user-defined functions expressed in a subset of the OpenCL C language operating on non-array data types.

2.4.4.2 Example

Figure 2.12 shows the LIFT IR of the dot-product program from Listing 2.4. The plain arrows show how objects reference each other. The top left node labelled *Lambda2* is the root node of the graph taking two parameters and its body implements dot-product as a sequence of function calls.

The dashed arrows visualises the way the data flows through the IR. The inputs x and y are first used as an input to the *zip* function which is then fed into a call to *split(128)*. Then the results of the split is fed into the *mapWrg* function. The function which is applied to each chunk of 128 elements is represented as a lambda which processes the input in three steps. First, the data is moved from global to local memory by performing a partial reduction (labelled *glbToLcl*). Then, the data flows to a function which iteratively reduces the elements in local memory (*iteration*). Finally, the data is moved back from local memory to global memory (*lclToGlb*), exits the *mapWrg* and the last *join* is applied to return the final result.

2.4.4.3 Lambda and Data Flow

Lambdas appear in several places in the graph and make the data flow explicit. For example, focusing on the *iteration* part of the graph, a `Lambda` node is used below this `Iterate` node. To understand what the lambda does, look back at Listing 2.4, lines 6– 9 copied here:

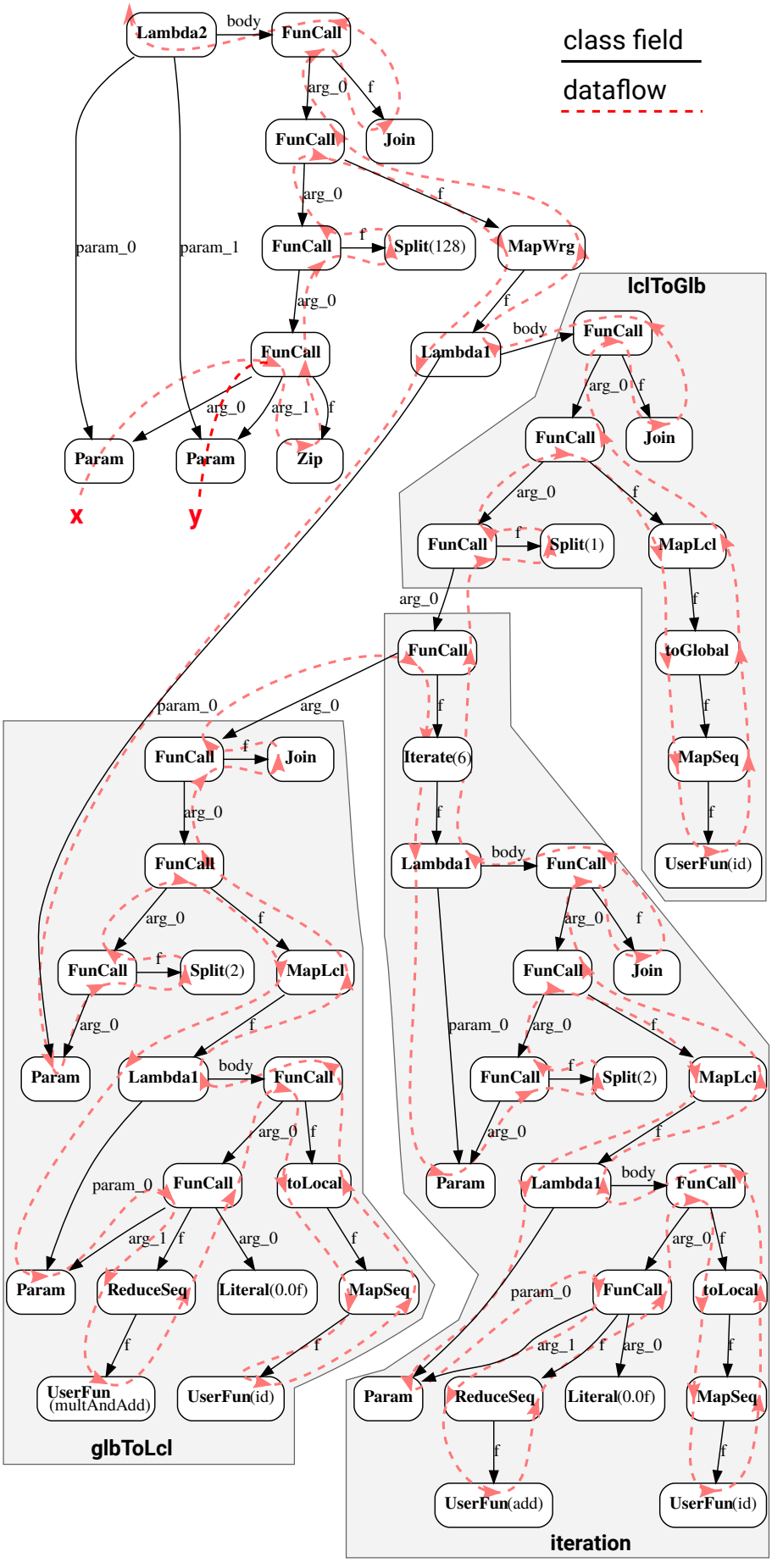


Figure 2.12: LIFT IR for dot-product example. From [Steu 17].

```
iterate(6)( join ∘ mapLcl(0)( ... ) ∘ split(2) )
```

This notation is only syntactic sugar for:

```
iterate(6)( λ p . join(mapLcl(0)(..., split(2, p))) )
```

The lambda (λ) makes the data flow explicit, i.e. the *iterate* pattern passing its input via the parameter *p* first to *split* which then passes it to *mapLcl* and *join*.

2.4.5 Open Problems and Challenges of LIFT Code Generation

This section discusses the design of the LIFT functional data-parallel language. It is similar in style to prior work [Brow 16, Maie 16] and is OpenCL specific. Via rewriting LIFT expresses precisely how programs are mapped to the OpenCL programming model, as seen for the dot product example.

However, it is not described in [Steu 15b] how OpenCL code is generated from LIFT programs once mapped into a low-level OpenCL specific form. While code generation is straightforward for the simple programs presented in [Steu 15b] it is unclear how this would scale to more complex applications. Chapter 4 will show how a naïve approach to code generation and especially array access generation leads to code that achieves only a fraction of the available performance. Chapter 4 then introduces a novel technique for generating high-performance code on par with handwritten and tuned code for real-world applications such as matrix-matrix multiplication.

The set of rewrite rules presented in [Steu 15b] is incomplete for expressing many optimisations relevant in practice such as tiling. Additionally, the number of different expressions evaluated for a single program in the first LIFT paper is only around 100 and it is unclear how a search strategy for finding good implementations will scale if hundreds of rule applications are needed to apply optimisations and many thousands of expression variations must be explored. Chapter 5 and Chapter 6 address these issues by introducing additional rules, grouping rules together as macro rules to express optimisations and presenting a search strategy for exploring tens of thousands of different versions of a program that is accelerated by using a performance prediction model.

2.5 Design-Space Exploration

As different GPUs are designed differently and have different performance characteristics, appropriate implementation choices need to be made when optimising a program for a particular device. These choices lead to a design space of alternative implementations that can be systematically explored to find implementations that perform well on a given device.

```
1  kernel void multiply(global float* input, global float* output, unsigned long N) {
2    for (size_t gid = get_global_id(0); gid < N / EPT; gid += get_global_size(0) {
3      for (int i = 0; i < EPT / VW; i++) {
4        unsigned index = gid * EPT / VW + i;
5  #if VW == 1
6        output[index] = input[index] * 2;
7  #elif VW == 2
8        vstore2(vload2(index, input) * 2, index, output);
9        // ...
10 #elif VW == 16
11       vstore16(vload16(index, input) * 2, index, output);
12 #endif
13     }
14   }
15 }
```

Listing 2.5: An OpenCL kernel multiplying all elements of the input by 2. The amount of elements processed by each thread is defined by `EPT` and the vector width is defined by `VW`. Values for both are meant to be chosen by an auto-tuner.

2.5.1 Auto-Tuning

Auto-tuning is a technique for systematically choosing values for parameters in a program to improve its performance with respect to some metric, such as time or throughput. These implementation parameters can, for example, be the number of elements processed by each thread, the local size of an OpenCL program or the width of vector data types.

As an example, Listing 2.5 shows a simple OpenCL kernel that multiplies every element of an array of `N` elements by two. The number of elements processed by each thread in a single iteration is defined by the parameter `EPT`. The vector width used for memory and arithmetic operations is defined by the parameter `VW`. The value of both, `EPT` and `VW`, can be chosen or tuned for a particular device. The parameter `EPT` can take all values that divide the input size `N`, so that all elements would be processed. `VW` can take the value of 1, which means no vectorisation, as well as the values of 2, 4, 8, 16 that are legal vector widths in OpenCL. Additionally, constrains can be placed between values of different parameters. In the case of the example in Listing 2.5, `VW` has to divide `EPT`.

It is the job of the auto-tuner to choose valid parameter combinations, as determined by the constraints for `EPT` and `VW`, while improving performance as determined by a cost function. The cost function evaluates the kernel with respect to the metric the kernel is being tuned for. It can be as simple as compiling and running the kernel with the chosen parameters and measuring the execution time.

Figure 2.13 shows an overview of how an auto-tuner operates. It takes as input the kernel being tuned, its parameters, legal values for the parameters and any constraints between the

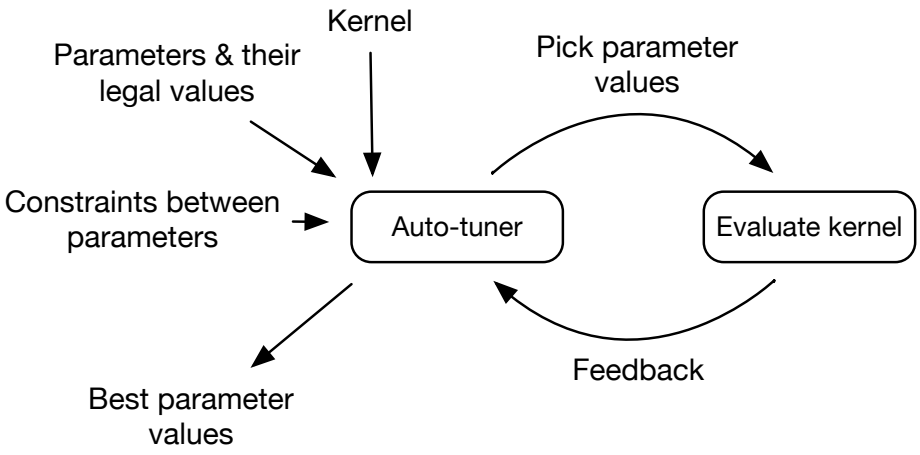


Figure 2.13: Auto-tuning loop

parameters. For example, the kernel from Listing 2.5, the parameters `EPT` and `VW`, their legal values, and the constraints `N` mod `EPT` $= 0$ and `EPT` mod `VW` $= 0$

First, the auto-tuner will pick a combination values for all parameters that is legal with respect to the constraints using some strategy. There exist many different search strategies. For example, the exhaustive search strategy evaluates all different combination of parameter values. A random search strategy picks random points in the search space to evaluate.

The cost function will be evaluated for the kernel being tuned with the chosen parameters. The auto-tuner will then pick a new set of parameters to evaluate. Using a more sophisticated search strategy, the auto-tuner can make the decision based on the feedback received from evaluating previous sets of parameters and how the value of the function changed based on changing the values of parameters.

For example, simulated annealing is a probabilistic search technique inspired by annealing in metallurgy. In each iteration of the search it picks a neighbouring configuration to the current one to evaluate. If the new configuration has better performance it will move to that point. However, to avoid getting stuck in local minima it also has a probability to move to a worse configuration that is based on the annealing *temperature*. The *temperature* is progressively reduced to zero as the search continues and the probability to accept a worse solution decreases with it.

The process of picking parameter values and evaluating the kernel using them will continue until a certain level of performance is reached, a set amount of time has elapsed, a certain number of combinations have been tried or all combinations have been evaluated. As output, the auto-tuner will produce the set of parameter values that gave the best performance.

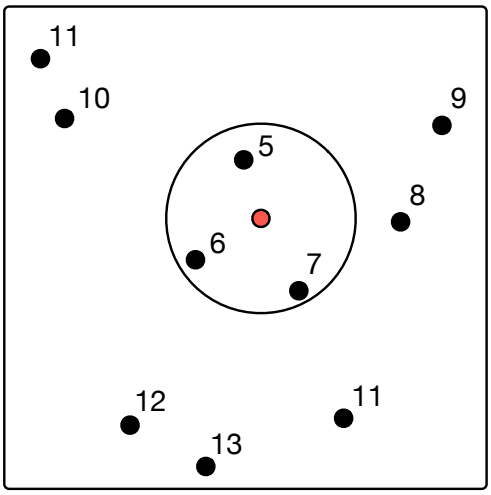


Figure 2.14: K-nearest neighbours algorithm with $k = 3$. The output value for the new point (red) is predicted as the average of the closest existing (black) points, so $(5+6+7)/3 = 6$.

2.5.2 Performance Modelling and Prediction

Determining the value of the cost function by running the program can be time consuming.

To address this problem, the cost function can be replaced by a different function that is quicker to evaluate and produces a cost prediction. One common approach is to produce the prediction based on some program characteristics. These characteristics are known as features and describe different aspects of the program. The prediction function should relate the feature values to the value of the cost function.

One approach for coming up with a prediction function is analytical and consists of manually devising a mathematical formula that will calculate the predicted value of the cost function based on the input features. For example, the input could be the number of different instructions and the function could predict the total number of cycles required based on how many cycles each instruction takes to execute.

Another approach is to use statistical methods to automatically learn the function. This is the approach taken in Chapter 6. It requires collecting data of different points in the space, their feature values as well as the values of their cost functions. There exist a number of different techniques for learning functions such as decision trees, linear regression, k-nearest neighbours, support-vector machines, neural networks and others.

The data is separated into a training set and a test set. The training set is used to learn the function. The test set is used to evaluate the learned function and is separated from the training set to check for over-fitting to the training data.

As an example, Figure 2.14 shows the k-nearest neighbours algorithm. K-nearest neighbours works on the assumption that points close by in the feature space have similar values for the metric we are interested in predicting. The algorithm will find k closest points in the

training set to the new point whose metric we are interested in predicting. In the example in Figure 2.14, k is set as 3 and the training set consists of the points in black. The new point for which we are interested in providing an prediction for is red and its 3 closest neighbours are highlighted in the circle. Once the closest neighbours have been found, the output value for the new point is predicted as the average of the output of the neighbours. So the prediction for the red point in the example would be $(5+6+7)/3 = 6$.

When k-nearest neighbours is applied to estimating the performance of programs, the algorithm assumes that programs close to each other in the feature space have similar performance. Therefore, a good selection of features that accurately characterises the performance of programs is crucial for this technique to work well.

2.6 Summary

This chapter discussed the technical background required to understand this thesis. First, it described the architectures of the different GPUs used in this thesis. Then it introduced the low-level OpenCL programming model for programming them, as well as the high-level LIFT language that will be used for optimising and generating OpenCL code. Finally, it presented an overview of different techniques for design-space exploration. The next chapter will give an overview of work related to this thesis.

Chapter 3

Related Work

This chapter presents work related to this thesis. Section 3.1 describes high-level approaches designed to ease GPU programming. Section 3.2 discusses compilers and compiler based optimisation techniques for GPUs. Section 3.3 presents related work on exposing optimisation choices and making decisions between them. Section 3.4 discusses performance modelling and prediction for GPUs. Section 3.5 summarises the chapter.

3.1 High-Level Approaches for GPU Programming

Most GPU programmers use low-level languages such as OpenCL and CUDA that are difficult and error prone to use, as they require paying attention to device specific details for correctness and performance. There exists rich literature on high-level approaches for GPU programming that aim to simplify the process of programming GPUs. This section gives an overview of different proposed library and language based solutions.

3.1.1 Libraries for High-Level GPU Programming

Many high-level approaches for GPU programming are inspired by *parallel patterns* [McCo 12] or *algorithmic skeletons*, a concept developed in the late 80's [Cole 89]. Algorithmic skeletons abstract over implementations for common parallel programming patterns and allow hiding the complexity of parallelism. A common way to implement algorithmic skeletons is to expose them as interfaces of libraries.

SkePU [Enmy 10], SkelCL [Steu 11], Muesli [Erns 12] and FastFlow [Aldi 11, Aldi 12] are all high-level skeleton based C++ libraries that target GPUs. They provide algorithmic skeletons like *map*, *reduce* and *scan* that are combined to create more complex applications. SkelCL targets OpenCL, Muesli and FastFlow target CUDA while SkePU supports both. SkelCL, Muesli and SkePU additionally support distributing execution across multiple GPUs.

Thrust [Bell 11] and Bolt [AMDI 14] are C++ libraries for parallel programming written

by NVIDIA and AMD, respectively. They intend to simplify GPU programming by providing parallel algorithmic skeletons with an interface similar to that of the C++ Standard Template Library (STL) but with CUDA or OpenCL implementations of the algorithms. Programmers familiar with the STL are able to easily take advantage of the GPU acceleration provided by both. As of C++17 [ISO 17], parallel versions of STL algorithms have been added to the official C++ standard and basic GPU implementations of them are available.

Library approaches are easy to use and do not require learning a new programming language. They are also straightforward to integrate into an existing software project. On the other hand, libraries limit users by tying them to fixed interfaces that are not suited for all uses.

3.1.2 Languages for High-Level GPU Programming

To provide more flexibility for GPU acceleration than the narrow interface offered by libraries as well as to provide custom syntax not tied to the library implementation language, a number of high-level languages for GPU programming have been designed and implemented. Languages often have a steeper learning curve and are harder to integrate into existing solutions but provide more acceleration and optimisation opportunities than fixed library interfaces.

StreamIt [Thie 02] is a data flow based programming language designed for programming streaming applications that has been extended to be able to target GPUs by compiling to CUDA [Udup 09]. A StreamIt program consists of *filters*, and data channels connecting filters for communication forming a graph of computation and their dependencies. Different graphs can be used to represent task and data-parallelism. The hierarchical primitives *pipeline*, *splitjoin* and *feedbackloop* are provided to compose filters into larger stream graphs.

Copperhead [Cata 11] is a data-parallel language and compiler embedded in Python and generating CUDA code. Nested parallelism is supported by mapping nested patterns to the GPU thread hierarchy.

Single Assignment C (SaC) [Grel 06, Guo 11] is a MATLAB-like high-level functional array programming language. The compiler automatically identifies `with`-loops (loops guaranteed to be free of dependencies) that are eligible to be executed on GPUs and compiles them to CUDA code.

The Lime language and compiler [Duba 12] is an extension of Java for targeting heterogeneous systems by providing high level abstractions for parallel tasks and communication. It adds support for immutable data types, as well as automatic optimisations for GPUs for improving locality, reducing bank conflicts and vectorisation. It uses the language's type system to make the analysis required for the optimisations simpler.

Halide [Raga 13] is a domain specific language embedded in C++ and an optimising compiler targeted at image processing applications. Optimised code is generated for CPUs as well as GPUs. Halide separates the functional description of the problem from the description of

the implementation, which is called a *schedule*. This allows re-targeting of Halide programs to different platforms by specifying different schedules. A function in Halide maps integer coordinates to scalar results or pixel values, and consist of arithmetic and logical operators, loads from other images and if-then-else expressions. Because the calculation of a function at each coordinate is independent Halide programs are inherently data-parallel. Image processing pipelines are constructed by chaining Halide functions.

Since Halide is designed for image processing applications this can make it akward to express other types of computation such as linear algebra whereas LIFT is more similar to general purpose functional programming languages. Combinators can be used in the *schedule* to determine which parts of the image should be produced sequentially or in parallel. Importantly, Halide allows defining how operations are executed in deep image processing pipelines whereas the optimisations developed in this thesis focus on what would be a single stage.

Accelerate [McDo 13] is a domain specific language embedded in Haskell for GPU programming. Various optimisations are applied internally, for example, fusion of computations for avoiding intermediate results. The implementation relies on templates of manually written CUDA kernels. Sharing recovery and array fusion optimisations are developed for compiling Haskell programs for GPUs and eliminating unnecessary intermediate data-structures.

HiDP [Zhan 13] is a data-parallel language that provides hierarchical constructs and maps them onto the hierarchical execution model for GPUs when generating CUDA code.

NOVA [Coll 14] is a functional programming language and compiler for targeting CPUs and GPUs developed by NVIDIA. It aims to provide performance portability as well as require no in-depth knowledge of the underlying hardware from the programmer. The compiler applies optimisations such as aggressive inlining and fusion. NOVA supports nested parallelism by flattening and unflattening vectors. LIFT supports these optimisations as well as more complex transformations as described in Chapter 5.

Futhark [Henr 14, Henr 17] is a purely functional data-parallel array language targeting GPUs by generating OpenCL or CUDA code. It provides algorithmic skeletons as primitives such as *map*, *scan*, *reduce* and *filter* and supports nested data-parallelism by flattening while still allowing further locality optimisations. By using a type system extension based on uniqueness types it supports in-place array updates to avoid creating unnecessary copies while still preserving the purity of the language.

SYCL [Khro 19c] is an open standard that enables single source C++ programming for heterogeneous devices using completely standard C++. It builds on the concepts of OpenCL but provides additional simplifying abstractions as well as type safety between host and device code. While OpenCL GPU kernel code is typically provided as a string and compiled at runtime, a SYCL compiler statically extracts and compiles the kernel portions of the C++ code to an intermediate representation such as SPIR or SPIR-V. SYCL is considerably higher level

than OpenCL itself but lower level than the other approaches discussed in this section.

While languages provide opportunities to express programs that do not fit narrow library interfaces to be offloaded to GPUs they also come with additional demands. In particular, a language targeting GPUs needs a sophisticated compiler to compile and optimise it.

3.2 Compilers for GPU Programming

Since GPUs behave differently than CPUs, compilers for them need to use techniques suited for generating GPU code. By raising the layer of abstraction, high-level languages for programming GPUs help users to take advantage of GPU acceleration. But the higher layer of abstraction also places a greater burden on the compiler to efficiently handle the complexity that is being hidden from the user. This section first discusses GPU compilers for general purpose languages, such as CUDA or OpenMP. It then discusses polyhedral compilation techniques, before looking at related work in compiling tensor algebra applications. Finally, this section will shift focus and discuss frameworks for building parallel compilers and different compiler intermediate representations.

3.2.1 General Purpose GPU Compilation

There are a variety of different general purpose parallel programming models that are suitable for GPU code generation. Compilers have been designed for these models in both, industry and academia.

OpenMP-to-GPGPU [Lee 09] was an early compiler for translating OpenMP applications into CUDA applications to improve programmability as well as allow existing OpenMP applications to take advantage of GPU acceleration. It operates in two phases. The first phase transforms the OpenMP program into a form more suitable for GPUs. The second phase translates the OpenMP region into a CUDA kernel function and applies further CUDA specific optimisations.

Other work has used a similar approach to compile OpenMP programs to OpenCL [Grew 13]. This approach was combined with a runtime system to choose whether it is more beneficial to run the generated and optimised kernel on the GPU or the original OpenMP code on the CPU.

Bones [Nugt 14a] is a pattern based GPU compiler that automatically detects algorithm species in sequential C code and maps them to parallel algorithmic skeletons. An algorithm species classifies algorithms based on memory access patterns in loop nest based on static analysis using the polyhedral model or array reference characterisations. The pattern implementations are pre-written and not performance portable.

LambdaJIT [Lutz 14] is a C++11 JIT-compiler to parallelise and optimise lambda functions used with STL algorithms. Additionally, it can re-target lambdas to offload computation to

GPUs. Because of additional information available at runtime, it can also perform partial specialisation and fuse function compositions. This type of JIT-compilation based on extra information available at runtime could complement the work presented in the thesis which focuses on compile time optimisation.

`gpucc` [Wu 16] is an open-source CUDA compiler based on LLVM and Clang. It develops or improves several general and CUDA specific optimisations to reach performance on par with or faster than NVIDIA's proprietary `nvcc` compiler. At the time of its release it also supported more recent C++ features than `nvcc`.

PACXX [Haid 16] provides a SYCL-like interface for GPU programming using single source C++. It also provides mechanisms for multi-stage programming to embed runtime values into kernels for specialising them using JIT-compilation in a type safe manner. PACXX is based on LLVM and Clang and supports generating Nvidia's pseudo-assembly language Parallel Thread Execution (PTX) for targeting NVIDIA GPUs and SPIR for targeting GPUs with an OpenCL implementation.

Using techniques specific to particular domains can provide compilers with additional information that is exploitable for optimisation.

3.2.2 Polyhedral GPU Compilation

Polyhedral compilers [Xue 94, Boul 98, Bast 04] represent loop nests as mathematical structures called polyhedra. Transformations are performed on these structures and then converted back into equivalent but transformed loop nests. The polyhedral model is also used for parallelisation and vectorisation.

C-to-CUDA [Bask 10] and PPCG [Verd 13] are both polyhedral GPU compilers. They create a polyhedral model of the parallel nested loops in the source code and perform advanced loop optimisations such as tiling and blocking to static loop nests with affine loop bounds and subscripts. The work in Chapter 5 shows how to rewrite rules are used to apply these types of optimisations in a functional language at a much higher level in the compiler.

Polly [Gros 12] is an implementation of the polyhedral model for loop and data locality optimisation in the LLVM infrastructure. It works on LLVM IR to analyse memory access patterns and perform classical loop transformations like tiling and blocking. It has been used to detect parallel loops for generating GPU code [Miku 14] and to optimise OpenCL programs and reduce divergence [Moll 16].

Pencil [Bagh 15] is an intermediate language defined as a restricted subset of C99. It is intended as an implementation language for libraries and a compilation target by DSLs. It also relies on the use of the polyhedral model to optimise code and is combined with an auto-tuning framework.

While the polyhedral model and compilers employing it are effective at performing loop

transformations, it is not suitable for all types of problem domains and optimisations.

3.2.3 Tensor Algebra Specific Compilers

Compilers for specific domains use knowledge about that domain to apply optimisations that would not be possible or not legal in a general setting. One such domain that has risen to prominence recently is performing mathematical operations on tensors, a generalisation of vectors and matrices to higher dimensions, in the context of machine learning.

The Tensor Algebra COmpiler (`taco`) [Kjol 17] is a C++ library and compiler for automatically generating and optimising kernels for compound tensor algebra operations on dense and sparse tensors. `taco` uses a novel internal representation called iteration graphs to describe how to iterate over non-zero values of the tensor expression. It develops merge lattices that describes how to merge index data structures of sparse tensors that are used in the same tensor expression.

Accelerated Linear Algebra (XLA) [XLAT 17] is a domain-specific compiler for TensorFlow's computational graphs of linear algebra operations targeting and optimising for CPUs, GPUs and custom accelerators. It uses JIT compilation to analyse the graph at runtime and specialise the computation and fuse operations. Optimisation is performed in two stages. The first stage performs target independent optimisations like operation fusion and analysis for memory allocation. The second stage of optimisation takes place in the backend to perform target specific optimisations, such as further operator fusion beneficial for the specific device or replacing certain operations with optimised library calls. Finally, LLVM IR is generated and LLVM is invoked to perform further low-level optimisation and native code generation.

Glow [Rote 18] is a machine learning compiler for heterogeneous hardware. Neural networks are lowered to a dataflow graph based two-phase IR. The high-level graph IR is constructed when a neural net is loaded and supports some basic transformations such as constant propagation. The graph contains functions and storage nodes. The low-level IR is used for preforming optimisation on low-level memory operations and scheduling, such as transforming buffers in place instead of making copies or replacing copies with device specific DMA operations.

Domain specific knowledge has the ability to enable further optimisation of programs but building these type of compilers is more complex as it also requires introducing this knowledge as well as the transformations to exploit it.

3.2.4 Compiler Frameworks

As building compilers is complicated and time consuming, there have been efforts to build frameworks that ease producing compilers for domain specific uses.

Delite [Brow 11, Suje 14] is a compiler framework for creating DSLs. Delite provides implementation of high-performance parallel patterns, optimisations and code generators that can be reused across DSLs. It has been extended to explore different strategies for mapping data-parallel computations onto GPU hardware [Lee 14].

LMS [Romp 12] is a library-based generative programming approach that lowers the effort of creating program generators for writing in a high-level generic style while producing efficient and specialised programs. It makes a reusable and extensible compiler framework available at the library level. Code generators and the generated code are expressed in a single program. Projects using LMS include Delite (already described) and Spiral (see Section 3.3).

AnyDSL [Leis 18] is a framework based on partial evaluation for writing high-performance domain-specific libraries targeting CPUs as well as GPUs. Code generation techniques (GPU mapping or vectorisation) are exposed as compiler known higher-order functions which will be partially evaluated to avoid costly closure allocations at runtime. Users control the partial evaluation with annotations, allowing to instantiate generic implementations with target specific code at runtime to generate code specialised for the target hardware.

3.2.5 Intermediate Representations

Compilers do not work on source code directly but use internal data structures that accurately represent the program and provide the means to manipulate it for the purpose of optimisation. Different representations of programs have implications on the type of transformations they are suitable for.

LLVM IR [Latt 04] is a widely used single static assignment (SSA) based strongly typed intermediate representation that forms the core of the LLVM project. It is part of a large and mature project with a large number of analyses, optimisation passes and backends and has found widespread use in industry. It includes backends for AMD and NVIDIA GPUs.

INSPIRE [Jord 13] is a high-level parallel intermediate representation that supports parallel language constructs for thread identification, spawning and merging threads as well as communicating between threads. It is suitable for representing programs written in different parallel programming standards, such as OpenMP, Cilk, OpenCL and MPI.

SPIR (Standard Portable Intermediate Representation) [Khro 14] is a binary intermediate representation that was designed to be used for parallel OpenCL programs and is based on LLVM IR. One of its goals is to be able to distribute partly-compiled device independent binaries instead of source code. It was later redesigned as SPIR-V [Khro 19b] to be independent of LLVM IR and to be able to represent graphical shaders as well as compute kernels. In addition to OpenCL, it is now also used by SYCL, Vulkan [Khro 19d] and OpenGL [Khro 19a].

Thorin [Leis 15] is a graph-based, higher-order, functional IR based on continuation-passing style that is suitable for representing both, imperative as well as functional programs. It also

introduces lambda mangling, that takes the place of classical transformations like tail-call elimination, loop unrolling, loop peeling and inlining and simplifies their implementation significantly. Thorin is used as the intermediate representation of the AnyDSL project.

Distributed Multiloop Language (DMLL) ([Brow 16] is an intermediate language based on parallel patterns for targeting distributed heterogeneous systems. It introduces transformations to optimise for different devices and analyses for determining a distribution of work between devices. DMLL is implemented on top of the Delite DSL framework.

Compilers do a lot in terms of optimising programs for GPUs, especially when it comes to low-level details, but they have their limits and need to be complemented by other techniques to achieve higher performance than is reachable by a compiler alone.

3.3 Exploration of the Optimisation Space

For programs to be able to be automatically optimised and reach high-performance on a variety of devices with different performance characteristics, there needs to be a way to expose different optimisation choices and a way to explore the effect of those choices. This section discusses a number of approaches that have been proposed to automatically choose implementation parameters as well as projects that offer choice over optimisations.

3.3.1 Auto-Tuning

Auto-tuning is a way to choose parameter values of tunable kernels and provide users with the ability to adapt kernels to suit different devices. It has been used to tune libaries such as ATLAS [Whal 98a] for linear algebra and FFTW [Frig 05] for fast fourier transforms as well as in compilers. [Kisu 00, Agak 06] There exist a large number of auto-tuning projects in the literature and a few of them will be highlighted in this section.

OpenTuner [Anse 14] is a framework for creating domain-specific multi-objective auto-tuners. It supports a variety of parameters and search techniques, as well as user-defined ones providing domain specific knowledge. The user-provided domain specific knowledge is crucial for many problems. It uses several search techniques simultaneously, automatically assigning more work to the ones that find better performing configurations.

CLTune [Nugt 15] is a state-of-the-art generic auto-tuner for OpenCL kernels. It supports search strategies such as simulated annealing and particle swarm optimisation to deal with high-dimensional parameter spaces that may have many non-linearities.

ATF [Rasc 17] is a language-independent auto-tuning framework built on top of OpenTuner which enables the exploration of huge search spaces with inter-parameter constraints.

The Petabricks language [Phot 13] discussed below has been extended to run parts of the algorithm on different heterogeneous devices and the different algorithmic and mapping con-

figurations are explored using an evolutionary algorithm.

Auto-tuners are limited in the scope of the choices they can make and are restricted to parameter based tuning. All available choices have to be provided to the auto-tuner by its user and they lack the ability to automatically and drastically change the tunable kernels.

3.3.2 Exposing and Making Optimisation Choices

As auto-tuning on its own is too limited to achieve performance portability, there are various projects and frameworks to expose different algorithmic versions and differently optimised versions of programs to tune. This provides more flexibility and exposing the choices means that various options can also be automatically explored and appropriate ones selected.

Petabricks [Anse 09] allows the user to provide several algorithmic choices or implementations of the same algorithm. The compiler and runtime try to figure out the best combinations of them to use for a given processor. Petabricks has also been extended to generate OpenCL code [Phot 13].

Spiral [Pusc 05, Ofen 13] is a project that aims to generate and optimise digital signal processing algorithms for a large variety of devices. It uses DSL rewriting for loop optimisation and parallelisation as well as expressing algorithmic choices. More recently, it has also been extended to handle linear algebra [Spam 14].

Halide [Raga 13] *schedules* determine how a computation expressed as a Halide function should be performed. It provides primitives for performing computations in a dimension sequentially or parallel, for unrolling and vectorisation, dimension reordering and splitting dimensions into two. The schedule also specifies where temporary results should be stored or recomputed in the pipeline. Automatic scheduling based on locality and parallelism-enhancing transformations with choices using cost models has also been developed [Mull 16].

Tangram [Chan 16, Gonz 19] is a kernel synthesis framework based on *spectrums* and *codelets*. A *spectrum* specifies a computation, while a *codelet* provide specific implementations of them and they are interchangeable. Codelets can themselves invoke other codelets and interchanging them exposes different levels of composition to enable finding the best fit for different devices. Each codelet might contain parameters that can be tuned at either compile time or runtime.

Tiramisu [Bagh 19] is a polyhedral compiler to generate high-performance code for domains such as image processing, stencils, linear algebra and deep learning targeting CPUs, GPUs and distributed architectures. It introduces a scheduling language similar to that of Halide but with new extensions for partitioning computation, communication and synchronisation. Tiramisu uses a four-level IR to separate algorithms, loop transformations, data layouts and communication.

Locus [Teix 19] introduces a language to specify optimisation sequences separately from

the application itself. Code regions need to be marked in the source code of the application and given identifiers. The Locus DSL can then be used to specify, which optimisation sequences should be explored for the different regions.

The ability to automatically make choices about how a program should be implemented is crucial for adapting to different devices. Exploring different variations of the same program can be time consuming.

3.4 GPU Performance Modelling & Prediction

One approach to reducing the time it takes to explore and pick program implementations for a specific device is to predict the time it takes to run instead of actually running it. Performance modelling is a long standing field and there have been many projects for resource analysis and cost models for functional languages and algorithmic skeletons [Trin 13]. This section focuses on GPUs and describe different approaches to modelling the performance of programs running on GPUs.

3.4.1 Analytical Performance Modelling

One approach to performance modelling is to describe the device and its runtime behaviour as mathematical equations so that the time taken to execute a program can be directly calculated.

CuMAPz [Kim 11] is a compile time analysis tool that helps programmers to increase the memory performance of CUDA programs. It estimates the effects of performance-critical memory behaviours such as data reuse, coalesced accesses, channel skew, bank conflict and branch divergence. GROPHECY [Meng 11] uses the MWP-CWP model [Hong 09] (Memory Warp Parallelism – Computation Warp Parallelism) to estimate the GPU performance of skeleton-based applications and the effects of different transformations on the performance to choose the best transformation. GPUPerf [Sim 12] is a version of the analytical MWP-CWP model enhanced with understanding of cache effects, special functional units, parallelism and binary-level analysis. It provides a way of understanding performance bottlenecks and predicting the effects of different optimisations when applied to a program. The boat hull model [Nugt 12] is a modified version of the roofline model that is based on an algorithm classification and produces a roofline model for each class of algorithm. It uses the estimated amount of data and computation needed by the algorithm and the theoretical bounds of the device. It is also extended with data transfer costs to enable comparing running a program on the host or offloading it to an accelerator.

GPU cache models [Nugt 14b] have been built by extending reuse distance theory with parallel execution, memory latency, limited associativity, miss-status holding-registers and warp divergence and memory coalescing to model cache behaviour and predict miss rates based on a

memory trace aquired using Ocelot [Diam 10]. COMPASS [Lee 15] introduces a language for creating analytical performance models that analyse the amount of floating point and memory operations based on static code features. Coloured petri nets [Mado 16] have been proposed for GPGPU performance modelling by simulating both, the program and the hardware by moving tokens between different nodes in the net. Another approach [Beau 17] builds an analytical performance model to determine the lower bound on execution time. The lower bound is used to prune implementations that can not achieve the good performance from the optimisation space. Low-level GPU ISA solving and assembly microbencharking [Zhan 17] has been used to collect data about architectural features and performance.

Sensitivity Analysis via Abstract Kernel Emulation [Hong 18] aims to predict execution time and determine resource bottlenecks for a given NVIDIA GPU kernel binary. It emulates a small number of thread blocks for the target GPU and extrapolates from the results to determine execution time of the whole kernel.

Analytical models describe low-level details of the hardware to model performance using a model written by a hardware expert. They typically use low-level kernel representations to make their predictions. Developing analytical models requires a significant amount of time and effort and it is far from trivial to adapt them to new hardware devices. In contrast, the approach presented in Chapter 6 based on machine-learning is fully automatic.

3.4.2 Statistical Performance Modelling

To automate building performance models without requiring detailed knowledge about the hardware and its behaviour statistical methods and machine learning can be used.

Early work [Duba 07] extracts static code features and uses a machine learning model that only needs a small number of training samples per program to predict the performance of optimisation sequences and therefore find good transformation sequences. Principal component analysis, cluster analysis and regression modelling have been used [Kerr 10] to generate predictive models for GPUs and CPUs. Predictive modelling has also been applied in polyhedral compilation [Park 11] to predict speedups for different combinations of polyhedral transformations based on hardware performance counters. Graph-based program characterisation [Park 12] has also been used for polyhedral compilation to predict the speedups of optimisation sequences. Features are collected for every basic block in the control-flow graph (CFG) and the whole graph is used as input to the model.

Clustering on similarity of a graph-based intermediate representation [Demm 12] has been used to cluster similar programs and detect similarities that are not obvious for humans looking at the code. Programs in clusters react similarly to optimisations so one program from a cluster can be used to develop optimisations also beneficial for the others. Another approach [Stoc 12] uses machine learning models trained on assembly level features to choose a good combina-

tion of transformations for vectorisation. The model predicts the performance of different vectorised versions and uses the prediction to rank them and output the one with the best predicted performance.

MaSiF [Coll 13] uses principal component analysis (PCA) and the k-nearest neighbour algorithm to auto-tune skeleton parameters for programs written using TBB and FastFlow. To tune a new program, it's features are extracted, the k closest near optimal programs are computed. PCA is then applied to the near optimal parameter values, the mean and eigenvectors are used to conduct the search. Stargazer [Jia 12] uses step-wise linear regression together with cubic splines to estimate the performance of programs on different GPU designs in GPGPU-Sim [Bakh 09]. Starchart [Jia 13] uses random sampling and building regression trees to divide the whole optimisation space into smaller subspaces. The model predicts the performance or power usage, based on program parameters such as thread-block sizes, different data layouts and usage of caches or scratchpad memory.

Regression trees have also been used as part of a hybrid method for autotuning [Pric 15] as the fitness function for a genetic algorithm to make the auto-tuning converge quicker into the optimal or near-optimal solution.

Statistical models in compilers traditionally use features extracted from a deep stage in the compilation pipeline. The work in Chapter 6 instead extracts them at a considerably higher-level from a functional IR.

3.5 Summary

This chapter presented an overview of research work related to this thesis, starting from high-level library and language based approaches for GPU programming. Various compilers from academia and industry along with compiler frameworks and intermediate representations were described next, followed by auto-tuning for parameter selection and techniques for exposing and choosing between optimisation choices. Finally, analytical and statistical techniques for GPU performance modelling were discussed.

While the presented work has made great progress towards ease of programming and achieving high performance for GPUs, the needs identified in Chapter 1 required to achieve performance portability have still not been fully addressed in a single approach. The rest of this thesis will present techniques developed towards a first solution of the performance portability problem. The next chapter addresses one of the needs identified in Chapter 1 by describing techniques to compile functional LIFT programs where optimisations have been explicitly encoded into efficient imperative OpenCL code.

Chapter 4

High-Performance GPU Code Generation

4.1 Introduction

This chapter describes the process of compiling the functional data-parallel LIFT language, which explicitly expresses OpenCL-specific constructs, into efficient imperative OpenCL code. This functional language is built on top of lambda-calculus and expresses a whole computational kernel as a series of nested and composed function calls to built-in parallel patterns. As discussed in Chapter 2, it is equipped with a limited dependent type system that reasons about array sizes and value ranges for variables, preserving important semantic information from the high-level patterns. The information available in the types is used at multiple stages during compilation, such as when performing array allocation, index calculation, and even synchronisation and control flow simplification.

One traditional downside of a functional language is that all operations are represented as functions which produce intermediate results, which require additional storage. This issue is well known in the functional community [Wadl 90]. In LIFT it is addressed by fusing chains of composed or nested functions that only affect the data layout (e. g. *zip* or *gather*). This involves recording information about the accessed data in a *view* structure which is then used to emit the appropriate array access expression. These indexing expressions are simplified using a symbolic algebraic simplifier that relies on type information (e. g. array length and value ranges). This compilation process generates highly efficient GPU code competitive with hand-tuned kernels. Without this level of performance, any automatic exploration of optimisation space using LIFT could not match the performance of hand tuned kernels.

This chapter presents the following contributions:

- it shows how semantic information embedded in the high-level functional language is exploited in various phases of the compilation process, such as *memory allocation*, *ar-*

ray access generation and optimisation, *synchronisation* minimisation and *control-flow simplification*, to generate highly efficient low-level imperative OpenCL code;

- it demonstrates that common but complex optimisations in GPU programming are expressible in LIFT programs;
- it demonstrates that the performance of the generated GPU code is on par with manually optimised OpenCL code.

The rest of the chapter is organised as follows: Section 4.2 motivates our approach. Section 4.3 discusses the compiler implementation and optimisations for high-performance code generation. Section 4.4 gives some intuition of how the different optimisations used by the benchmarks in the evaluation are expressed in LIFT. Section 4.5 and Section 4.6 present the experimental setup and evaluation before Section 4.7 concludes the chapter.

The author contributed to and extended an existing implemention of type checking and address space inference and memory allocation, described in section 4.3.1 and section 4.3.2. The author redesigned or independently developed generating and optimising array accesses, barrier eliminaion and the code generation and the control flow simplification performed during code generation, described in section 4.3.3, section 4.3.4 and section 4.3.5.

4.2 Motivation

The problem of producing efficient GPU code has been well studied over the years. Figure 4.1 gives an overview of the different approaches related to the work presented in this chapter. Loop-based auto-parallelisation techniques have been extensively studied for languages like C [Bask 10, Gros 12, Verd 13, Miku 14, Bagh 15]. Recent work on polyhedral compilation [Bagh 15] for instance has pushed the boundaries of such techniques for GPU code generation. However, these techniques only operate on loops and require certain property such as affine indices to work effectively.

In the last decade, there has been a shift towards algorithmic skeletons and Domain Specific Languages (DSLs). These approaches offer the advantage of exploiting high-level and domain-specific information. The simplest approaches are based on parametric library implementation of skeletons such as Thrust [Bell 11] and SkelCL [Steu 11]. However, these approaches are not portable and more importantly, cannot optimise across library calls.

A different approach consists of lowering the applications to a functional representation which is then compiled into GPU code. This process involves the mapping of parallelism, performing optimisations such as fusion of operations and finally code generation. This approach is used by a many systems such as Copperhead [Cata 11], Delite [Brow 11], Accelerate [Chak 11, McDo 13], LiquidMetal [Duba 12], HiDP [Zhan 13], Halide [Raga 13] and

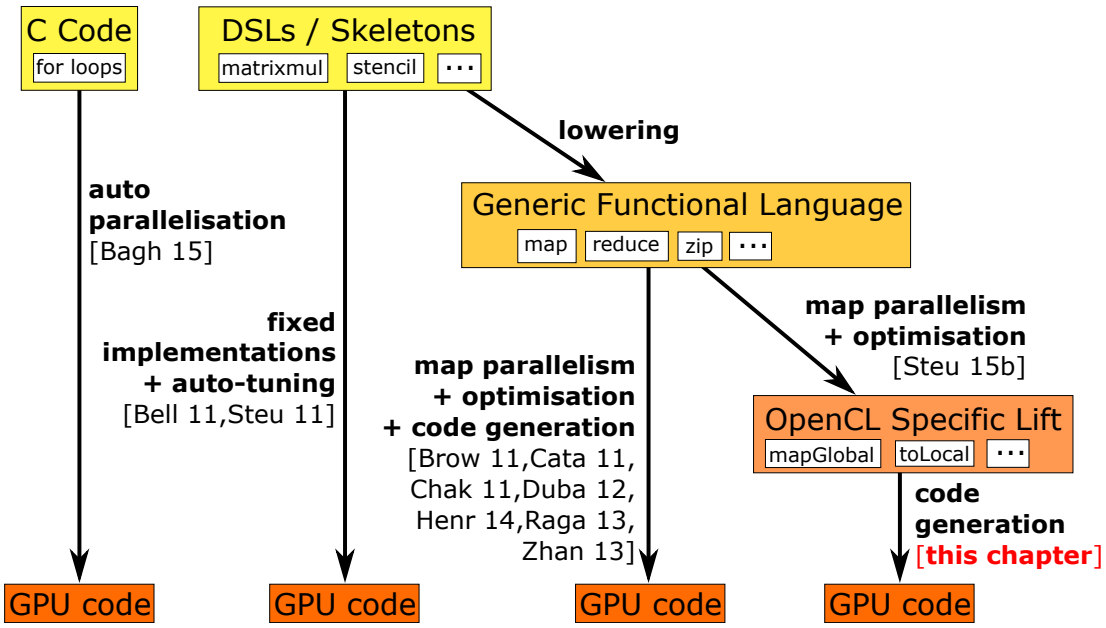


Figure 4.1: GPU code generation landscape.

NOVA [Coll 14].

The drawback of such approaches is that the mechanism to map the parallelism and optimise the code is performed within the code generator and, in general, uses a fixed strategy driven by heuristics. This means that it is challenging to achieve performance portability due to the large gap between the functional language and the GPU code that will eventually be produced. In contrast, LIFT is a language which encodes OpenCL-specific constructs. The decisions of how to optimise code and map the parallelism in LIFT are taken during the conversion from the high-level generic form to the low-level OpenCL-specific form. This clearly separates the concerns of optimisation and parallelism mapping from the actual process of code generation.

This chapter demonstrates that LIFT is capable of expressing many different OpenCL mappings and optimisations in a pattern-based and functional style. GPUs are programmed using imperative languages and the functional LIFT language still needs to be transformed to such a form. The main contribution of this chapter is to demonstrating novel techniques used by the LIFT compiler to produces efficient imperative OpenCL code once the original program has been lowered and mapped into a LIFT program that explicitly encodes optimisations. While a naïve code generation approach would be straightforward to implement, the evaluation shows that generating high performance OpenCL code is non-trivial and relies on using the semantic information of the parallel patterns encoded in the language.

Figure 4.2: Overview of the LIFT compilation stages.

4.3 Compilation Flow

Figure 4.2 shows the stages involved in compiling the functional LIFT language into efficient imperative OpenCL code. The compilation starts by analysing type information and making sure the program is well-typed. Type information is also used heavily in the subsequent memory allocation and efficient array accesses generation passes and is therefore the key for high-performance code generation.. The barrier elimination stage minimises the number of synchronisations required for correct parallel code to avoid unnecessary synchronisation overheads. Finally, the OpenCL code is generated using code snippets and the final optimisation of simplifing control flow is performed.

4.3.1 Type System and Analysis

The LIFT compiler implements a limited form of a *dependent type system* which keeps track of the length and shapes of nested arrays. Besides *array types*, the type system supports *scalar types* (e. g. `int`, `float`), *vector types*, and *tuple types*. While vector types correspond to OpenCL vector data types (e. g. `int2`, `float4`) tuples are represented as `structs`. Array types can be nested to represent multi-dimensional arrays. In addition, arrays carry information about the length of each dimension in their type. This length information comprises of arithmetic expressions of operations on natural numbers larger than zero and named variables which are unknown at compile time. For example, given an array x of length n where the type of the elements is `float` the type of x is written as $[\texttt{float}]_n$. Applying the *split(m)* pattern to the array x results in the type $[[\texttt{float}]_m]_{n/m}$.

The types of function bodies are automatically inferred from the parameter types by traversing the IR following the data flow.

4.3.2 Address Space Inference and Memory Allocation

A straightforward memory allocator would allocate a new output buffer for every single `FunCall` node. However, this would be very inefficient as data layout patterns, such as *split*, only change the way memory is accessed but do not modify the actual data. Memory allocation is, therefore, only performed for functions actually modifying data. These are `FunCall` nodes where the called function is a `UserFun` node. For these nodes, the compiler uses the array length information from the type as well as the target address space to compute the size of the memory buffer required. When a data layout pattern is encountered, an internal data structure called a

view is created, which remembers how memory should be accessed by the subsequent functions. Details of the *views* are discussed in section 4.3.3.

Address Space Inference Memory is allocated in one of three OpenCL *address spaces* and before the size of the buffer can be calculated, the address space for the buffer needs to be determined. The address space affects which threads can access the data and how the size of an allocation in that address space needs to be specified. Global memory can be accessed by all threads and the size of the allocation is specified as the total amount of memory used by all threads. Local memory can be accessed by threads in a single work-group and the size of the allocation is specified as the amount used by a single work-group. Private memory can only be accessed by a single thread and the size of a private memory allocation is the amount of memory used by a single thread.

The algorithm for inferring address spaces places all the arguments to the LIFT program to global memory and then traverses the program following the data-flow, propagating and assigning address spaces the output of all functions will reside in. Patterns that do not perform any computation just propagate the address space of their arguments. Patters that do perform computation either get the address space for their output based on the address spaces of their arguments or from *toPrivate*, *toLocal*, *toGlobal* they are nested in.

More details can be found in [Steu 17].

Size Computation and Memory Allocation Once the address spaces have been determined, the sizes of buffers are computed using the address space and type information. To do this the IR is traversed depth-first following the data-flow, while keeping track of the amount of memory that needs to be allocated for different address spaces based on the patterns and types encountered during the traversal, as described in Algorithm 1.

The algoritm takes a lambda reprsenting a LIFT program and returns a map that for every node contains information about which bufers it accesses and what size the buffer needs to be. The algorithm starts by taking the program lambda and allocating memory for the inputs of the program and inserting them to the map. The main function of the algorithm is then called in line 4 to propagate those and all other allocated memories through the whole IR while keeping track of the number of elements that would need to be allocated, based on the array length information from the type. These numbers are tracked separately for the three address spaces that might be encountered.

As before, when inferring the address spaces, the three different cases of `Expr` nodes: `Literal`s, `Param`s and `FunCall`s are considered separately. Memory for `Literal`s can be allocated based on the size of their type in bytes as seen in line 6. Memory for `Param`s has to be set when their function is called, as already seen above during address space allocation, so the algorihm just returns. When a new `FunCall` node is entered, its input memories entered

input : Lambda expression representing a program

output: A map containing memory information for all expressions

```
allocateMemoryProg(in: lambda)
1  memoryMap = {}
2  foreach param in lambda.params do
3  |  memoryMap = insert(memoryMap, param, allocateMemory(param.type.size, param.as))
4  return allocateExpr(lambda.body, 1, 1, 1, memoryMap)

allocateExpr(in: expr, in: numGlb, in: numLcl, in: numPvt, in: memoryMap)
5  switch expr do
6  |  case Literal return insert(memoryMap, expr, allocateMemory(expr.type.size, expr.as));
7  |  case Param return memoryMap;
8  |  case FunCall(f, args)
9  |  |  foreach arg in args do
10 |  |  |  memoryMap = allocateExpr(arg, numGlb, numLcl, numPvt, memoryMap);
11 |  |  switch f do
12 |  |  |  case Lambda(body)
13 |  |  |  |  memoryMap = allocateLambda(f, args, numGlb, numLcl, numPvt, memoryMap);
14 |  |  |  |  return insert(memoryMap, expr, lookup(memoryMap, body));
15 |  |  |  case toPrivate(f) or toLocal(f) or toGlobal(f)
16 |  |  |  |  memoryMap = allocateLambda(f, args, numGlb, numLcl, numPvt, memoryMap);
17 |  |  |  |  return insert(memoryMap, expr, lookup(memoryMap, f.body));
18 |  |  |  case MapGlb(f) or MapWrg(f)
19 |  |  |  |  memoryMap = allocateLambda(f, args, numGlb * lengthOfArray(expr.type), numLcl, numPvt,
   |  |  |  |    memoryMap);
20 |  |  |  |  return insert(memoryMap, expr, lookup(memoryMap, f.body));
21 |  |  |  case MapLcl(f) or MapSeq(f)
22 |  |  |  |  length = lengthOfArray(expr.type);
23 |  |  |  |  if args.as.containsPrivateMemory or f.body.as.containsPrivateMemory then
24 |  |  |  |  |  numPvt = numPvt * length;
25 |  |  |  |  memoryMap = allocateLambda(f, args, numGlb * length, numLcl * length, numPvt,
   |  |  |  |    memoryMap);
26 |  |  |  |  return insert(memoryMap, expr, lookup(memoryMap, f.body));
27 |  |  |  case Reduce(f)
28 |  |  |  |  memoryMap = allocateLambda(f, args, numGlb, numLcl, NumPvt, memoryMap);
29 |  |  |  |  memoryMap = replace(memoryMap, lookup(memoryMap, f.body), lookup(memoryMap,
   |  |  |  |    args.head)); // Last write to the initial value
30 |  |  |  |  return insert(memoryMap, expr, lookup(memoryMap, f.body));
31 |  |  |  case UserFun
32 |  |  |  |  numElems = getNumElementsForAS(expr.as, numGlb, numLcl, numPvt);
33 |  |  |  |  return insert(memoryMap, expr, allocateMemory(numElems * expr.type.size, expr.as))
34 |  |  |  case Iterate(f, _)
35 |  |  |  |  numBytes = // Calculate swap buffer size
36 |  |  |  |  memoryMap = insert(memoryMap, f, allocateMemory(numBytes, args.head.as));
37 |  |  |  |  memoryMap = allocateLambda(f, args, numGlb, numLcl, numPvt, memoryMap);
38 |  |  |  |  return insert(memoryMap, expr, lookup(memoryMap, f.body));
39 |  |  |  otherwise do return insert(memoryMap, expr, lookup(memoryMap, args));

allocateLambda(in: lambda, in: args, in: numGlb, in: numLcl, in: numPvt, in: memoryMap)
40 foreach p in lambda.params and a in args do
41 |  memoryMap = insert(memoryMap, p, lookup(memoryMap, a))
42 return allocateExpr(lambda.body, numGlb, numLcl, numPvt, memoryMap)
```

Algorithm 1: Recursive memory allocation algorithm

to the map by visiting its arguments (line 10). Once the input memories are inserted into the map, the type of the function being called is inspected to decide what to do next. If the function has child nodes, e. g. a *map*, the memory of the `FunCall` node will be updated based on them. Otherwise, e. g. for a *split*, the argument memory is propagated to the current node (line 39).

For *Lambda* the input memories are propagated to the parameters and then the body of the lambda is visited using the `allocateLambda` helper function in line 13. Since *toPrivate*, *toLocal* and *toGlobal* only affect the address space, they are ignored and treated almost the same as *Lambda* after the address space inference has run.

When a *map* node is encountered, the type information is used to update the amount of memory required. This happens as follows for different OpenCL specific *map* nodes. For a *mapGlb* or *mapWrg* (line 18), the amount of global memory required is updated by multiplying the current amount with the length of the array being mapped over. Only the global amount is updated, since local and private memory are by definition available only to a single work-group or work-item (thread). For a *mapLcl* or *mapSeq* (line 21), the amount of global *and* local memory required are updated in the same manner as before. Additionally, if anything using private memory appears as an input or an output, the amount of private memory required is also updated. When the amounts have been updated, the input memories are propagated and memory is recursively allocated for the body of the *map* node using the same `allocateLambda` helper function. A *reduce* (line 27) is handled almost like an address space pattern, except the memory allocated for the final write (the memory of the body of the lambda) is replaced in the whole map with the memory for the initial value of the accumulator in line 29. An *iterate* needs a swap buffer to be allocated, to be able to use double buffering.

Finally, when a `FunCall(UserFun)` node is encountered (line 31), a new memory object needs to be allocated to hold its result. The number of elements to allocate (*numGlb*, *numLcl* or *numPvt*, which contain information about how the `UserFun` is nested inside other patterns) is chosen based on the address space inferred for the current `FunCall` node. A new memory object in the required address space is allocated and its size is calculated by multiplying the size of the return type with the number of elements required for that address space. The algorithm keeps traversing the IR and propagating the newly allocated memory objects.

As an example, consider the function in Listing 4.1. When the first user function in line 3 is reached *numGlb* will be $N \times M \times 4$, *numLcl* will be $M \times 4$, and *numPvt* will be 4 as *mapWrg*, *mapLcl* and *mapSeq* have been visited. As the input and output of *mapLcl* read and write global memory, *numPvt* has not been updated. The final size of the allocation depends on which concrete address space pattern *toAddressSpace* is, *toGlobal*, *toLocal* or *toPrivate*. So, if the pattern is *toGlobal*, then the size of the allocation will be *numGlb* or $N \times M \times 4$ bytes.

The objects denoting the allocated buffers, that were propagated through the IR, tie together the accesses to the same locations in the final allocated OpenCL buffers.

```
1 f(x: [[[float]4]M]N) =
2 (mapWrg(0)(mapLcl(0)(mapSeq(toGobal(id)) ∘
3    mapSeq(toAddressSpace(id)))))(x)
```

Listing 4.1: Memory allocation with different address spaces. Depending on `toAddressSpace` the memory needs to be allocated differently.

```
1 partialDotProduct(x: [float]N, y: [float]N) =
2 (join ∘ mapWrg(0)( ...
3   join ∘ mapLcl(0)( ...
4    reduceSeq(0, λ(a,xy). a + (xy0 × xy1))) ∘ split(2)
5  ) ∘ split(128))( zip(x, y) )
```

Listing 4.2: Partial dot product.

4.3.3 Multi-Dimensional Array Accesses

In the LIFT IR, arrays are not accessed explicitly but implicitly; the patterns determine which thread accesses which element in memory. This design simplifies the process of lowering high-level programs to the LIFT IR and guarantees that data races are avoided by construction since no arbitrary accesses into memory are permitted and threads will not try to write to the same location. However, this introduces two main challenges when compiling the LIFT IR: First, avoiding unnecessary intermediate results arising from functions which only change the data layout; And, secondly, generating efficient accesses to multi-dimensional arrays which have a flat representation in memory.

Example Consider the dot product example in Listing 4.2. We are interested in understanding how the arrays `x` and `y` are accessed inside the lambda in line 4 and, ultimately, how to generate code to express these accesses. This is not obvious, as the arrays are first combined using *zip* and then split into chunks of size 128 in line 5. When processing a single chunk inside a work group (*mapWrg* in line 2), the array is further split into smaller chunks of two elements (line 4) and every local thread (*mapLcl* in line 3) performs a sequential reduction. Individual elements of the arrays are accessed using the `xy` variable. The $\texttt{xy}_0$ indicates an access to the first element of the tuple, which is an element of array `x`.

***View* Construction** A *view* in the LIFT IR describes an internal data structure which stores information for generating array accesses. Most patterns produce *view*s, but importantly, functions that only change the data layout will not produce any code to allocate memory and copy data to a new array in the specified fashion, but will only produce a *view*.

Most patterns have a *view* corresponding to them but a single *view* type can be used for

View	Contents	Generated for Pattern
MemoryView	variable used for the memory	N/A
ArrayAccessView	predecessor, iteration variable *i*	*map*, *reduce*
ArrayExitView	predecessor, iteration variable *i*	*map*, *reduce*
ZipView	predecessor views being zipped	*zip*
TupleAccessView	predecessor, tuple component accessed	*get*
SplitView	predecessor, size of dimension being created *s*	*split*
JoinView	predecessor, size of dimension being joined *s*	*join*
AsVectorView	predecessor, vector width introduced *w*	*asVector*
AsScalarView	predecessor, vector width removed *w*	*asScalar*
ReorderView	predecessor, reordering function *f*	*gather*, *scatter*

Table 4.1: Summary of the views used in the LIFT compiler, along with their contents and the patterns they are used for.

several different patterns. A summary of the different types of views and the patterns that generate them is shown in Table 4.1. For example, all *reduce* and *map* patterns generate ArrayAccessViews when they are entered during the traversal. This is because for all of them only the variable to use for the access, the type and the predecessor view need to be stored. The ArrayAccessView indicates that the next inner dimension will be the next one accessed. When *map* and *reduce* IR nodes are exited during the traversal an ArrayExitView is generated. It is the opposite of an ArrayAccessView and signals that the next outer dimension is now being accessed. The reordering patterns, *gather* and *scatter*, either generate a ReorderView which contains the predecessor and the reorder function or are ignored depending on whether read or write accesses are being generated. *split*, *join*, *asVector*, and *asScalar* all have corresponding view types that store the chunk size or vector width. The address space patterns are examples of patterns that do not emit a view and get ignored at this stage.

To generate the read array access for the $\texttt{xy}_0$ expression from our example, the IR is traversed following the data flow until the `FunCall` node for the user function is encountered as that is the point where the access is made. For each node a *view* representing how the particular node influences the array access is constructed. The resulting *view* structure is shown on the left hand side of Figure 4.3 where each view is connected to its predecessor view. For example, the ZipView has two predecessors, since the two arrays `x` and `y` have been combined. Each *map* and *reduce* pattern results in a ArrayAccessView which represents an access in one dimension of the array by the function and stores the iteration variable that is used for the access in this dimension. Nested ArrayAccessViews, therefore, correspond to accesses to multi-dimensional arrays.

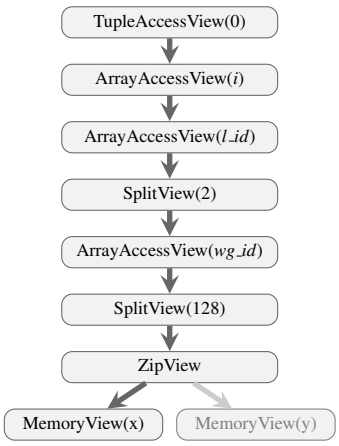


Figure 4.3: *View*s constructed for the generation of the first memory access of dot product.

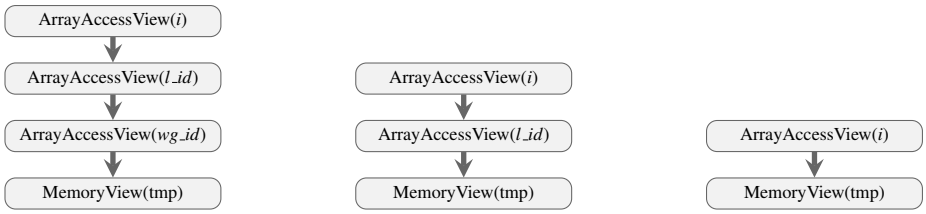
```
1 f(x: [[[float]4]]M]N) =
2 (mapWrg(0)(mapLcl(0)(mapSeq(toGobal(id)) ∘
3    mapSeq(toAddressSpace(id)))))(x)
```

Listing 4.3: Fresh view creation after a user function. Depending on `toAddressSpace` the new view needs to be created differently.

After a `FunCall` containing a user function is encountered, a fresh view structure needs to be created before continuing the traversal of the IR, in case there is another user function that needs to read the result of this one. That is the case in line 3 of the example in Listing 4.3. Furthermore, the new view is different depending on which particular address space pattern *toAddressSpace* is.

To be able to create a new view after encountering a user function, information about how every user function is nested in *map* and *reduce* patterns is needed. Specifically, the iteration variables that have been used until this point and their range is needed. The new views also need to match the address space that is used. For example, because local memory in OpenCL only exists within a single group, then it can never be accessed by an iteration variable corresponding to a *mapWrg*. Similarly, because private memory only exists within a single thread, it can never be accessed with a thread id. In Listing 4.3, assuming *mapWrg* uses the iteration variable `wg_id` ranging from `0` to `N` for indexing, *mapLcl* uses the iteration variable `l_id` ranging from `0` to `M` for indexing, and the *mapSeq* uses the iteration variable `id` ranging from `0` to `4` for indexing. To create a new view for accessing global memory, information about all three *map*s is required; for local memory information about the *mapLcl* and *mapSeq* is required, and for private memory, just information about the *mapSeq* is required.

Using the address space and nesting information a new MemoryView is created. Figure 4.4 shows the new views that will be created for the example in Listing 4.3. One ArrayAccessView is created for every iteration variable that is needed for the current address space. This means



(a) Global memory (b) Local memory (c) Private memory

Figure 4.4: New views for different address spaces before continuing traversal for the example in Listing 4.3.

that the MemoryView for global memory will be created with the type $[[[\texttt{float}]_4]_M]_N$ and contain the variables `wg_id`, `l_id` and `i` as shown in Figure 4.4a. For local memory, the MemoryView will be created with the type $[[\texttt{float}]_4]_M$ and the will contain the variables `l_id` and `i` as shown in Figure 4.4b. And for private memory, the MemoryView will be with the type $[\texttt{float}]_4$ and contain just the variable `i` as shown in Figure 4.4c. Once the new view is created, traversal continues as before.

To generate the *view* for the write access storing the result of the reduction function $(\texttt{a} + (\texttt{xy}_0 \times \texttt{xy}_1))$ in Listing 4.2, the IR is traversed in the opposite order. Because of the reverse direction of traversal, the dual for a read view is created. I.e. for a *join* a SplitView is created and for a *split* a JoinView is created. For reorder patterns, this time *gather* is ignored and a ReorderView is created for *scatter*, resulting in the reordering being applied on the correct access. When encountering a user function, fresh views are created for all its arguments. For our example, the *view* for generating the store would look very similar to the loads, but without the TupleAccessView, the ZipView and with only one MemoryView for the output.

***View* Consumption** Once the *view* structures are constructed, all information required for generating memory accesses is available. Consuming a *view* follows the exact same procedure for reads and writes. An array index expression is calculated by consuming this information in the opposite order of construction, i. e. top-to-bottom. This process is illustrated on the right hand side of Figure 4.5 with the resulting array access at the bottom. The constructed view is shown on left hand side. The *Tuple Stack* on the right side contains information about tuple access which determine which array is being accessed. The *Array Stack* in the middle records information about which element of the array is being accessed.

Starting from the top with two empty stacks, the TupleAccessView(0) is processed first and pushes the first component of a tuple, i. e. *0*, onto the tuple stack. Then an ArrayAccessView pushes a new variable (i) on the stack indexing the array in one dimension. Another ArrayAccessView pushes another index variable (l_id) on the stack. The SplitView pops two indices

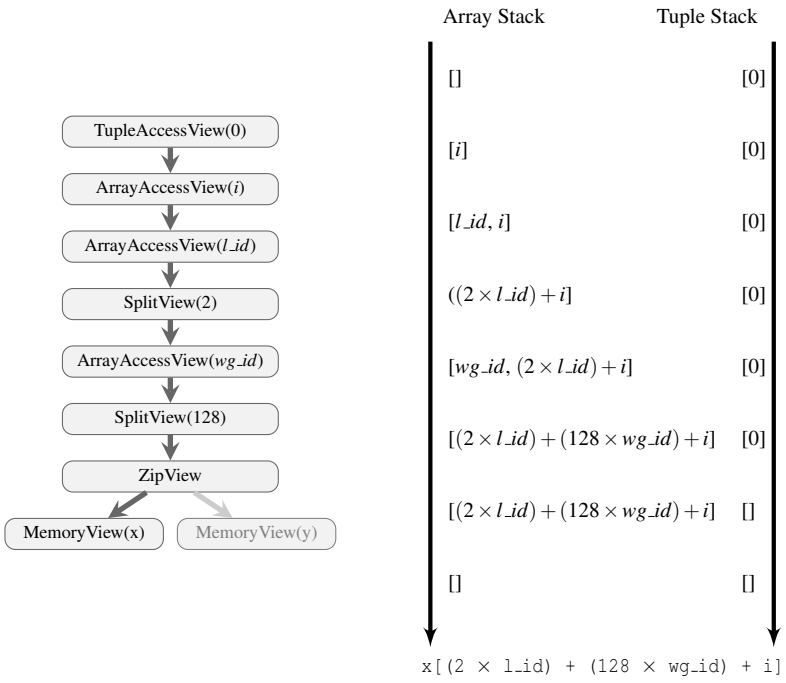


Figure 4.5: *View*s constructed for the generation of the first memory access of dot product (on the left) and consumption of views to generate an array index (on the right).

from the stack and combines them into a one dimensional index using the split factor, linearising the array access. The ZipView pops from the tuple stack and uses this information to decide which view should be visited next: the MemoryView(x). Finally, a memory view is reached which is used to emit the final index to the memory of input x.

In addition to the examples already seen, other *view*s modify the index as follows, before visiting its child view.

- AsScalarView pops the top index i off the stack, divides it by the vector width w, and pushes the new index i/w to the stack
- AsVectorView pops the top index i off the stack, multiplies it by the vector width w, and pushes the new index $i \times w$ to the stack
- ReorderView pops the top index i off the stack, applies its reorder function f, and pushes the new index $f(i)$ to the stack
- JoinView pops the top index i off the stack, distributes it into two indices using the split factor s, and pushes the new indices i mod s and i/s to the stack in that order
- ArrayExitView pops the top index i off the stack and recursively replaces its iteration variable in all its child views with i to propagate the correct variable to use for the access

```
1 matrixTranspose(x: [[float]_M]_N) =
2 (mapWrg(0)(mapLcl(0)(id)) ∘
3  split(N) ∘ gather(λ(i) → i/N + (i mod N) × M) ∘ join)(x)
```

Listing 4.4: Matrix transposition in LIFT. Without simplifying the array accesses obtained from the input view, the arithmetic expressin is needlessly complicated.

```
1 ((((wg×N+l)/N)+(((wg×N+l) mod N)×M))/M)×M+(((wg×N+l)/N)+(((wg×N+l) mod N)×M)) mod M
2 ((  wg         +         l       ×M) /M)×M+(  wg         +         l       ×M) mod M
3                          l              ×M+   wg
```

Figure 4.6: Simplification process of automatically generated array indices.

Simplifying Array Accesses Following the approach described above will generate correct array indices, however, this naïve treatment leads to long and overly complex expressions. This issue is illustrated using matrix transposition, expressed in LIFT as shown in Listing 4.4. Here the *join*, *gather* and *split* patterns flatten the two-dimensional matrix, rearrange the indices with a stride before splitting the array in two dimensions. When generating the read accesses for the *id* function, following the methodology introduced above, the array index shown in Figure 4.6 line 1 is obtained. While this array index expression is correct it is also quite long compared to the index a human could write for performing a matrix transposition, shown in line 3.

However, a standard compiler would be unable to simplify this expression since important information about value ranges is missing. In contrast, the LIFT compiler is able to derive the simplified form using a symbolic simplification mechanism exploiting domain knowledge. The simplification process follows a set of algebraic rules exploiting properties of arithmetic operators supported in the compiler (additions, multiplications, integer divisions, fractions, powers and logarithms). A small subset of the rules supported is shown below:

$$x/y = 0, \quad \text{if } x < y \text{ and } y \neq 0 \tag{4.1}$$

$$(x \times y + z)/y = x + z/y, \quad \text{if } y \neq 0 \tag{4.2}$$

$$x \bmod y = x, \quad \text{if } x < y \text{ and } y \neq 0 \tag{4.3}$$

$$(x/y) \times y + x \bmod y = x, \quad \text{if } y \neq 0 \tag{4.4}$$

$$(x \times y) \bmod y = 0, \quad \text{if } y \neq 0 \tag{4.5}$$

$$(x + y) \bmod z = (x \bmod z + y \bmod z) \bmod z, \quad \text{if } z \neq 0 \tag{4.6}$$

The type system exploits domain specific knowledge by inferring range information for every variable. For iteration variables, the range takes form of addition, and has information about

the start, stop, and step of the variable. For example, the `wg_id` variable corresponding to the iteration variable used in the `for` loop generated for the *mapWrg*, starts at `get_group_id(0)`, stops at `M`, which is the row length of the input matrix, and adds `get_num_groups(0)` to itself after every iteration. The OpenCL built-in `get_group_id(0)` itself ranges from 0 up to `get_num_groups(0)`. Similarly, the `l_id` variable corresponding to the iteration variable used in the `for` loop generated for the *mapLcl* loop iteration variable, has values between `get_local_id(0)` and `N`, since it indexes an array split in chunks of `N`, and is incremented by `get_local_size(0)` after every iteration. The OpenCL built-in `get_local_id(0)` itself ranges from 0 up to `get_local_size(0)`, possibly enabling more simplification opportunities. If the local and/or global thread counts are statically known, `get_local_size(0)` and/or `get_num_groups(0)` can be replaced with those values.

The expression $(\texttt{wg\_id} \times \texttt{N} + \texttt{l\_id}) \bmod \texttt{N}$ can, therefore, be simplified to `l_id` using rule 4.6 to distribute the modulo followed by rules 4.3 and 4.5 to simplify the remaining modulo operations. A traditional OpenCL compiler is not able to simplify this code, as it is missing the information that `l_id` is positive and smaller than `N`. Lines 2 and 3 in Figure 4.6 show the expression after a few simplification steps. This results in the same compact array index a human would write.

In one case, disabling the simplification led to the generation of several MB of OpenCL code. By applying arithmetic simplification concise indices are generated which reduce code size and speed up execution as costly operations such as division and modulo can often be simplified away. The performance benefits will be investigated in Section 4.6.

4.3.4 Barrier Elimination

When different threads access the same memory location they must synchronise their accesses to ensure memory consistency. When compiling a LIFT program, this corresponds to generating an appropriate synchronisation primitive after each occurrence of a parallel *map* pattern. For *mapLcl* an OpenCL barrier is emitted synchronising all threads in the work group. It also acts as a memory fence, ensuring the data is accessible to all threads in the work group after synchronising. OpenCL does not support synchronisation across work groups and the only way to share data between work groups is to launch several kernels. A `return` is therefore emitted after the *mapGlb* and *mapWrg* patterns.

Under certain circumstances these barriers are not required, for example, when there is actually no sharing of data between threads because each thread continues to operate on the same memory locations or when there are two consecutive barriers emitted due to nested *map* patterns.

The LIFT compiler takes a conservative approach to avoiding data races, where a barrier is emitted by default and is only removed if it can infer from the context that it is not required.

```
1 zip(mapLcl(...), mapLcl(...))
```

(a) LIFT program snippet

```
1  // first argument of zip
2  for (int l_id_0 = get_local_id(0); l_id_0 < N; l_id_0 += get_local_size(0)) {
3    // ...
4  }
5  barrier(CLK_LOCAL_MEM_FENCE); // unnecessary barrier
6
7  // second argument of zip
8  for (int l_id_0 = get_local_id(0); l_id_0 < N; l_id_0 += get_local_size(0)) {
9    // ...
10 }
11 barrier(CLK_LOCAL_MEM_FENCE);
```

(b) Corresponding generated OpenCL

Figure 4.7: Barriers when generating code for several *mapLcl* that are arguments to a *zip*.

```
1 mapLcl(0)(mapLcl(1)(...))
```

(a) LIFT program snippet

```
1 for (int l_id_0 = get_local_id(0); l_id_0 < N; l_id_0 += get_local_size(0)) {
2   for (int l_id_1 = get_local_id(1); l_id_1 < N; l_id_1 += get_local_size(1)) {
3     // ...
4   }
5   barrier(CLK_LOCAL_MEM_FENCE); // unnecessary barrier, potentially invalid code
6 }
7 barrier(CLK_LOCAL_MEM_FENCE);
```

(b) Corresponding generated OpenCL

Figure 4.8: Barriers when generating code for nested *mapLcl*.

One insight for this barrier elimination process is the fact that for composed *mapLcl* functions, LIFT only allows sharing of data when using the *split*, *join*, *gather*, *scatter*, *asVector*, or *asScalar* patterns. These patterns are the only ones changing the access locations and cause different threads to read the data. Therefore, the compiler looks for sequences of sequentially composed *mapLcl* calls which have no *split*, *join*, *gather*, *scatter*, *asVector*, or *asScalar* between them and marks them specially. These marked *mapLcl* function calls will not emit a barrier in the OpenCL code generation stage.

All but the last barrier are eliminated in the situation where several *mapLcl* appear as arguments of a *zip* (Figure 4.7a). These barriers can be eliminated since the two arguments of a *zip* cannot read each other's output and can therefore be executed completely independently. In Figure 4.7b we can see two loops generated by two arguments to a *zip* and the barrier in line 5 is unnecessary and is eliminated.

Similarly, if there is a *mapLcl* nested inside another *mapLcl* (Figure 4.8a) the barrier generated by the inner *mapLcl* is eliminated, as by definition it cannot access its own output locations and the barrier after the outer loop will be taken before any of its output locations are read. In Figure 4.8b we can see two loops generated by two *mapLcl* where one is nested inside the other. The barrier in line 5 is unnecessary and is eliminated. Furthermore, if the number of local threads in dimension 0 is more than `N` or does not divide `N`, then the barrier in the nested loop is in a divergent control flow region, not all threads will take the barrier which can cause a deadlock. According to the OpenCL specification, a barrier must be encountered either by all threads of a work-group executing the kernel or by none at all, so the originally generated code with two barriers is not valid OpenCL without removing the unnecessary barrier in the nested loop.

If thread counts are known at compilation time then the presence of a barrier in a divergent control flow region that could not be removed is detected by the LIFT compiler and an error is raised. In a production environment, runtime checks can be emitted for the case where thread counts are unknown at compilation time. In the experiments presented in this chapter, the thread counts are always known and the checks are performed at compile time.

In the LIFT program in Figure 4.9a , a local memory array is first written to and then read from. As seen in the generated code in Figure 4.9b, in every iteration of the outer loop in line 2, generated by a *mapWrg*, the same locations in `arr` are reused. If the number of groups is known, it can be determined how many times the outer loop will be executed. If it is executed only a single time, i. e. the number of groups is `N / 128`, then the barrier in line 11 is removed. If not, the threads will need to synchronise after reading from `arr`, as some threads could be running faster than others and start overwriting the data in the next iteration of the loop.

4.3.5 OpenCL Code Generation

The final stage in the LIFT compilation pipeline is the OpenCL code generation where low-level optimisations are performed to precisely control the generated code. Computational kernels in OpenCL are passed to the device compiler as strings so a string containing the OpenCL C code for a program is the output of the LIFT compiler.

To generate the OpenCL code of an entire LIFT program, all the user functions are first emitted.

Secondly, the kernel signature is emitted and pointers or arrays for temporary buffers are emitted based on the memory allocation results. To generate the body of the kernel, the LIFT IR graph is traversed following the data flow and a matching OpenCL code snippets are generated for all computational patterns, as well as function calls, array accesses and calls to user functions.

As an example, the generated kernel for the dot product example in Listing 4.5 is shown in

```
1  mapWrg(mapLcl(toGlobal(...)) ∘ ... ∘ mapLcl(toLocal(...)))
```

(a) LIFT program snippet

```
1  local arr[128];
2  for (int wg_id = get_group_id(0); wg_id < N / 128; l_id_0 += get_num_groups(0)) {
3    for (int l_id = get_local_id(0); l_id < 128; l_id += get_local_size(0)) {
4      arr[l_id] = /* ... */;
5    }
6    barrier(CLK_LOCAL_MEM_FENCE);
7    // ...
8    for (int l_id = get_local_id(0); l_id < 128; l_id += get_local_size(0)) {
9      /* ... */ = arr[127 - l_id];
10   }
11   barrier(CLK_LOCAL_MEM_FENCE); // potentially unnecessary barrier
12 }
```

(b) Corresponding generated OpenCL

Figure 4.9: Barriers when generating code for several composed *mapLcl* in a *mapWrg*.

Listing 4.6 with only minor cosmetic changes made by hand for presentation purpose (renamed variables, removed comments, removed extra parenthesis).

The kernel signature in lines 1 to 3 of Listing 4.6 contains entries for all inputs, outputs, temporary buffers for intermediate results, and the lengths of all arrays. Storage for local buffers whose size is known at compile time is emitted into the kernel code as seen in lines 4 to 6. No OpenCL code is generated for patterns such as *split* and *toLocal* since their effect have been recorded in the views and allocated memory, respectively. For the different *map* patterns, `for` loops are generated, which for the parallel variations will be executed in parallel by multiple work groups or threads, such as the loop in in line 8. For the *reduceSeq* pattern, a loop with an accumulation variable (e. g. in line 11) is generated calling its function in every iteration. The code generated for *iterate* spans lines 20 to 34 with double buffering initializing two pointers in line 21 and swapping the pointers after each iteration in lines 32 and 33.

Control Flow Simplification The LIFT compiler performs control flow simplification using the extra semantic information available in patterns and types. A straightforward implementation would emit a `for` loop for every *map*, *reduce* and *iterate* pattern. Fortunately, the LIFT compiler often statically infers if the number of threads for a *map* is larger, equal or lower than the number of elements to process. This is the case in lines 23 and 35 of Listing 4.6 which correspond to the *mapLcl* in line 7 and 5 in the original Listing 4.5. Instead of a `for` loop, an `if` condition has been generated. This is possible because `get_local_id(0)` returns a non-negative number and given the local thread count is 64, only some threads will execute the body. If it is inferred that the loop executes exactly once by every thread the loop is elim-

```
1  kernel void KERNEL(const global float *restrict x,
2                     const global float *restrict y,
3                     global float *z, int N) {
4    local float tmp1[64];
5    local float tmp2[64];
6    local float tmp3[32];
7    float acc1; float acc2;
8    for (int wg_id = get_group_id(0); wg_id < N/128; wg_id += get_num_groups(0)) {
9      {
10       int l_id = get_local_id(0);
11       acc1 = 0.0f;
12       for (int i = 0; i < 2; i += 1) {
13         acc1 = multAndSumUp(acc1,
14                      x[2 * l_id + 128 * wg_id + i],
15                      y[2 * l_id + 128 * wg_id + i]);
16       }
17       tmp1[l_id] = id(acc1);
18     }
19     barrier(CLK_LOCAL_MEM_FENCE);
20     int size = 64;
21     local float *in = tmp1; local float *out = tmp2;
22     for (int iter = 0; iter < 6; iter += 1) {
23       if (get_local_id(0) < size / 2) {
24         acc2 = 0.0f;
25         for (int i = 0; i < 2; i += 1) {
26           acc2 = add(acc2, in[2 * l_id + i]);
27         }
28         out[l_id] = id(acc2);
29       }
30       barrier(CLK_LOCAL_MEM_FENCE);
31       size = size / 2;
32       in   = (out == tmp1) ? tmp1 : tmp3;
33       out  = (out == tmp1) ? tmp3 : tmp1;
34       barrier(CLK_LOCAL_MEM_FENCE); }
35     if (get_local_id(0) < 1) {
36       z[wg_id] = id(tmp3[l_id]);
37     }
38     barrier(CLK_GLOBAL_MEM_FENCE);
39   }
40 }
```

Listing 4.6: Compiler-generated OpenCL kernel for the dot product example shown in Listing 4.5

```
1  partialDotProduct(x: [float]_N, y: [float]_N) =
2    join ∘ mapWrg(0)(
3      join ∘
4      toGlobal(mapLcl(0)(mapSeq(id))) ∘
5      split(1) ∘
6      iterate(6)( join ∘
7                  mapLcl(0)( toLocal(mapSeq(id)) ∘
8                             reduceSeq(0, add) ) ∘
9                  split(2) ) ∘
10     join ∘ mapLcl(0)( toLocal(mapSeq(id)) ∘
11                       reduceSeq(0, multAndSumUp) ) ∘
12     split(2) ) ∘
13   split(128, zip(x, y))
```

Listing 4.5: LIFT implementation of a partial dot product

inated completely, which is the case in line 10 which corresponds to the *mapLcl* in line 10 in Listing 4.5.

Performing control flow simplification is beneficial in two ways: first, execution time is improved as additional jump instructions from the loop are avoided; and, secondly, in general fewer registers are required when loops are avoided.

Private Memory Arrays To ensure that arrays in private memory are allocated to registers on GPUs by the device compiler and not spilled into global memory, they are unrolled into variables instead of declaring them as OpenCL C arrays as seen in Listing 4.7.

This also means, that any `for` loops generated by computational patterns operating on such arrays need to be fully unrolled. If a `for` loop needs to be unrolled is determined by inspecting the input and output address spaces of the corresponding *map* or *reduce* pattern. If any of the address spaces is private memory, the pattern is marked to be unrolled during code generation. Since variable length arrays are not supported in OpenCL C, the length of private arrays always has to be a constant and the bounds of loops operating on those arrays are known, which means full unrolling is always possible.

To unroll the loop, its body is simply emitted the number of times it has iterations while keeping track of the current iteration counter when emitting the body. For all memory accesses the current iteration counter is substituted into the access index, possibly simplifying them even further. Accesses to the unrolled private arrays get the current constant index after substituting the current iteration appended to the array name to generate an access to the correct variable.

Vector Component Access When using vector types in OpenCL, it is possible to access their components. This is done by appending `.s` and the numeric index of the component

```
1  // array declaration and use
2  int arr[4];
3
4  for (int i = 0; i < 4; i++) {
5    /* ... */ = arr[i];
6  }
7
8  // unrolled array declaration and use
9  int arr_0, arr_1, arr_2, arr_3;
10
11 {
12   /* ... */ = arr_0;
13   /* ... */ = arr_1;
14   /* ... */ = arr_2;
15   /* ... */ = arr_3;
16 }
```

Listing 4.7: Private memory array declaration and use

```
1  float4 vec = /* ... */;
2  float sum = 0.0f;
3
4  sum += vec.s0; sum += vec.s1; sum += vec.s2; sum += vec.s3;
```

Listing 4.8: Vector component use when performing a vectorised reduction.

being accessed to the vector variable name. The numeric index must be within bounds of the vector type to be legal. This is useful for example when performing a vectorised reduction and then adding up the components as in Listing 4.8 and therefore important for hardware that has SIMD units and benefits from vectorised code.

As with our unrolled private memory, the components cannot be accessed by indexing into them using a loop iteration variable. This means that loops that access components of vectors have to be unrolled as well, as seen in line 4 of the example. Unrolling is performed much in the same way as described for private memory arrays. To detect if this needs to be done, the type of the memory object was allocated for and the type of the type the pattern is accessing are compared. If the original type was a vector type and it is now being accessed as a scalar type, the generated loop is marked to be unrolled.

4.3.6 Summary

This section described how LIFT IR is compiled to OpenCL. It used a number of examples to discuss how types and address spaces are inferred, memory is allocated, concise and efficient array accesses are generated, barriers are eliminated, and, finally, how the dot-product OpenCL kernel with simplified control flow shown in Listing 4.6 is generated.

4.4 Expressing Optimisations Structurally in LIFT

This section describes how different optimisations commonly used in OpenCL for GPUs can be expressed using the LIFT primitives. The different optimisations include parallelism mapping, vectorisation and the use of distinct address spaces provided by OpenCL and the hardware. The individual optimisations are investigated, along with how they are expressed functionally and the corresponding generated OpenCL code.

4.4.1 Mapping of Parallelism

In OpenCL, programmers have different choices on how to map the computation to the hardware, which directly affects performance. The programmer might decide to group threads (*work-items*) into work groups and use their associated *local ids* together with their *work-group ids* to distribute the work. Often it is also possible to use the *global ids* of work items independently of their work-group ids.

In LIFT, using the different layers of the parallelism hierarchy is expressed by using different low-level variations of the *map* pattern that were described in Section 2.4. All variations share the same high level semantics: applying a function to each element of the input array to produce the output array. The low-level variations differ in their OpenCL implementations, where the computation might be performed sequentially (*mapSeq*), or in parallel, distributing the workload across work groups (*mapWrg*), local work items (*mapLcl*) or global work items (*mapGlb*).

Figure 4.10 shows one possible mapping of parallelism for matrix multiplication. In Figure 4.10a, the `mapGlb(0)` primitive is used to perform a computation for every row of A. Nested inside is the `mapGlb(1)` primitive which maps over the columns of B. The used `mapGlb` primitives indicate, that a work item with the global ids `g_id_0` and `g_id_1` will process a combination of a row of A and a column of B.

Figure 4.10b shows the corresponding OpenCL code generated for this expression. The two for loops correspond to the map primitives. In the generic case it is unclear how many global work items will be launched at execution time, therefore, `for` loops are emitted and a single work item might process multiple data elements. For matrix multiplication (and many other applications) it is common to specialise the OpenCL kernel so that it only works if a matching global size is selected at execution time. To support this, array length information is used to statically prove that each work item executes the loop exactly once and avoid generating the loop altogether as described in Section 4.3. The resulting OpenCL code is shown in Figure 4.10c.

```
parallelismMapping(A: [[float]_M]_N, B: [[float]_K]_N]]) =
  mapGlb(0)(λ rowOfA .
   mapGlb(1)(λ colOfB . ... )(B) )(A)
```

(a) Functional expression using the `mapGlb` primitive.

```
1 kernel void KERNEL(...) {
2   for (int g_id_0 = get_global_id(0); g_id_0<N; g_id_0 += get_global_size(0)) {
3     for (int g_id_1 = get_global_id(1); g_id_1<N; g_id_1 += get_global_size(1)) {
4     ...
5     } } }
```

(b) Generated OpenCL code for an arbitrary global size.

```
1 kernel void KERNEL(...) {
2   int g_id_0 = get_global_id(0);
3   int g_id_1 = get_global_id(1);
4   ...
5 }
```

(c) Generated OpenCL code for fixed global size.

Figure 4.10: Exploiting parallelism using global work items.

4.4.2 Vectorisation

Vectorised Memory Operations Vectorising load and store instructions helps to better utilise the memory bandwidth by issuing larger memory transfers with a single instruction. For example, AMD suggests vectorising copying memory in a vectorised fashion in their example codes [AMDI 15]. The instructions might have specific requirements for alignment, such as requiring addresses to be aligned to a multiple of the access size. OpenCL provides specific `vload` and `vstore` built-in functions for loading or storing vector values from arrays of scalar values.

In LIFT, vectorised memory operations are decomposed into two parts, as shown in Figure 4.11a: first, interpreting the initially scalar array as a vectorised array using `asVector`; secondly, copying the data by applying the vectorised identity function `id4` to every element of the vectorised array. In the example `toPrivate` indicates a copy into the private memory. The length of arrays are kept track of in their types. Assuming that A in the example is an array of N `float` values. Therefore, its type is written as $[\texttt{float}]_N$. After applying `asVector(4)` to it, an array with type $[\texttt{float4}]_{N/4}$ is obtained. This length information is used when generating indices in OpenCL as was descirbed in section 4.3.3.

The generated OpenCL code is shown in Figure 4.11b. The `id4` function is declared in the first line and models a copy operation in the functional expression. It will be inlined and, therefore, optimised away by the OpenCL compiler. After vectorising the array its `float4` values are loaded using `vload4` built-in functions. As arrays in private memory are not necessary

```
vectorLoads(A: [float]_N) =
  ( ... ∘ toPrivate(mapSeq(id4)) ∘ asVector(4) )(A)
```

(a) Functional expression using the `asVector` primitive.

```
1 float4 id4(float4 x) { return x; }
2
3 kernel void KERNEL(const global float* A) {
4   ...
5   float4 elemsOfA_0 = id4(vload4(index_0, A));
6   float4 elemsOfA_1 = id4(vload4(index_1, A));
7   ...
8 }
```

(b) Generated OpenCL code using `vload` instructions.

Figure 4.11: Vectorised memory operations.

stored in registers, the array is unrolled into private variables. The first two variables are shown in lines 5 and 6. To unroll the array, its size has to be statically known, which is the case for arrays obtained through fixed size tiling. Symbolic computations are used to compute indices like $index_0$ using the length information stored in the array's type.

Vectorised Arithmetic Operations Vectorising arithmetic operations is one of the most important optimisations on Mali GPUs due to its SIMD architecture. The vectorisation of the dot-product computation will be discussed as an example, which is used as a building block in matrix multiplication as seen in Listing 5.3.

The dot product is represented functionally by combining two arrays using the `zip` primitive. It is followed by `map(mult)` which performs a pairwise multiplication before `reduce(0, add)` adds up all the intermediate results. Figure 4.12a shows a vectorised version of the dot product. The `vectorize(4, mult)` primitive is used to vectorise the multiplication with a vector width of 4. After performing the vectorised pairwise multiplication, all values are added up to compute the scalar result by first interpreting the vectorised data as scalar, and then by performing a reduction using scalar addition.

The generated OpenCL code is shown in Figure 4.12b. The vectorised function `mult4` performs the multiplication operation on two `float4` values. The `add` function in line 2 is not vectorised and operates on scalar `float` values. This example OpenCL code assumes that only two `float4` values are combined and multiplied producing a temporary `tmp` in line 8. The following two lines reduce the vector by accessing its individual components to produce the final result.

```
vectorDotProduct(A: [float]_N, B: [float]_N) =
  ... λ (elemsOfA: [float4]_I, elemsOfB: [float4]_I) .
    (... ∘ reduceSeq(0.0f, add) ∘
      asScalar ∘ mapSeq(vectorize(4, mult)))(
        zip(elemsOfA, elemsOfB))
```

(a) Functional expression performing a vectorised dot product.

```
1  float4 mult4(float4 l,float4 r){ return l*r;}
2  float  add(float l,float r){ return l+r;}
3
4  kernel void KERNEL(const global float* A,
5                     const global float* B) {
6   ...
7   float4 elemsOfA = /* ... */; float4 elemsOfB = /* ... */;
8   float4 tmp = mult4(elemsOfA, elemsOfB);
9   float acc = 0.0f;
10  acc = add(acc,tmp.s0); acc = add(acc,tmp.s1);
11  acc = add(acc,tmp.s2); acc = add(acc,tmp.s3);
12  ...
13 }
```

(b) Generated OpenCL code using vector arithmetic instructions.

Figure 4.12: Vectorised arithmetic operations.

```
1 reusePattern(x: float, y: [float]_N, ...) =
2   ... map(... x ...)(y) ...
```

Listing 4.9: Pattern for reuse. `x` is used for calculating every element of `map(...)(y)`.

4.4.3 Using Different Address Spaces

OpenCL provides several distinct memory regions typically (but not necessarily) corresponding to different physical memories on the device. Global memory accessible to all work-items and work-groups and is typically allocated to DRAM on a desktop GPU. Local memory is shared by a work-group and for example on NVIDIA GPUs, allocated to memory local to a streaming multiprocessor. Private memory is only accessible to a single work-item and is typically allocated to registers. This section discusses reasons for using these different address spaces and how their use is expressed in LIFT.

Data Reuse An important optimisation for GPUs is copying data that is reused to a faster memory space from global memory before using it. This reduces the number of accesses to the slow global memory and reduces the latency to access data. This can be data cooperatively copied to local memory that is used by threads in a group, or data copied to private memory that is used several times by a single thread, or a combination of both.

```
1 rewrittenReuse(x: float, y: [float]_N, ...) =
2   ( λ x_private .
3     ... map(... x_private ...)(y) ...
4   ) ∘ toPrivate(id(x))
```

Listing 4.10: Pattern for reuse. `x` is used for calculating every element of `map(...)(y)` and is copied to private memory beforehand.

```
1 matrixVector(A: [[float]_M]_N, x: [float]_M) =
2   map(λ a .
3     reduce(0.0f, add) ∘ map(mult) ∘ zip(a, x)
4   )(A)
```

Listing 4.11: Matrix-vector multiplication. The vector `x` is reused for calculating every element of the output vector.

An example of reuse occurring in a program can be seen in Listing 4.9. If `x` in line 1 is used inside the *map* in line 2 then it will be reused when calculating every element of the *map*'s output and is therefore a candidate to be copied to a faster address space. The best address space depends on the parallelism mapping that is chosen. The argument `x` could be a scalar value, a tuple or even an array. For arrays, different address spaces also impose restrictions on the number of elements that can reside there at any given time. It will be shown later in Section 5.5 how combinations of different patterns and rules are used to automatically reshape the computation to create smaller chunks of data that can fit in faster address spaces.

Listing 4.10 shows the same example, but this time `x` is copied to private memory assuming it's a scalar beforehand, to reduce the cost of loading it for calculating element of the output of the *map*.

Listing 4.11 shows the same basic reuse pattern occurring in matrix-vector multiplication. It can be seen that the input `x` is reused for calculating every element of the output vector y and is a candidate for copying to a faster memory space.

Changing Data Access Patterns GPU performance is very sensitive to the access patterns that are used to load data from global memory. In particular, most desktop GPUs prefer the data accesses to be coalesced, i. e. consecutive threads access consecutive memory locations. This enables the hardware to coalesce a number of different accesses into a single memory load. Sometimes it is necessary to use different access patterns, but it can be possible to use other address spaces to keep accesses to global memory coalesced.

In the example in Listing 4.12, the writes to global memory are coalesced, but the reads are not due to *gather* changing the order in which elements are read. It is possible to replace the *gather* with a *scatter* and move it to the other side of the expression to make the reads

```
1 reverseNotCoalesced(a: [float]_N) = (mapGlb(id) ∘ gather(reverse))(a)
```

Listing 4.12: Example of data access patterns. Writes are coalesced while the reads are not.

```
1 reverseCoalesced(a: [float]_N) =
2   (join ∘ mapWrg(
3     mapLcl(toGlobal(id)) ∘ gather(reverse) ∘ mapLcl(toLocal(id))
4   ) ∘ gather(reverse) ∘ split(64))(a)
```

Listing 4.13: Rewritten reverse, where all accesses to gloal memory are coalesced

coalesced, but then the writes will not, as scatter reorders the writes.

A better option, is to reshape the computation and process the data in smaller chunks, such that global accesses are still coalesced as in Listing 4.13. Accesses to global memory are now all coalesced as the 64 element chunk of the array is still accessed consecutively while the chunks themselves are being accessed in reverse order. These non-coalesced accesses are into local memory, where coalescing does not matter. A very similar transformation can be applied, for example, to matrix transposition except the piece of data copied to local memory will be a 2-dimensional array.

Temporary Results and Communication As described earlier, everything is by default allocated in global memory and this can lead to large intermediate results. If the result of a user function is consumed by the same thread, then the result should be allocated in private memory (`double(toPrivate(add)(x,y))` vs `double(add(x,y))`). If the result is too big for private memory or needs to be communicated to other threads in the group, local memory should be used, for example, when performing an iterative tree based reduction.

4.4.4 Summary

This section has discussed a number of GPU optimisations and how they are expressed in LIFT. It discussed how parallellism is mapped to the GPU thread hierarchy; how vectorisation of memory and arithmetic optimisations is performed; and using different address spaces for various purposes.

4.5 Experimental Setup

Two GPUs are used for the evaluation: an AMD Radeon R9 295X2 with AMD APP SDK 2.9.214.1 and driver 1598.5, as well as an Nvidia GTX Titan Black with CUDA 8.0.0 and driver 367.35. All experiments are performed using single-precision floating point values. The median runtime of 10 executions is reported for each kernel measured using the OpenCL

profiling API. Data transfer times are ignored as the focus is on the quality of the kernel code. For benchmarks with multiple kernels, the individual kernel runtimes are summed up.

4.6 Experimental Evaluation

This section evaluates the quality of the code generated by the LIFT compiler using 12 OpenCL hand-optimised kernels collected from various sources shown in Table 4.2. These represent GPU programs from different fields such as physics simulations (N-Body, MD), statistics and machine learning (KMeans, NN), imaging (MRI-Q), stencil (Convolution), and universally useful linear algebra primitives (ATAX, GEMV, GESUMMV, MM). The characteristics of the reference implementations are described in Table 4.2. Local and private memory denote their usage for storing data that is reused. The vectorisation of memory or compute operations is indicated as well as global memory coalescing. Iteration space shows the thread organisation dimensionality when running the kernel.

4.6.1 Code Size

Table 4.2 also shows the code size in lines of code for each benchmark. For LIFT we distinguish between the low-level LIFT which is the input for the LIFT compiler discussed in this chapter and the high-level LIFT discussed in Chapter 2.

The numbers show that writing high-performance OpenCL kernels is extremely challenging with 768 lines required for an optimised matrix multiplication kernel. The benchmarks in LIFT are up to 45× shorter, especially the portable high-level programs. The low-level LIFT programs are slightly longer as they encode optimisation choices explicitly.

4.6.2 Expressing OpenCL Optimisations in LIFT

The reference OpenCL implementations encode GPU specific optimisations. Each implementation is represented in LIFT by mimicking the optimisation and implementation choices of the OpenCL reference code. We are interested in testing the ability to represent differently optimised programs using the LIFT patterns presented in Section 2.4. This section gives a brief overview of different patterns of computation and communication are encoded.

The *N-Body* implementation from the NVIDIA SDK makes use of *local memory* to store particle locations accessed by multiple threads. In LIFT this is represented by copying the particle locations using *map(id)* nested inside the *toLocal* pattern. How the data is copied in the local memory is controlled by selecting one of the *mapSeq*, *mapLcl*, and *mapGlb* patterns. The AMD implementation does not use local memory but vectorises the operations expressed using a combination of *mapVec* and *asVector*.

Program	Source	Input Size (Small and Large)	Characteristics					Code size		
			Local memory	Private memory	Vectorisation	Coalescing	Iteration space	OpenCL	High-level LIFT	Low-level LIFT
N-Body, A	NVIDIA SDK	16K, 131K particles	✓	✓	✓	✓	1D	139	34	49
N-Body, B	AMD SDK	16K, 131K particles		✓	✓	✓	1D	54	34	34
MD	SHOC	12K, 74K particles		✓		✓	1D	50	34	34
K-Means	Rodinia	0.2M, 0.8M points				✓	1D	32	25	25
NN	Rodinia	8M, 34M points				✓	1D	18	7	7
MRI-Q	Parboil	32K, 262K pixels		✓		✓	1D	41	43	43
Convolution	NVIDIA SDK	$4K^2$, $8K^2$ images	✓			✓	2D	92	48	48
ATAX	CLBlast	$4K^2$, $8K^2$ matrices	✓			✓	1D	426	30	64
GEMV	CLBlast	$4K^2$, $8K^2$ matrices	✓			✓	1D	213	15	32
GESUMMV	CLBlast	$4K^2$, $8K^2$ matrices	✓				1D	426	30	64
MM	CLBlast, AMD	$1K^2$, $4K^2$ matrices		✓	✓	✓	2D	768	17	38
MM	CLBlast, NVIDIA	$1K^2$, $4K^2$ matrices	✓	✓	✓	✓	2D	768	17	65

Table 4.2: Overview, Characteristics, and Code size of the benchmarks

The *Convolution* benchmark applies *tiling* to improve performance by exploiting locality. Overlapping tiles, required by stencil applications, are created using the *slide* pattern. Two-dimensional tiles are achieved by a clever composition of *slide* with *map* and matrix transposition, which itself is expressed using *split*, *join*, and *gather*. These 2D tiles are then cooperatively copied into the local memory using the *toLocal(mapLcl(id))* pattern composition.

The CLBlast implementation of matrix-vector multiplication (SGEMV) carefully loads elements from the global memory using *coalesced memory accesses*. In LIFT the *gather* pattern is used to influence which thread loads which element from memory and by choosing the right permutation accesses to the global memory are coalesced.

The MM implementations from CLBlast applies slightly different optimisations for both GPUs. For NVIDIA CLBlast uses a combination of *tiling* in local memory, *register blocking*, and *vectorisation* of global and local memory operations. For AMD it also uses register blocking and vectorisation but not tiling in local memory. In LIFT, tiling and register blocking are represented by compositions of the *split* and *map* patterns together with a matrix transposition, which is itself expressed as combination of *split*, *scatter*/*gather* and *join* as seen in section 4.3.3. The LIFT vectorise patterns are used for vectorisation.

LIFT has proven to be powerful and flexible enough to represent this set of benchmarks and their versatile GPU optimisations. The next section investigates the performance obtained when generating OpenCL code from low-level LIFT programs.

4.6.3 Performance Evaluation

Figure 4.13 shows the relative performance of the LIFT generated code compared to the manually written OpenCL code on two GPUs. For each benchmark, the performance of the hand-written OpenCL implementation is compared with the performance of the generated kernel from the corresponding LIFT program. The different bars represent the performance obtained with different optimisations enabled and will be explained in the next section.

Concentrating on the right-most, dark red bar in each sub-plot, it can be seen that the code generator is able to achieve performance on-par with hand-written OpenCL kernels in most cases. This clearly demonstrates that the functional LIFT is able to express all the low-level details necessary to produce very efficient OpenCL code. The performance of the generated code is on average within 5% of the hand-written OpenCL implementation, which is quite a feat, considering how sensitive the underlying OpenCL compilers are.

4.6.4 Evaluation of Optimisation Impact

The differently colored bars in Figure 4.13 show the impact of code generator optimisations discussed in Section 4.3. As can be seen, applying none of the optimisations discussed in this chapter, leads to an average performance of only half the baseline. In extreme cases,

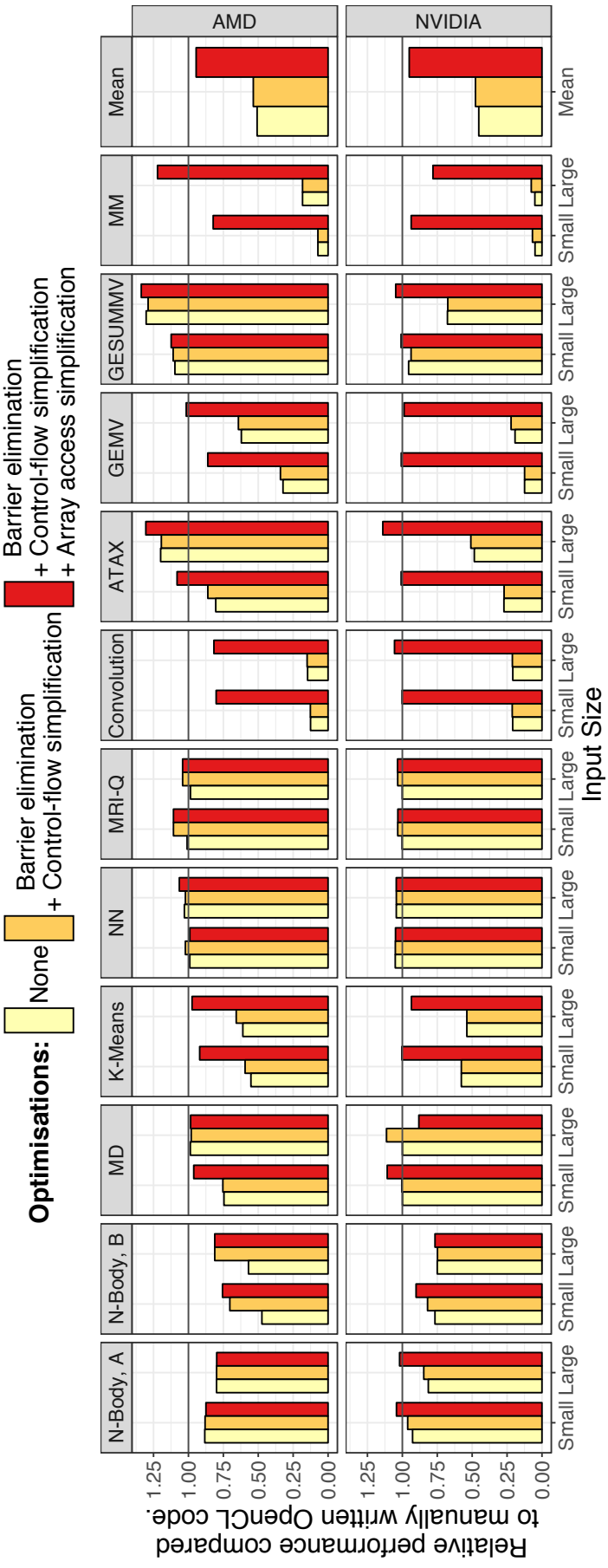


Figure 4.13: Speedup of generated code compared to OpenCL reference implementations.

such as matrix multiplication and convolution, the generated code can be as much as 10x or even 20x slower than the baseline. For convolution, for instance, this is due to the complexity of the memory accesses expressions resulting from using the *slide* primitive. However, as can be seen on the figure, the effect of array access simplification on performance is very impressive, demonstrating the importance of this optimisation. In addition, disabling array access simplification generally leads to larger kernel code, up to 7MB of source code in the case of matrix multiplication. There is one case where the array access simplification surprisingly makes the perfomance worse. The kernel code does not offer clues as to why this is the case, as the simplified case contains less aritmetic operations for calculating the array indices and the rest of the kernel is identical.

Surprisingly, the barrier elimination and control-flow simplification seems to have little effect on performance on both machines. The largest impact is for the AMD version of N-Body where the simplification of control plays an important role since this AMD implementation does not use local memory. The control simplification is able to produce a kernel with a single loop (the reduction) which corresponds to the human-written implementation. On the other hand, without the simplification of control-flow enabled, three loops are produced which results in a 20% slowdown.

4.7 Conclusion

This chapter has presented the compilation flow of LIFT, a functional data-parallel intermediate representation for OpenCL. It addresses the need of compiling a functional program with optimisations explicitly encoded into efficient imperative OpenCL code that actually runs on the devices identfied in Chapter 1 and described the code generation techniques that are the first contribution of this thesis.

By design, LIFT preserves high-level semantic information which can be exploited by the LIFT compiler to generate efficient OpenCL code. However, as seen in this chapter, generating efficient code is far from trivial and requires the careful application of optimisations such as array access simplification.

The experimental evaluation shown that the optimisations presented in this chapter have a significant impact on the performance of more complex applications with a performance improvement of over 20 times. Therefore, these optimisations are crucial to achieving high performance and producing code on par with hand-tuned OpenCL kernels. It follows that the automatic structural optimisation techniques based on rewriting presented in the next chapter could not achieve high performance without the optimisations for generating efficient OpenCL code presented in this chapter.

Chapter 5

Creating and Exploring the Optimisation Space with Rewrite Rules

5.1 Introduction

Producing high-performance GPU code is notoriously hard with low-level hardware features that are directly exposed to programmers, requiring expert knowledge to achieve high performance. The memory hierarchy needs to be managed explicitly and memory accesses have to be carefully handled to avoid memory bank conflicts and ensure coalescing. The code also explicitly controls the mapping of parallelism at multiple levels: work-groups, threads, warps, and vector units. Since each type of device comes with its own performance characteristics, requiring different optimisations, the resulting low-level device-tailored code is ultimately not performance portable. This problem is further exacerbated with mobile GPUs since optimisations beneficial for desktop GPUs (e. g. AMD, Nvidia GPUs) can negatively impact performance on mobile GPUs, as will be seen later in this chapter.

Auto-tuners have been proposed to address performance portability issues on GPUs. They are generally based on a specialised parametric implementation of a computational kernel, such as matrix multiplication, and the tuning process explores the performance space on the targeted hardware. However, auto-tuners have two major drawbacks. First, writing the parametric implementation for a given kernel requires non-negligible effort from the programmer. Secondly, and more importantly, the implementation is limited by a finite set of parameters which might not be good at expressing complex composition of optimisations. As already shown in Chapter 1, this can result in far from optimal performance when the parametric implementation is run on a device it was not originally designed for. In other words, auto-tuning alone is not sufficient to solve the performance portability problem.

In [Steu 15b], the authors propose to use a functional intermediate representation in the compiler and to express algorithmic and optimisation choices in a unified rule-rewriting sys-

tem. The functional representation provides an abstraction to reason about parallel programs at the algorithmic level. The rewrite rules define the optimisation space in a formal way, transforming the program seamlessly between different algorithmic forms and then into an low-level OpenCL-specific form. The previous chapter already showed how programs in the low-level OpenCL form are compiled into efficient imperative OpenCL code.

This chapter builds upon the work by [Steu 15b] to show how it is applied in practice and scaled to larger applications using matrix multiplication as a case study. Matrix multiplication is arguably one of the most studied applications in computer science. It is a fundamental building block of many scientific and high performance computing applications. Even though it has been studied extensively for many years, traditional compiler techniques still do not deliver performance portability automatically. Naïve implementations of matrix multiplication deliver very poor performance on GPUs; programmers are forced to manually apply advanced optimisations to achieve high performance (see Section 5.2). These optimisations are not portable across different GPUs, making manual optimisation costly and time-consuming.

This chapter combined with Chapter 4 presents a fully automated compilation technique which generates high performance GPU code for matrix multiplication for different GPUs from a single portable source programs. This approach achieves this by combining algorithmic and GPU specific optimisations to generate thousands of provably correct implementations. Using a pruning strategy, 50,000 OpenCL kernels implementing matrix multiplication are generated and run on GPUs from AMD and Nvidia. The best implementations found match or exceed the performance of several high-performance GPU libraries on all platforms.

Additionally, this chapter shows that an auto-tuner designed primarily for desktop-class GPUs is unable to achieve the full performance potential on mobile GPUs. As an example, using the auto-tuner with the ARM Mali GPU results in a 40% performance loss compared using a hand-tuned version written by an expert. In contrast, the rewrite-based approach delivers performance on par with the best hand-tuned version on each of the three platforms tested. This is possible due to the generic nature of the rewrite-based code generation technique, which allows encoding generic optimisations that are combined during the exploration process. This includes vectorisation and the use of built-in functions, which are highly beneficial for the Mali GPU.

This chapter makes the following key contributions:

- Demonstrates rewrite rules being used to optimise a complex application by expressing well-known optimisations as provably correct and composable *macro-rules* (sequences of rewrite rules);
- An automated technique for generating high-performance code from a single portable high-level representation of matrix multiplication;

```
1 kernel mm(global float* A, B, C, int N, K, M) {
2   int gid0 = global_id(0);
3   int gid1 = global_id(1);
4   float acc = 0.0f;
5   for (int i=0; i<K; i++)
6     acc += A[gid1*K+i]*B[i*M+gid0];
7   C[gid1*M+gid0] = acc;
8 }
```

Figure 5.1: Naïve OpenCL kernel for matrix multiplication.

- Experimental evidence that the rewrite based approach is performance portable and matches the performance of highly tuned CUDA and OpenCL implementations on different GPUs, succeeds where auto-tuners fail to deliver and even outperforms hand-tuned code on Mali.

The remainder of the chapter is structured as follows. Section 5.2 provides a motivation. Section 5.3 discusses optimisations for matrix multiplication and how they are represented functionally in LIFT. Section 5.4 presents rewrite rules that serve as building blocks and Section 5.5 describes how they are combined to perform more complex optimisations. Section 5.6 explains the automatic program space exploration strategy used. Section 5.7 and Section 5.8 present the experimental setup and results. Finally, Section 5.9 concludes the chapter.

Some of the rewrite rules described in Section 5.4 were pre-existing from [Steu 15b] (Listing 5.4, Listing 5.9, Listing 5.10 and Listing 5.19). The others are original contributions of the author. The macro rules and the automatic exploration strategy, described in Section 5.5 and Section 5.6 are the author's original contributions.

5.2 Motivation

This section illustrates the shortcomings of existing GPU compilers to produce high-performance code from easy to write naïve implementations as well as the shortcomings of complex auto-tuned implementations using matrix multiplication as an example. This results in a difficulty of writing high performing OpenCL programs requiring in-depth knowledge of various hardware characteristics.

The difficulty to achieve high performance motivates the need for new compilation techniques capable of automatically producing code close to manually optimised implementations from an easy to write high-level program.

Naïve Version Figure 5.1 shows the OpenCL kernel of a naïve matrix multiplication implementation using a 2D thread space. The rows of matrix A and the columns of matrix B are mapped to the first and second dimension of the iteration space using the thread indices `gid0`

```
1  kernel mm_amd_opt(global float* A, B, C, int K, M, N) {
2  local float tileA[512]; tileB[512];
3
4  private float acc_0;      ...; acc_31;
5  private float blockOfB_0; ...; blockOfB_3;
6  private float blockOfA_0; ...; blockOfA_7;
7
8  int lid0 = get_local_id(0); lid1 = get_local_id(1);
9  int wid0 = get_group_id(0); wid1 = get_group_id(1);
10
11 for (int w1 = wid1; w1 < M/64; w1 += get_num_groups(1)) {
12  for (int w0 = wid0; w0 < N/64; w0 += get_num_groups(0)) {
13
14   acc_0 = 0.0f; ...; acc_31 = 0.0f;
15   for (int i = 0; i < K/8; i++) {
16    vstore4(vload4(lid1*M/4+2*i*M+16*w1+lid0,A)
17            ,16*lid1+lid0, tileA);
18    vstore4(vload4(lid1*N/4+2*i*N+16*w0+lid0,B)
19            ,16*lid1+lid0, tileB);
20    barrier(CLK_LOCAL_MEM_FENCE);
21
22    for (int j = 0; j < 8; j++) {
23     blockOfA_0 = tileA[0+64*j+lid1*8];
24     // ... 6 more statements
25     blockOfA_7 = tileA[7+64*j+lid1*8];
26     blockOfB_0 = tileB[0 +64*j+lid0];
27     // ... 2 more statements
28     blockOfB_3 = tileB[48+64*j+lid0];
29
30     acc_0  += blockOfA_0 * blockOfB_0;
31     acc_1  += blockOfA_0 * blockOfB_1;
32     acc_2  += blockOfA_0 * blockOfB_2;
33     acc_3  += blockOfA_0 * blockOfB_3;
34     // ... 24 more statements
35     acc_28 += blockOfA_7 * blockOfB_0;
36     acc_29 += blockOfA_7 * blockOfB_1;
37     acc_30 += blockOfA_7 * blockOfB_2;
38     acc_31 += blockOfA_7 * blockOfB_3;
39    }
40    barrier(CLK_LOCAL_MEM_FENCE);
41   }
42
43   C[ 0+8*lid1*N+64*w0+64*w1*N+0*N+lid0] = acc_0;
44   C[16+8*lid1*N+64*w0+64*w1*N+0*N+lid0] = acc_1;
45   C[32+8*lid1*N+64*w0+64*w1*N+0*N+lid0] = acc_2;
46   C[48+8*lid1*N+64*w0+64*w1*N+0*N+lid0] = acc_3;
47   // ... 24 more statements
48   C[ 0+8*lid1*N+64*w0+64*w1*N+7*N+lid0] = acc_28;
49   C[16+8*lid1*N+64*w0+64*w1*N+7*N+lid0] = acc_29;
50   C[32+8*lid1*N+64*w0+64*w1*N+7*N+lid0] = acc_30;
51   C[48+8*lid1*N+64*w0+64*w1*N+7*N+lid0] = acc_31;
52 } } }
```

Figure 5.2: Hand-optimised OpenCL kernel for fast matrix multiplication on an AMD GPU.

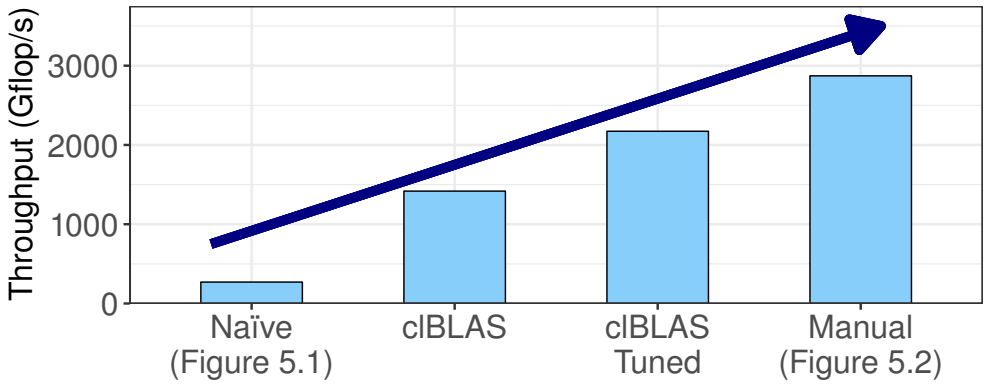


Figure 5.3: Performance comparison of matrix multiplication implementations on an AMD GPU.

and `gid1`. The for-loop performs the dot-product of a row of A and a column of B in 6. The final statement stores the result into matrix C.

While this version is easy to write, no existing compiler generates efficient code from it, despite many years of fruitful research on automatic compiler optimisations. Advanced optimisations like the usage of local memory, tiling, or register blocking are not applied automatically by traditional compilers. A lot of static analysis is needed to determine whether these optimisations are applicable. In contrast, more higher level information is available to the LIFT compiler.

Manually Optimised Version Figure 5.2 shows a manually optimised version of matrix multiplication tuned for an AMD GPU. This version performs a tiled matrix multiplication [Mats 12, McKe 69] using local memory. Register blocking [McKe 69] is used where each tile is further partitioned into smaller blocks stored in registers. Please notice that Figure 5.2 shows a shortened version omitting similar declarations (e. g., see line 4) and statements (e. g., see line 24). The original source code is 268 lines long.

The implementation in Figure 5.2 takes advantage of many hardware-specific features such as vectorised loads and local memory, which involves the use of synchronisation primitives. The parallelism is decomposed and mapped in a very specific way, taking advantage of the thread hierarchy and increasing registers usage using register blocking. In more detail, copying the tiles into local memory is performed in lines 16–20. Lines 23–25 and lines 26–28 perform register blocking for tile of A and B respectively. Lines 30–38 perform a partial dot-product between a block of tile A and B and accumulate temporary results in private memory. Once all the partial dot-products have been computed and accumulated, lines 43–51 store the final result of the dot-product into global memory.

Performance Comparison Figure 5.3 shows the performance comparison of the two versions of matrix multiplication shown in Figure 5.1 and Figure 5.2 together with two versions

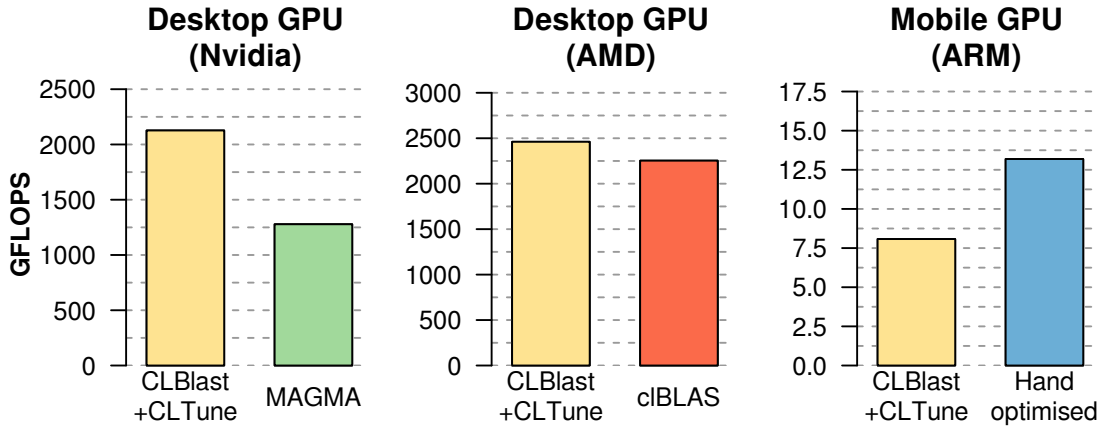


Figure 5.4: Performance comparison between auto-tuned (left bar) and hand-optimised (right bar) code. Higher is better.

from the AMD clBLAS library. The clBLAS library provides an expert written implementation of matrix multiplication. In addition, a tuning script is provided for automatically choosing implementation parameters for specific GPUs.

It can be seen in Figure 5.3 that the clBLAS library version performs 5× better than the naïve version, the tuned library version 8× better and the hand-optimised version even 10× better. It is obvious from this data – and maybe not very surprising – that current OpenCL compilers fail to automatically reach the performance of optimised libraries or hand-tuned kernels staring from a naïve version. Manual optimisations are still crucial in OpenCL to achieve high performance and it is often possible to beat highly optimised library implementations with manual optimisations and specialisations.

Ideally, programmers should write simple programs like the naïve version and automatically obtain the performance of the hand-tuned one.

Auto-Tuning Automatic tuning techniques have been applied quite successfully to matrix multiplication for over 20 years starting with PHiPAC [Bilm 97] and ATLAS [Whal 98b]. However, auto-tuners rely on a parametric implementation (or a parametric code generator) that is highly specialised to the target machine. This approach is well-suited in cases where little variation exists between different processing units but falls short when the target processing units exhibit significant variations. This section illustrates this problem using the CLBlast library auto-tuned using CLTune, a state-of-the-art auto-tuner which has been shown [Nugt 15] to achieve competitive performance on several GPUs.

Figure 5.4 shows the performance achieved by CLBlast on three different platforms; two desktop-class GPUs (Nvidia and AMD) and one mobile GPU (ARM Mali). For each platform, the performance of the auto-tuner is compared with the best open source reference imple-

mentation available at the time: MAGMA [Dong 14] on Nvidia, clBLAS on AMD and code written and optimised by ARM's engineers [Gron 14] on the Mali GPU. The auto-tuner is able to achieve significant performance on both desktop GPUs, clearly beating the hand-written MAGMA and slightly outperforming AMD's clBLAS.

However, the auto-tuner is unable to achieve the full performance potential on the mobile GPU resulting in a 40% performance loss. This shortfall is explained by the fact that CLBlast has been primarily designed for desktop-class GPUs and includes optimisations that are beneficial on these machines but detrimental on the Mali GPU. Examples of these include the use of local memory and coalesced accesses. The Mali GPU does not have a separate faster scratch-pad memory and local memory is mapped into global memory. This means that using local memory just introduces an unnecessary copy of the data in the same physical memory space without the added benefit of faster memory. Desktop GPUs coalesce the adjacent memory accesses of different threads together into fewer memory requests while the Mali GPU relies on caching to provide performance. Using the preferred data access pattern of a device is crucial for achieving high performance.

While it is conceptually not difficult to realise what needs to be done to reach a higher-level of performance for some specific machine, it is extremely hard to write a parametric kernel which exposes these choices as a finite set of parameters. Especially given that a library enabled for auto-tuning, such as CLBlast, is already even more complex than the hand-optimised kernel in Figure 5.3 with more than 1500 lines of parametric OpenCL code just for matrix multiplication.

Towards High-Performance Code from High-Level Programs This chapter argues that automatically producing high-performance code is possible if starting from a high-level functional program representation and keep it in the compiler pipeline for as long as possible. To achieve this, compiler optimisations are encoded as rewrite rules which transform the program into semantically equivalent optimised forms for different types of hardware. The rewrite rules express choices available to the compiler such as how parallelism is exploited, where data is stored, or if vectorisation is applied.

This design offers two main advantages: first, a functional representation ensures that high-level semantic information is available to the compiler, reducing the need for complicated static analysis; secondly, the transformations expressed by the rewrite rules are composable and provably correct, guaranteeing correctness of the generated specialised code. As will be seen, this design based on a functional representation of programs leads to a compiler that produces high-performance code like that shown in Figure 5.2 from a high-level program comparable to the one shown in Figure 5.1 and it also succeeds where the auto-tuner fails.

```
1 matrixMultiplication(A : [[float]_M]_K, B : [[float]_K]_N) =
2   map(λ rowOfA .
3     map(λ colOfB .
4       ( reduce(0.0f,add) ∘ map(mult) )( zip(rowOfA, colOfB) )
5     )( transpose(B) )
6   )(A)
```

Listing 5.1: Matrix multiplication expressed functionally. This is the input from which the LIFT compiler generates efficient OpenCL code targeted for the different GPUs.

5.3 Optimising Matrix Multiplication

This section discusses how matrix multiplication represented in the LIFT IR is optimised and transformed into forms exploiting GPU features explicitly. The basic implementation of matrix multiplication in LIFT is shown in Listing 5.1 and it corresponds to the illustration in Figure 5.5a where every element of C is computed as the dot product of the corresponding row from A and column from B.

5.3.1 Traditional Optimisations

Register Blocking Register blocking [McKe 69] is an optimisation technique for matrix multiplication where the idea is to swap nested loops such that a data item is loaded into a register and during the execution of the inner loop, this item is reused while iterating over a block of data from the other matrix. Figure 5.5b shows register blocking for matrix multiplication. Here each element of the highlighted column of B is reused n times while iterating over a single column of the highlighted block of A.

It can also be applied to both matrices as shown in Figure 5.5c. Now n elements from A and m elements from B can be copied to a faster address space to be reused.

Tiling Tiling is a common optimisation used on CPUs and GPUs [Cao 14, Mats 12, McKe 69] and is highly beneficial for the matrix-matrix multiplication use case application. The idea behind tiling is to increase data locality and fit small portions of data into a faster memory region. Then the computations are performed while the data is in the fast memory region.

In the context of matrix multiplication, the aim is to create 2D tiles for the output matrix C. The LIFT compiler achieves this by splitting each input matrix along both dimensions, so that they are decomposed into multiple *tiles*, which are copied to a faster memory space before being multiplied. Figure 5.5d visualises this situation. It is similar to 2 dimensional register blocking as a tile of the output is still calculated but the amount of data reused can now also be adjusted in another dimension with parameter k. The highlighted tiles of matrices A and B are copied to local memory, then multiplied and summed up to compute a tile of matrix C.

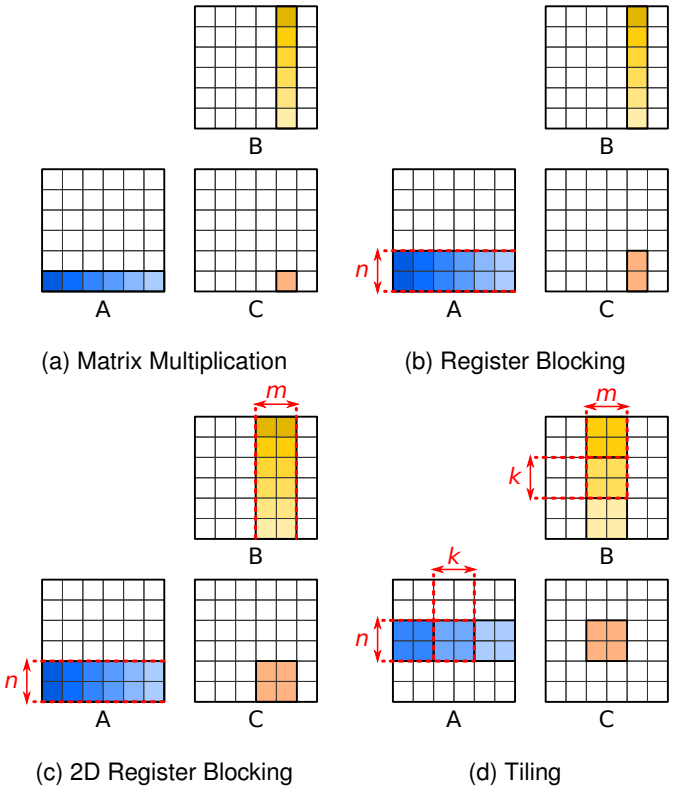


Figure 5.5: Matrix multiplication and examples classical optimisations for it.

Register blocking can be used when processing a tile, combining the two optimisations.

5.3.2 Manually Optimising Matrix Multiplication for Mali

As an example of implementing the tiling optimisaton, this section discusses optimising matrix multiplication for Mali. It first investigates a hand-optimised OpenCL kernel which implements the tiling optimisation described before. Then it shows that the functional representation is suitable for expressing the same optimisations structurally.

ARM has published a paper where they discuss optimisation techniques for their Mali GPU [Gron 14]. One of the applications investigated is the general matrix multiplication for which multiple optimised OpenCL kernels are presented. Listing 5.2 shows the best performing version developed by ARM's engineers [Gron 14]. To keep the discussion simple a slightly simpler version is shown, which concentrates on the actual matrix multiplication and omits the scalar values α and β used in the BLAS formulation of GEMM.

OpenCL kernel analysis The OpenCL kernel shown in Listing 5.2 applies **vectorisation** and **tiling** in private memory as its two main optimisations. The for loop in line 8 iterates over tiles (or *blocks*) comprising of 2 `float4` elements from matrix `A` and `B`. These elements

```
1  kernel void mm(global float4* const A,
2                 global float4* const B,
3                 global float2* C, uint n) {
4    uint i = get_global_id(0);
5    uint j = get_global_id(1);
6    uint nv4 = n >> 2;
7    float4 ab = (float4)(0.0f);
8    for (uint k = 0; k < nv4; ++k) {
9      float4 a0 = A[ 2*i   *nv4+k];
10     float4 a1 = A[(2*i+1)*nv4+k];
11     float4 b0 = B[ 2*j   *nv4+k];
12     float4 b1 = B[(2*j+1)*nv4+k];
13     ab += (float4)(dot(a0, b0), dot(a0, b1), dot(a1, b0), dot(a1, b1)); }
14   uint ix = 2*i*(n>>1) + j;
15   C[ix]           = ab.s01;
16   C[ix + (n>>1)] = ab.s23; }
```

Listing 5.2: Optimised OpenCL matrix multiplication kernel. This listing shows the `blockedNT` version from [Gron 14].

are loaded into private variables in lines 9–12. The dot products of all four combinations of `float4` elements from matrix `A` and `B` are computed using the OpenCL built-in `dot` function (lines 13 and 13) resulting in four separate intermediate results. These are combined into a single `float4` value (line 13) which is added to the accumulation variable `ab` (declared in line 7).

The vectorisation of the addition operation in line 13 is independent of the use of vector data types for the elements of matrix `A` and `B`. Instead, the tiling of 2 values from `A` and 2 values from `B` leads to 4 intermediate results which are added to the accumulation variable using a vector addition. After the loop, the results are written to global memory in two instructions (lines 15 and 16) using a vector width of 2.

Optimised matrix multiplication expressed functionally Listing 5.3 shows a functional expression in LIFT resembling the optimised implementation shown in Listing 5.2. Starting from the top, the tiling optimisation is expressed by splitting matrices `A` (line 23) and `B` (line 22) by a factor of 2. This groups 2 rows of `A` and 2 columns of `B` together. The `mapGlb` primitives used in lines 2 and 3 express the mapping of parallelism to global threads in OpenCL: every global thread processes a pair of *2 rows of* `A` and *2 columns of* `B`.

To complete the **tiling** of `A`, a block of 2 rows of `A` is first transposed (line 20), each row is split into chunks of 4 elements and then transposed back to obtain tiles with 2×4 `float` values. The same process is applied to `B` in line 21. The `zip` (line 20) combines the tiles of `A` and `B` together. These pairs of tiles are then processed by the `reduceSeq` in line 5 which corresponds to the for loop in the OpenCL kernel.

```
1  matrixMultiplication(A, B) =
2    mapGlb(0)(λ 2RowsOfA .
3      mapGlb(1)(λ 2ColsOfB .
4        toGlobal(mapSeq(id2)) ∘ asVector(2) ∘ asScalar ∘
5        reduceSeq(init = (float4)0.0f, λ (acc, (tileOfA, tileOfB)) .
6          (λ 2x2DotProducts .
7              mapSeq(add4(acc)) ∘
8              join ∘ asVector(4, 2x2DotProducts) ) ∘
9          (λ (tileOfA_p, tileOfB_p) .
10            mapSeq(λ rowOfTileOfA .
11             mapSeq(λ colOfTileOfB .
12                reduceSeq(0.0f, add) ∘
13                asScalar ∘
14                mapSeq(mult4 ,
15                zip(rowOfTileOfA, colOfTileOfB) ),
16              tileOfB_p),
17            tileOfA_p) ) ∘
18         ⟨ mapSeq(toPrivate(id4), asVector(4, tileOfA)),
19           mapSeq(toPrivate(id4), asVector(4, tileOfB))⟩ ,
20       zip( transpose ∘ split(4) ∘  transpose(2RowsOfA),
21            transpose ∘ split(4) ∘  transpose(2ColsOfB) ))
22     split(2, B)),
23   split(2, A))
```

Listing 5.3: Low-level functional expression resembling the OpenCL kernel presented in Listing 5.2.

When processing a single pair of a *tile of* `A` and a *tile of* `B` inside of the reduction, the pairs are copied into the private memory in lines 18–19. The `asVector(4)` primitive **vectorises the data** by turning 4 individual `float` values of a tile into a single `float4` value. This section corresponds to the lines 9–12 in Listing 5.2 where values from matrices `A` and `B` are loaded into private variables.

For each combination of a *row of a tile of* `A` and a *column of a tile of* `B`, each represented by a `float4` value, the dot product computation is performed in lines 15–17. The dot product is expressed as a combination of the `zip`, `mapSeq` and `reduceSeq` primitives. The `zip` (line 15) combines the two `float4` values from the tiles of `A` and `B`, before the `mapSeq(mult4)` (line 14) performs the **vectorised multiplication** of the two values. To finish the dot product computation, `reduceSeq(0.0f, add)` (line 17) adds up the multiplied values after they have been turned back into scalar values using the `asScalar` primitive (line 14). This section corresponds to the four occurrences of the `dot` function in line 13 in Listing 5.2.

To complete the reduction over multiple tiles, the computed intermediate result must be added to an accumulation variable. To achieve this, the computed 2×2 *dot products* are flattened into a one dimensional array using the `join` primitive (line 8). The resulting array of 4 `float` values is vectorised, using the `asVector(4)` primitve and added to the accumulation

$$\texttt{map}(f) \circ \texttt{map}(g) \Longrightarrow \texttt{map}(f \circ g)$$

Listing 5.4: Rule definition for fusing two *map* primitives

variable `acc` in line 7. This section corresponds to the **vectorised += operation** in Listing 5.2 (line 13).

Finally, to write the computed results back to the global memory the vector width is changed using `asScalar` and `asVector(2)` before the actual copy operation in line 4. This last section corresponds to the lines 15 and 16 from Listing 5.2.

This example should give some intuition on how optimised programs are expressed functionally. This representation enables the automatic transformation of the high-level program in Listing 5.1 into low-level expressions such as Listing 5.3 using rewrite rules, as the rest of the chapter shows.

5.3.3 Summary

This section first discussed traditional optimisations for matrix multiplication in general and then optimisations for the Mali GPU in particular, showing how it is implemented in OpenCL and in LIFT. The next section introduces the rewrite-rules that enable the LIFT compiler to automatically combine various optimisations and transform high-level programs into optimised functional low-level expressions from which OpenCL code is generated.

5.4 Rewrite Rules

A *rewrite rule* is a well-defined transformation of an expression represented in the LIFT IR (see also Chapter 2; section 2.4.3). Each rule encodes a simple – and provably correct – rewrite. Simple rules like the ones presented in this section, are the building blocks for more complex optimisations that are described in Section 5.5. As an example of a rule, the *map-fusion rule* in Listing 5.4 combines two successive *map* primitives into a single one.

In addition to the rules describing purely algorithmic transformations, there are also rules lowering the algorithmic primitives to OpenCL specific primitives. For example, the algorithmic *map* primitive can be mapped to any of the OpenCL specific *map* primitives, i. e., *mapWorkgroup*, *mapLocal*, or *mapSeq*, as long as the OpenCL thread hierarchy is respected.

Interesting interactions exist between the algorithmic and OpenCL specific rules. For example, the algorithmic *split-join* rule in Listing 5.5 transforms a *map* primitive following a divide-and-conquer style.

Here the *split(n)* primitive divides the input into chunks of size *n*, which are processed by the outer *map* and each single chunk is processed by the inner *map*. Finally, the *join* primitive

```
map(f) ⟹ join ∘ map(map(f)) ∘ split(n)
```

Listing 5.5: Divide-and-conquer style *split-join* rule for the *map* primitive.

```
join ∘ mapWrg(mapLcl(f)) ∘ split(n)
```

Listing 5.6: Nested expression mapped to the OpenCL thread hierarchy.

collects and appends all results. This rule transforms a flat one-dimensional *map* primitive into a nested expression which can easily be mapped to the OpenCL thread hierarchy, as seen in Listing 5.6.

This interaction allows the LIFT compiler to explore different strategies of mapping algorithmic expressions to the GPU hardware. In the example above, the parameter n directly controls the amount of work performed by the workgroups and local threads which is an important tuning factor.

In the following subsections, some more rewrite rules are described. They are required in order to be able to express rich optimisations such as described in Section 5.3 as sequences of rewrite rules, which are crucial for applications like matrix multiplication. These rules are applied to simplify the expression, to avoid unnecessary intermediate results, vectorise functions, or to ensure memory coalescing when accessing global GPU memory.

5.4.1 Fusion Rules

A fusion rule combining two *map* primitives was shown earlier in Listing 5.4. A similar rule to fuse a combination of *mapSeq* and *reduceSeq* also exists, as shown in Listing 5.7. This rule avoids an intermediate array produced by the *mapSeq(g)* primitive, as the function g is applied to all elements of the input array inside the reduction immediately before the element is combined with the reduction operator f.

In addition to fusing composed *map* and *reduce* primitives, rules to fuse two *map* primitives in a *zip* pattern are also defined, as shown in Listing 5.8. As the lengths of the arrays being *zip*ped have to be equal then they can be fused and moved out of the *zip*.

```
reduceSeq(z, f) ∘ mapSeq(g) ⟹ reduceSeq(z, λ(acc, x) . f(acc, g(x)))
```

Listing 5.7: Rule definition for fusing *mapSeq* and *reduceSeq* primitives.

```
zip(map(f, a), map(g, b)) ⟹
  map(λ x . (f(get(0, x)), g(get(1, x))), zip(a, b))
```

Listing 5.8: Rule definition for horizontally fusing two map primitives that are arguments to a *zip*.

```
map(g) ⟹ scatter(f^-1) ∘ map(g) ∘ gather(f)
```

Listing 5.9: Rule definition for changing the access patterns of a *map* primitive by reordering both, the input and output.

5.4.2 Memory Access Patterns

The *gather* and *scatter* primitives allow specifying an index function to reorder an array. It is important to point out, that this reordering is not performed in the generated code by producing a reordered array. Instead, the index computation required to perform the reordering is delayed until the next primitive accesses the input array. This is similar to lazy evaluation and was discussed more thoroughly in Chapter 4; section 4.3.3. Therefore, a *gather* or *scatter* primitive effectively controls how the following primitive will access its input or output array.

This design can be taken advantage of by applying the rewrite rule in Listing 5.9.

This rule rewrites an arbitrary *map* primitive to access its input array in a fashion dictated by the reordering function f. A common reordering function to use changes the accesses to be strided, enabling memory coalescing. To ensure correctness, the reordering has to be undone, by reordering the computed array with the inverse index function as used before. In situations where each thread processes multiple data elements, this transformation ensures that these elements are read and written in a coalesced way.

5.4.3 Vectorisation Rules

The basic rule in Listing 5.10 is applied to make use of the vector units in the hardware. It rewrites a *map* primitive into a vectorised version. For example, this rule can be applied to vectorise copying of a tile into local memory which is a technique advocated by AMD in their example OpenCL codes [AMDI 15].

The rewrite rule in Listing 5.11 describes the vectorisation of a *map* primitive following a *zip* of the concrete operation performed by the function f or the concrete vector width n. It is easy to see that this rule is correct, since the result of both expressions is an array of scalar values computed by applying the function f to pairs of elements from a and b.

```
map(f) ⟹ asScalar ∘ map( vectorize(n, f) ) ∘ asVector(n)
```

Listing 5.10: Rule definition for vectorising the operation performed by a *map* primitive.

```
  map(f, zip(a, b) )
⟹
  asScalar ∘ map(vectorize(n, f), zip(asVector(n, a), asVector(n, b)))
```

Listing 5.11: Rule definition for vectorising the operation of a *map* primitive that has several inputs combined by a *zip*.

```
 reduce(z, ⊕, a )
⟹
  reduce(z, ⊕) ∘ asScalar ∘
    reduce(asVector(n, z), vectorize(n, ⊕), asVector(n, a))
```

Listing 5.12: Rule definition for vectorising the operation of a *reduce* primitive.

Similarly, a rule for vectorising a reduction is defined in Listing 5.12. The rewritten expression performs a reduction on the vectorised data using the vectorised operator $\oplus$ before the final result is computed by a scalar reduction of the components of the vectorised result of the first reduction. For this rewrite to be correct, the reduction is required operator $\oplus$ to be commutative, as the order in which elements are processed is changed.

The matrix-multiplication version shown in Listing 5.2 uses the OpenCL built-in function *dot* to perform a dot product of two `float4` values and return the result as a scalar. This function can be implemented more efficiently by the OpenCL compiler, e. g. by using specialised hardware instructions. As will be shown in the evaluation, this is highly beneficial on Mali. A rule to detect a sequence of patterns computing a dot product and rewrite it into a function call of the `dot` built-in function is easily defined in Listing 5.13.

For this rule to fire x and y must be of type $[\texttt{float4}]_N$. Now, instead of applying the `mult4` function, the `dot` built-in is applied which also sums the elements. The additional `reduceSeq` after applying the `mapSeq(dot)` adds together the partial results computed by applying the `dot` primitive to the accumulation variable which is initialised with z. This shows how a very specialised optimisation can be implemented as a simple generic rewrite rule.

```
  reduceSeq(z, add) ∘ asScalar ∘ mapSeq(mult4, zip(x, y) )
⟹
   reduceSeq(z, add) ∘ mapSeq(dot, zip(x, y))
```

Listing 5.13: Rule definition for detecting computing a dot product and rewriting it to use the `dot` built-in.

```
kernel void KERNEL(global T* in,
    global T* out, int N) {
  for (int i = 0; i < N; i++) {
    out[i] = f(in[i]);
  }
}
```

(a) Code generated for a *map* before applying the *split-join* rule.

```
kernel void KERNEL(global T* in,
    global T* out, int N, int n) {
  for (int i = 0; i < N/n; i++) {
    for (int j = 0; j < n; j++) {
      out[j + i * n] = f(in[j + i * n]);
    } }
```

(b) Code generated for a *map* after applying the *split-join* rule.

```
kernel void KERNEL(global T* in,
    global T* out, int N) {
  for (int i = 0; i < N; i++) {
    out[0] = ⊕(out[0], in[i]);
  }
}
```

(c) Code generated for a *reduce* before applying the equivalent to the *split-join* rule.

```
kernel void KERNEL(global T* in,
    global T* out, int N, int n) {
  for (int i = 0; i < N/n; i++) {
    for (int j = 0; j < n; j++) {
      out[0] = ⊕(out[0], in[j + i * n]);
    } }
```

(d) Code generated for a *reduce* after applying the equivalent to the *split-join* rule.

Figure 5.6: Examples of generated OpenCL code showing the effect of the *split-join* rule.

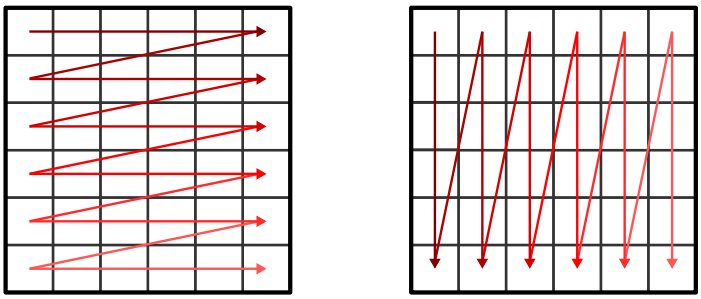
```
reduce(z, ⊕) ⟹
  reduceSeq(z, λ (acc, chunk) .
    head ∘ reduceSeq(acc, ⊕, chunk)
  ) ∘ split(n)
```

Listing 5.14: Rule definition for distributing a *reduce* primitive.

5.4.4 Split Reduce Rule

The split-join rule presented in Listing 5.5 for map has the effect on generated code as shown in the examples from Figure 5.6a and Figure 5.6b. Instead of a single loop, two are now generated. The outer loop is generated from the map that operates on chunks of the original array and the inner loop is generated from the map that works on elements of the chunk. To achieve the same effect for a reduce, the rule in Listing 5.14 is defined. The effect of the rule on the generated code is shown in Figure 5.6c and Figure 5.6d.

If the type of the input was $[\mathrm{T}]_N$, then now, after the *split(n)* it is $[[\mathrm{T}]_n]_{N/n}$. This means the outer reduce is working on chunks of the original array. It calls another reduce on the chunk and the current partial result in the accumulator. That means that in every iteration of the outer reduce, n elements are reduced into the accumulator. If the original array is $[x_1, x_2, \cdots, x_N]$, then after the *split(n)* it is $[[x_1, x_2, \cdots, x_n], [x_{n+1}, \cdots], [\cdots, x_N]]$. After the first iteration of the outer *reduceSeq*, the accumulator will contain the sum of the first chunk, $z \oplus x_1 \oplus x_2 \oplus \cdots \oplus x_n$, and this will be the initial accumulator value for the second iteration. After the second iteration the



(a) Regular (b) Transposed

Figure 5.7: Multi-dimensional array access patterns

accumulator will contain the sum of the first two chunks, $z \oplus x_1 \oplus x_2 \oplus \cdots \oplus x_n \oplus x_{n+1} \oplus \cdots \oplus x_{2n}$. Finally, after the outer reduce has finished the final result will be $z \oplus x_1 \oplus x_2 \oplus \cdots \oplus x_n \oplus x_{n+1} \oplus \cdots \oplus x_N$ which is the same as the result of the left hand side before rewriting.

Since the type of chunk and z are different for the outer reduce ($[\mathtt{T}]_n$ and $\mathtt{T}$) then the reduction operator is not associative and, therefore, it can no longer be a *reduce* but has to be a *reduceSeq*. The customising function of both *reduce* and *reduceSeq* has to return the same type as the accumulator. The *head* in the customising function is an artefact of *reduce* returning a single element array $[\mathtt{T}]_1$ and is required to access the single element of the result array and make the right-hand of the rule well typed.

5.4.5 Interchange Rules

An important building block for optimisations such as tiling, is the ability to change the order in which different dimensions of arrays are accessed. Multi-dimensional arrays in LIFT are stored in a row-major form by default, so they are accessed row by row as shown in Figure 5.7a for a 2-dimensional array. The same access pattern is used by both, the reads and writes, when applying a function f to every element of a 2-dimensional array using nested *maps* ($map(map(f))$). The rest of this section introduces rules for changing the order of accesses for a number different cases. The order memory is accessed is important, as it has a big impact on performance. On desktop GPUs, we are interested in rearranging memory accesses to achieve coalescing.

Mapping over a multi-dimensional array The rule in Listing 5.15 changes the access pattern when mapping over a multi-dimensional array. Assuming the input type is $[[\mathtt{T}]_M]_N$, where $\mathtt{T}$ could be any type, and N and M specify the number of rows (N) and columns (M). The right hand side of the rule first transposes the matrix, so the type is $[[\mathtt{T}]_N]_M$, then applies f to

```
map(map(f)) ⟹ transpose ∘ map(map(f)) ∘ transpose
```

Listing 5.15: Rule definition for interchanging the order in which nested *map* primitives operate.

```
kernel void KERNEL(global T* in, global
    T* out, int M, int N) {
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < M; ++j) {
      out[j + M * i] = f(in[j + M * i]);
    } } }
```

(a) Before

```
kernel void KERNEL(global T* in, global
    T* out, int M, int N) {
  for (int j = 0; j < M; ++j) {
    for (int i = 0; i < N; ++i) {
      out[j + M * i] = f(in[j + M * i]);
    } } }
```

(b) After

Figure 5.8: Examples of generated OpenCL code demonstrating the effect of the `map(map(f))` interchange rule.

every element of the matrix and then transposes the resulting matrix, restoring the original type $[[\mathtt{T}]_M]_N$. In both cases f is applied to every element and type input and output types are the same.

As transpose is defined in terms of patterns that do not actually change the data in memory but just the access patterns, then the expression on the right hand side accesses the input with the pattern shown in Figure 5.7b. The outer map now works over the columns and the inner over the rows. The effect of applying the rule on the generated OpenCL code is swapping the two for loops that are generated from the 2 maps as seen in Figure 5.8. Note that the indices for accessing the input and output stay the same, indicating that only the traversal order has changed.

Mapping over separate arrays The rule in Listing 5.16 is for the case where the two maps don't operate on a single matrix but two separate arrays. Assuming the type of `A` is $[\mathtt{T}_1]_N$ and the type of `B` is $[\mathtt{T}_2]_M$, since the output of the map has the same length as the input, then the type of output of the left hand side of the rule is $[[\mathtt{T}_3]_M]_N$. If the maps are interchanged such that the outer one works on `B` and inner one on `A`, the output type will be $[[\mathtt{T}_3]_N]_M$. To restore the original type, the output needs to be transposed. The effect on the generated OpenCL code is the same as before, swapping the two generated for loops and accessing the output in a column by column fashion as shown in Figure 5.7b.

Mapping over a multi-dimensional in the presence of *zip* Listing 5.17 shows an example of a rule for the case where the argument of the inner map is a zip. Since the zip moves out, the arguments to f need to be replaced appropriately. The access pattern to `A` and the output changes as also described previously while the one to `x` stays the same. A very similar rule is

```
 λ (A: [T1]N, B: [T2]M) .
   (λ r: [[T3]M]N . r) ∘ map(λ a . map(λ b . f(a,b), B), A)
⟹
 λ (A: [T1]N, B: [T2]M) .
   (λ r: [[T3]M]N . r) ∘ transpose ∘
   (λ r: [[T3]N]M . r) ∘ map(λ b . map(λ a . f(a,b), A), B)
```

Listing 5.16: Rule definition for interchanging two *map* primitives when they do not operate on a single matrix.

```
 λ (A: [[T1]M]N, x: [T2]M) .
   (λ r: [[T3]M]N . r) ∘ map(λ row: [T1]M .
     map(λ elemElem: (T1, T2) .
       f(get(0, elemElem), get(1, elemElem)),
     zip(row,x)),
   A)
⟹
 λ (A: [[T1]M]N, x: [T2]M) .
   (λ r: [[T3]M]N . r) ∘ transpose ∘
   (λ r: [[T3]N]M . r) ∘ map(λ colElem: ([T1]N, T2) .
       map(λ elem: T1 . f(elem, get(1, colElem)), get(0, colElem)),
     zip(transpose(A), x))
```

Listing 5.17: Rule definition for interchanging two *map* primitives when the row of the matrix accessed is *zip*ped with another array.

defined for the case where the zip originally appears as an argument to the outer map.

Mapping a reduce over a multi-dimensional array Listing 5.18 shows the rule for interchanging a *reduce* that is nested inside a *map*. The left hand side has the effect of one by one summing all the rows of a matrix and producing a vector of the sums. It is also possible to do the reduction such that at each iteration of the reduction a whole column is processed and every element added to the partial sum for that row, which is the case in the right hand side of the rule. Firstly, as in this case the *reduce* directly produces an array the initial value for the accumulator needs to be changed as well. To do this, a new literal array is created where every element is the same as the original initial value ($[z, z \cdots z]$). Secondly, to access a column of `A` in every iteration of the reduction, it is transposed before being passed as an argument. Now, when performing the reduce, `col` and `acc` are zipped to match every element of the column with the corresponding partial sum. The result of the zip is then mapped over and the element is added to the partial sum. The final result after the reduce needs to be transposed again to restore the same type as before applying the rule ($[[T]_1]_N$).

```
λ (A: [[T]M]N) .
  (λ r: [[T]1]N . r) ∘ map(λ row: [T]M .
    reduce(z, λ (elem: T, acc: T) . ⊕(elem, acc))(row) )(A)
⟹
λ (A: [[T]M]N) .
  (λ r: [[T]1]N . r) ∘ transpose ∘
  (λ r: [[T]N]1 . r) ∘ reduce([z,z⋯z], λ (col: [T]N, acc: [T]N) .
    map(λ elemAcc: (T, T) .
      ⊕(get(0, elemAcc), get(1, elemAcc)),
    zip(col, accArr)),
  transpose(A) )
```

Listing 5.18: Rule definition for interchanging a *map* and a *reduce* primitive.

Assume the input is $[[x_{1,1},x_{1,2},\cdots,x_{1,M}],[x_{2,1},x_{2,2},\cdots,x_{2,M}],\cdots,[x_{N,1},x_{N,2},\cdots,x_{N,M}]]$. Then on the left hand side, the first iteration of the map will produce $[x_{1,1}\oplus x_{1,2}\oplus\cdots\oplus x_{1,M}]$, the second $[x_{2,1}\oplus x_{2,2}\oplus\cdots\oplus x_{2,M}]$, with the final result as $[[x_{1,1}\oplus x_{1,2}\oplus\cdots\oplus x_{1,M}],[x_{2,1}\oplus x_{2,2}\oplus\cdots\oplus x_{2,M}],\cdots,[x_{N,1}\oplus x_{N,2}\oplus\cdots\oplus x_{N,M}]]$. On the right hand side, the input is processed column by column. After accumulator the first iteration of the reduce $[[x_{1,1}],[x_{2,1}],\cdots,[x_{N,1}]]$, after the second $[[x_{1,1}\oplus x_{1,2}],[x_{2,1}\oplus x_{2,2}],\cdots,[x_{N,1}\oplus x_{N,2}]]$ until finally computing the same final result as before.

5.4.6 Simplification Rules

Applying rewrite rules can leave a large number of superfluous data layout patterns in the program, which unnecessarily complicate the program and also hinder applying fusion rules.

Listing 5.19 contains some examples of simplification rules that remove data layout patterns which undo each other. As a result of the patterns undoing each other, these rules have no effect on the generated OpenCL code. Their removal is still important to enable the application of other rewrite rules, such as fusion rules. In the notation used in this chapter, some restrictions that ensure that applying the rules is legal are not shown. For example, for the second case of the `split-join` rule, the array that is being `join`ed needs to have n elements in the inner dimension. Similarly, for the second case of the `asVector-asScalar` rule, the vector length of the input must be n. Lastly, the rules involving `gather` and `scatter` are limited to reordering functions f where it is known the reordering is being undone.

Since *transpose* is defined in terms of *split*, *join* and *gather*/*scatter* the rule in Listing 5.20 is defined as a shortcut. This an example of a macro rule, which are described more thoroughly in the next section.

Applying the rules presented previously can leave behind other patterns that have no computational effect. For example, the rule in Listing 5.21 removes a *map* pattern that does not

```
join ∘ split(n) ⟹ ε
split(n) ∘ join ⟹ ε

asScalar ∘ asVector(n) ⟹ ε
asVector(n) ∘ asScalar ⟹ ε

gather(f⁻¹) ∘ scatter(f) ⟹ ε
scatter(f⁻¹) ∘ gather(f) ⟹ ε
```

Listing 5.19: Definitions of some simplification rules removing superfluous data layout patterns.

```
transpose ∘ transpose ⟹ ε
```

Listing 5.20: Rule definition for removing two *transpose*s that undo each other.

perform any computation.

It is also possible to encode more complex transformation as rewrite rules, although they border on abusing the rewrite rule system. For example, for simplifying unnecessarily complex *zip* constructs that can arise as a result of interchange and fusion transformations. These rules differ from other rewrite rules, as changes need to be propagated to the part of the program that consumes the output of the *zip*. For example, when replacing `zip(a, zip(b, c))` with `zip(a, b, c)` to access an element of `c`, `get(1, get(1, ...))` needs to be replaced with `get(2, ...)`). As a result, the implementation limits them to a subset where the `get` calls can be substituted appropriately.

5.4.7 Enabling Rules

Sometimes a rule is not applicable but other rules can be applied to make it possible to apply the rule. These rules are called enabling rules and they are useful for simplifications as well as interchange rules.

For example, interchanging the two *map*s in the following expression, `map(join ∘ map(`f`) ∘ split(`n`))`, is not possible, as the rule does not apply. To be able to transform the expression so that the rule is applicable, the *map* fission rule is defined and described in Listing 5.22. This makes it possible to rewrite the expression to `map(join)∘map(map(`f`))∘map(split(`n`))`, so that the interchange rule is now applicable.

```
map(λ x . x) ⟹ ε
```

Listing 5.21: Rule definition for removing a *map* that only returns its input and therefore has no effect.

$$\texttt{map}(f \circ g) \implies \texttt{map}(f) \circ \texttt{map}(g)$$

Listing 5.22: Rule definition for fissioning a single *map* primitive into two to make other rules applicable.

For simplifications, an example of an enabling rule is the split-join rule. For example, in the expression $\texttt{split}(n) \circ \texttt{map}(f) \circ \texttt{join}$, the split-join rule can be applied, followed by the appropriate simplification rule twice to get $\texttt{map}(\texttt{map}(f))$. When the split-join rule is used in this fashion f contains no user functions and only other data-layout changing pattens.

5.4.8 Implementation

As the LIFT language and compiler are implemented in Scala (see Chapter 2), then implementation of rewrite rules relies on features such as pattern matching and partial functions. This allows expressing the rules in a form quite similar to the notation used above. The rules pattern match on the IR representation of LIFT, operate on `Expr` nodes and return new `Expr` nodes, so have the Scala type `PartialFunction[Expr, Expr]`. Using partial functions allows querying whether a rule is applicable at a given `Expr` node using its `isDefinedAt(x: Expr): Boolean` member function.

When primitives require a value (e. g. split factor) to be specified then it is introduced to the expression as a symbolic variable. Not committing to a particular value at this stage makes it possible to explore different values that lead to different performance later in the exploration.

Pattern matching also supports adding additional constraints for a case to match the pattern. This can be done by adding an `if` condition containing any code to check those constraints after the pattern. This mechanism is used to check, for example, that the OpenCL thread hierarchy is respected, that simplifications are valid based on array lengths or vector widths or that the type that a rule is attempting to vectorise can in fact be vectorised.

After applying a rule, the program surrounding the `Expr` node where it was applied needs to be rebuilt as the structure of the IR is immutable and the replacement cannot be performed in place. As the goal is to create several versions of the same program then doing replacement in place would also destroy the original program preventing the application of different rules to it.

Rebuilding is performed by traversing the whole program, keeping track of the current node, the node to replace and the node to perform the replacement with. If the current node is the one to replace, the new node is returned. Otherwise, if the current node is a function call, the replacement is first attempted in the arguments. If the call is to a function which has child nodes, child nodes are visited next. If the replacement occurred in the arguments or in the child node, a new `FunCall` node is instantiated with the new arguments or body. All *map* and

reduce nodes have a helper member function to create a new node of the same type but with a new body to help perform the replacement in the child nodes. If no replacement occurred, the existing node is returned.

5.5 Macro Rules and Encoding Optimisations

By design, each rewrite rule encodes a simple transformation. As discussed in the previous section, more complex optimisations are achieved by composition.

To guide the automatic rewrite process rewrite rules are grouped together into *macro rules* which encode sequences of rules to apply, as it can require tens or hundreds of rule applications to perform an optimisation. The space created by applying that many rules is huge and defining macro rules enables enables cutting the optimisation space to a much smaller size without excluding interesting optimised implementations.

A macro rule aims to achieve a particular optimisation goal, such as apply tiling or blocking or a smaller steps of those optimisations. These macro rules are also more flexible than the simple rules as try to apply different sequences of rewrites to achieve their optimisation goal, whereas a simple rewrite rule always performs exactly the same transformation. For example, it might be required to first rewrite the source expression into a form where the rewrites performing the actual optimisation (e. g., tiling) can be applied, as described in section 5.4.7.

5.5.1 Map Interchange Macro Rule

An example of a macro rule performing a single step of an optimisation is one for performing an interchange of two *map* patterns. As seen in section 5.4.5, there are a number of different rules that perform the same action but in slightly different circumstances. Therefore, a macro rule is defined that chooses the correct one to apply depending on which one of them applies. Additionally, it will automatically apply the fissioning rule to make the rule applicable if necessary.

In the two examples below, none of the interchange rules apply immediately but both of them can be rewritten into a form where one of them is applicable by applying the fission rule. The macro rule will determine the position of the nested map primitive and apply the fission rule appropriately. In the first example, the rule needs to be applied twice, and in the second only once.

```
map(join() ∘ map(f) ∘ split(n))

map(λ a . join() ∘ map(f) ∘ zip(a, ⋯))
```

The examples after applying the split-join rule where an interchange rule can now be applied are below. After applying the appropriate fission rules, the macro rule will pick the

$$\begin{bmatrix} [a, & b, & c, & d, & e, & f], \\ [g, & h, & i, & j, & k, & l], \\ [m, & n, & o, & p, & q, & r], \\ [s, & t, & u, & v, & w, & x] \end{bmatrix}$$

(a) $[[\mathrm{T}]_6]_4$

$$\begin{bmatrix} \begin{bmatrix} [a, & b, & c, & d, & e, & f], \\ [g, & h, & i, & j, & k, & l] \end{bmatrix}, \\ \begin{bmatrix} [m, & n, & o, & p, & q, & r], \\ [s, & t, & u, & v, & w, & x] \end{bmatrix} \end{bmatrix}$$

(b) $[[[\mathrm{T}]_6]_2]_2$

$$\begin{bmatrix} \begin{bmatrix} [[a, & b], [c, & d], [e, & f]], \\ [[g, & h], [i, & j], [k, & l]] \end{bmatrix}, \\ \begin{bmatrix} [[m, & n], [o, & p], [q, & r]], \\ [[s, & t], [u, & v], [w, & x]] \end{bmatrix} \end{bmatrix}$$

(c) $[[[[\mathrm{T}]_2]_3]_2]_2$

$$\begin{bmatrix} \begin{bmatrix} \begin{bmatrix} [a, & b], \\ [g, & h] \end{bmatrix}, \begin{bmatrix} [c, & d], \\ [i, & g] \end{bmatrix}, \begin{bmatrix} [e, & f], \\ [k, & l] \end{bmatrix} \end{bmatrix}, \\ \begin{bmatrix} \begin{bmatrix} [m, & n], \\ [s, & t] \end{bmatrix}, \begin{bmatrix} [o, & p], \\ [u, & v] \end{bmatrix}, \begin{bmatrix} [q, & r], \\ [w, & x] \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

(d) $[[[[\mathrm{T}]_2]_2]_3]_2$

Figure 5.9: Creating 2×2 tiles in a 4×6 matrix. The the data structure and its type after each rule application is shown.

appropriate interchange rule to apply. For the top one, the basic one is chosen, and for the bottom one, the zip one is chosen.

```
map(join()) ∘ map(map(f)) ∘ map(split(n))

map(join()) ∘ map(λ a . map(f) ∘ zip(a, ⋯))
```

Another macro rule is defined to perform this interchange transformation for when a reduce is involved. As with the one described, it will apply fission rules and then choose the appropriate reduce interchange rule to apply.

5.5.2 Basic Tiling

Tiling a matrix transposition or any other computation in the form *map(map(f))* is achieved by applying split-join macro rule twice and the interchange macro rule once. The sequence of rules for tiling transformation and their effect when applied on a program is shown in Figure 5.10. The corresponding input matrix for every step is shown in Figure 5.9.

To walk through the sequence, consider the example program in Figure 5.10a. Its the input is a matrix of type $[[\mathrm{T}]_6]_4$ and is shown in Figure 5.9a.

The tiling starts by applying the split-join rule to the outer dimension to get the program in Figure 5.10b. The input matrix to the nested *map(map(map(f)))* now has the type $[[[\mathrm{T}]_6]_2]_2$ and is shown in Figure 5.9b.

This followed by another split-join on the inner dimension to get the program in Fig-

```
1 tilingExample(A: [[T]6]4) = map(map(f), A)
```

(a) Starting program. The input to the nested maps has type $[[\mathrm{T}]_6]_4$.

↓ Apply split-join

```
1 tilingExample(A: [[T]6]4) =
2   join ∘
3   map(map(map(f))) ∘
4   split(2, A)
```

(b) The input to the nested maps has type $[[[\mathrm{T}]_6]_2]_2$.

↓ Apply split-join

```
1 tilingExample(A: [[T]6]4) =
2   join ∘ map(map(join)) ∘
3   map(map(map(map(f)))) ∘
4   map(map(split(2))) ∘ split(2, A)
```

(c) The input to the nested maps has type $[[[[\mathrm{T}]_2]_3]_2]_2$.

↓ Apply interchange

```
1 tilingExample(A: [[T]6]4) =
2   join ∘ map(map(join)) ∘ map(transpose) ∘
3   map(map(map(map(f)))) ∘
4   map(transpose) ∘ map(map(split(2))) ∘ split(2, A)
```

(d) Final tiled program. The input to the nested maps has type $[[[[\mathrm{T}]_2]_2]_3]_2$.

Figure 5.10: Creating 2×2 tiles when mapping over a 4×6 matrix. The LIFT program after each rule application is shown.

```
tile(n, k, A) =
  map(transpose) ∘ map(map(split(k)), split(n, A))
```

Listing 5.23: Using a combination of *split*, *transpose* and *map* to tile a matrix.

ure 5.10c. The input matrix to the nested *map(map(map(map(f))))* now has the type $[[[[\mathtt{T}]_2]_3]_2]_2$ and is shown in Figure 5.9c.

Finally, the interchange rule is applied on the two middle dimensions to get the program in Figure 5.10d. The input matrix to the nested *map(map(map(map(f))))* now has the type $[[[[\mathtt{T}]_2]_2]_3]_2$ and is shown in Figure 5.9d.

The innermost two dimensions of the resulting four dimensional array, with the type $[[\mathtt{T}]_2]_2$, now form the set of 2×2 tiles. This sequence of rules is one of the macro rules that encodes a larger optimisation.

5.5.3 Optimising Matrix Multiplication with Macro Rules

Like the tiling rule presented in section 5.5.2, applying the tiling and blocking optimisations to matrix multiplication consists of applying split-join and interchange rules in the appropriate order. Turning a textbook matrix multiplication into an optimised version using macro rules is shown in Figure 5.11.

Building the tiles Listing 5.23 shows *map* and *split* and the high-level function *transpose* presented earlier used to produce a tiled representation of matrix A (or B). The first *split(n)* divides A into chunks of n rows. The second *split(k)* divides the columns of A into chunks of k columns each. Finally, the *transpose* reorganises the created chunks into 2D tiles of size $n \times k$. The reuse of unmodified primitives illustrates the power of composition and shows that larger building block can be build on top of a very small set of primitives. This makes the design of the compiler easier since the compiler only need to handle a very small set of primitives and does not need to know about higher-level building blocks such as *transpose* or *tile*.

Combining everything As a result of applying split-join and interchange rules, these basic principles are applied to both matrices and *zip* is used to combine them, the second expression shown in Figure 5.11 is obtained. The *tile* function is used twice to tile both matrices in the last two lines. Line 11 combines a row of tiles of matrix A with a column of tiles of matrix B using the *zip* primitive. The computation of dot-product remains unchanged (line 8) and is nested in two *map* primitives, now operating on pairs of tiles instead on entire matrices. To compute matrix multiplication in a tiled fashion, the intermediate results computed by multiplying the pairs of tiles have to be added up. This is done using the *reduce* primitive introduced in line 4 combined with the $+$ operation used in line 5 to add up two tiles.

Naïve matrix multiplication

```
matrixMultiplication(A : [[float]_M]_K, B : [[float]_K]_N) =
map(λ arow .
  map(λ bcol .
    reduce(0, +) ∘ map(×) ∘ zip(arow, bcol)
  , transpose(B))
, A)
```

↓ Apply tiling macro rule

```
matrixMultiplication(A : [[float]_M]_K, B : [[float]_K]_N) =
untile ∘ map(λ rowOfTilesA .
 map(λ colOfTilesB .
  reduce(0, λ (tileAcc, (tileA, tileB)) .
   map(map(+)) ∘ zip(tileAcc) ∘
   map(λ as .
    map(λ bs .
     reduce(0, +) ∘ map(×) ∘ zip(as, bs)
    , tileB)
   , tileA)
  zip(rowOfTilesA, colOfTilesB))
 ) ∘ tile(m, k, transpose(B))
) ∘ tile(n, k, A)
```

↓ Apply blocking macro rule

```
matrixMultiplication(A : [[float]_M]_K, B : [[float]_K]_N) =
untile ∘ map(λ rowOfTilesA .
 map(λ colOfTilesB .
  reduce0, (λ (tileAcc, (tileA, tileB)) .
   map(map(+)) ∘ zip(tileAcc) ∘
   map(λ aBlocks .
    map(λ bs .
     reduce(0, +) ∘
     map(λ (aBlock, b) .
       map(λ (a) .  a × b
       , aBlock)
     ) ∘ zip(transpose(aBlocks), bs)
    , tileB)
    , split(l, tileA))
  , zip(rowOfTilesA, colOfTilesB))
 ) ∘ tile(m, k, transpose(B))
) ∘ tile(n, k, A)
```

Figure 5.11: Transforming matrix multiplication by combining optimisations using macro rules.

This complex transformation is achieved by applying macro rules that are composed of simple rewrite rules like the ones presented in Section 5.4. As each of these simple rules is provably correct, by composition the bigger transformations are automatically valid as well. This is a major advantage compared to traditional compiler techniques, where complex analysis is required to apply such big optimisation steps.

Blocking Blocking is represented by swapping nested *map* primitives as shown in the third expression in Figure 5.11. Like the macro rule for tiling matrix multiplication, the macro rule for blocking uses split-join and interchange rules to achieve its goal.

It starts by *split*ing *tileA* on line 14 to form multiple blocks of rows. For combining multiple rows of *tileA* with a single column of *tileB* the resulting blocks of rows of A (*aBlocks*) is transposed before using *zip* on line 12. Then *map* is applied (line 9) to obtain a pair of elements of *tileA* (*aBlock*) together with a single element of *tileB* (*b*). The element *b* is reused on line 10 while iterating over *aBlock* using the *map* primitive.

While it might seem like the macro rules are just compositions of existing rules, it is worth noting that they are driven by the goal of achieving specific optimisations for the hardware devices. This makes them an important structuring mechanism while at the same time avoiding an explosion in the size of the search space and making it feasible to explore these optimisatons automatically. The next section will describe how these macro rules are used to explore the optimisation space.

5.6 Automatic Exploration Strategy

Having defined optimisations as rewrite rules, it is now possible to explore the space automatically by applying a combination of rules to the input program. However, the resulting space is extremely large, even potentially unbounded, which opens up a new research challenge. This is in stark contrast to classical auto-tuners which have a much smaller space to explore due to their parametric nature. However, this is also the main reason why auto-tuners sometimes fail to achieve high-performance as seen in the motivation; they are bound by the fixed set of parameter chosen by the implementer and cannot search beyond these. In contrast, the LIFT rewrite-based approach is able to combine the various optimisations expressed as rules in any way and can, therefore, explore a far larger amount of implementations unreachable with classic auto-tuning.

A simple and heuristic-based pruning strategy to tackle the space complexity problem is presented here. Future research will investigate more advanced techniques to fully automate the pruning process, e. g. using combinations of micro-benchmarking and machine learning.

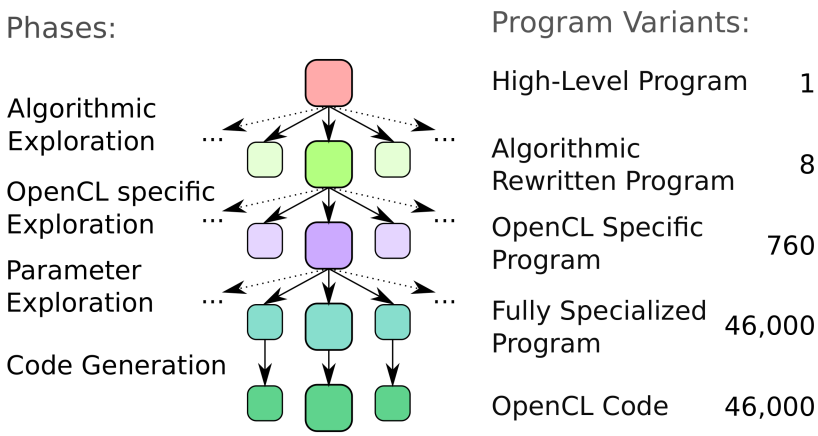


Figure 5.12: Exploration and compilation strategy and the number of program variants generated for desktop class GPUs

For matrix multiplication, the exploration starts from the high level expression shown in Listing 5.1. Rewrite rules are automatically applied until a low-level expression is produced such as Listing 5.3 from which OpenCL code is generated.

Figure 5.12 gives an overview of the exploration and compilation strategy. For desktop class GPUs 46,000 OpenCL kernels are generated from a single high-level program.

5.6.1 Algorithmic Exploration Using Macro Rules

To explore different algorithmic optimisation choices, the algorithm shown in Algorithm 2 is used. `allRules` in line 1 contains all the rules that will be applied during the exploration. The vectorisation optimisations discussed in section 5.4.3 as well as 1D and 2D register blocking, and tiling presented in Section 5.5 are encoded and contained in the list.

To generate the 8 algorithmically rewritten programs, 20,000 program variations are considered. These variations are produced by applying these macro rules at all valid locations in line 2 starting from the high-level matrix multiplication program in Listing 5.1. The recursive algorithm (`applyRulesRecursively`) for applying the macro rules is described in more detail below. As inputs, it takes the current program, the number of recursive calls that should be performed and the rules applied so far. First, some rule are removed from consideration in line 6. This removes rules that have already been applied twice. Next, in line 7, all nodes in `program` are inspected for all `rulesToTry` to see if they can be applied there to produce a list of node and rule pairs. Each pair contains a node and a rule can be applied there. In line 8 all these rule applications are performed to create new algorithmically rewritten variations of `program`.

Finally, the algorithm either just returns the rewritten variations (line 9) or recursively calls itself on all the newly created variations to create more (line 11).

After creating the algorithmically different variations, most variations have unnecessary

data-layout patterns in them. This inhibits the possibility of fusing map and reduce primitives and using less memory and/or a faster address space for storing temporary results and will therefore be automatically removed if possible. The expression below is an example where the interchange rule has been applied twice, first on $map(reduce)$ and then on $map(map)$. The *reduce* and *map* could be fused if there were no transpose patterns between them.

```
transpose ∘ reduce(z, λ (acc, elem) . map(⊕)(zip(acc, elem))) ∘
  transpose ∘ transpose ∘ map(map(f)) ∘ transpose
```

Therefore, the variations are simplified in line 3 of Algorithm 2. The algorithm for simplifying an expression will attempt to fuse as many map and reduce primitives as possible and eliminate as many data-layout patterns as possible. It starts by trying to apply fusion rules to the program. Once none of the fusion rules can be applied, it will try to apply simplification rules. If there are simplification rules that can be applied, it will do so and then attempt to apply fusion rules again. If no fusion or simplification rules can be applied, the algorithm will attempt to apply different enabling rules. For each version produced by applying a different enabling rule it will attempt fusion and simplification rules again. Once none of them can be applied, it will pick the one with the fewest nodes as the one to keep working on and attempt to apply enabling rules again. If no fusion or simplification rules apply, it will try to apply more enabling rules up to a maximum of 5. If none of that enables more fusion or simplification, the algorithm terminates and returns the current program.

In order to reduce the search space, programs which are unlikely to deliver good performance on the GPU are discarded using two heuristics in line 4 of Algorithm 2. The first heuristic limits the depth of the nesting in the program: some rules are always applicable, however they are unlikely to improve performance after exploiting all levels and dimensions of the OpenCL thread hierarchy. Using the first heuristic the number of rewritten programs to focus on is reduced to about one hundred. The second heuristic looks at the distance between the addition and multiplication operations after applying the simplification algorithm. A small distance increases the likelihood of fusing these two instructions together and avoiding intermediate results. The number of expressions after applying the second heuristic is reduced to 8, which are then passed to the next phase.

5.6.2 OpenCL Specific Exploration

For each algorithmically rewritten program, different mapping strategies to the GPU are explored. The mapping is divided into two parts, the mapping of parallelism to the OpenCL thread hierarchy and the mapping of memory to the OpenCL memory hierarchy. The algorithm for performing this part of the exploration is shown in Algorithm 3.

First, parallelism mappings are applied in line 1. The exploration is restricted to a few fixed parallelism mappings. For desktop GPUs the two outermost `map` primitives are turned

```
   input  : A single LIFT program
   output: A list of rewritten LIFT programs
1  allRules = [...]

   rewrite(program)
2  rewritten = applyRulesRecursively(program, 5, [])
3  simplified = map(simplify, rewritten)
4  filtered = filter(checkHeuristics, simplified)
5  return filtered

   applyRulesRecursively(program, explorationLevel, rulesAppliedSoFar)
6  rulesToTry = filterRules(allRules, rulesAppliedSoFar)
7  ruleLocationPairs = listAllPossibleRewritesForRules(program, rulesToTry)
8  rewritten = map(λ (rule, location) . (applyRuleAt(program, rule, location), rulesAppliedSoFar + rule),
     ruleLocationPairs)
9  if explorationLevel == 1 then return rewritten;
10 else return rewritten ++ flatMap(λ (program, rulesAppliedSoFar) .
11    applyRulesRecursively(program, explorationLevel- 1, rulesAppliedSoFar), rewritten);
```

Algorithm 2: Algorithm for performing algorithmic rewriting of a LIFT program using macro rules.

into *mapWorkgroup* primitives to perform these computations across a two-dimensional grid of work-groups. The next two `map`s are rewritten into *mapLocal* primitives to exploit the parallelism inside of a two-dimensional work-group. Finally, all further nested `map` primitives will be executed sequentially. This strategy is common in desktop GPU programming. For mobile GPUs, the two outermost *map*s are mapped to the global work items using *mapGlb* and the rest will be executed sequentially. The Mali GPU does not have physically distinct local memory, it is mapped to global memory, and when the local memory is not used in the expression then using work-groups is usually not beneficial.

For the memory hierarchy, the usage of local and private memory is explored. This is achieved by inserting copies to those address spaces at various locations in line 2 and line 3. The number of copies into each memory space is limited to two, to avoid expressions which perform many meaningless copies. The exploration attempts to insert copies into locations where data is reused or communicated between threads.

Starting from the 8 algorithmically rewritten programs, the exploration automatically generates 760 OpenCL specific programs for desktop GPUs and 31 for the Mali with a particular mapping decision encoded using these heuristics.

5.6.3 Parameter Exploration

Most *low-level* (OpenCL specific) programs contain parameters, e. g., the argument to *split(n)* controlling the size of a tile or a block. Selecting these is similar to classical auto-tuning

input : A single LIFT program
output: A list of OpenCL specific LIFT programs

```
rewrite(program)
1 rewritten = applyParallelismMappings(program)
2 rewritten = rewritten ++ flatMap(insertCopiesToLocal, rewritten)
3 rewritten = rewritten ++ flatMap(insertCopiesToPrivate, rewritten)
4 return rewritten
```

Algorithm 3: Algorithm for performing OpenCL specific rewriting of a LIFT program.

techniques. An automatic exploration of these parameters is performed by exhaustively picking all possible parameter values in a reasonable range.

Thread counts also need to be chosen for each kernel.They are picked in such a way to try and minimise the number of loops generated in the final OpenCL code after the control flow simplification optimisations described in Chapter 4 are applied. To do this, the whole program is traversed and the bounds of all *mapGlb*, *mapWrg* and *mapLcl* are examined. Each dimension is considered separately. If none of the patterns appear in a dimension, both, the local and global thread counts are set to 1. If *mapGlb* then the global thread count will be set to the most common range and the work-group size will be explored as described later. If *mapLcl* and *mapWrg*, then the local thread count will be set to the most common range for *mapLcl* and number of groups to the most common range of *mapWrg*. The global thread count will be the local thread count and number of groups multiplied. Because of parallelism mappings enabled, either only *mapGlb* or *mapWrg* and *mapLcl* will appear in the program.

Furthermore, the exploration makes sure that the parameters picked will not generate an OpenCL kernel requiring too much private, local, or global memory. Parameter combinations leading to an unreasonably small or high number of work-groups or local threads are also discarded. As the focus is on the Mali GPU, the vector width is fixed to 4 to prune the space.

For the 760 low-level OpenCL specific programs, around 46,000 fully specialised programs are generated and for the 31 low-level expression, 677 specialised programs are generated. For all these programs OpenCL code is generated using the techniques described in Chapter 4.

Once an OpenCL kernel has been generated, the work-group size runtime parameter, i. e. the number of threads in a work-group, needs to be chosen if it was not chosen by the algorithm described before. For matrix multiplication, work groups are two-dimensional and the number of threads has to be selected in both dimensions. A search of the work-group size within the range allowed by OpenCL is conducted on Mali. This resulted in $11,628$ unique combinations of runtime parameters and optimisations on Mali.

5.6.4 Summary

By defining rewrite rules and expressing larger optimisations using them, tens of thousands of OpenCL kernels are automatically generated, all of which are correct by construction. This enables exploring combinations of the tiling and register blocking optimisations combined with strategies for mapping expressions to GPUs and numerical parameters. Section 5.8 discusses performance results, but first, the experimental setup is briefly discussed.

5.7 Experimental Setup

Three platforms are used to compare against high-performance BLAS libraries: 1) a Nvidia GTX Titan Black (Kepler architecture) using CUDA 6.0 and driver 331.79; 2) a AMD Radeon HD 7970 (Tahiti architecture) using AMD APP SDK 2.9.214.1 and driver 1526.3; 3) an ODROID XU3 board with a Samsung Exynos5422 system on a chip containing a Mali-T628 MP6 GPU.

The code generation technique is evaluated using matrix multiplication with differently sized square and rectangular matrices ($512^2 * 512^2$, $1024^2 * 1024^2$, $2048 \times 512 * 512 \times 2048$, $512 \times 2048 * 2048 \times 512$) of single precision floating point values.

The mobile Mali-T628 MP6 GPU is separated into two OpenCL devices. The first device with 4 cores is used with the Mali SDK 1.1 OpenCL implementation. DVFS is disabled and the clock frequency is locked at 600 MHz.

For all experiments, the median performance in GFLOPS of at least 5 executions for each kernel is reported. The runtimes are measured using the device high resolution timers.

5.8 Experimental Evaluation

This section evaluates the rewrite based approach using matrix multiplication as a case study. It first investigates the results obtained by executing the automatically generated OpenCL kernels on two different desktop GPUs and a mobile GPU that is designed considerably differently from the desktop ones. Items of interest are the kernels with the highest performance and if a universally good kernel which can be used for all input sizes and GPUs exists.

Unless specified otherwise, $1024^2 * 1024^2$ is used as the default input size.

5.8.1 Performance Portability and Performance Comparison Against Libraries and Auto-tuning

To investigate portability of performance across different classes of GPUs the rewrite-based approach is compared against the CLBlast[1] `insert` [1] d190bec from https://github.com/CNugteren/CLBlast

library that is auto-tuned with the state-of-the-art CLTune[2] `insert` [2] ad94a3d from https://github.com/CNugteren/CLTune

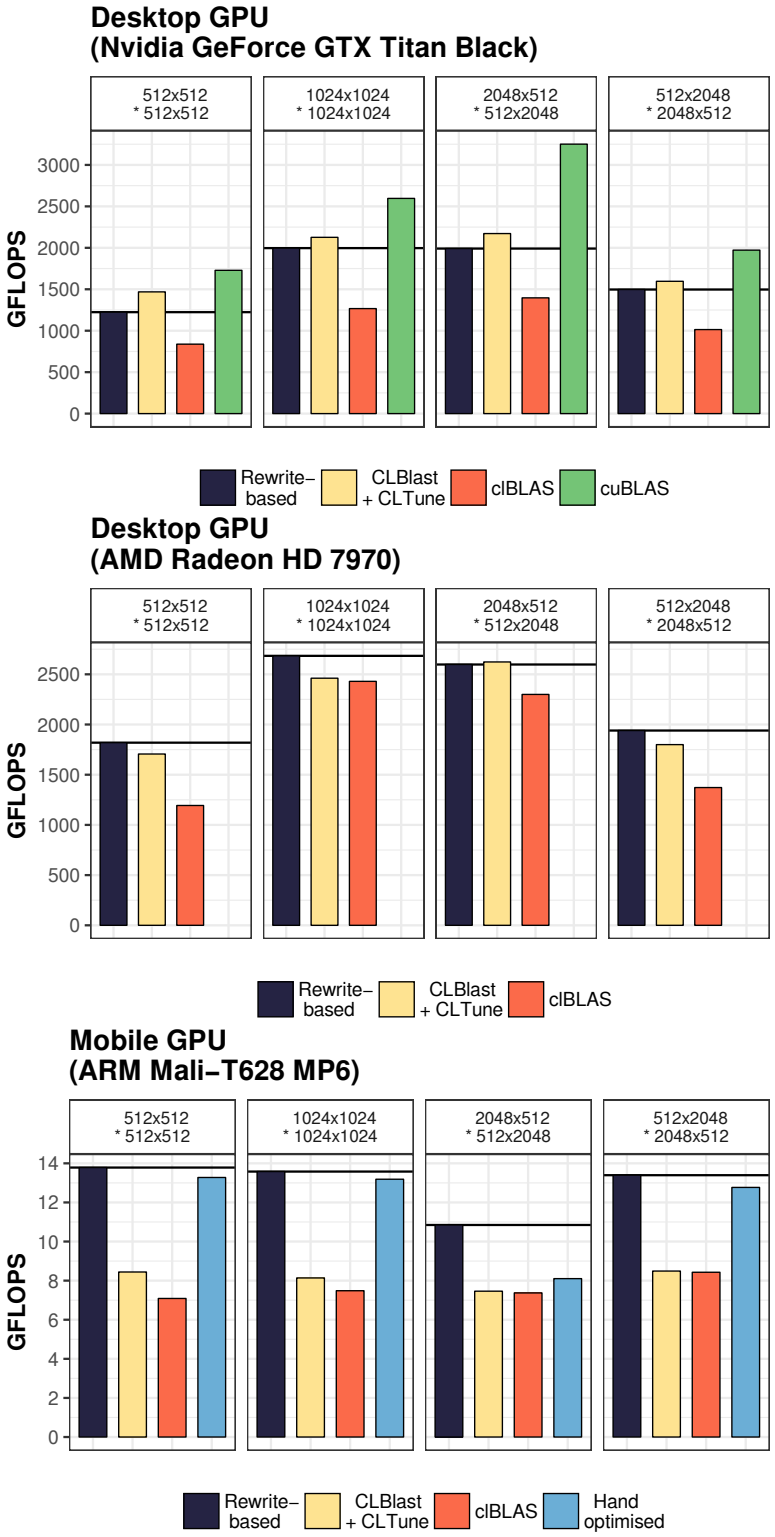


Figure 5.13: Performance of matrix multiplication on two desktop GPUs and one mobile GPU for different input sizes. The *rewrite-based* approach is the only one that achieves performance portability across desktop-class *and* mobile GPUs.

		Run on		
		Nvidia	AMD	Mali
Tuned for	Nvidia	100.0 %	27.5 %	N/A
	AMD	20.5 %	100.0 %	11.6 %
	Mali	4.2 %	14.4 %	100.0 %

Table 5.1: Performance portability of kernels ($1024^2 * 1024^2$)

[Nugt 15] on three GPUs from AMD, ARM, and Nvidia. As reference points, the clBLAS[3] library developed by AMD using OpenCL, as well as an implementation particularly tuned for each architecture are used: the hand tuned version shown in Listing 5.3 on Mali, cuBLAS on Nvidia, and clBLAS on AMD.

[3] `insert` d16f7b3 from https://github.com/clMathLibraries/clBLAS

Figure 5.13 shows the performance comparison of all implementations on four different input sizes and shapes. The auto-tuned CLBlast library delivers high performance on the two desktop GPUs, achieving performance higher than clBLAS on the AMD GPU. On Nvidia, CLBlast achieves about 80% of the performance of cuBLAS for three inputs sizes. That is a very good number, as the proprietary cuBLAS relies on advanced assembly-level optimisations which cannot be implemented using CUDA or OpenCL [Lai 13]. However, on the mobile Mali GPU the auto-tuning approach is less successful, achieving only about 60% of the performance of the hand optimised implementation on three inputs and 25% slower than the LIFT results on the other input.

This shows that performance portability is not achieved purely using auto-tuning. By investigating the tuned OpenCL kernel used by CLBlast, it could be seen that the built-in *dot* function or vectorised operations are not used which is crucial for achieving high performance on Mali (see section 5.8.3). On the desktop GPUs these optimisations are not required as there is no hardware support for vectorisation. Furthermore, the overall structure of the kernel is similar to the one used for the desktop GPUs, clearly showing that CLBlast was developed for these GPUs and applied to Mali as an afterthought.

The LIFT rewrite-based approach delivers high performance on the desktop GPUs *and* on the mobile GPU. Performance on the desktop GPUs is very close (Nvidia) or even slightly better (AMD) compared to CLBlast on all input sizes. Crucially, the rewrite-based approach consistently achieves a large performance improvement on the Mali GPU compared to CLBlast (up to 1.7× better). It is able to outperform any other implementation on Mali, especially for the third input size where choosing a larger tile size increases the amount of work per thread which is beneficial for this type of matrix shape.

The key for achieving high performance is the support for architecture specific optimi-

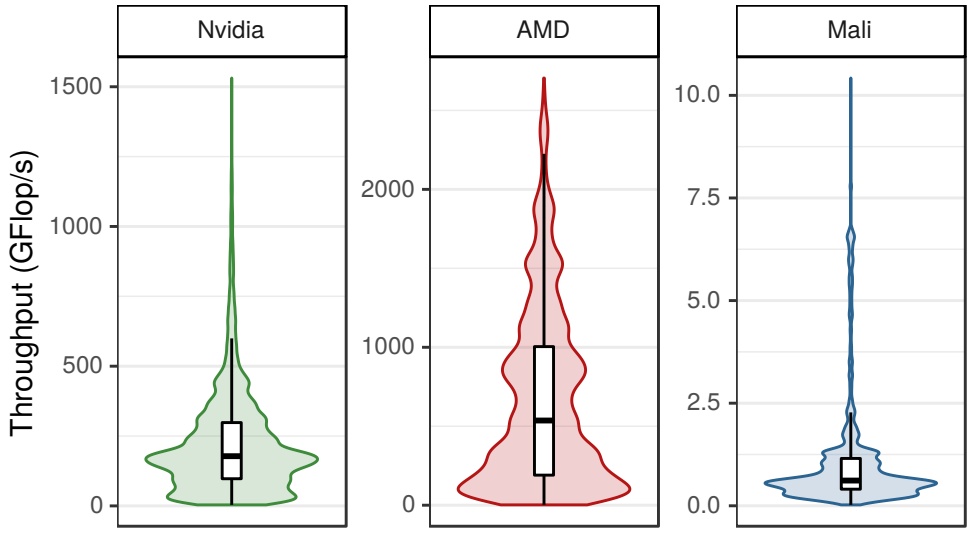


Figure 5.14: Distribution of performance for generated kernels.

sations expressed as generic rewrite rules and the ability to generate structurally-different OpenCL kernels. In fact, when running the best OpenCL kernel generated for Mali on the Nvidia GPU only 4% of the performance is obtained compared to running the kernel optimised for this GPU (i. e. 25x slower) as seen in Table 5.1. Conversely, running the kernel optimised for the desktop class AMD GPU on Mali results in only 11% of the performance achieved with the best kernel generated for the embedded GPU (i. e. 9x slowdown). The Nvidia kernel does not even run on Mali due to insufficient hardware resources.

On the desktop GPUs the rewrite-based approach generates kernels exploiting the hierarchical organisation of threads, local memory, tiling, and the fused multiply-add instruction, whereas on the mobile GPU, a flat organisation of threads, vectorisation, and the *dot* built-in are crucial. These very different OpenCL kernels are derived from a single high-level expression of matrix multiplication using rewrites.

5.8.2 Space Exploration

Figure 5.14 shows the distribution of performance as well as the median and quartiles for generated kernels on the three test platforms for the $1024^2 \times 1024^2$ input size. All three graphs show a similar shape with poor performance for most kernels. The maximal performance is only reached by a few generated kernels. This highlights the difficulty of optimising matrix multiplication kernels: only a few kernels find the right balance for applying the tiling and blocking optimisations, make good use of the local memory, vectorisation, and choose well suited implementation parameters.

For a matrix of size 1024^2, a single kernel execution takes on average 10ms (AMD) and 26ms (Nvidia). Even the execution of tens of thousands of OpenCL kernels can, therefore, be

```
1  matrixMultiplication(A, B) =
2    join ∘ mapGlb(0)(λ nRowsOfA .
3      transpose ∘ join ∘ mapGlb(1)(λ mColsOfB .
4        transpose ∘ map(transpose) ∘ transpose ∘ toGlobal(mapSeq(mapSeq(mapSeq(id)))) ∘
5        reduceSeq(init = make2DArray(n,m, 0.0f), λ (accTile, (tileOfA, tileOfB)) .
6          mapSeq(λ (accRow, rowOfTileOfA) .
7            join ∘ mapSeq(λ (acc, colOfTileOfB) .
8              reduceSeq(acc, add) ∘
9              mapSeq(λ (vectorA, vectorB) . dot(vectorA, vectorB),
10             zip(asVector(k, rowOfTileOfA),
11                 asVector(k, colOfTileOfB))),
12           zip(accRow, transpose(tileOfB)) ),
13         zip(accTile, transpose(tileOfA))),
14       zip( split(k, transpose(nRowsOfA)),
15            split(k, transpose(mColsOfB)) )),
16     split(m, B)),
17   split(n, A))
```

Listing 5.24: The best performing low-level expression automatically derived from the high-level expression in Listing 5.1 using rewrite rules.

performed in a reasonable time frame. Overall the exhaustive execution of all 46,000 OpenCL kernel took less than an hour on Tahiti and Kepler, including the overheads of data transfers and validation. The generation of all kernels with the LIFT prototype compiler implemented in Scala took about 2 hours and 40 minutes. The compilation of all generated OpenCL kernels to binaries took 20 minutes for Nvidia and 1 hour for AMD.

For the Mali GPU, following the strategy described in Section 5.6, the generation of the 677 OpenCL kernels from 31 functional expressions took less than half an hour while performing the 11,628 executions took about a day.

5.8.3 Performance Comparison Against Manually Optimised Kernel on the Mali GPU

Figure 5.15 shows a performance comparison of three matrix multiplication implementations on the Mali GPU. The first bar shows the performance of the generated OpenCL kernel from the expression resembling the manually optimised kernel (Listing 5.3) whose performance is shown as the last bar. The second bar shows the best OpenCL kernel generated by automatically deriving the expression shown in Listing 5.24 from the five-line long high-level expression of matrix multiplication (Listing 5.1).

Listing 5.24 shows the best performing expression found automatically for the 1024^2x1024^2 input size. By investigating the expression it can be seen that vectorisation has been applied (lines 10–11), the *dot* built-in function is used (line 9), and tiling is performed with the `split` and `transpose` in lines 14–17. The vector width (k) and tile sizes ($n \times k$ and $m \times k$) are still

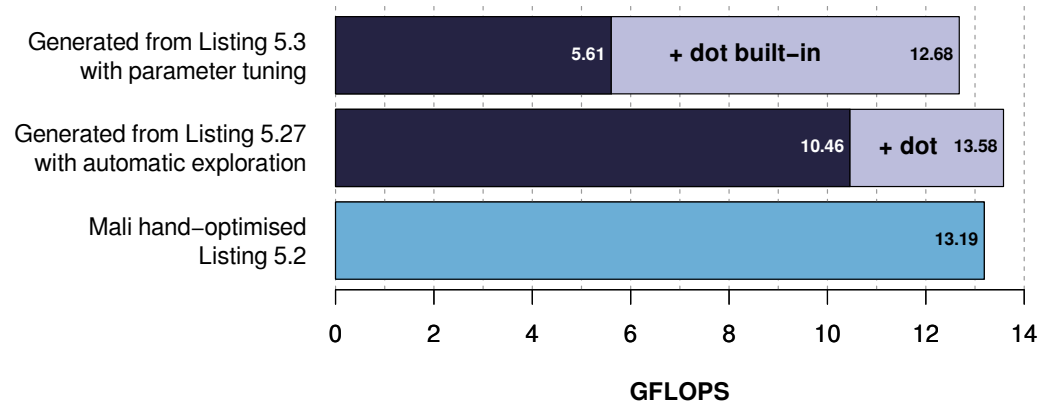


Figure 5.15: Performance of different matrix multiplication kernels on the Mali GPU. The fully automated exploration technique produces a kernel that outperforms the manually optimised kernel.

parameters that are picked prior to OpenCL code generation. After conducting the parameter exploration, the values leading to the fastest kernel are $k = 4$ and $n = m = 2$.

The expression is similar to the one resembling the manually optimised OpenCL kernel (Listing 5.3). Nevertheless, there are a few notable differences. For example the tiles are not explicitly copied to private memory and, therefore, more loads to the global memory are issued. However, as will be seen in the next subsection, this does not affect performance negatively as these accesses are probably cached. In fact, the generated OpenCL kernel is *faster* than the kernel which explicitly copies the data into private memory.

For the two OpenCL kernels generated by the automatc exploration process in Figure 5.15 the lighter bar indicates the performance benefit measured when including the rewrite rule for introducing the *dot* built-in function.

It can be seen that the performance of the OpenCL kernel generated via automatic exploration even slightly outperforms the manually optimised kernel. The rewrite rule that introduces the *dot* built-in turns out to be crucial, giving an extra 30% of performance as can be seen.

The kernel generated from the expression resembling the manually optimised implementation (the first bar) achieves 96% of the performance of the manually written kernel. Again, the usage of the *dot* built-in function is crucial for achieving high performance for matrix multiplication on Mali.

5.9 Conclusion

This chapter has shown how a system of rewrite rules encodes algorithmic and low-level transformations and is used to generate different implementations of the same program. This approach can easily apply composition of optimisations which leads to very high-performance

code, even on a mobile GPU. The rewrite-based technique makes it easy to add optimisations such as expressing the OpenCL dot-product built-in function, which makes a large performance difference on the Mali GPU. By applying the rewrite rules automatically, the system generates thousands of semantically equivalent OpenCL kernels.

Performing the same optimisations in an auto-tuner parametric implementation requires a significant effort to build complex parametric implementations and ultimately ends up being specialised for a certain class of devices. Indeed, this chapter has shown that the classical auto-tuning technique for matrix multiplication is not performance portable when presented with a GPUs whose architecture is significantly different.

Using matrix multiplication as a case study, the chapter has shown how the rewrite-based approach is able to generate 46,000 differently optimised OpenCL kernels for desktop GPUs and 677 OpenCL kernels for the Mali GPU. Out of these, the best generated kernels provide performance on par or even better than state-of-the-art high-performance OpenCL library implementations on Nvidia and AMD GPUs. By investigating the performance among kernels, it has been shown that sampling a small fraction of the generated kernels is sufficient to achieve high performance. Furthermore, among all generated OpenCL kernels there was not a single portable kernel providing good performance across all investigated GPUs.

Overall, the results have shown that only the rewrite-based code generator offers true performance portability across desktop GPUs and the Mali mobile GPU.

The next chapter will describe statistical performance prediction techniques to speed up optimsation space exploration and finding program variants with good performance faster.

Chapter 6

Performance Prediction for Accelerated Exploration of Optimisation Spaces

6.1 Introduction

As seen in the previous chapter, the rewrite based approach of LIFT leads to a huge and complex optimisation space that can take hours or days to explore. This chapter solves this problem by altogether avoiding the need to run candidate programs in the first place. It proposes to use a performance model that directly predicts the performance of programs by computing static features from the LIFT IR. This removes the necessity for compiling programs into OpenCL, then to a device binary, and finally running them on the GPU, which accounts for the majority of the time spent during exploration when performed as in described in Chapter 5.

The use of performance modeling for GPUs is not novel in itself and there have been numerous papers on the subject [Hong 09, Nugt 12, Nugt 14b, Zhan 17, Hong 18]. However, unlike previous approachs, this chapter shows how low-level GPU-specific features are extractable from a high-level functional IR. Concretely, this chapter demonstrates that the LIFT IR is amenable to extraction of low-level features which are useful in predicting performance. In particular, it shows how cache locality information, which is very important for predicting performance, is extractable at this level. The extraction of this information relies on the use of the rich information stored in the LIFT type system together with the ability to reason about array indices in a symbolic manner.

Using the features extracted, a performance predictor is built using a the kNN (k-Nearest Neighbours) machine-learning technique. The results show that this approach leads to a highly accurate machine-learning model for the stencil domain, an important class of high-performance code. The model achieves a high correlation of 0.8 and 0.9 on average on a GPU from Nvidia

and AMD, respectively. Using the model to search the space requires less than 5 runs in the majority of the cases to achieve performance within 90% of the best available. In comparison, a random search requires over 100 runs in the majority of the cases.

To summarise, the chapter makes the following three contributions:

- It shows how low-level GPU specific features are extracted from the high-level functional LIFT program;
- It presents a k-nearest neighbour based machine-learning model that predicts program performance;
- It provides experimental evidence that the model makes accurate predictions and is able to drastically reduce the time it takes to explore the optimisation space when used to drive the exploration process.

The rest of this chapter is organised as follows: Section 6.2 motivates this work. Section 6.3 explains how low-level hardware features are extracted from the high-level LIFT IR and Section 6.4 presents the performance model. Section 6.5 shows the experimental setup, Section 6.6 shows an analysis of the features and the performance achieved by the model while Section 6.7 shows how the model is used to drive the optimisation space exploration and that the model is able to drastically speed up the time it takes to find points that perform well. Finally, Section 6.8 concludes the chapter.

All the contributions presented in this chapter are the author's original work.

6.2 Motivation

Current LIFT Exploration Approach As described in the previous chapter, LIFT uses a system of rewrite rules to explore the space of possible GPU implementations. Figure 6.1 a presents an overview of how LIFT explores the optimisations. First, a high-level expression representing the program to be compiled is used as an input to the compiler. This generic high-level expression does not encode any optimisations. Then, the rewriting takes place and the LIFT exploration module applies rewrite rules to search the space randomly. This results in a set of *transformed* expressions where optimisations have been applied and parallelism has been mapped.

These transformed expressions are then fed into the LIFT code generator which produces OpenCL kernel source code. These kernels are compiled with the vendor-provided OpenCL compiler and OpenCL binaries are produced. Finally, all binaries are executed, the performance is recorded and the best found kernel is reported.

This automatic process is very time consuming as it produces a large number of kernels. For instance, for this chapter, up to 1,000 kernels are generated per benchmark. In addition,

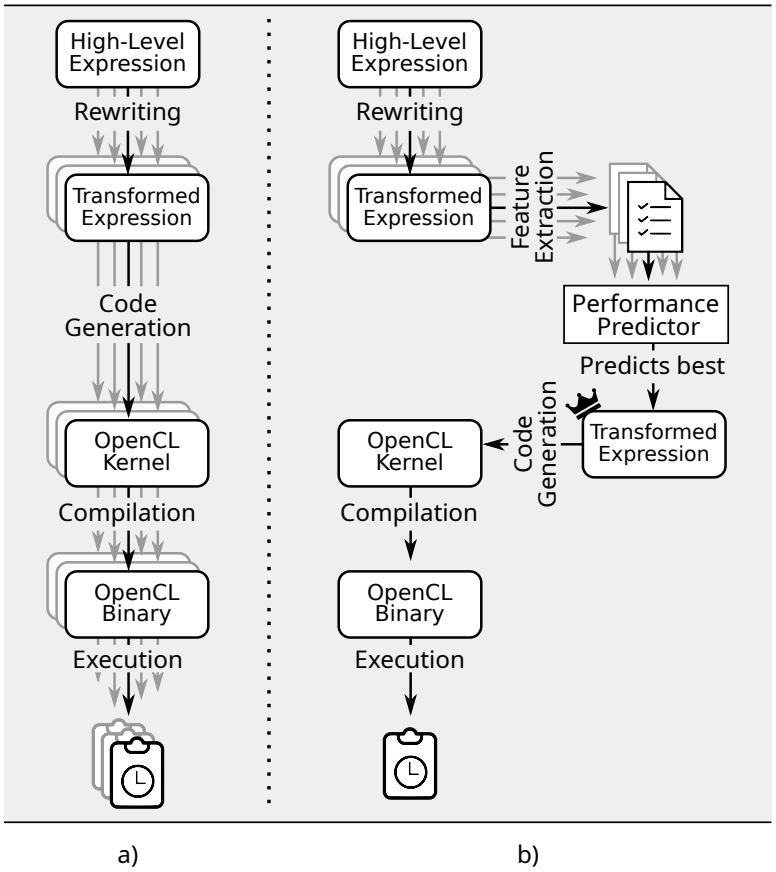


Figure 6.1: LIFT compilation and exploration. a) Current strategy which compiles and executes all transformed expressions. b) Improved strategy which uses a performance model to rank the transformed expressions and only compile and execute the best one.

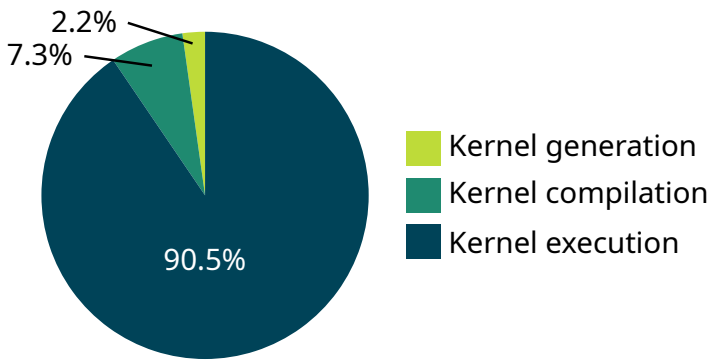


Figure 6.2: Time breakdown for the LIFT exploration process. *Kernel generation* includes time to rewrite and compile LIFT expressions to OpenCL kernels. *Kernel compilation* is the vendor-provided OpenCL compilation time. *Kernel execution* is the time required to execute all generated kernels.

some LIFT generated kernels are executable with a configurable number of threads which is additionally explored using heuristics leading up to a total number of 10,000 kernel executions.

Time breakdown Figure 6.2 shows the percentages of the time spent in the different stages of the current LIFT compilation and exploration. Unsurprisingly, the last part of LIFT's workflow, the kernel execution, requires by far the most time (up to 90%). For this chapter, executing all kernels for a single application, including the exploration of thread configurations took up to 41 minutes while all kernels were generated in less than a minute, which is about 2% of the overall time.

Using a Performance Predictor for Exploration The major bottleneck for searching the space is clearly the OpenCL compilation and execution time of the generated kernels, which represents 98% of total time. This chapter addresses this bottleneck by using a trained performance predictor directly on the transformed LIFT expression. Figure 6.1b shows how the exploration strategy is modified to integrate a performance model.

Once the transformed expressions have been produced, the idea is to extract features that are informative about performance. These features are then fed into a model which ranks the transformed expressions based on their predicted performance, which is almost instantaneous. Then, the transformed expression with the fastest predicted performance is selected, the corresponding kernel is generated, compiled and finally executed.

While this approach seems very simple, the challenges are two-fold. First, features that are informative about performance need to be identified, such as memory access patterns. Then, the features need to be extracted from the high-level functional LIFT IR. As will be seen, the LIFT IR encodes all the information necessary to calculate low-level GPU-specific features.

Type	Feature
Parallelism	global size (dimensions 0, 1 and 2)
	local size (dimensions 0, 1 and 2)
Memory	amount of local memory allocated
	global stores per thread
	global loads per thread
	local stores per thread
	local loads per thread
	average cache lines per access per warp
Control Flow & Synchronisation	barriers per thread
	if statements per thread
	for loop bodies executed per thread

Table 6.1: List of extracted features

The next section explains how features are extracted.

6.3 Feature Extraction

This chapter proposes to use a performance model to predict the performance of transformed LIFT expressions on GPUs in order to identify the best performing variant. This performance model relies on static features extracted directly from the high-level LIFT IR. Although the features are extracted at the high-level, they represent low-level features related to OpenCL and GPU hardware in general.

This section explains how low-level GPU-specific features are extracted from the high-level LIFT IR. A summary of the features can be seen in Table 6.1. They broadly fall into three categories: parallelism, memory and control-flow.

6.3.1 Parallelism

The amount of threads used to execute a program generally indicates how much of the work is performed in parallel. For a fixed input size, a low thread count means that the threads will be doing a lot of sequential work while a high thread counts corresponds to less sequential work. Both global and local thread counts across the three available thread dimensions are included. The local thread count affects how large each work-group will be, which may affect the amount of data reused or the number of groups that can run concurrently. Nothing special need to be done to extract these features since they are runtime parameters and readily available.

6.3.2 Memory

This section covers the features related to memory. This includes the amount of memory allocated, the number of memory accesses and memory access patterns.

6.3.2.1 Local memory usage

It is generally desirable to start as many threads as possible to fully utilise all cores of the target machine reaching maximum *occupancy*. Occupancy is typically maximised when multiple work-group execute concurrently on the core. More concurrent work-groups typically translates to more threads executing concurrently, which ultimately helps hiding memory latency.

The number of work-groups that execute simultaneously on a core depends on the amount of resources used by each work-group. One important resource is the amount of fast local memory (shared memory) used by the work-group. Therefore, it is crucial to determine this quantity.

Extracting the amount of memory used in a LIFT program is straightforward. The whole program is traversed once, collecting all the local memory allocations and simply summing up these numbers.

6.3.2.2 Number of Memory Accesses

In many cases, performance is greatly affected by the number and type of memory operations. An application that exhibits a large amount of data re-usage for instance, might be able to exploit the fast local memory. In such case, the slow global memory might be accessed very rarely, for instance just once to load the data into fast local memory. The program can simply reuse the local data several times, dramatically reducing the number of global memory accesses which usually translates in an increase in performance.

Algorithm The LIFT code generator only produces loads and stores to memory when a user function is called. Therefore, counting the number of loads and stores boils down to counting how often each user function is called. As can be seen in Algorithm 4, a depth-first traversal is performed on the IR while keeping track of the number of times the body of patterns that generate loops is executed. Once a user-function is reached, the feature extractor simply updates the total number of loads and stores. In addition to this, the extractor keeps track of the type of memory being accessed, local or global, using the *toLocal* and *toGlobal* patterns. The information about the address space is encoded directly into the IR and is populated by another pass that runs prior to feature extraction. The number of global/local loads and stores is then normalised by the number of total threads.

input : Lambda expression representing a program

output: Numbers of different types of memory accesses.

```
countAccesses(in: lambda)
1  counts = { totalLoad: {local = 0, global = 0}, totalStore: {local = 0, global = 0}}
2  return countAccessesExpr(lambda.body, 1, counts)

countAccessesExpr(in: expr, in: iterationCount, in: counts)
3  switch expr do
4  | case FunCall(f, args)
5  | | foreach arg in args do
6  | | | counts = countAccessesExpr(arg, iterationCount, counts)
7  | | switch f do
8  | | case Lambda(body)
9  | | | return countAccessesExpr(body, iterationCount, counts)
10 | | case toPrivate(Lambda(body)) or toLocal(Lambda(body)) or toGlobal(Lambda(body))
11 | | | return countAccessesExpr(body, iterationCount)
12 | | case MapSeq(Lambda(body)) or MapGlb(Lambda(body)) or MapLcl(Lambda(body)) or
   |   MapWrg(Lambda(body))
13 | | | return countAccessesExpr(m.f.body, iterationCount * args(0).length)
14 | | case ReduceSeq(Lambda(body))
15 | | | return countAccessesExpr(body, iterationCount * args(1).length)
16 | | case Iterate(Lambda(body), count)
17 | | | return countAccessesExpr(body, iterationCount * count);
18 | | case UserFun
19 | | | foreach arg in args do
20 | | | | counts.totalLoad[arg.addrsSpace] += iterationCount
21 | | | counts.totalStore[arg.addrsSpace] += iterationCount
22 | | | return counts
23 | | otherwise do return counts // Nothing to count;
24 | otherwise do return counts // Nothing to count;
```

Algorithm 4: Pseudo-code for counting the total number of loads/stores for each type of memories.

```
1 example(arg0: [float]N, arg1: [float]N) =
2   mapWrg(
3     mapLcl(toGlobal(multByTwo)) ∘ mapLcl(toLocal(add))
4   )(split(64) ∘ zip(arg0, arg1))
```

Listing 6.1: Example program for demonstrating extracting memory access counts.

Example As an example, consider the program in Listing 6.1. The algorithm starts with the top level lambda and soon encounters the *mapWrg* primitive. At this point in the algorithm, line line 12, `n` will be $N/64$ (the length of the outer dimension of the input after the *split*). The algorithm calls recursively *countAccessesExpr* with $N/64$ as the *iterationCount*. When visiting either of the *mapLcl* in line 3, `n` will this time be 64 (the length of the inner dimension of the input after the *split*).

When the `add` user function is visited, global loads will be updated twice, since the add function has two inputs (the tuple is automatically unboxed). Since at this point, the *iterationCount* is $N/64 * 64 = N$, the total number of global loads is $N * 2$ and the total number of local stores is N. When the `multByTwo` user function is visited, local reads and global store will both be updated once, resulting in N local loads and N global stores.

When the algorithm terminates, it has determined that the LIFT program in Listing 6.1 performs $N * 2$ loads from global memory, N stores to global memory, N loads from local memory and N stores to local memory.

If the program is run with N threads, such that every thread computes one element of the output, then the output of the algorithm is used to calculate that every thread performs $N * 2/N = 2$ loads from global memory, $N/N = 1$ stores to global memory, $N/N = 1$ loads from local memory and $N/N = 1$ stores to local memory.

6.3.2.3 Memory Access Patterns

The way a program accesses memory has a profound impact on performance. For example, GPUs coalesce several memory requests into a single one when threads from the same warp access a single cache line (for example, typically 128 bytes on NVIDIA GPUs and 64 bytes on AMD GPUs). Coalescing memory accesses is critical for exploiting the large memory bandwidth that GPUs offer. It is, therefore, important to extract information about memory access patterns to have any hope of building an accurate performance predictor.

General Algorithm In order to determine the average number of cache lines accessed per warp per memory access, the feature extractor recursively traverses the IR, keeping track of the iteration count as for counting memory accesses. When a memory access is encountered, it determines the number of unique cache lines accessed by the warp as follows. First, it

generates the actual index expression using the existing mechanism of the LIFT compiler that was described in Chapter 4. If the expression contains no thread id, it means all the threads are accessing the exact same address and therefore the same cache line.

When the expression contains a thread id, a new index expression is generated for each thread in the warp by adding a constant to the thread id (threads in a warp have consecutive ids). The original array index expressed as a function of the thread id is denoted as $access(tid)$. Given n, the number of threads in a warp, the set of array indices accessed by the warp is given by the following:

$$\{access(tid+0), access(tid+1), \cdots, access(tid+n-1)\}$$

This list of indices relates to the different addresses accessed in memory by a warp. Now, given the cache line size s (assuming it is expressed as a multiple of the data size being accessed), the list of cache lines accessed can be computed:

$$\{access(tid+0)/s, access(tid+1)/s, \cdots, access(tid+n-1)/s\}$$

Finally, the elements in the list can be subtracted from each other to identify which ones are equal (result of the subtraction is 0) and count the number of unique accesses.

Implementation details The approach explained above is conceptually correct, however, it relies on having the ability to simplify arithmetic expressions symbolically. While the LIFT arithmetic simplifier supports a significant set of simplifications, it is far from being complete and fails in some cases to simplify some subtractions. In such cases the features extractor might fail to recognise that some accesses are identical. The following paragraphs explain a few workarounds for this problem that are used inside the feature extractor.

The first issue encountered, is that it is extremely difficult to calculate the set of unique cache lines by subtraction. Conceptually, one could take the first access $access(tid+0)/s$, subtract every other accesses by it and hope that the algebraic simplifier would be able to return 0 in the case where two accesses are identical. Unfortunately, simplifying expressions as simple as

$$(tid+0)/s-(tid+1)/s$$

which is equal to 0 when $s>1$, is far from trivial given that $/$ represents the integer division.

To overcome this challenge, the approach is slightly modified and an extra step is added. Before dividing by s, all the relative array accesses are first calculated as an offset of the first access by simple subtraction. The intuition behind this is two-fold. First, it is much easier to simplify a subtraction if it does not contain terms with integer division. Secondly, only the distances between the accesses rather than their absolute locations matter for identifying the number of unique cache line accessed.

```
1 example(in: [float]_N) = mapGlb(mapSeq(f))(split(n)(in))
```

Listing 6.2: Example program for demonstrating extracting memory access patterns.

So if the original accesses are

$$\{tid+0, tid+1, \cdots\}$$

they become

$$\{(tid+0)-(tid+0), (tid+1)-(tid+0), \cdots\}$$

which simplifies trivially to $\{0,1,\cdots\}$. Then, the division is performed as before, which leads to $\{0/s, 1/s, \cdots\}$ which trivially simplifies to $\{0,0,\cdots\}$. Now it is much easier to identify the unique cache lines.

Another practical issue has to do with the *pad* pattern which is used to implement boundary conditions in stencil programs. This introduces a lot of ternary operators `?:` to check at every memory access if the element is in bounds. This operator makes it again harder for the simplifier when subtracting memory accesses with each other. To overcome this, the feature extractor focuses on the common case which most accesses use. This is done by removing the ternary operator, substituting it with the index for the common case and therefore ignoring the rarely taken edge case. This substitution is only possible because the semantics of the *pad* pattern makes it clear which option is the common case and which case is the edge case. It would not work for arbitrary ternary operators.

Example Consider the example program from Listing 6.2. The array index being read for the argument of f is `i + n * gl_id` where i is the iteration variable of the *mapSeq* and `gl_id` the global thread id. Depending on the split factor n, a different number of cache lines will be accessed by a warp. With a split factor of $n = 1$, a single cache line would be accessed since the accesses within a warp are consecutive. However, if the split factor is larger than the warp size, then each warp will be touching a different cache line.

Assuming a cache line size of 32 words, 32 thread per warp and 1 word for a float, then the cache line indices within a warp are:

$$\{(i+n*gl\_id), (i+n*(gl\_id+1)), \cdots, (i+n*(gl\_id+31))\}$$

Using the trick presented earlier, all indices can be expressed as an offset from the first one:

$$\left\{\begin{array}{c} (i+n*gl\_id)-(i+n*gl\_id), \\ i+n*(gl\_id+1)-(i+n*gl\_id), \\ \cdots, \\ i+n*(gl\_id+31))-(i+n*gl\_id)\} \end{array}\right\}$$

which simplifies trivially to: $\{0, n, \cdots, n*31\}$. Now dividing by the cache line size results in $\{0, n/32, \cdots, n*31/32\}$.

If the split factor n is 1, this results in a set of 32 zeros, meaning all the thread in the warp access a single cache line. When the split factor $n = 4$, this will results in the following list: $\{0,0,0,0,1,1,1,1,\cdots,7,7,7,7\}$. Since it has 8 unique values, the warp touches 8 cache lines for this memory access.

6.3.3 Control Flow and Synchronisation

Another important factor that often limits performance for GPUs is the amount of control flow and synchronisation. Control flow, such as `if` and `for` loop statement typically produce branching instructions which is notoriously bad for performance on GPUs. The different branches are counted separately, as the branches from `if` statements are not taken by all threads and will cause divergent execution. Similarly, the presence of barriers is detrimental to performance since all execution can only process once all the threads have reached the barrier. For this reason, the features extractor determines the total number of branches resulting from `if` statements and `for` loops; and barriers that will be encountered by all threads. As with memory accesses, the numbers are the normalised by the total number of threads.

Algorithm This algorithm is similar to the algorithm used to count the number of memory operations and is shown in Algorithm 5. The algorithm recursively traverses the IR starting from the root in line 2, keeping track of the number of times branches or barriers are executed.

Whenever a patterns that might produce a loop (e. g. *iterate*, *mapLocal*, *reduceSeq*) is encountered the `updateCounts` function is called (in line 15, line 19 and line 22). The `updateCounts` function checks whether a branch will be emitted and update the global branch counters, taking into account the current iteration count. This is handled in line 27.

The algorithm also detects special cases where loops might not be emitted. The first case is when only a single statement is emitted, and can be seen in line 28. It happens when a *mapSeq* iterates over an array of size 1 and it is clear that a loop is not required. This is also the case if *mapLocal*, *mapWrg* or *mapGlobal* iterates over the same number of elements, as there are local threads, work-groups or global threads respectively, then a loop is also not required. The second case is when a `for` loop is unrolled because it is accessing a private memory array that has been flattened into variables or it is accessing elements of a vector. The cases where only a single statement or an unrolled loop is emitted are handled in line 28.

The final case is more subtle and involves *mapLocal*, *mapWrg* or *mapGlobal*. If the size of the input array is smaller than the number of local threads, work-groups or global threads, respectively, the code generator will emit an `if` statement instead of a loop since the loop can at most be executed once per thread or work-group. The number of branches from `if` loops are

```
   input : Lambda expression representing a program
   output: Numbers of barrier calls and branches from for loops and if statements.
countBranchesAndBarriers(in: lambda)
1  counts = {forBranchCount = 0, ifBranchCount = 0, barrierCount = 0}
2  return countBranchesAndBarriersExpr(lambda.body, 1, counts)

countBranchesAndBarriersExpr(in: expr, in: iterationCount, in: counts)
3  switch expr do
4      case FunCall(f, args)
5          foreach arg in args do
6              counts = countBranchesAndBarriersExpr(arg, iterationCount, counts)
7          switch expr.f do
8              case Lambda(body)
9                  return countBranchesAndBarriersExpr(body, iterationCount, counts)
10             case toPrivate(Lambda(body)) or toLocal(Lambda(body)) or toGlobal(Lambda(body))
11                 return countBranchesAndBarriersExpr(body, iterationCount, counts)
12             case MapSeq(Lambda(body)) or MapGlb(Lambda(body)) or MapLcl(Lambda(body)) or
                MapWrg(Lambda(body))
13                 if f is MapLcl and f.emitBarrier then counts.barrierCount += iterationCount;
14                 n = args(0).length
15                 updateCounts(f, iterationCount, n);
16                 return countBranchesAndBarriersExpr(body, iterationCount * n, counts)
17             case ReduceSeq(Lambda(body))
18                 n = args(1).length
19                 updateCounts(f, iterationCount, n);
20                 return countBranchesAndBarriersExpr(body, iterationCount * n, counts)
21             case Iterate(Lambda(body), count)
22                 updateCounts(f, iterationCount, count);
23                 return countBranchesAndBarriersExpr(body, iterationCount * count, counts)
24             otherwise do return counts // Nothing to count;
25     otherwise do return counts // Nothing to count;
updateCounts(in: pattern, in: iterationCount, in: n, inout: counts)
26 if pattern.emitIf then counts.ifBranchCount += iterationCount;
27 else if pattern.emitFor then counts.forBranchCount += iterationCount * n;
28 else // Single statement or unrolled loop, nothing to count;
```

Algorithm 5: Pseudo-code for counting branches resulting from `for` loops and `if` statements and `barrier` primitives resulting from *mapLcl* patterns.

counted in line 26.

For determining the number of barriers, the algorithm only need to look at the occurrences of *mapLcl*. This is the only pattern that might emit a barrier since OpenCL only has local barriers for synchronising threads within a work-group. Global barriers to synchronise across work-groups are "emulated" by multiple kernel launches. The number of encountered barriers is updated in line 13. As described in Chapter 4, the LIFT code generator has an optimisation which detects unnecessary barriers and tags the call to *mapLcl* when it is not required. Therefore, this barrier elimination pass is run before feature extraction and this information is used to ignore the *mapLcl* which have been marked as not requiring a barrier.

6.3.4 Summary

This section has shown how low-level GPU-specific features are extracted from the LIFT IR. Memory-related features, control flow and synchronisation features, are extracted using information about the length of arrays stored in the type. In addition, it has shown how the fine-grained memory feature related to cache lines accesses can be computed using the power of the LIFT symbolic arithmetic expressions. The next section explains how a performance model can be built using these features.

6.4 Performance Model

Having seen how hardware-specific features are extracted from the high-level LIFT IR, this section now focuses on how to build a performance predictor. A performance model based on k-nearest neighbours (kNN) is chosen. A kNN model makes prediction based on the distance between programs in the feature space. Intuitively, LIFT expressions that exhibit similar features are likely to have similar performance.

6.4.1 Output Variable

The prediction output is throughput ($1/time$) normalised per input/program by the maximum achievable. This is done to ensure that performance is comparable across programs since different programs might exhibit different number of operations.

6.4.2 Principal Component Analysis

Given that a kNN model works best with a small number of features, PCA (Principal Component Analysis) is used to reduce the dimensionality of the feature space. Prior to applying PCA, the features are first normalised. This step is necessary since different features have very different ranges of values, and the same feature can have a very different range of values between different input sizes.

Features, such as the amount of global memory used and the number of global threads are first normalised with respect to the input size. The features are then centred and scaled, such that each feature has a mean of 0 and a standard deviation of 1. Finally, PCA is applied and the principal components that explain 95% of the variance are retained. In effect, this compresses the feature space by removing redundant features.

6.4.3 K-Nearest Neighbours Model

A k-nearest neighbours model makes a prediction of a new data point by finding the k closest points to it using Euclidean distance and averaging their responses to make a prediction. In this case, the distance metric is determined by how close the feature vectors are from one another.

The kNN model does not require any special training. The execution time of rewritten LIFT expressions, together with their features, are simply collected and added into a database. To predict the performance of a newly unseen LIFT expression or even a new program, the k closest neighbours are simply looked up and their responses are averaged to form a new prediction. In the experiments in this chapter, $k = 5$ is used.

6.4.4 Making Predictions

To be able to make prediction about new programs, data points from a group of training programs are first collected. For each program, an exploration of their optimisation space is conducted and the features and corresponding performance are stored.

Given a new program, the following procedure is used to make a prediction:

1. For each rewritten program:
   (a) The features are extracted, normalised and projected based on the PCA calculated from the training data;
   (b) The model predicts the performance using the average of the k-nearest neighbours.
2. The predictions are used to sort the different rewritten programs
3. The fastest predicted rewritten program is generated, compiled and executed

6.5 Experimental Setup

Platform The experimental setup consists of two GPUs, an NVIDIA Titan Black and an AMD Radeon R9 295X2. The Nvidia platform uses driver version 367.35 and OpenCL 1.2 (CUDA 8.0.0). The AMD platform uses OpenCL 2.0 AMD-APP (1598.5).

Benchmark	Points in Neighbourhood	Points Used	# grids
Stencil2D	9	9	1
SRAD1	9	5	1
SRAD2	9	3	2
Hotspot2D	9	5	2
Gradient	9	5	1
Jacobi2D 5 pt	9	5	1
Jacobi2D 9 pt	9	9	1
Gaussian	25	25	1

Table 6.2: Stencil benchmarks used in the evaluation along with some of their characteristics.

Benchmarks and Space All the 2D stencil benchmarks from [Hage 18] are used and they are listed in Table 6.2. All experiments are performed using single precision floating point numbers with matrix sizes from 512^2 to 8192^2.

Model evaluation The performance model is evaluated using leave-one-out cross-validation, a standard evaluation methodology for machine learning techniques. When evaluating the performance of the model a given benchmark, the traning data consists of all the data collected from all benchmarks, except the one being evaluated (it is left-out). This guarantees that the model is never tested on data used for training.

6.6 Feature and Model Analysis

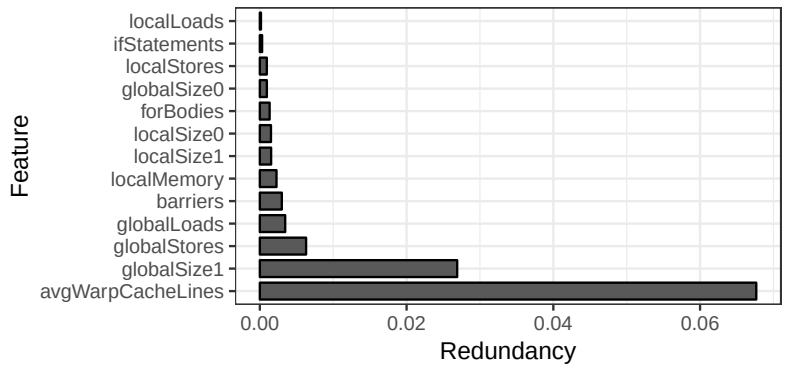
Before looking at how the performance model is used to speed up the optimisation space exploration, an analysis of the features is first performed and the accuracy of the model's predictions are evaluated

6.6.1 Features Analysis

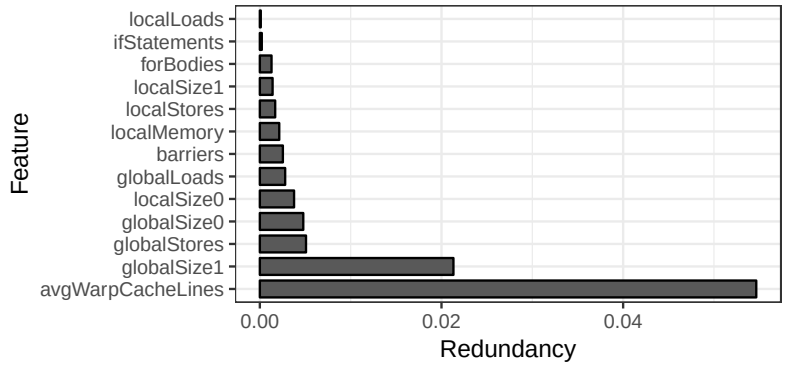
The redundancy metric (R) is used to analyse which features are the most informative about performance:

$$R = \frac{I(X,Y)}{H(X)+H(Y)}$$

The redundancy metric normalises the mutual information (I) by the sum of the entropy (H) of the two variables, X and Y. This ensures that different features can be compared with one another.



(a) NVIDIA



(b) AMD

Figure 6.3: Normalised mutual information (redundancy) between each feature and performance.

Intuitively, mutual information quantifies how much information observing one variable gives about another variable. In this case, each feature is compared with the output to predict: performance. A higher value between a certain feature and the output indicates that the feature is going to be very useful for making predictions.

Figure 6.3 shows the normalised mutual information between the feature and performance for NVIDIA and AMD GPUs. As expected, one of the most important features on both platform is the average number of cache lines accessed per warp. This feature, which represents locality, is extremely important for stencil benchmarks.

The next most important feature for both machines is the globalSize in dimension 1. This feature is directly related to the number of threads that execute and, therefore, the amount of parallel work performed. It is also used to determine if the kernels are launched using a 2D or 1D iteration space (in the 1D case, the globalSize1 will be 1). Then, comes the number of global stores, followed closely by the number of global loads. This basically corresponds to the number of memory accesses performed into the slow global memory.

For both platforms, barriers and control flow (for loops) seem to have only a medium

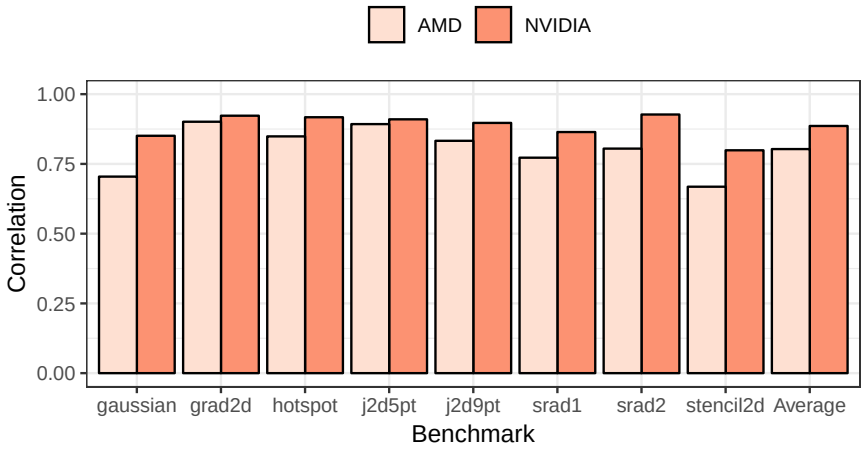


Figure 6.4: Correlation between the predicted and actual throughput when predicting a new program.

relation to performance, whereas the number of if-statements does not seem very relevant at all. Focusing on the least important features, the number of local loads does not seem to have much relation to performance. One explanation could be that since the local memory is anyway very fast, having fewer or more local loads might not make much of a difference in terms of performance, especially compared to the number of global memory operations.

6.6.2 Performance Model Correlation

This section now analyses the accuracy of the model using the coefficient of correlation. Figure 6.4 shows the correlation between the predicted throughput and the actual throughput, averaged across input sizes. As can be seen, for all programs the correlation coefficient is very high in the range $[0.7 - 0.9]$, which is a sign that the predictor works as expected. On average, the model achieves a correlation of 0.9 on Nvidia and 0.8 on AMD.

6.6.3 Summary

This section has shown that the most important features for performance prediction on the GPU are related to memory access pattern, amount of parallelism and number of global memory accesses. The section has also shown that the model's predictions correlate highly with the actual performance. The next section will show how the model can be used to speed up the optimisation space exploration of the benchmarks.

6.7 Optimisation Space Exploration

The previous section showed that the performance model makes reasonably accurate predictions. This section will describe how the performance model is used to drive the exploration process and speed it up but first, it will characterise the optimisation space for the different benchmark, input size and GPU combinations.

6.7.1 Optimisation Space Characterisation

The space exploration is conducted by generating transformed LIFT expression using rewrite rules and combining them with different thread-counts. This leads to up to 10,000 design points per program and input size combination.

Figure 6.5 shows a density plot of the normalised performance for all design points for all program, input size and GPU combinations. The figure show that in most cases, only a small number of kernels achieve good performance. It also shows that on all programs on AMD, fewer kernels achive good performance than on NVIDIA.

6.7.2 Model-Based Exploration

This section shows how the optimisation space is explored with the help of the predictor and how it speeds up finding good points in the space.

To use the performance model to drive the exploration, the OpenCL specific program variations need to be created so the features could be extracted. The model is then queried for a performance prediction for all the variants in the optimisation space so they could be ranked from the best predicted performance to the worst. If the model was perfectly accurate, then only the best predicted point could be picked, code generated for it and then executed, as shown in Figure 6.1b. However, as the predictions made by the model are not quite that accurate a few points might need to be evaluated, starting from the one with the best predicted performance, to find a program variant that performs well.

Figure 6.6 shows the normalised best performance achieved so far as a function of the number of points evaluated using the K-NN driven exploration, as well as the performance achieved using a purely random evaluation order. As can be seen, using the predictor, it is possible to very quickly achieve 100% of the performance available in the space for all programs. In comparison, the random strategy struggles to reach even 50% of the performance available in some cases after having explored 3% of the whole space.

6.7.3 Space Exploration Speedups

Figure 6.7 shows the exploration speedup achived when using the model compared to random search to achieve 90% of the available performance in the design space. A speedup of $10x$

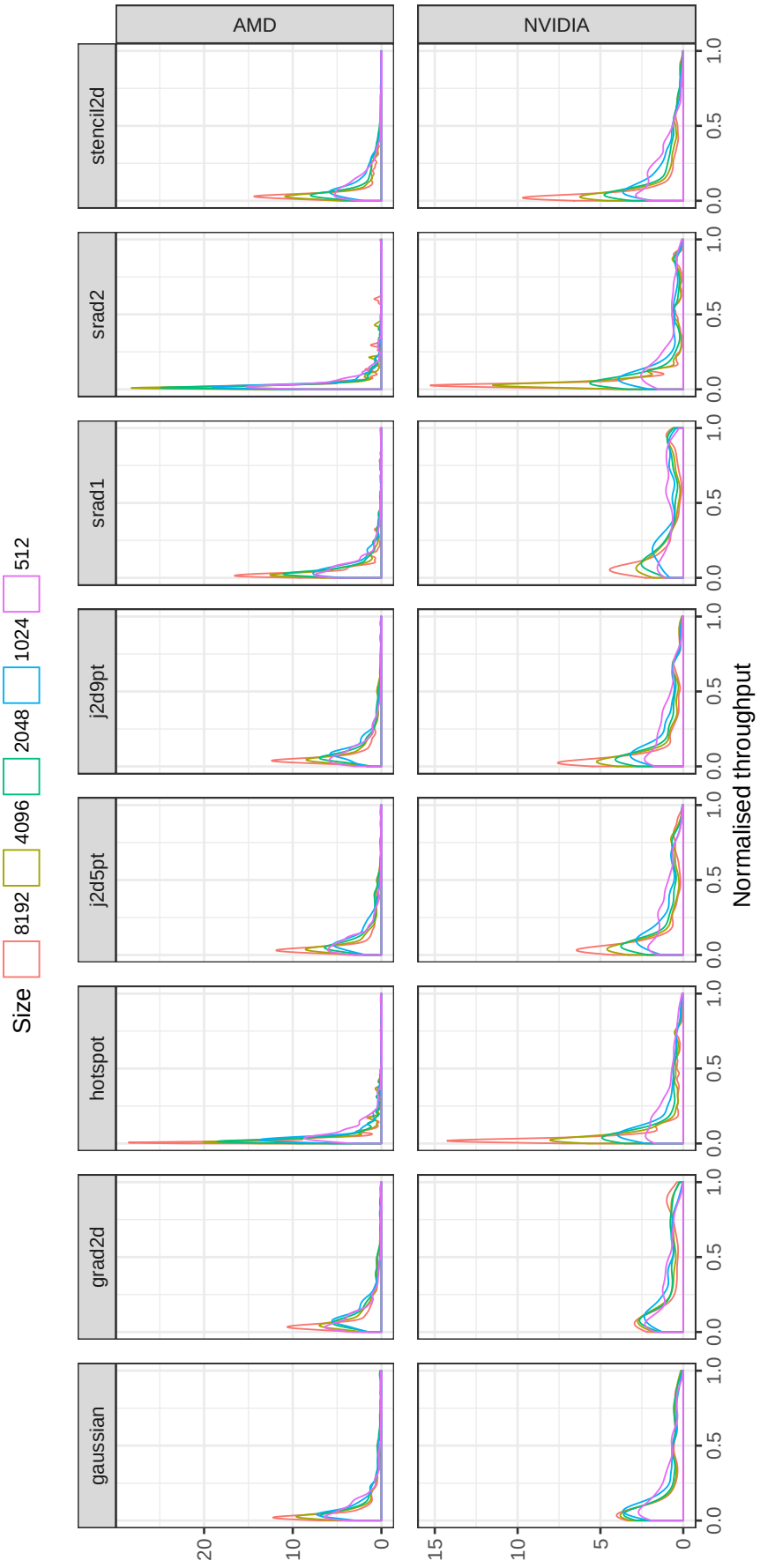


Figure 6.5: Performance distribution of kernels for all programs, input sizes and both GPUs.

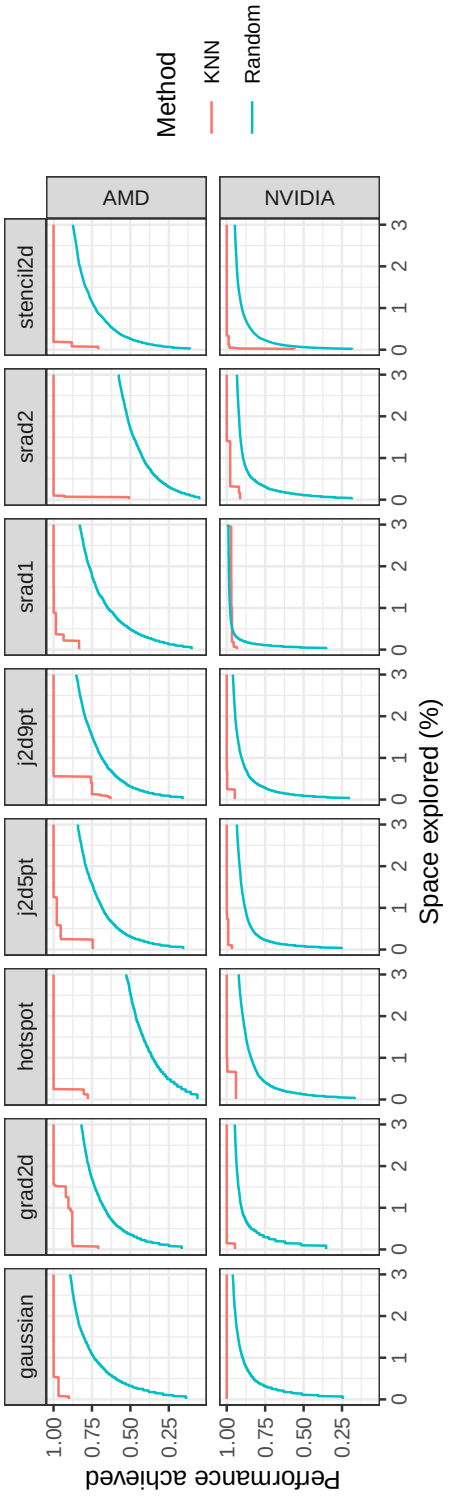
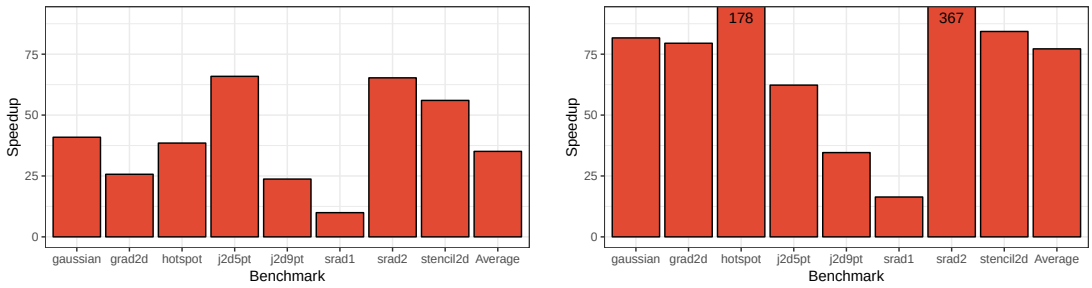
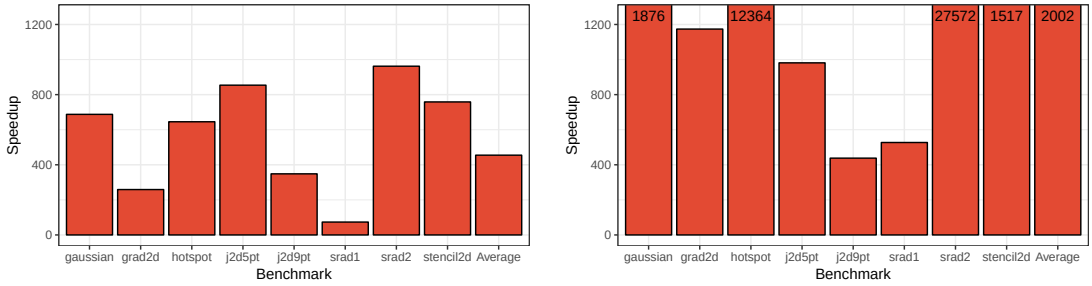


Figure 6.6: Achieved performance when exploring the space for a 4K input size using a model trained on other programs.



(a) NVIDIA (b) AMD

Figure 6.7: The reduction in the amount of the search space needed to explore to reach at least 90% of the available performance.

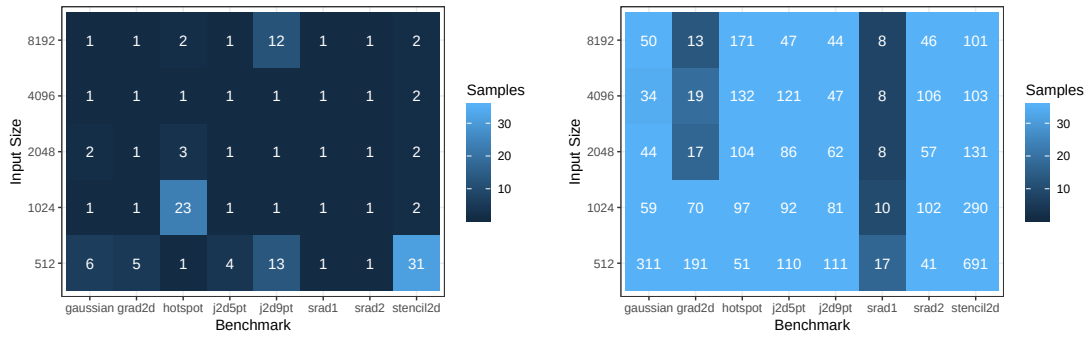


(a) NVIDIA (b) AMD

Figure 6.8: The reduction in the amount of the kernel execution time needed to reach at least 90% of the available performance.

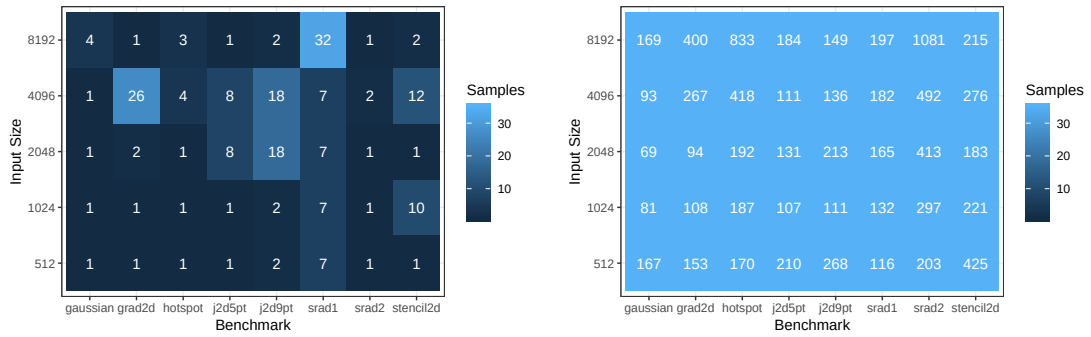
means that the performance model needs $10x$ less runs than random to achieve 90% of the performance. As can be seen, using the performance model brings large speedup across all programs. The per program speedups are calculated as the geometric means of speedups across all input sizes. On Nvidia, using the performance model requires $37x$ less runs than random sampling. On AMD, there are even bigger savings, since the model requires $75x$ less runs than random samplig.

Figure 6.8 shows the speedup of the kernel execution time needed to execute the required number of samples to reach 90% of the available performance when using the model for exploration compared to random sampling. The kernel execution time speedups are even higher, as the model not only requires less runs than random sampling but also manages to avoid picking the worst performing points in the space. On Nvidia, using the performance model requires more than $400x$ less time than random sampling. On AMD, the speedup is higher again and the model needs $2000x$ less time than random sampling.



(a) KNN (b) Random

Figure 6.9: The number of samples needed to reach 90% of the available performance on NVIDIA.



(a) KNN (b) Random

Figure 6.10: The number of samples needed to reach 90% of the available performance on AMD.

6.7.4 Detailed Results

Finally, this last section shows more detailed results per program and input size. Figure 6.9 and Figure 6.10 shows the actual number of runs required to reach 90% of the performance across programs and input sizes. As can be seen, only 1 run is necessary in the majority of the cases for Nvidia and 2 for AMD. In contrast, random needs over 60 runs for Nvidia and over 180 for AMD in most cases.

The average number of runs using the model is 3 for Nvidia and 5 for AMD. In comparison, random requires on average 97 runs for Nvidia and 240 for AMD. These results clearly shows that the performance model is working extremely well in the majority of the cases.

Interestingly, there are a couple of outliers programs/input size combination that requires over 30 runs for the model-based approach. In both cases, stencil2d on Nvidia and srad1 on AMD, this is when the largest or smallest input sizes are used. We believe that in such cases, the behaviour of these programs probably changes drastically with the input size. For

instance, the data might actually fit entirely in the cache for the smallest input size of stencil2d and, therefore, change drastically the behaviour of the application for this input size. Since the features have no notion of working-set size, the model might be unable to pick up this change of behaviour. However, even in such cases, the model-based exploration is still ahead of random. For stencil2d, the model needs 31 runs while random needs 691, a 21x speedup!

6.7.5 Evaluation Summary

Overall, the results have demonstrated that the performance model is able to make accurate performance predictions from features extracted from the high-level LIFT IR. The model-based approach outperforms random search in all cases, with an average of 75x exploration speedup on AMD and 37x on Nvidia.

6.8 Conclusion

This chapter has demonstrated that it is possible to extract low-level hardware-specific features from the LIFT high-level functional IR. It has shown how type information, such as array length, is useful for computing certain features. The ability to reason symbolically about array indices also enables the extraction of very fined-grained features such as the number of accessed cache lines per warp. These low-level features can be extracted at very high-level, without requiring any profiling or performance counters.

The chapter also demonstrated how a performance model can be built to make accurate performance predictions about different program variants. Using an Nvidia and AMD GPUs, and stencil applications, we have seen that the model is able to predict points in the space that are within 90% of the best within one or two runs in the majority of the cases. When compared to a random search strategy, the model requires on average 75x less runs than random on AMD and 37x less on Nvidia. When looking at the time needed to execute the kernels needed to reach that level of performance, the gains are even bigger, with a 2000x improvement on AMD and a 400x improvement on NVIDIA.

Chapter 7

Conclusion

This thesis has proposed methods for tackling the performance portability problem using the high-level programming language LIFT, a system of rewrite rules and a performance model. Chapter 4 presented the compilation techniques necessary to produce high-performance imperative code from functional programs written in LIFT. Chapter 5 presented the LIFT rewrite system and how rewrite rules are combined into *macro rules* to express complex optimisations. Chapter 6 presented techniques for extracting GPU specific features from the functional language and using them to build a performance model to accelerate optimisation space exploration. Section 7.1 summarises the main contributions of this thesis. Section 7.2 analyses the limitations of this thesis and discusses possible future extensions to this work.

7.1 Contributions

This section summarises the main contributions of the previous three chapters.

7.1.1 High-Performance GPU Code Generation

Chapter 4 presented techniques for compiling a functional LIFT program with optimisation choices explicitly encoded into highly efficient imperative OpenCL code on par with hand-written versions applying the same optimisations. Performance critical details such as address spaces, memory allocation, array accesses or barriers are not explicitly represented in LIFT but have to be explicitly dealt with in OpenCL code. The presented techniques for address space inference, memory allocation, array access generation and barrier elimination are necessary for generating OpenCL code and making sure it is efficient.

From the results of 11 benchmarks it can be seen that the presented optimisations are crucial to achieving performance on par with handwritten code and without them only a fraction of the performance is reached for complex applications like matrix multiplication.

7.1.2 Creating and Exploring the Optimisation Space with Rewrite Rules

Chapter 5 demonstrated that expressing complex optimisations as sequences of rewrite rules called *macro rules* to make it feasible to explore them automatically. The chapter first looked at how classical optimisations for matrix multiplication are represented in the LIFT language. It then introduced additional rules to be able to apply them in an automated fashion before grouping them together as macro rules to cut through the optimisation space. The chapter introduced an exploration strategy to apply rewrite rules to generate program variants and then explore the options to find high-performing implementations.

Using matrix multiplication as a case study, starting from a single high-level program the compiler automatically generates highly optimised and specialised implementations for desktop and mobile GPUs with very different types of architectures achieving performance portability where existing solutions missed out.

7.1.3 Peformance Prediction for Accelerated Exploration of Optimisation Spaces

Chapter 6 presented building a performance model to be used for exploring the optimisation space created by applying rewrite rules. It showed how to extract low-level GPU specific features about parallelism, memory accesses, synchronisation and control-flow directly from a functional LIFT program without needing to compile it to OpenCL, as existing techniques would require. The chapter then showed how these features are used to build a performance model capable of making accurate predictions about the performance of previously unseen programs.

Using 8 benchmarks the results demonstrated that using the performance model to explore the optimisation space reduces the number of runs needed to achieve good performance by 30 to 75x on average when compared to random search. The results also showed that unlike random search, the model is able to avoid exploring the worst performing points and improving the time needed to execute kernels by even more. On average from 400x to 2000x less kernel execution time is required when using the model compared to using random search.

Together the contributions allow achieving performance portability from a single high-level program expressed in the LIFT language. The high-level program is automatically rewritten into different versions which will be quickly explored using the performance model to find high-performance implementations for different GPUs, resulting in performance on par with highly tuned device-specific libraries.

7.2 Critical Analysis and Future Work

7.2.1 Limitations

While the techniques presented in this thesis are a first important step towards achieving performance portability, there are a number of practical limitations. A few important ones are highlighted here.

The arithmetic simplification for array index generation and barrier elimination work on a case by case basis and consequently, there are cases where the arithmetic simplification or barrier elimination could be improved. A better solution would be to research more formal methods that can give strong guarantees about their outcomes. First work in this direction is described in section 7.2.2

The *macro rules* for tiling and register blocking are quite application specific and sensitive to the program structure while the ideas behind them of splitting dimensions and interchanging the order in which they are mapped over are more general and applicable for a wider range of applications. A possible solution for making *macro rules* more general is described in section 7.2.3.

The accuracy of the k-nearest neighbour based performance model is very sensitive to the input features. If applying it to new hardware with different performance characteristics, it is likely that new features will be required that describe those characteristics. Adding new input features is again likely to affect the performance of the model on existing platforms. A possible approach to solve the problem of choosing input features is described in section 7.2.4.

7.2.2 Formalising Translation to OpenCL

While the rewrite rules are provably correct [Steu 15a], there are no such strong formal guarantees about the translation to OpenCL. Some work has been done towards achieving this by introducing imperative primitives corresponding to the functional ones and providing a formal translation between them along with a correctness proof [Atke 17]. However, at the time of writing, the functionality is not yet at a level where it could be able to replace the approach of Chapter 4 as barrier handling, for example is missing.

7.2.3 DSL for Expressing *Macro Rules*

Chapter 5 showed that grouping rules as *macro rules* enables the expression of complex optimisations like tiling. However, their implementation is complex and hard to understand. A better way to express optimisations as compositions of rewrite rules to form *macro rules* would be to have a separate language for describing them. Composition, conditional application would ease constructing *macro rules* that are flexible enough to handle a variety of situations in real world programs. In addition to making them easier to represent, this would also make it much

easier to reason about them. The language along with meaningful diagnostics would greatly simplify the development of *macro rules*.

7.2.4 Feature Selection for Building Performance Models

New program features are likely to be required for accurate predictions, especially when considering additional platforms and applications from other domains. Features about different memory access patterns, cache behaviour or computational intensity could help generalise the methodology for building performance models for different domains and devices. Since machine learning and specifically k-nearest neighbour are very sensitive to the features used there is need of a system to choose appropriate features from the available ones for a new platform. Further experiments also showed that it is also possible to further improve the performance of the models by selecting a subset of the features but it is not clear how it could be done automatically in a way that transfers to previously unseen programs.

7.2.5 Making LIFT More Suited for Practical Use

While LIFT has been demonstrated to be able to achieve high-performance on a variety of programs, as a research compiler it makes some assumptions that would need to be removed in order to use it in practice. For example, a prevalent assumption is related to choosing split factors and vector widths which need to divide the input array lengths. Array lengths that do not meet that assumption are very likely to crop up in practice and will need to be handled. Possible solutions include padding the inputs to divide the parameters, inserting code to perform computations for elements that are on the boundaries that would currently be ignored, or inserting runtime checks and choosing a kernel with appropriate parameter values.

This thesis has focused on producing high-performance computational kernels but integrating LIFT into a project will require identifying kernels that could be accelerated, writing them in the LIFT language and writing the host code to run the kernels. Some work has been done towards automatically detecting and mapping parts of large legacy applications to LIFT for OpenCL code generation and acceleration based on computational idioms [Gins 18].

Bibliography

[Agak 06] F. V. Agakov, E. V. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. "Using Machine Learning to Focus Iterative Optimization". In: *Fourth IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2006), 26-29 March 2006, New York, New York, USA*, pp. 295–305, IEEE Computer Society, 2006. (Cited on page 42.)

[Aldi 11] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati. "Accelerating Code on Multi-cores with FastFlow". In: E. Jeannot, R. Namyst, and J. Roman, Eds., *Euro-Par 2011 Parallel Processing - 17th International Conference, Euro-Par 2011, Bordeaux, France, August 29 - September 2, 2011, Proceedings, Part II*, pp. 170–181, Springer, 2011. (Cited on page 35.)

[Aldi 12] M. Aldinucci, C. Spampinato, M. Drocco, M. Torquati, and S. Palazzo. "A parallel edge preserving algorithm for salt and pepper image denoising". In: K. Djemal and M. A. Deriche, Eds., *3rd International Conference on Image Processing Theory Tools and Applications, IPTA 2012, 15-18 October 2012, Istanbul, Turkey*, pp. 97–104, IEEE, 2012. (Cited on page 35.)

[AMDI 12] AMD Inc. "AMD Graphics Cores Next (GCN) Architecture Whitepaper". `https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf`, 2012. (Cited on page 10.)

[AMDI 14] AMD Inc. "Bolt C++ Template Library. C++ template library for heterogeneous compute.". `https://github.com/HSA-Libraries/Bolt`, 2014. (Cited on page 35.)

[AMDI 15] AMD Inc. "APP OpenCL Programming Guide". 2015. (Cited on pages 68 and 91.)

[Anse 09] J. Ansel, C. P. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. P. Amarasinghe. "PetaBricks: a language and compiler for algorithmic choice". In: M. Hind and A. Diwan, Eds., *Proceedings of the 2009 ACM SIGPLAN Conference*

on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009, pp. 38–49, ACM, 2009. (Cited on page 43.)

[Anse 14] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U. O'Reilly, and S. P. Amarasinghe. "OpenTuner: an extensible framework for program autotuning". In: J. N. Amaral and J. Torrellas, Eds., *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, pp. 303–316, ACM, 2014. (Cited on page 42.)

[ArmL 13] Arm Limited. "Mali-T600 Series GPU OpenCL – Dev. Guide". 2013. (Cited on page 11.)

[ArmL 19] Arm Limited. "Mali-T628 - ARM". `https://www.arm.com/ja/products/multimedia/mali-cost-efficient-graphics/mali-t628.php`, 2019. (Cited on page 12.)

[Atke 17] R. Atkey, M. Steuwer, S. Lindley, and C. Dubach. "Strategy Preserving Compilation for Parallel Functional Code". *CoRR*, Vol. abs/1710.08332, 2017. (Cited on page 142.)

[Bagh 15] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, S. van Haastregt, A. Kravets, A. Lokhmotov, R. David, and E. Hajiyev. "PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming". In: *2015 International Conference on Parallel Architectures and Compilation, PACT 2015, San Francisco, CA, USA, October 18-21, 2015*, pp. 138–149, IEEE Computer Society, 2015. (Cited on pages 39 and 48.)

[Bagh 19] R. Baghdadi, J. Ray, M. B. Romdhane, E. D. Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. P. Amarasinghe. "Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code". In: M. T. Kandemir, A. Jimborean, and T. Moseley, Eds., *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*, pp. 193–205, IEEE, 2019. (Cited on page 43.)

[Bakh 09] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. "Analyzing CUDA workloads using a detailed GPU simulator". In: *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2009, April 26-28, 2009, Boston, Massachusetts, USA, Proceedings*, pp. 163–174, IEEE Computer Society, 2009. (Cited on page 46.)

[Bask 10] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. "Automatic C-to-CUDA Code Generation for Affine Programs". In: R. Gupta, Ed., *Compiler Construction, 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, pp. 244–263, Springer, 2010. (Cited on pages 39 and 48.)

[Bast 04] C. 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*, pp. 7–16, IEEE Computer Society, 2004. (Cited on page 39.)

[Beau 17] U. Beaugnon, A. Pouille, M. Pouzet, J. A. Pienaar, and A. Cohen. "Optimization space pruning without regrets". In: P. Wu and S. Hack, Eds., *Proceedings of the 26th International Conference on Compiler Construction, Austin, TX, USA, February 5-6, 2017*, pp. 34–44, ACM, 2017. (Cited on page 45.)

[Bell 11] N. Bell and J. Hoberock. "Thrust: A Productivity-Oriented Library for CUDA". In: *GPU Computing Gems Jade Edition*, Morgan Kaufmann, 2011. (Cited on pages 35 and 48.)

[Bilm 97] J. A. Bilmes, K. Asanovic, C. Chin, and J. Demmel. "Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology". In: S. J. Wallach and H. P. Zima, Eds., *Proceedings of the 11th international conference on Supercomputing, ICS 1997, Vienna, Austria, July 7-11, 1997*, pp. 340–347, ACM, 1997. (Cited on page 83.)

[Boul 98] P. Boulet, A. Darte, G. Silber, and F. Vivien. "Loop Parallelization Algorithms: From Parallelism Extraction to Code Generation". *Parallel Computing*, Vol. 24, No. 3-4, pp. 421–444, 1998. (Cited on page 39.)

[Brow 11] K. J. Brown, A. K. Sujeeth, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. "A Heterogeneous Parallel Framework for Domain-Specific Languages". In: L. Rauchwerger and V. Sarkar, Eds., *2011 International Conference on Parallel Architectures and Compilation Techniques, PACT 2011, Galveston, TX, USA, October 10-14, 2011*, pp. 89–100, IEEE Computer Society, 2011. (Cited on pages 41 and 48.)

[Brow 16] K. J. Brown, H. Lee, T. Rompf, A. K. Sujeeth, C. D. Sa, C. R. Aberger, and K. Olukotun. "Have abstraction and eat performance, too: optimized heterogeneous computing with parallel patterns". In: B. Franke, Y. Wu, and F. Rastello,

Eds., *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016, Barcelona, Spain, March 12-18, 2016*, pp. 194–205, ACM, 2016. (Cited on pages 30 and 42.)

[Cao 14] C. Cao, J. J. Dongarra, P. Du, M. Gates, P. Luszczek, and S. Tomov. "clMAGMA: high performance dense linear algebra with OpenCL". In: S. McIntosh-Smith and B. Bergen, Eds., *Proceedings of the International Workshop on OpenCL, IWOCL 2013 & 2014, May 13-14, 2013, Georgia Tech, Atlanta, GA, USA / Bristol, UK, May 12-13, 2014*, pp. 1:1–1:9, ACM, 2014. (Cited on page 85.)

[Cata 11] B. Catanzaro, M. Garland, and K. Keutzer. "Copperhead: compiling an embedded data parallel language". In: C. Cascaval and P. Yew, Eds., *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, pp. 47–56, ACM, 2011. (Cited on pages 36 and 48.)

[Chak 11] M. M. T. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. "Accelerating Haskell array codes with multicore GPUs". In: M. Carro and J. H. Reppy, Eds., *Proceedings of the POPL 2011 Workshop on Declarative Aspects of Multicore Programming, DAMP 2011, Austin, TX, USA, January 23, 2011*, pp. 3–14, ACM, 2011. (Cited on page 48.)

[Chan 16] L. Chang, I. E. Hajj, C. I. Rodrigues, J. Gómez-Luna, and W. W. Hwu. "Efficient kernel synthesis for performance portable programming". In: *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, pp. 12:1–12:13, IEEE Computer Society, 2016. (Cited on page 43.)

[Cole 89] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989. (Cited on page 35.)

[Coll 13] A. Collins, C. Fensch, H. Leather, and M. Cole. "MaSiF: Machine learning guided auto-tuning of parallel skeletons". In: *20th Annual International Conference on High Performance Computing, HiPC 2013, Bengaluru (Bangalore), Karnataka, India, December 18-21, 2013*, pp. 186–195, IEEE Computer Society, 2013. (Cited on page 46.)

[Coll 14] A. Collins, D. Grewe, V. Grover, S. Lee, and A. Susnea. "NOVA: A Functional Language for Data Parallelism". In: L. J. Hendren, A. Rubinsteyn, M. Sheeran, and J. Vitek, Eds., *ARRAY'14: Proceedings of the 2014 ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming,*

Edinburgh, United Kingdom, June 12-13, 2014, pp. 8–13, ACM, 2014. (Cited on pages 37 and 49.)

[Demm 12] J. Demme and S. Sethumadhavan. "Approximate graph clustering for program characterization". *TACO*, Vol. 8, No. 4, pp. 21:1–21:21, 2012. (Cited on page 45.)

[Denn 74] R. H. Dennard, F. H. Gaensslen, H. nien Yu, V. L. Rideout, E. Bassous, Andre, and R. Leblanc. "Design of ion-implanted MOSFETs with very small physical dimensions". *IEEE Journal of Solid-State Circuits*, Vol. 9, No. 5, pp. 256–268, 1974. (Cited on page 1.)

[Diam 10] G. F. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. "Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems". In: V. Salapura, M. Gschwind, and J. Knoop, Eds., *19th International Conference on Parallel Architectures and Compilation Techniques, PACT 2010, Vienna, Austria, September 11-15, 2010*, pp. 353–364, ACM, 2010. (Cited on page 45.)

[Dong 14] J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and I. Yamazaki. *Numerical Computations with GPUs*, Chap. Accelerating Numerical Dense Linear Algebra Calculations with GPUs. Springer Intl. Publishing, 2014. (Cited on page 84.)

[Duba 07] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, and O. Temam. "Fast compiler optimisation evaluation using code-feature based performance prediction". In: U. Banerjee, J. Moreira, M. Dubois, and P. Stenström, Eds., *Proceedings of the 4th Conference on Computing Frontiers, 2007, Ischia, Italy, May 7-9, 2007*, pp. 131–142, ACM, 2007. (Cited on page 45.)

[Duba 12] C. Dubach, P. Cheng, R. M. Rabbah, D. F. Bacon, and S. J. Fink. "Compiling a high-level language for GPUs: (via language support for architectures and compilers)". In: J. Vitek, H. Lin, and F. Tip, Eds., *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pp. 1–12, ACM, 2012. (Cited on pages 36 and 48.)

[Enmy 10] J. Enmyren and C. 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*, Baltimore, Maryland, USA, pp. 5–14, ACM, 2010. (Cited on page 35.)

[Erns 12] S. Ernsting and H. Kuchen. "Algorithmic skeletons for multi-core, multi-GPU systems and clusters". *IJHPCN*, Vol. 7, No. 2, pp. 129–138, 2012. (Cited on page 35.)

[Frig 05] M. Frigo and S. G. Johnson. "The Design and Implementation of FFTW3". *Proceedings of the IEEE*, Vol. 93, No. 2, pp. 216–231, 2005. (Cited on page 42.)

[Gins 18] P. Ginsbach, T. Remmelg, M. Steuwer, B. Bodin, C. Dubach, and M. F. P. O'Boyle. "Automatic Matching of Legacy Code to Heterogeneous APIs: An Idiomatic Approach". In: X. Shen, J. Tuck, R. Bianchini, and V. Sarkar, Eds., *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, pp. 139–153, ACM, 2018. (Cited on page 143.)

[Gonz 19] S. G. D. Gonzalo, S. Huang, J. Gómez-Luna, S. D. Hammond, O. Mutlu, and W. Hwu. "Automatic Generation of Warp-Level Primitives and Atomic Instructions for Fast and Portable Parallel Reduction on GPUs". In: M. T. Kandemir, A. Jimborean, and T. Moseley, Eds., *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*, pp. 73–84, IEEE, 2019. (Cited on page 43.)

[Gras 14] I. Grasso, P. Radojkovic, N. Rajovic, I. Gelado, and A. Ramírez. "Energy Efficient HPC on Embedded SoCs: Optimization Techniques for Mali GPU". IEEE, 2014. (Cited on page 11.)

[Grel 06] C. Grelck and S. Scholz. "SAC - A Functional Array Language for Efficient Multithreaded Execution". *International Journal of Parallel Programming*, Vol. 34, No. 4, pp. 383–427, 2006. (Cited on page 36.)

[Grew 13] D. Grewe, Z. Wang, and M. F. P. O'Boyle. "Portable mapping of data parallel programs to OpenCL for heterogeneous systems". In: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013*, pp. 22:1–22:10, IEEE Computer Society, 2013. (Cited on page 38.)

[Gron 14] J. Gronqvist and A. Lokhmotov. "Optimising OpenCL kernels for the ARM Mali-T600 GPUs". In: *GPU Pro 5: Advanced Rendering Techniques*, A K Peters/CRC Press, 2014. (Cited on pages 11, 84, 86, and 87.)

[Gros 12] T. Grosser, A. Größlinger, and C. Lengauer. "Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation". *Parallel Processing Letters*, Vol. 22, No. 4, 2012. (Cited on pages 39 and 48.)

[Guo 11] J. Guo, J. Thiyagalingam, and S. Scholz. “Breaking the GPU programming barrier with the auto-parallelising SAC compiler”. In: M. Carro and J. H. Reppy, Eds., *Proceedings of the POPL 2011 Workshop on Declarative Aspects of Multicore Programming, DAMP 2011, Austin, TX, USA, January 23, 2011*, pp. 15–24, ACM, 2011. (Cited on page 36.)

[Hage 18] B. Hagedorn, L. Stoltzfus, M. Steuwer, S. Gorlatch, and C. Dubach. “High performance stencil code generation with lift”. In: J. Knoop, M. Schordan, T. Johnson, and M. F. P. O’Boyle, Eds., *Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, Vösendorf / Vienna, Austria, February 24-28, 2018*, pp. 100–112, ACM, 2018. (Cited on page 131.)

[Haid 16] M. Haidl, M. Steuwer, T. Humernbrum, and S. Gorlatch. “Multi-stage programming for GPUs in C++ using PACXX”. In: D. R. Kaeli and J. Cavazos, Eds., *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit, GPGPU@PPoPP 2016, Barcelona, Spain, March 12 - 16, 2016*, pp. 32–41, ACM, 2016. (Cited on page 39.)

[Henn 19] J. L. Hennessy and D. A. Patterson. “A new golden age for computer architecture”. pp. 48–60, 2019. (Cited on page 1.)

[Henr 14] T. Henriksen, M. Elsman, and C. E. Oancea. “Size slicing: a hybrid approach to size inference in futhark”. In: J. Berthold, M. Sheeran, and R. Newton, Eds., *Proceedings of the 3rd ACM SIGPLAN workshop on Functional high-performance computing, FHPC@ICFP 2014, Gothenburg, Sweden, September 4, 2014*, pp. 31–42, ACM, 2014. (Cited on page 37.)

[Henr 17] T. Henriksen, N. G. W. Serup, M. Elsman, F. Henglein, and C. E. Oancea. “Futhark: purely functional GPU-programming with nested parallelism and in-place array updates”. In: A. Cohen and M. T. Vechev, Eds., *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pp. 556–571, ACM, 2017. (Cited on page 37.)

[Hong 09] S. Hong and H. Kim. “An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness”. In: S. W. Keckler and L. A. Barroso, Eds., *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, pp. 152–163, ACM, 2009. (Cited on pages 44 and 117.)

[Hong 18] C. Hong, A. Sukumaran-Rajam, J. Kim, P. S. Rawat, S. Krishnamoorthy, L.-N. Pouchet, F. Rastello, and P. Sadayappan. "GPU Code Optimization Using Abstract Kernel Emulation and Sensitivity Analysis". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Philadelphia, PA, USA, pp. 736–751, ACM, 2018. (Cited on pages 45 and 117.)

[ISO 17] ISO. *ISO/IEC 14882:2017 Information technology — Programming languages — C++*. fifth Ed., 2017. (Cited on page 36.)

[Jia 12] W. Jia, K. A. Shaw, and M. Martonosi. "Stargazer: Automated regression-based GPU design space exploration". In: R. Balasubramonian and V. Srinivasan, Eds., *2012 IEEE International Symposium on Performance Analysis of Systems & Software, New Brunswick, NJ, USA, April 1-3, 2012*, pp. 2–13, IEEE Computer Society, 2012. (Cited on page 46.)

[Jia 13] W. Jia, K. A. Shaw, and M. Martonosi. "Starchart: Hardware and software optimization using recursive partitioning regression trees". In: C. Fensch, M. F. P. O'Boyle, A. Seznec, and F. Bodin, Eds., *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, United Kingdom, September 7-11, 2013*, pp. 257–267, IEEE Computer Society, 2013. (Cited on page 46.)

[Jord 13] H. Jordan, S. Pellegrini, P. Thoman, K. Kofler, and T. Fahringer. "INSPIRE: The insieme parallel intermediate representation". In: C. Fensch, M. F. P. O'Boyle, A. Seznec, and F. Bodin, Eds., *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, United Kingdom, September 7-11, 2013*, pp. 7–17, IEEE Computer Society, 2013. (Cited on page 41.)

[Karr 11] R. Karrenberg and S. Hack. "Whole-function vectorization". In: *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011*, pp. 141–150, IEEE Computer Society, 2011. (Cited on page 23.)

[Kerr 10] A. Kerr, G. F. Diamos, and S. Yalamanchili. "Modeling GPU-CPU workloads and systems". In: D. R. Kaeli and M. Leeser, Eds., *Proceedings of 3rd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU 2010, Pittsburgh, Pennsylvania, USA, March 14, 2010*, pp. 31–42, ACM, 2010. (Cited on page 45.)

[Khro 12] Khronos OpenCL Working Group. "The OpenCL Specification, Version: 1.2, Document Revision: 19". https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf, 2012. (Cited on pages 13, 14, and 15.)

[Khro 14] Khronos Group. "The SPIR Specification: Standard Portable Intermediate Representation, Version 1.2". https://www.khronos.org/registry/SPIR/specs/spir_spec-1.2.pdf, 2014. (Cited on page 41.)

[Khro 19a] Khronos Group. "The OpenGL Graphics System: A Specification". http://khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf, 2019. (Cited on page 41.)

[Khro 19b] Khronos Group. "SPIR-V Specification, Version 1.4, Revision 1, Unified". https://www.khronos.org/registry/spir-v/specs/unified1/SPIRV.pdf, 2019. (Cited on page 41.)

[Khro 19c] Khronos OpenCL Working Group - SYCL subgroup. "SYCL Specification: SYCL integrates OpenCL devices with modern C++, Version: 1.2.1, Document Revision: 5". https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf, 2019. (Cited on page 37.)

[Khro 19d] Khronos Vulkan Working Group. "Vulkan 1.1.109 - A Specification". https://www.khronos.org/registry/vulkan/specs/1.1/pdf/vkspec.pdf, 2019. (Cited on page 41.)

[Kim 11] Y. Kim and A. Shrivastava. "CuMAPz: a tool to analyze memory access patterns in CUDA". In: L. Stok, N. D. Dutt, and S. Hassoun, Eds., *Proceedings of the 48th Design Automation Conference, DAC 2011, San Diego, California, USA, June 5-10, 2011*, pp. 128–133, ACM, 2011. (Cited on page 44.)

[Kisu 00] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. "Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation". In: *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques (PACT'00), Philadelphia, Pennsylvania, USA, October 15-19, 2000*, pp. 237–248, IEEE Computer Society, 2000. (Cited on page 42.)

[Kjol 17] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. P. Amarasinghe. "The tensor algebra compiler". *PACMPL*, Vol. 1, No. OOPSLA, pp. 77:1–77:29, 2017. (Cited on page 40.)

[Lai 13] J. Lai and A. Seznec. "Performance upper bound analysis and optimization of SGEMM on Fermi and Kepler GPUs". In: *Proceedings of the 2013 IEEE/ACM In-*

ternational Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013, pp. 4:1–4:10, IEEE Computer Society, 2013. (Cited on page 112.)

[Latt 04] C. Lattner and V. S. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pp. 75–88, IEEE Computer Society, 2004. (Cited on page 41.)

[Lee 09] S. Lee, S. Min, and R. Eigenmann. "OpenMP to GPGPU: a compiler framework for automatic translation and optimization". In: D. A. Reed and V. Sarkar, Eds., *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2009, Raleigh, NC, USA, February 14-18, 2009*, pp. 101–110, ACM, 2009. (Cited on page 38.)

[Lee 14] H. Lee, K. J. Brown, A. K. Sujeeth, T. Rompf, and K. Olukotun. "Locality-Aware Mapping of Nested Parallel Patterns on GPUs". In: *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom, December 13-17, 2014*, pp. 63–74, IEEE Computer Society, 2014. (Cited on page 41.)

[Lee 15] S. Lee, J. S. Meredith, and J. S. Vetter. "COMPASS: A Framework for Automated Performance Modeling and Prediction". In: L. N. Bhuyan, F. Chong, and V. Sarkar, Eds., *Proceedings of the 29th ACM on International Conference on Supercomputing, ICS'15, Newport Beach/Irvine, CA, USA, June 08 - 11, 2015*, pp. 405–414, ACM, 2015. (Cited on page 45.)

[Leis 15] R. Leißa, M. Köster, and S. Hack. "A graph-based higher-order intermediate representation". In: K. Olukotun, A. Smith, R. Hundt, and J. Mars, Eds., *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015, San Francisco, CA, USA, February 07 - 11, 2015*, pp. 202–212, IEEE Computer Society, 2015. (Cited on pages 28 and 41.)

[Leis 18] R. Leißa, K. Boesche, S. Hack, A. Pérard-Gayot, R. Membarth, P. Slusallek, A. Müller, and B. Schmidt. "AnyDSL: a partial evaluation framework for programming high-performance libraries". *PACMPL*, Vol. 2, No. OOPSLA, pp. 119:1–119:30, 2018. (Cited on page 41.)

[Lutz 14] T. Lutz and V. Grover. "LambdaJIT: a dynamic compiler for heterogeneous optimizations of STL algorithms". In: J. Berthold, M. Sheeran, and R. Newton, Eds., *Proceedings of the 3rd ACM SIGPLAN workshop on Functional high-*

performance computing, FHPC@ICFP 2014, Gothenburg, Sweden, September 4, 2014, pp. 99–108, ACM, 2014. (Cited on page 38.)

[Mado 16] S. Madougou, A. L. Varbanescu, and C. de Laat. "Using colored petri nets for GPGPU performance modeling". In: G. Palermo and J. Feo, Eds., *Proceedings of the ACM International Conference on Computing Frontiers, CF'16, Como, Italy, May 16-19, 2016*, pp. 240–249, ACM, 2016. (Cited on page 45.)

[Maie 16] P. Maier, J. M. Morton, and P. Trinder. "JIT costing adaptive skeletons for performance portability". In: D. Duke and Y. Kameyama, Eds., *Proceedings of the 5th International Workshop on Functional High-Performance Computing, FHPC@ICFP 2016, Nara, Japan, September 22, 2016*, pp. 23–30, ACM, 2016. (Cited on page 30.)

[Mats 12] K. Matsumoto, N. Nakasato, and S. G. Sedukhin. "Performance Tuning of Matrix Multiplication in OpenCL on Different GPUs and CPUs". In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis, Salt Lake City, UT, USA, November 10-16, 2012*, pp. 396–405, IEEE Computer Society, 2012. (Cited on pages 82 and 85.)

[McCo 12] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming*. Morgan Kaufmann Publishers Inc., 1st Ed., 2012. (Cited on page 35.)

[McDo 13] T. L. McDonell, M. M. T. Chakravarty, G. Keller, and B. Lippmeier. "Optimising purely functional GPU programs". In: G. Morrisett and T. Uustalu, Eds., *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pp. 49–60, ACM, 2013. (Cited on pages 37 and 48.)

[McKe 69] A. C. McKellar and E. G. C. Jr. "Organizing Matrices and Matrix Operations for Paged Memory Systems". *Commun. ACM*, Vol. 12, No. 3, pp. 153–165, 1969. (Cited on pages 82 and 85.)

[Meng 11] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath, and T. D. Uram. "GROPHECY: GPU performance projection from CPU code skeletons". In: S. Lathrop, J. Costa, and W. Kramer, Eds., *Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12-18, 2011*, pp. 14:1–14:11, ACM, 2011. (Cited on page 44.)

[Miku 14] D. Mikushin, N. Likhogrud, E. Z. Zhang, and C. Bergstrom. "KernelGen - The Design and Implementation of a Next Generation Compiler Platform for Accelerating Numerical Models on GPUs". In: *2014 IEEE International Parallel &*

Distributed Processing Symposium Workshops, Phoenix, AZ, USA, May 19-23, 2014, pp. 1011–1020, IEEE Computer Society, 2014. (Cited on pages 39 and 48.)

[Moll 16] S. Moll, J. Doerfert, and S. Hack. "Input space splitting for OpenCL". In: A. Zaks and M. V. Hermenegildo, Eds., *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pp. 251–260, ACM, 2016. (Cited on page 39.)

[Moor 65] G. E. Moore. "Cramming more components onto integrated circuits". *Electronics 38.8*, 1965. (Cited on page 1.)

[Mull 16] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian. "Automatically scheduling halide image processing pipelines". *ACM Trans. Graph.*, Vol. 35, No. 4, pp. 83:1–83:11, 2016. (Cited on page 43.)

[Nugt 12] C. Nugteren and H. Corporaal. "The boat hull model: enabling performance prediction for parallel computing prior to code development". In: J. Feo, P. Faraboschi, and O. Villa, Eds., *Proceedings of the Computing Frontiers Conference, CF'12, Caligari, Italy - May 15 - 17, 2012*, pp. 203–212, ACM, 2012. (Cited on pages 44 and 117.)

[Nugt 14a] C. Nugteren and H. Corporaal. "Bones: An Automatic Skeleton-Based C-to-CUDA Compiler for GPUs". *TACO*, Vol. 11, No. 4, pp. 35:1–35:25, 2014. (Cited on page 38.)

[Nugt 14b] C. Nugteren, G. van den Braak, H. Corporaal, and H. E. Bal. "A detailed GPU cache model based on reuse distance theory". In: *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*, pp. 37–48, IEEE Computer Society, 2014. (Cited on pages 44 and 117.)

[Nugt 15] C. Nugteren and V. Codreanu. "CLTune: A Generic Auto-Tuner for OpenCL Kernels". In: *IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip, MCSoC 2015, Turin, Italy, September 23-25, 2015*, pp. 195–202, IEEE Computer Society, 2015. (Cited on pages 42, 83, and 112.)

[NVID 14] NVIDIA Corporation. "NVIDIAs Next Generation CUDA Compute Architecture: Kepler GK110/210". `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf`, 2014. (Cited on page 9.)

[Ofen 13] G. Ofenbeck, T. Rompf, A. Stojanov, M. Odersky, and M. Püschel. "Spiral in scala: towards the systematic construction of generators for performance libraries". In: J. Järvi and C. Kästner, Eds., *Generative Programming: Concepts and Experiences, GPCE'13, Indianapolis, IN, USA - October 27 - 28, 2013*, pp. 125–134, ACM, 2013. (Cited on page 43.)

[Park 11] E. Park, L. Pouchet, J. Cavazos, A. Cohen, and P. Sadayappan. "Predictive modeling in a polyhedral optimization space". In: *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011*, pp. 119–129, IEEE Computer Society, 2011. (Cited on page 45.)

[Park 12] E. Park, J. Cavazos, and M. A. Alvarez. "Using graph-based program characterization for predictive modeling". In: C. Eidt, A. M. Holler, U. Srinivasan, and S. P. Amarasinghe, Eds., *10th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2012, San Jose, CA, USA, March 31 - April 04, 2012*, pp. 196–206, ACM, 2012. (Cited on page 45.)

[Peyt 01] S. Peyton Jones, A. Tolmach, and T. Hoare. "Playing by the rules: rewriting as a practical optimisation technique in GHC". In: *Haskell Workshop*, 2001. (Cited on page 24.)

[Phot 13] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. P. Amarasinghe. "Portable performance on heterogeneous architectures". In: V. Sarkar and R. Bodík, Eds., *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pp. 431–444, ACM, 2013. (Cited on pages 42 and 43.)

[Pric 15] J. Price and S. McIntosh-Smith. "Improving Auto-Tuning Convergence Times with Dynamically Generated Predictive Performance Models". In: *IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip, MCSoC 2015, Turin, Italy, September 23-25, 2015*, pp. 211–218, IEEE Computer Society, 2015. (Cited on page 46.)

[Pusc 05] M. Püschel, J. M. F. Moura, J. R. Johnson, D. A. Padua, M. M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. "SPIRAL: Code Generation for DSP Transforms". *Proceedings of the IEEE*, Vol. 93, No. 2, pp. 232–275, 2005. (Cited on page 43.)

[Raga 13] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. P. Amarasinghe. "Halide: a language and compiler for optimizing parallelism, locality, and

recomputation in image processing pipelines". In: H. Boehm and C. Flanagan, Eds., *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pp. 519–530, ACM, 2013. (Cited on pages 36, 43, and 48.)

[Rasc 17] A. Rasch, M. Haidl, and S. Gorlatch. "ATF: A Generic Auto-Tuning Framework". In: *19th IEEE International Conference on High Performance Computing and Communications; 15th IEEE International Conference on Smart City; 3rd IEEE International Conference on Data Science and Systems, HPCC/SmartCity/DSS 2017, Bangkok, Thailand, December 18-20, 2017*, pp. 64–71, 2017. (Cited on page 42.)

[Romp 12] T. Rompf and M. Odersky. "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs". *Commun. ACM*, Vol. 55, No. 6, pp. 121–130, 2012. (Cited on page 41.)

[Rote 18] N. Rotem, J. Fix, S. Abdulrasool, S. Deng, R. Dzhabarov, J. Hegeman, R. Levenstein, B. Maher, N. Satish, J. Olesen, J. Park, A. Rakhov, and M. Smelyanskiy. "Glow: Graph Lowering Compiler Techniques for Neural Networks". *CoRR*, Vol. abs/1805.00907, 2018. (Cited on page 40.)

[Sim 12] J. Sim, A. Dasgupta, H. Kim, and R. W. Vuduc. "A performance analysis framework for identifying potential benefits in GPGPU applications". In: J. Ramanujam and P. Sadayappan, Eds., *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*, pp. 11–22, ACM, 2012. (Cited on page 44.)

[Spam 14] D. G. Spampinato and M. Püschel. "A Basic Linear Algebra Compiler". In: D. R. Kaeli and T. Moseley, Eds., *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, February 15-19, 2014*, p. 23, ACM, 2014. (Cited on page 43.)

[Steu 11] M. Steuwer, P. Kegel, and S. Gorlatch. "SkelCL - A Portable Skeleton Library for High-Level GPU Programming". In: *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May 2011 - Workshop Proceedings*, pp. 1176–1182, IEEE, 2011. (Cited on pages 35 and 48.)

[Steu 15a] M. Steuwer. *Improving programmability and performance portability on many-core processors*. PhD thesis, Universität Münster, 2015. (Cited on pages 24 and 142.)

[Steu 15b] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach. "Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code". In: K. Fisher and J. H. Reppy, Eds., *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pp. 205–217, ACM, 2015. (Cited on pages 3, 4, 18, 30, 78, 79, and 80.)

[Steu 17] M. Steuwer, T. Remmelg, and C. Dubach. "Lift: a functional data-parallel IR for high-performance GPU code generation". In: V. J. Reddi, A. Smith, and L. Tang, Eds., *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, pp. 74–85, ACM, 2017. (Cited on pages 27, 29, and 51.)

[Stoc 12] K. Stock, L. Pouchet, and P. Sadayappan. "Using machine learning to improve automatic vectorization". *TACO*, Vol. 8, No. 4, pp. 50:1–50:23, 2012. (Cited on page 45.)

[Suje 14] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. "Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages". *ACM Trans. Embedded Comput. Syst.*, Vol. 13, No. 4s, pp. 134:1–134:25, 2014. (Cited on page 41.)

[Teix 19] T. S. F. X. Teixeira, C. Ancourt, D. A. Padua, and W. Gropp. "Locus: A System and a Language for Program Optimization". In: M. T. Kandemir, A. Jimborean, and T. Moseley, Eds., *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*, pp. 217–228, IEEE, 2019. (Cited on page 43.)

[Thie 02] W. Thies, M. Karczmarek, and S. P. Amarasinghe. "StreamIt: A Language for Streaming Applications". In: R. N. Horspool, Ed., *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, pp. 179–196, Springer, 2002. (Cited on page 36.)

[Trin 13] P. W. Trinder, M. I. Cole, K. Hammond, H. Loidl, and G. Michaelson. "Resource analyses for parallel and distributed coordination". *Concurrency and Computation: Practice and Experience*, Vol. 25, No. 3, pp. 309–348, 2013. (Cited on page 44.)

[Udup 09] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil. "Software Pipelined Execution of Stream Programs on GPUs". In: *Proceedings of the CGO 2009, The*

Seventh International Symposium on Code Generation and Optimization, Seattle, Washington, USA, March 22-25, 2009, pp. 200–209, IEEE Computer Society, 2009. (Cited on page 36.)

[Verd 13] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor. "Polyhedral parallel code generation for CUDA". *TACO*, Vol. 9, No. 4, pp. 54:1–54:23, 2013. (Cited on pages 39 and 48.)

[Wadl 90] P. Wadler. "Deforestation: Transforming Programs to Eliminate Trees". *Theor. Comput. Sci.*, Vol. 73, No. 2, pp. 231–248, 1990. (Cited on page 47.)

[Whal 98a] R. C. Whaley and J. J. Dongarra. "Automatically Tuned Linear Algebra Software". In: *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, San Jose, CA, pp. 1–27, IEEE Computer Society, 1998. (Cited on page 42.)

[Whal 98b] R. C. Whaley and J. J. Dongarra. "Automatically Tuned Linear Algebra Software". In: *Proceedings of the ACM/IEEE Conference on Supercomputing, SC 1998, November 7-13, 1998, Orlando, FL, USA*, p. 38, IEEE Computer Society, 1998. (Cited on page 83.)

[Wu 16] J. Wu, A. Belevich, E. Bendersky, M. Heffernan, C. Leary, J. A. Pienaar, B. Roune, R. Springer, X. Weng, and R. Hundt. "gpucc: an open-source GPGPU compiler". In: B. Franke, Y. Wu, and F. Rastello, Eds., *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016, Barcelona, Spain, March 12-18, 2016*, pp. 105–116, ACM, 2016. (Cited on page 39.)

[Xi 98] H. Xi and F. Pfenning. "Eliminating Array Bound Checking Through Dependent Types". In: *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, Montreal, Quebec, Canada, pp. 249–257, ACM, 1998. (Cited on page 18.)

[XLAT 17] XLA Team. "XLA - TensorFlow compiled.". `https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html`, 2017. (Cited on page 40.)

[Xue 94] J. Xue. "Automating Non-Unimodular Loop Transformations for Massive Parallelism". *Parallel Computing*, Vol. 20, No. 5, pp. 711–728, 1994. (Cited on page 39.)

[Zhan 13] Y. Zhang and F. Mueller. "Hidp: A hierarchical data parallel language". In: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013*, pp. 7:1–7:11, IEEE Computer Society, 2013. (Cited on pages 37 and 48.)

[Zhan 17] X. Zhang, G. Tan, S. Xue, J. Li, K. Zhou, and M. Chen. "Understanding the GPU Microarchitecture to Achieve Bare-Metal Performance Tuning". In: V. Sarkar and L. Rauchwerger, Eds., *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Austin, TX, USA, February 4-8, 2017*, pp. 31–43, ACM, 2017. (Cited on pages 45 and 117.)