THE UNIVERSITY *of* EDINBURGH

Guided Rewriting and Constraint Satisfaction for Parallel GPU Code Generation

Naums Mogers

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
The University of Edinburgh
2023

Abstract

Graphics Processing Units (GPUs) are notoriously hard to optimise for manually due to their scheduling and memory hierarchies. What is needed are good automatic code generators and optimisers for such parallel hardware. Functional approaches such as Accelerate, Futhark and LIFT leverage a high-level algorithmic Intermediate Representation (IR) to expose parallelism and abstract the implementation details away from the user. However, producing efficient code for a given accelerator remains challenging. Existing code generators depend on the user input to choose a subset of hard-coded optimizations or automated exploration of the implementation search space. The former suffers from the lack of extensibility, while the latter is too costly due to the size of the search space. A hybrid approach is needed, where a space of valid implementations is built automatically and explored with the aid of human expertise.

This thesis presents a solution combining user-guided rewriting and automatically generated constraints to produce high-performance code. The first contribution is an automatic tuning technique to find a balance between performance and memory consumption. Leveraging its functional patterns, the LIFT compiler is empowered to infer tuning constraints and limit the search to valid tuning combinations only.

Next, the thesis reframes parallelisation as a constraint satisfaction problem. Parallelisation constraints are extracted automatically from the input expression, and a solver is used to identify valid rewriting. The constraints truncate the search space to valid parallel mappings only by capturing the scheduling restrictions of the GPU in the context of a given program. A synchronisation barrier insertion technique is proposed to prevent data races and improve the efficiency of the generated parallel mappings.

The final contribution of this thesis is the guided rewriting method, where the user encodes a design space of structural transformations using high-level IR nodes called *rewrite points*. These strongly typed pragmas express macro rewrites and expose design choices as explorable parameters. The thesis proposes a small set of reusable rewrite points to achieve tiling, cache locality, data reuse and memory optimisation.

A comparison with the vendor-provided handwritten kernel library ARM Compute and the TVM code generator demonstrates the effectiveness of this thesis' contributions. With convolution as a use case, LIFT-generated direct and GEMM-based convolution implementations are shown to perform on par with the state-of-the-art solutions on a mobile GPU. Overall, this thesis demonstrates that a functional IR yields well to user-guided and automatic rewriting for high-performance code generation.

Lay Summary

Graphics Processing Units (GPUs) are special-purpose processors initially designed to compute the colour of the pixels on the computer screen. GPUs are uniquely suited to perform many operations in parallel instead of sequentially, as is more prevalent in general-purpose computer processors. This capability motivated their adoption in other applications, such as machine learning, which are computed much faster in parallel.

Producing programming code for parallel processors is challenging. Manual development results in the fastest software but requires expertise in programming a specific processor. Reusing programs from software libraries written by experts shifts the burden onto the library developers to keep producing programs for new applications and processors.

Automatic code generation has been used to produce custom-tailored programs for specific processors. However, code generators struggle to solve many problems involved in developing correct and efficient parallel code. The popular approaches rely either on an engineer to hard-code their expertise into code generators, or mechanically try many possible versions of the same program to find the best one. The former approach is too expensive to maintain for new platforms and applications. The latter takes too long.

This thesis proposes techniques to address the problems of both approaches and combine them in a single system. The explorative approach has been improved to detect program variants that produce incorrect results. By avoiding these candidates, the code generator takes less time to find efficient programs. The manual approach has been improved by providing human experts with a way to guide the code generator in broad strokes in optimising a program without overconstraining the system to a specific processor.

The techniques presented in this thesis are based on a particular way to represent the application called functional programming. Functional expression captures the user's intent, helping the code generator optimise the program without deviating from the original purpose of the computation. This thesis argues that representing the applications in such a way helps solve the issue of parallel processor programmability.

Acknowledgements

I want to thank my adviser Christophe Dubach for his guidance throughout my studies. I have been continuously motivated by his curiosity; he has gone above and beyond in shaping my research path.

I am grateful to the entire Lift team, a welcoming bunch. Thank you to Michel Steuwer, who makes you feel included and is always there to help. Among others, Toomas Remmelg's work on the Lift compiler has been fundamental to my project, and I am grateful for his assistance in the early stages. Larisa Stoltzfus, Christof Schlaak and Lu Li have been fantastic to work with. Long chats with Federico Pizzuti have elevated my days as a PhD student; befriend the man if you get a chance.

I have appreciated the friendly and supportive environment created by the School of Informatics members. Many thanks to Murray Cole, Rodrigo Caetano de Oliveira Rocha, Patrick Hudson, Karen Pinto-Csaszar, Valentin Radu, and Gregor Hall, among many others.

This work was supported by the Engineering and Physical Sciences Research Council (grant EP/L01503X/1), EPSRC Centre for Doctoral Training in Pervasive Parallelism at the University of Edinburgh, School of Informatics. The work has made use of resources provided by the Edinburgh Compute and Data Facility.

The privilege of pursuing a PhD was given to me by my parents Jevgenija Vetsteina-Mogere and Ilja Mogers. From inspiring me to challenge myself to, let's be honest, helping me with my school homework, they have supported me unconditionally all the way. I do what I do because of them.

Finally, I can't help but thank my partner Tân Nazaré, not only because they mentioned me in their thesis but for keeping my sanity throughout this project. They've gone to great lengths to support me when it mattered most, and this thesis was finished thanks to them. But do not hold them accountable for the consequences of this work.

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 used in this thesis has been published in the following papers:

- Naums Mogers, Valentin Radu, Lu Li, Jack Turner, Michael O'Boyle, and Christophe Dubach. "Automatic generation of specialized direct convolutions for mobile GPUs". In: *Proceedings of the 13th Annual Workshop on General Purpose Processing using Graphics Processing Unit*. 2020, pages 41–50. DOI: 10.1145/3366428.3380771. This study was conceived by all of the authors. I carried out the implementation of the proposed techniques in the LIFT compiler and wrote the overwhelming majority of the publication's text.

- Naums Mogers, Lu Li, Valentin Radu, and Christophe Dubach. "Mapping parallelism in a functional IR through constraint satisfaction: a case study on convolution for mobile GPUs". In: *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*. 2022, pages 218–230. I carried out the design of the language features, the implementation of the proposed techniques in the LIFT compiler and wrote the overwhelming majority of the publication's text.

(*Naums Mogers)*

Table of Contents

Chapter 1

Introduction

Recent decades saw a shift in hardware design towards increasing the complexity of processing units from single-chip to multicore architectures. The industry has acted so in response to the "power wall" [Asa06] – the increased costs of performance scaling through transistor density increase. Battery costs in mobile devices, data centre energy consumption and heat dissipation challenges have motivated distributing the computational resources across multiple cores [Bos11].

The fundamental shift in computational resource parallelisation coincided with the specialisation of hardware units. Graphics processing, machine learning and networking applications have fueled the development of massively parallel GPUs, ASICs and FPGAs. The resulting platforms often combine multiple architectures within the same system to allow offloading domain-specific computations to dedicated accelerators.

GPUs gained widespread availability due to the pervasiveness of graphics processing. The GPU chips feature multiple Streaming Multiprocessor cores (SMs) with many floating-point Arithmetic Logic Units (ALUs) and large memories. This architecture proved a good fit outside of its original domain for the applications requiring a large number of independent arithmetic operations and little context switching.

Meanwhile, the computational demand has grown due to the advent of increasingly resource-intensive applications such as Machine Learning (ML), computer vision and scientific computing. ML alone has been used for speech recognition [Col11], sentiment analysis [Dos14], sentence modelling [Kal14] and visual classification [Sim14]. With practical adoption of these methods predicated on utilising accelerators such as GPUs, parallel programming transitioned from the toolset of systems engineers to that of the application developers.

However, parallelising computation has unique challenges not often encountered in

sequential programming. Efficient task partitioning and scheduling, distributed memory management and synchronisation are some of the problems that need to be addressed. For software engineers, these challenges ended the free performance lunch [Sut05], which promised applications regular performance improvements through increasing CPU clock speeds and memory and disk access rates. The increase in the computational capability of the platform has ceased to improve performance unless the software is well-parallelised and tailored to the hardware architecture [Wan18]. Expertise in parallel programming is required to avoid resource underutilisation, deadlocks, livelocks and race conditions. The programmability of the GPU has to be improved to lower the barrier of entry into parallel computation.

1.1 The Programmability Challenge

Hardware vendors addressed the computational demand by exposing the scheduling and memory management functionality through parallel programming models such as CUDA [Nic08] and OpenCL [Khr22]. These interfaces are sufficiently fine-grained to enable high processor and cache utilisation through tiling, vectorisation and memory coalescing. However, architectural differences among parallel accelerators restrict the efficiency and correctness of low-level implementations to a small set of platforms. Manually porting software across platforms, therefore, incurs high development costs. This rigidity has been addressed by decoupling the problem specification from the implementation.

1.1.1 Separation of Concern

Two high-level programming approaches have emerged: static optimised kernel libraries and automated code generation. Handwritten kernel libraries provide suites of target-specific implementations of popular algebraic [Nvi07; Wha01; Xia12] and domain-specific [Che14; Arm21; Kha19] procedures. This approach shifts most of the optimisational burden from the user to the library developers.

This thesis tackles automated code generation, which has been used to achieve high performance across multiple platforms at a fraction of the cost of manual development. Code generators pursue performance portability through high-level Intermediate Representations (IRs).

High-level IRs expose implicit parallelism in the application without specifying

a particular scheduling policy. Relaxed representation allows the compiler to switch between parallel programming models based on the target platform. These IRs are also sufficiently expressive to capture the algorithmic intent of the user. Rich algorithmic representation encodes the information required to alter the program's structure safely. Data dependencies, expressive types and high-level iteration space characteristics allow the compiler to partition and reorder the computations without breaking the program semantics.

Several high-level programming abstractions have been proposed to capture algorithmic properties useful for automatic code generation. Algorithmic skeleton [Col89; Col04] frameworks expose high-level computation patterns such as *map*, *reduce* and *scan* to the user as C++ templates [Dea08; Ald11; Ste11]. Platform-specific programs are generated automatically by embedding user-provided functions into the parallel implementations of the user-chosen algorithmic skeletons. However, the successful application of algorithmic skeletons is limited by the front-end languages of the skeleton libraries. This limitation motivated the development of IRs, where the language captures the implicit parallelism as part of the programming model.

Computational graphs [Lea17; Wei17; Rot18; Cyp18] and functional patterns [Hen17; McD13; Ste17] have been successfully used to capture implicit parallelism. These programming abstractions enable radical program transformations that would otherwise require expensive static analysis: operation fusion, tiling, access coalescing, and automatic tuning. With such expressive power comes the curse of dimensionality: the design space of alternative implementations for each program is so large that finding efficient candidates is prohibitively expensive. We now look into the automated reasoning techniques that optimise parallel code with varying design space coverage and search efficiency.

1.1.2 Automatic Code Generation

Polyhedral compilers exploit loop-level parallelism [Ver13; Vas18; Bag19]. They derive the geometric representation of the iteration space and parallelise loops using linear algebra transformations. Although these approaches produce highly efficient kernels, the polyhedral model is unsuitable for problems that depend on non-affine loops [Pre19].

Accelerate [McD13], Futhark [Hen17], LIFT [Ste17] and Spiral [Fra18] use automated rewriting systems to tailor programs to the target platforms. These approaches

leverage the advantages of the functional representation: limited side effects, strong dependent type systems and functional patterns. Although these systems eventually generate high-performance solutions, finding optimal solutions takes a long time.

PetaBricks [Pho13] and Tangram [Cha16; De 19] avoid long search times by depending on the user to define the algorithm and provide alternative implementations. Similarly, TVM users define the optimisations using Halide schedules [Che18b; Ada19; Zhe20a]. These approaches either put too much pressure on the user to provide good initial choices [Sot19] or overconstrain the heuristic search through strong assumptions about the target hardware.

Another challenge with automated design space exploration is ensuring the transformation correctness. This applies to auto-tuning of the numeric parameters [Ans14], parallel mapping search [Zhe20a] and algorithmic transformations [Pho13]. Avoiding invalid transformations is essential to ensure the preservation of the input program semantics and to avoid spending time evaluating incorrect candidates.

Despite the advances in automatic parallelisation methods, finding a balance between mechanic exploration and informed optimisation remains an open problem. The former delivers extensive design coverage, while the latter finds optimal solutions faster. This thesis tackles the challenge of balancing explorative and heuristic code generation methods. For automatic tuning and parallelisation, it proposes two techniques to detect invalid implementations early and restrict the exploration to valid implementations only. For algorithmic transformations, the thesis proposes a way to incorporate user expertise in the optimisation process without sacrificing design space coverage.

1.2 Contributions

This thesis tackles several challenges in parallel code generation. Firstly, it addresses the need to avoid the evaluation of invalid implementations during auto-tuning. Secondly, it proposes a way to capture the parallel programming model of a given platform to generate a space of valid parallel mappings automatically. Finally, it provides a high-level optimisational abstraction for the user to outline a large space of structural program transformations.

This work leverages the functional data-parallel language and compiler LIFT. The proposed techniques are demonstrated using Convolutional Neural Networks (CNNs) on mobile GPUs as a case study. This choice is motivated by the challenges of opti-

mising a complex application on a resource-constrained platform. Applications with non-straightforward memory access patterns highlight the limitations of static code analysis due to non-obvious data dependencies and obscured high-level algorithms. Platforms with fewer computational resources – *e.g.*, mobile GPUs prioritising power efficiency at the expense of the operating memory – limit the choice of available optimisations. A memory-bound application on a resource-constrained platform makes for a challenging optimisation target.

Specifically, the following contributions are made:

Auto-tuning The thesis shows how tuning constraints are automatically inferred from strongly typed functional patterns to tune computational kernels automatically. This addresses the combinatorial explosion problem during design space exploration by leveraging a functional IR to constrain the search. The thesis discusses how search optimisation can be leveraged to explore the trade-off between performance and memory consumption. The technique is shown to produce code automatically for direct convolution, exploring a large optimisation space of 1,000 points with LIFT, where the best candidate achieves a speedup of $10\times$ and memory saving of $\times 3.6$ over the vendor-provided ARM Compute Library on the ARM Mali GPU.

Parallelism Mapping To address the need for automatic code parallelisation, the thesis reformulates parallelisation mapping as a constraint satisfaction problem. Specifically, it proposes a technique to generate parallelisation constraints specific to a given program. These constraints restrict the search to the parallel mappings valid in a given parallel programming model. Also proposed is a synchronisation barrier insertion method to avoid data races and reduce the synchronisation overheads in the generated programs. Generated programs are shown to achieve performance on par with the state-of-the-art code generator TVM [Che18a] with memory savings of more than $2\times$.

Guided Rewriting A user-guided parametric rewriting mechanism is proposed to address the need for a way to inject loosely-defined heuristics into the search. This work focuses on encoding good design choices directly in the IR without burdening the user with low-level implementational details and overconstraining the problem. With a small set of rewrite points, a single convolution expression is used to express two convolution methods, tiling, prefetching, data reuse, memory optimisation, access coalescing and OpenCL kernel fission. The compiler uses the rewrite points to create

new parallelisation and tuning opportunities automatically. The generated GEMM-based convolution implementations achieve 78% performance of the hand-optimised ARM Compute kernels.

1.3 Thesis Outline

The rest of this thesis is organised as follows:

Chapter 2 presents the technical background on the convolutional layers of CNNs; GPU programming and the OpenCL programming model in particular; the functional data-parallel language and compiler LIFT, which targets OpenCL and is used to implement the techniques proposed in this thesis with convolution as a use case.

Chapter 3 critically discusses related work in GPU code generation. It describes the explicitly parallel approaches, which are based on low-level programming abstractions or optimised kernel libraries. It also introduces ways to detect and capture implicit parallelism in applications and covers the relevant parallel code optimisation techniques.

Chapter 4 provides a novel account of expressing low-level convolution optimisations in a high-level functional IR of LIFT. It describes a tuning constraint inference technique based on functional patterns. The chapter shows how constraints restrict the tuning search to valid implementations and showcase the results of multi-objective search-based optimisation. The chapter also discusses improvements of the LIFT memory allocation method to optimise intermediate memory buffers.

Chapter 5 describes the proposed automatic parallelisation technique based on constraint satisfaction. It provides a detailed account of expressing OpenCL programming model restrictions as arithmetic constraints. The constraints are generated automatically based on the LIFT IR. This work shows that functional patterns both expose plenty of parallelism and enable safe parallelisation. The chapter also describes a synchronisation barrier insertion method which leverages functional representation to find correct and efficient barrier placements.

Chapter 6 extends the language with rewrite points as a way for the user to guide the optimisation process. The chapter defines the rewrite points from the user's per-

spective – as an IR primitive – and from the compiler engineer's perspective – as a parametric optimisation interface. The chapter describes how the compiler applies rewrite points in the top-down rewriting process to transform the expression; it then defines eleven rewrite points, which are used to encode a large design space on the example of convolution. The chapter analyses the generated implementations of direct and GEMM-based convolution and shows that one set of rewrite points encodes good choices for both convolution algorithms.

Chapter 7 concludes the thesis with a critical analysis of its contributions. The chapter highlights the limitations of the proposed techniques and suggests avenues for future work.

Chapter 2

Background

This chapter provides the background information required for the rest of this thesis. Three topics of interest are covered. The first section introduces convolution – the core operation of a CNN and the main use case of this thesis. Due to its high computational and memory requirements, convolution provides an interesting challenge for optimising code generation. The section provides two popular algorithms used to perform convolution and discusses their differences. Since one algorithm is preferable for low memory budgets, and the other – for high-bandwidth requirements – both are worthwhile optimisation targets.

The second section discusses GPUs as a target platform for high-performance applications. GPU is the target platform of the code generation techniques presented in this thesis. An overview of memory and execution models is provided, paving the way for the discussion of hardware-specific optimisation methods in the later chapters. The section then focuses on OpenCL – an open standard for parallel programming heterogeneous systems including GPUs [Khr22]. The functional portability of OpenCL makes it a good target language for code generation; OpenCL's lack of performance portability motivates further efforts on automatic optimisation methods.

Finally, the chapter introduces LIFT – a functional data-parallel language and a compiler used to demonstrate the contributions of this thesis. The chapter discusses the type system and the two levels of the LIFT IR. We will see in the later chapters how beneficial a rich algorithmic representation is for automatic code generation; an overview of LIFT IR forms the basis of this argument. A brief overview of the LIFT compiler internals helps understand the proposed static analysis methods. LIFT code generation is demonstrated with an example program; the chapter closes with a discussion of the limitations of the current approach motivating the rest of the thesis.

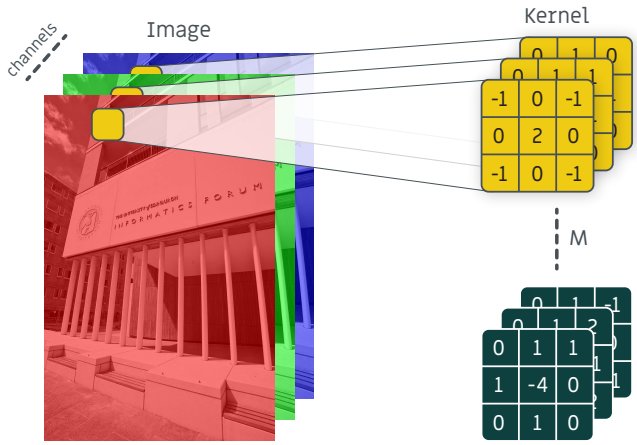


Figure 2.1: Convolutional layer stacking M kernels to be convoluted over a three channel-image by sliding the kernels over the image – direct convolution. Photo from [Win08].

2.1 Convolution

CNNs are the tool of choice for most computer vision problems such as classification and segmentation. They are composed of stacked layers of convolutions over multi-channel inputs. Each input is expressed as C_{in} channels of 2D images and produces C_{out} output channels, as illustrated in Figure 2.1. Each output channel contains a *feature map* – a tensor characterising the spatial distribution of visual features in the input image. Feature maps are produced by sliding *convolutional kernels* across the spatial dimensions of the image. Kernels contain weights encoding features. Each slided window is convolved with the kernel weights by multiplying pixel values and corresponding weights across all input channels. An output value is produced by summing all weighted values of the slided window. The weights of the kernels are said to be *learnable* and are usually optimised by gradient descent over convex objective functions.

In computer vision, the first image passed to a convolutional neural network has three channels: red, green and blue. They get transformed in scale and value based on the learned kernel weights at each layer, traversing the network of several convolutional layers. The set of C_{out} output feature maps is passed as an input to the next convolutional layer, until the end of the network. For classification tasks, the outputs of convolutional layers – feature maps – are flattened into a vector and passed to one or more affine transforms. For example, the popular ILSVRC Image Classification contest [Den09] has 1000 classes, so networks trained on this task will output vectors

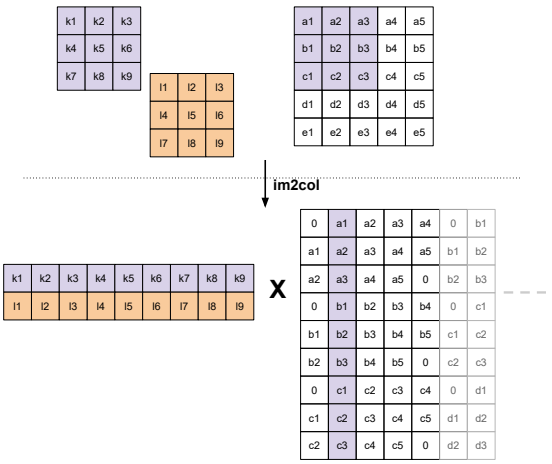


Figure 2.2: GEMM input preparation: im2col transformation applied to two 3×3 kernels and a portion of the larger input matrix generated from the smaller original 5×5 input image.

of length 1000. These affine transformations account for very little of the total inference time. Convolution affects run time more: for example, in the ImageNet-winning SENet [Hu18] architecture, convolution accounts for 99.99% of total floating point operations. This makes convolution a worthwhile target of optimisation, hence the focus of this thesis is on this layer.

The two widely adopted convolution algorithms are the General Matrix Multiply (GEMM) method and direct convolution. We now look at the differences between the two algorithms.

2.1.1 General Matrix Multiply

The convolution operation is commonly implemented as matrix multiplication due to the availability of highly optimised GEMM routines available in libraries for both CPU (openBLAS) and GPU (CLBlas, cuDNN). The GEMM method uses the Image To Column (im2col) operation to produce a copy of each sliding window stacked with other windows as columns in a single matrix. This allows performing convolution by multiplying the reshaped input and weights using an efficient GEMM routine, for which many Basic Linear Algebra Subprograms (BLAS) libraries provide optimised kernels.

Figure 2.2 presents the im2col operation, where two 3×3 kernels are convoluted on a single channel 5×5 image. The input image has 25 elements and the two kernels have 9 elements each. To perform GEMM, kernels are unrolled into two rows, and

```
input[C][H][W]; kernels[M][K][K][C]; output[M][H][W];
for h in 1 to H do
  for w in 1 to W do
    for o in 1 to M do
      sum = 0;
      for i in 1 to K do
        for j in 1 to K do
          for c in 1 to C do
            sum += input[c][h+i][w+j] * kernels[o][i][j][c];
      output[o][w][h] = sum;
```

Listing 2.1: Direct convolution expressed in an imperative style.

through im2col the input is mapped to the input-patch matrix which is $9\times$ larger than the original image assuming padding of 1 and stride of 1. im2col increases memory consumption due to data duplication in memory: in CNN architectures such as VGG, ResNet and GoogleNet, the sliding windows overlap causing im2col to duplicate data in memory.

2.1.2 Direct Convolution

The direct convolution method is based on a stencil algorithm, which updates elements based on their neighbouring values. Each convolution kernel has a receptive field of spatial size ($kernel_{width} \times kernel_{height}$) in 2D, usually square, $K \times K$, and a depth to match the input number of channels C, across all M kernels. On an input image size $C \times H \times W$ the direct convolution is performed with nested loops as shown in Listing 2.1.

Although the direct approach uses less memory, the input access patterns are more complicated than in GEMM, requiring careful optimisation of memory access patterns.

2.1.3 Memory Footprint

Figure 2.3 shows the actual run-time memory footprint required by the largest layer in the most popular deep neural networks used for image classification. GEMM requires consistently more memory than direct convolution (one order of magnitude) due to the increased memory size of the transformed input. In VGG layer 2, im2col enlarges the input from 13 MB to 116 MB. This puts a significant strain on resource-constrained platforms such as mobile GPUs, where memory is both small and shared for multiple

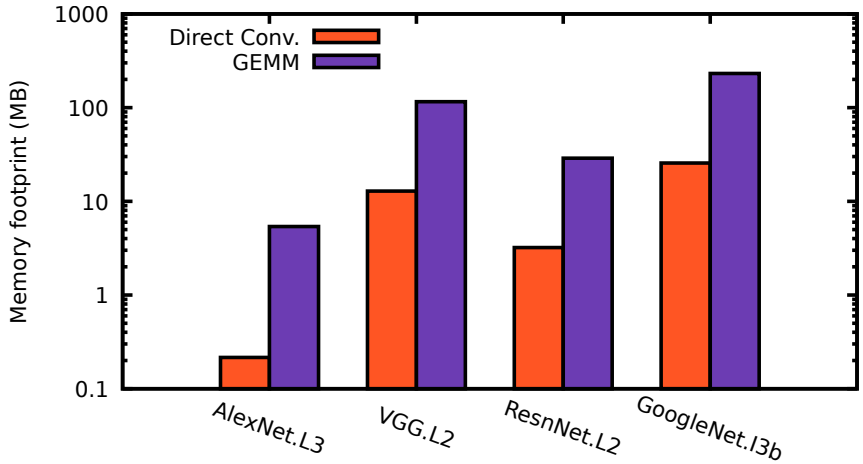


Figure 2.3: Runtime memory footprint of largest layers in some of the most popular deep neural networks. GEMM memory footprint includes the extra memory required by im2col.

tasks.

The difference in memory consumption can be illustrated as follows. For M kernels of size $K \times K$, applied over an image of C channels with width W and height H (padding P elements to the input image and stride S to indicate the kernel sliding distance), direct convolution requires a storage space of $M \times K^2 \times C$ for the weights and $C \times H \times W$ for the image. With the im2col in preparation for the GEMM operation, the memory space required by the input-patch matrix is

$$K^2 \times C \times (\frac{H+2P-K}{S}+1) \times (\frac{W+2P-K}{S}+1)$$

which is at least K^2 times larger than the original input for strides of 1 and padding of $K/2$. In the VGG-16 CNN [Sim14], the difference in memory consumption is at least $9\times$.

As demonstrated by the difference in memory consumption, there is a trade-off between GEMM-based and direct convolution. BLAS libraries provide well-optimised GEMM implementations, which achieve high performance at the cost of high memory consumption. As we will see in Chapter 4, it is nevertheless possible to generate an optimised direct convolution with a performance comparable to that of GEMM. We now turn our attention to the topic of GPU programming, which provides the technical background for the proposed technique to generate fast direct convolution.

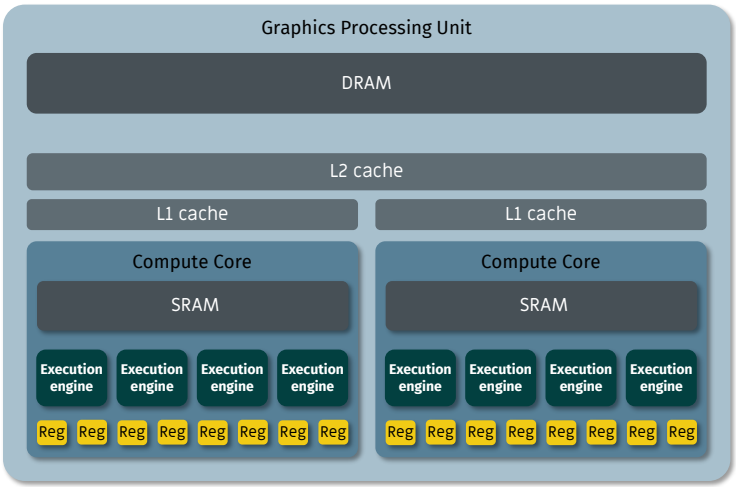


Figure 2.4: Simplified overview of a GPU architecture. The number of compute and memory units is not to scale.

2.2 GPU Programming

Graphics Processing Units are popular among embarrassingly parallel and streaming applications, where the same computation is performed repeatedly on large data sets. Although originally designed for graphics applications, GPUs rose to prominence as a general-purpose programming platform of choice rivalling Central Processing Unit (CPU). Compared to CPU's complex cores capable of performing different tasks simultaneously, GPU provides simple compute cores with a lot of ALUs and operating memory. The less centralised Single Instruction/Multiple Data (SIMD) architecture of GPUs allows scaling computations across the increasingly large numbers of compute cores with low synchronisation overheads and energy consumption.

2.2.1 GPU Architecture

Figure 2.4 shows a simplified GPU architecture. A GPU features multiple compute cores containing hundreds of execution units each. All compute cores share Dynamic random-access memory (DRAM) and an L2 cache. Each core usually has an L1 cache and on-chip Static random-access memory (SRAM) shared among the threads executed on that core. Registers are the fastest unit of memory, but they are usually scarce; the more registers are used per thread, the fewer threads are run in parallel. DRAM is the slowest memory – each access requiring several hundred clock cycles – while SRAM is usually faster due to its architecture and on-chip placement.

All threads scheduled on one core are split into units called warps, otherwise known

as thread blocks and wavefronts. A warp is the largest unit executing in lockstep, where multiple operations are performed in parallel at the time cost of one operation. A part of the warp might stay idle temporarily due to thread divergence, which is usually caused by the conditional execution blocks entered by only a part of a warp.

Each access of the DRAM returns several values to fill an entire cache line. Threads of the same warp may share a cache line if they access elements that fit in the same cache line; in that case, the overhead of only one memory access is incurred.

Threads are organised in groups; each group is further split into one or more warps. Each compute core can host multiple thread groups in parallel depending on the group size and memory consumption.

Beyond a shared SRAM, compute cores provide thread groups with another crucial feature: thread synchronisation using barriers. Threads that reach a software barrier in their control flow must wait until all threads in the group reach the same barrier. The barriers are set using memory fences as follows:

```
barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE);
```

Where the memory fence determines how operations are to be ordered. A barrier on the global memory orders operations which read or write global values; the local memory fence orders the local memory accesses. Barriers are useful to synchronise across multiple warps in a group. Lockstep execution means that the sequential operations are well-ordered across threads of the same warp, therefore barriers are not required to synchronise a warp.

Mobile and desktop GPUs have a few notable differences. Mobile GPUs often do not have SRAM memory – a block of DRAM is allocated for each core instead. Although desktop GPUs support larger thread groups, mobile GPUs put more focus on vectorisation. Coarse-grained ALUs and memory controllers are provided to execute multiple operations, loads and stores at the same time.

A Mali G72 GPU features 12 compute cores supporting up to 384 threads each. Warps contain four threads called *warps*. Through vectorised loads of consecutive data, a single quad fills an entire cache line of 64 bytes at the price of one access. Each compute core of Mali G72 provides 64 bytes of register memory. The L2 cache of 524 KBytes supports a DRAM of 4 GBytes.

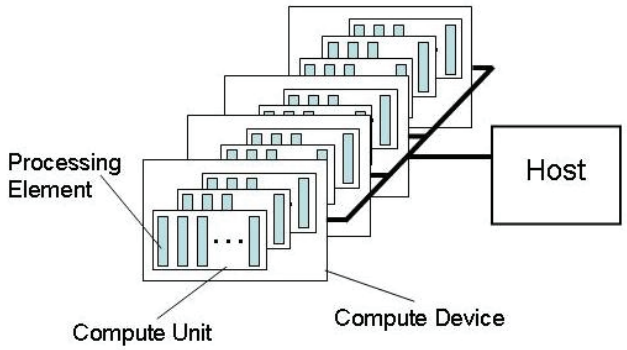


Figure 2.5: OpenCL platform model. From [Khr22].

2.2.2 OpenCL Programming Model

OpenCL is an open industry standard for programming heterogeneous systems supporting CPUs, GPUs, Field Programmable Gate Arrayss (FPGAs) and Application-Specific Integrated Circuits (ASICs) [Khr22]. OpenCL is a language implemented as a dialect of C; vendors supporting OpenCL provide OpenCL compilers and drivers. OpenCL code is designed for portability – the same program can be compiled for different targets with little or no changes.

2.2.2.1 Platform Model

The OpenCL platform model depends on two main components: the host and devices, where each device can have multiple compute units (Figure 2.5). In GPU programming, CPU is the host and the GPU is the OpenCL device; GPU cores are OpenCL compute units.

An OpenCL host orchestrates execution: schedules task execution across OpenCL devices, allocates memory and schedules data transfers between devices. An OpenCL program code consists of the host code (typically C++) and one or more OpenCL kernels. An OpenCL kernel is a C-like function which is executed on a device; it takes pointers to DRAM input and output buffers as arguments. Host code tracks the start and the end of a kernel execution through OpenCL events.

2.2.2.2 Memory Model

The OpenCL standard defines its own set of terms abstracting away the differences among the accelerators. The memory model shown in Figure 2.6 defines four types

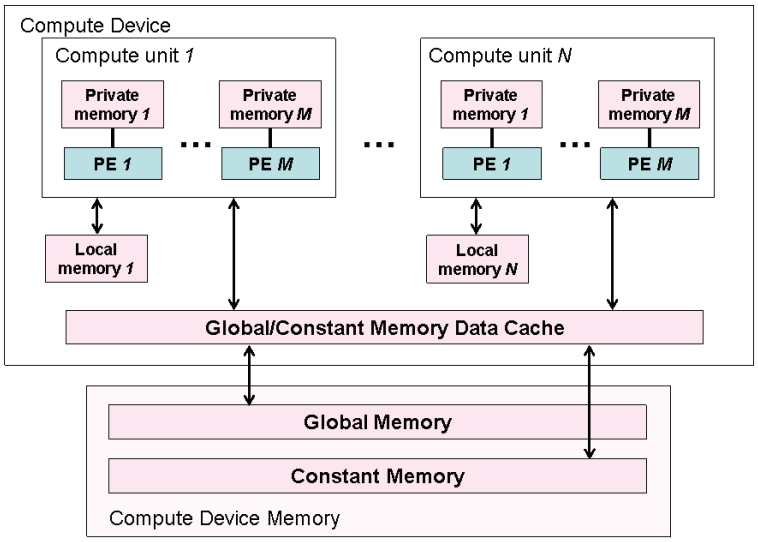


Figure 2.6: OpenCL memory model. From [Khr22].

of memories. Global memory is specific to the OpenCL device and is shared among all its processing units – in GPUs, global memory corresponds to DRAM. Constant memory is similar to global memory, but it is read-only for the OpenCL device.

Local memory is limited in scope to a single processing unit; on desktop GPUs, it corresponds to SRAM. On a mobile GPU like Mali G72, local memory is allocated in DRAM. Each compute unit has private memory corresponding to registers on a GPU; it is limited in scope to a processing element (thread). Only global and constant memories are accessible by the host to write inputs and read outputs.

2.2.2.3 Execution Model

The scheduling units of OpenCL are work groups and work items; on GPUs, they correspond to thread groups and threads, respectively. Work items within the same work group are executed concurrently on the same compute core and share local memory. Work items within the same work group can be synchronised using an OpenCL barrier. A barrier ensures valid ordering of interdependent read-write operations.

OpenCL is a SIMD programming model, therefore all work items in all work groups execute the same OpenCL kernel. Work groups and work items are specialised to access different memory regions using scheduling indices. Each thread can query three indices: its local index within a work group, the index of its work group, and the global index across all threads. Global and work group indices are used to access specific regions of global and constant memories. Local indices are used to access

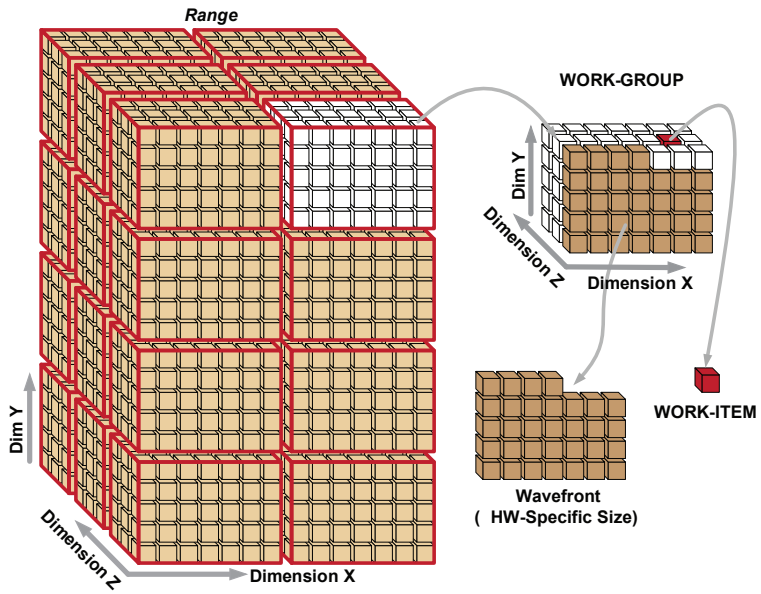


Figure 2.7: OpenCL work group and work item indexing space. From [Mun11].

local memory. The three types of indexing are also referred to as scheduling domains: global, work group and local.

OpenCL indexing space is shown in Figure 2.7. Global, work group and local indices are defined in three dimensions each: 0, 1 and 2 (otherwise referred to as dimensions X, Y and Z). 3D-indexing provides a concise way to subdivide work within the same domain, *e.g.*, a single work group can be further specialised three-way. Indexing dimension 0 (X) is the most fine-grained in terms of scheduling: threads with consecutive indices in dimension 0 are likely to be part of the same warp.

As a dialect of C, OpenCL inherits the notion of `for`-loops, which are not inherently sequential or parallel. An example of a sequential `for`-loop is `for (int i = 0; i < ub; i++) {..}`. `for`-loops can be used to split sequential work among parallel threads. In the following example, the loop counter iterates over the multiples of the thread index until the upper bound is reached:
`for (int i = get_local_id(0); i < ub; i += get_group_size(0)) {..}`.

Thanks to its wide adoption, OpenCL programs are functionally portable – little effort is required to perform the same computation correctly on a different architecture. However, OpenCL code is not portable without a loss in execution speed: optimising a program for one architecture hinders its performance on others. We now look at the LIFT code generator, which produces platform-specific OpenCL programs to achieve performance portability.

2.3 LIFT

The design goal of LIFT is to raise the programming abstraction and enable automatic performance optimisations on massively parallel accelerators, such as GPUs. LIFT provides a high level Intermediate Representation (IR) [Ste17], and a compiler that automatically translates the high-level IR to low-level target code. The LIFT IR is functional where operations are side-effect free, enabling the composition of LIFT primitives naturally. Optimisation choices are encoded using a system of rewrite rules that capture the algorithmic and hardware-specific optimisations.

2.3.1 The LIFT Language

The LIFT functional data-parallel language abstracts away the complexities of hardware, shifting the optimisation burden from users to the compiler. This separation of concern is particularly useful in the ML field. ML engineering requires broad domain knowledge on its own; ML practitioners should not have to acquire performance programming expertise to produce high-performance ML programs.

The functional patterns of LIFT create an algorithmic representation of the given problem, helping the compiler perform radical optimising program transformations. LIFT IR is discussed in detail in previous work [Piz22; Sto21; Hag18; Ste17]. LIFT includes hardware-agnostic algorithmic primitives and low-level primitives, which encode specific hardware details.

2.3.1.1 Type System

The LIFT language supports the following types: scalars, vectors, tuples and multidimensional arrays. Scalars are integers and floating point numbers; the data type size can be adjusted for the target platform. LIFT uses a limited form of a dependent type system, where the array and vector sizes are included in the type definitions. When LIFT's functional patterns affect the array sizes, the changes are propagated through the expression by the type checker. Such expressivity prevents out-of-bounds accesses and helps preserve expression semantics during rewriting.

LIFT types are denoted in cursive as T, where T is any supported type. $(\mathtt{x} : T)$ describes an argument or an expression $\mathtt{x}$ of type T. T_i and T_j denote unrelated types of arguments i and j. The type of an array of size n and element type T is denoted as $[T]_n$, where n is a symbolic arithmetic expression. A two-dimensional array type is denoted

as $[[T]_{n2}]_{n1}$, where *n1* is the length of the outer dimension, and *n2* is the length of the inner dimension. An alternative multidimensional array notation is $[T]_{\overrightarrow{N}}$, where $\overrightarrow{N}$ is a vector of array lengths: $\overrightarrow{N} = \langle n0, n1 \rangle$. The number of dimensions in the array of type $[T]_{\overrightarrow{N}}$ is denoted as $\|\overrightarrow{N}\|$, *i.e.*, the length of the vector $\overrightarrow{N}$.

The overhead arrow notation $\overrightarrow{N}$ refers to the vectors of array dimension sizes only; this notation is used only in type annotations to express multidimensional arrays concisely. The vector data type is denoted as $\langle T \rangle_n$, where n is the vector length and T is the vector element type. (T, U) denotes the type of a LIFT tuple with elements of type T and U.

2.3.1.2 High-Level Algorithmic Patterns

The main high-level algorithmic primitives supported by LIFT and used in this thesis are listed in Figure 2.8. These algorithmic primitives only express *what* needs to be computed, shielding programmers from any hardware-specific implementational details. For example, the **`map`** pattern does not enforce a particular parallelisation strategy or order of traversal. Such functional representation preserves the algorithmic information for as long as possible, leaving it up to the compiler to lower the abstract patterns into hardware-specific primitives through rewriting.

The double arrow ($\Rightarrow$) denotes a function: $(a : T,\ b : U) \Rightarrow V$ is a type of a function that takes two arguments of types T and U respectively and returns a value of type V. The subsequent discussions also refer to functions as lambdas.

Control Structures **`map`** and **`reduce`** are LIFT's two main higher-order functions; both are compiled to OpenCL for-loops. **`map`** applies the argument function on each element of the argument array. **`reduce`** "folds" the result using an initialised accumulator and a binary function.

Data Layout Patterns A number of primitives in LIFT express data layout transformation patterns that only affect the shape of the data without changing its value. Data grouping is achieved with **`tuple`** and **`zip`**, which pairs the corresponding elements of input arrays; the resulting tuples can be unpacked using **`get`**.

`split` subdivides an input array into chunks and **`join`** flattens the two outer array dimensions. **`slide`** creates an extra array dimension by sliding a window of a given size across the outer input dimension with a given step [Hag18]. **`slideStrict`** does

$$\textbf{map} : (f : T \Rightarrow U,\ x : [T]_n) \Rightarrow [U]_n$$
$$\textbf{reduce} : (init : U,\ f : (U,T) \Rightarrow U,\ x : [T]_n) \Rightarrow [U]_1$$

(a) Control structures

$$\textbf{tuple} : (x1 : T_1,\ \ldots,\ xn : T_n) \Rightarrow (T_1,\ \ldots,\ T_n)$$
$$\textbf{zip} : (x1 : [T_1]_n,\ \ldots,\ xm : [T_m]_n) \Rightarrow [(T_1,\ \ldots,\ T_m)]_n$$
$$\textbf{get} : (i : int,\ x : (T_1,\ \ldots,\ T_m)) \Rightarrow T_i$$
$$\textbf{split} : (m : int,\ x : [T]_n) \Rightarrow [[T]_m]_{n/m}$$
$$\textbf{join} : (x : [[T]_m]_n) \Rightarrow [T]_{m \times n}$$
$$\textbf{slide} : (size : int,\ step : int,\ x : [T]_n) \Rightarrow [[T]_{size}]_{\lfloor \frac{n-size+step}{step} \rfloor}$$
$$\textbf{slideStrict} : (size : int,\ step : int,\ x : [T]_n) \Rightarrow [[T]_{size}]_{\frac{n-size+step}{step}}$$
$$\textbf{transpose} : (x : [[T]_m]_n) \Rightarrow [[T]_n]_m$$
$$\textbf{transposeW} : (x : [[T]_m]_n) \Rightarrow [[T]_n]_m$$
$$\textbf{pad} : (l : int,\ r : int,\ value : T',\ x : [T]_n) \Rightarrow [T]_{l+n+r}$$

(b) Data layout patterns

$$\textbf{value} : (val : string,\ type : \text{“}T\text{”})$$

(c) The constant value pattern

Figure 2.8: High-level algorithmic patterns of the LIFT IR. In **pad**, the value is broadcast if it has fewer dimensions than the argument; the base type (the type of the innermost array elements) must match that of the argument. In **value**, the arguments are a string encoding the constant value and the LIFT type annotation of the value, respectively.

the same with an additional restriction on the arguments: *size* and *step* should be such that the sliding windows cover the entire input without breaking array bounds.

The **transpose** primitive swaps two outer array dimensions upon the subsequent read operation, while **transposeW** swaps two outer array dimensions upon the preceding write operation. **pad** pads an array with a fixed value, l times to the left and r times to the right. **value** declares a constant with a given value and type.

LIFT keeps track of these virtual transformations using its view system. Views are memory mapping structures which can be used to emit index expressions taking into account the history of data layout transformations. For example, the **pad** primitive produces a `ViewPad` view, which wraps the subsequent memory access into an inline

$$
\begin{aligned}
\textbf{mapND}_2 &: (f : T \Rightarrow U,\ x : [[T]_m]_n) \Rightarrow [[U]_m]_n \\
\textbf{zipND}_2 &: (x1 : [[T_1]_m]_n,\ \ldots,\ xk : [[T_k]_m]_n) \Rightarrow [[(T_1,\ \ldots,\ T_k)]_m]_n \\
\textbf{splitND}_2 &: (m1 : int,\ m2 : int,\ x : [T]_n) \Rightarrow [[[T]_{m2}]_{m1}]_{n/(m1 \times m2)} \\
\textbf{joinND}_3 &: (x : [[[T]_{n3}]_{n2}]_{n1}) \Rightarrow [T]_{n3 \times n2 \times n1} \\
\textbf{slideND}_2 &: (size1 : int,\ step1 : int,\ size2 : int,\ step2 : int, \\
&\quad x : [[T]_m]_n) \Rightarrow [[[[T]_{size2}]_{size1}]_{\lfloor \frac{m-size2+step2}{step2} \rfloor}]_{\lfloor \frac{n-size1+step1}{step1} \rfloor} \\
\textbf{slideStrictND}_2 &: (size1 : int,\ step1 : int,\ size2 : int,\ step2 : int, \\
&\quad x : [[T]_m]_n) \Rightarrow [[[[T]_{size2}]_{size1}]_{\frac{m-size2+step2}{step2}}]_{\frac{n-size1+step1}{step1}} \\
\textbf{transposeND}_3 &: (i : int,\ j : int,\ k : int,\ x : [[[T]_{n3}]_{n2}]_{n1}) \Rightarrow [[[T]_{nk}]_{nj}]_{ni} \\
\textbf{transposeWND}_3 &: (i : int,\ j : int,\ k : int,\ x : [[[T]_{n3}]_{n2}]_{n1}) \Rightarrow [[[T]_{nk}]_{nj}]_{ni} \\
\textbf{padND}_2 &: (t : int,\ b : int,\ l : int,\ r : int,\ value : T, \\
&\quad x : [[T]_m]_n) \Rightarrow [[T]_{l+m+r}]_{t+n+b} \\
\textbf{take} &: (m : int,\ x : [T]_n) \Rightarrow [T]_m
\end{aligned}
$$

Figure 2.9: High-level LIFT macros. Macros are expanded automatically using LIFT primitives and serve the purpose of syntactic sugar. Higher-dimensional versions of these macros are generated programmatically.

if-conditional. Depending on the access index, the conditional returns either the array element or the padding constant. We will see in Section 2.3.2.2 how the views are built during code generation based on the LIFT expression.

While other LIFT patterns produce loops and memory allocation calls, data layout transformers affect the array index expressions and buffer sizes. Layout transformations in memory are delayed as much as possible until a function reads or writes, which is when views are converted into arithmetic array indexing expressions. The view system in LIFT is backed by a handmade arithmetic simplifier library.

2.3.1.3 Macros

The LIFT IR is restricted to a small set of primitives to keep the compiler lean. A higher-level IR is provided for convenience using macros that are automatically expanded to equivalent LIFT expressions. Macros shine during rewriting, where the structure of the transformed expression varies based on the dimensionality of the original expression, which is not known in advance.

For instance, the two-dimensional **map** that operates on a 2D array is defined in

terms of **map** as follows:

$$(\textbf{mapND}_2\,\texttt{f})\ \texttt{<<}\ \texttt{x} \longmapsto (\textbf{map}\,(\textbf{map}\,\texttt{f}))\ \texttt{<<}\ \texttt{x}$$

Where $\longmapsto$ denotes semantics-preserving transformation, *e.g.*, macro expansion or rewriting. Two angular brackets (<<) denote function application, *i.e.*, `g << y` reads the same as `g(y)`: "apply function `g` on the argument `y`".

The multidimensional version of **zip** combines arguments across multiple dimensions, provided that the corresponding dimensions are equal in size. For example, $\textbf{zipND}_2$ takes two-dimensional arguments (where the elements can also be arrays) and produces a two-dimensional array of pairs. $\textbf{zipND}_2$ is expanded as follows:

$$\begin{aligned}&\textbf{zipND}_2\,(\texttt{x}_1, \ldots, \texttt{x}_k) \longmapsto\\ &\textbf{map}\,(\texttt{tuple1D} \Rightarrow \textbf{zip}\,(\textbf{get}\,(0, \texttt{tuple1D}), \ldots, \textbf{get}\,(\texttt{k}, \texttt{tuple1D}))\\ &\quad)\ \texttt{<<}\ \textbf{zip}\,(\texttt{x}_1, \ldots, \texttt{x}_k)\end{aligned}$$

Reshaping Macros An argument is split k times as follows:

$$(\textbf{splitND}_k\,(\texttt{m1}, \ldots, \texttt{mk})\ \texttt{<<}\ \texttt{x}) \longmapsto (\textbf{split}\,(\texttt{m1})\ \texttt{o}\ \ldots\ \texttt{o}\ \textbf{split}\,(\texttt{mk})\ \texttt{<<}\ \texttt{x})$$

Where `o` denotes function composition, *i.e.*, `f o g << y` reads the same as `f(g(y))`. The reverse of **splitND** is **joinND**. The LIFT primitive **join** is equivalent to $\textbf{joinND}_2$. Flattening three dimensions is expressed as follows:

$$(\textbf{joinND}_3\ \texttt{<<}\ \texttt{x}) \longmapsto (\textbf{join}\ \texttt{o}\ \textbf{join}\ \texttt{<<}\ \texttt{x})$$

Reordering Macros The two-dimensional $\textbf{slideND}_2$ is applied on a two-dimensional argument, sliding a rectangular window across the two corresponding dimensions:

$$\begin{aligned}&(\textbf{slideND}_2\,(\texttt{size1}, \texttt{step1}, \texttt{size2}, \texttt{step2})\ \texttt{<<}\ \texttt{x}) \longmapsto\\ &\textbf{map}\,(\textbf{transpose})\ \texttt{o}\ \textbf{slide}\,(\texttt{size1}, \texttt{step1})\ \texttt{o}\ \textbf{map}\,(\textbf{slide}\,(\texttt{size2}, \texttt{step2}))\ \texttt{<<}\ \texttt{x}\end{aligned}$$

Where transposition reorders dimensions intuitively: the outer dimensions correspond to sliding windows, and the inner dimensions correspond to window elements.

Multidimensional transposition reorders the input dimensions according to the new order specified by the arguments. k dimensions are numbered from zero to $k-1$, with zero referring to the outermost dimension. The new dimension order is specified through the position of the dimension indices in the **transposeND** arguments. The

original order is specified as `(0,1,2,3)`; **transposeND**$_4$`(0,1,2,3)` is a no-op, while **transposeND**$_2$`(1,0)` is equivalent to **transpose**.

For example, the following expression makes the innermost dimension of a 4D input (dimension #3) the outermost; the second innermost dimension (#2) is moved outwards once:

```
1  (transposeND4(3, 0, 2, 1) << x) ⟼
2   map(map(transpose)) o
3   transpose o map(transpose) o map(map(transpose)) << x
```

Where line 3 shifts the innermost dimension (#3) to the top, and line 2 shifts the second innermost dimension (#2) up by one. Like other macros, **transposeND** is expanded programmatically by the compiler based on the argument values.

While **transpose** and **transposeND** change how the argument is read by the subsequent function, **transposeW** and **transposeWND** affect the writes of the preceding function. We will see in Section 2.3.2.2 how the former translates into an input view transformation and the latter – into the output view transformation. The **transposeWND** macro is expanded the same as **transposeND** using **transposeW** instead of **transpose**.

Array Length Modifiers **padND**$_2$ appends the provided value at the four edges of the two-dimensional argument:

```
1  (padND2(top, bottom, left, right, value("c", T)) << (x: [[T]m]n)) ⟼
2   map(pad(left, right, value("c", T))) o
3   pad(top, bottom, value("c", T)) << x
```

Where **value** is an expression with a constant value of type T equal to `c`.

The reverse operation to padding is truncation. LIFT supports the **take** macro, which preserves the first m elements of the argument and discards the rest. **take** is achieved using the **pad** primitive with negative amount of padding:

```
(take(m) << (x: [T]n)) ⟼ (pad(0, -(n-m), value("0", T)) << x)
```

During negative padding, the value passed to **pad** is ignored. The type checker propagates the reduced array length through the rest of the LIFT expression, the loops make fewer iterations, and the "depadded" values are never accessed.

$$\begin{aligned}
\texttt{Lambda} &: (x_1 : T_1,\ x2 : T_2,\ \ldots) \Rightarrow U \\
\textbf{let} &: (x : T) \Rightarrow U \\
\texttt{UserFun} &: (x_1 : T_1,\ x2 : T_2,\ \ldots) \Rightarrow U \\
\textbf{oclKernel} &: (f : (T_1,\ T_2,\ \ldots \Rightarrow U),\ x_1 : T_1,\ x2 : T_2,\ \ldots) \Rightarrow U
\end{aligned}$$

Figure 2.10: Function literals of the LIFT IR. `Lambda` and `UserFun` refer to classes of functions; **`let`** and **`oclKernel`** are IR primitives.

```
1 map(elX ⇒ (elG ⇒
2     map(elY ⇒ f(elG, elY)) << y
3   ) << g(elX)
4 ) << x
```

(a)

```
1 map(elX ⇒ let(elG ⇒
2     map(elY ⇒ f(elG, elY)) << y
3   ) << g(elX)
4 ) << x
```

(b)

```
1 map(elX ⇒
2   map(elY ⇒ f(g(elX), elY)) << y
3 ) << x
```

(c)

Figure 2.11: Examples illustrating the difference between an ordinary LIFT lambda and the **`let`** primitive. Due to argument inlining, the expression (a) is equivalent to (c).

2.3.1.4 Function Literals

LIFT depends on four classes of function literals that are used as the top-level Abstract Syntax Tree (AST) node, composed with and nested in other function literals and patterns.

Lambda Anonymous functions are first-class citizens of the IR. A lambda has parameters and a body expression, which defines what happens with the parameters. The notation `(f : T ⇒ U)` implies a lambda with a parameter of type T and a body expression of type U. An alternative notation omits the parameter declaration and application when the parameter is used only once at the top level of the function body. For example, the anonymous function of a **`map`** in **`map`**`(x ⇒ f << x)` is sometimes shortened as follows: **`map`**`(f)`.

Let The **`let`** primitive declares a special type of lambda, where the lambda argument is evaluated before the lambda body. On the other hand, the regular `lambda` evaluates the arguments that are used only once upon the first usage. Consider the example in

Figure 2.11a, where a function `f` is applied on all pairs of elements of `x` and `y`; elements of `x` are also preprocessed using function `g`. The ordinary LIFT lambda inlines the function argument to simplify the expression. However, this results in evaluating `g` once for every element of `y` and `x`, therefore the example (a) is equivalent to (c). The **`let`** primitive in (a) prevents argument inlining and evaluates `g(elX)` before entering the **`map`** over `y`.

User Function A special type of lambda is a User function (UF) whose body is defined in the target-specific language, *e.g.*, OpenCL C. A UF performs operations on data in memory in the order defined by LIFT control structures and data layout transformers. UFs include OpenCL primitives such as `dot` and arithmetic operands; an identity function `id` issues a copy operation to a new buffer. An expression containing a UF is considered "concrete" because it results in operations on memory; otherwise, the expression is "abstract".

OpenCL Kernel Function A lambda wrapped in **`oclKernel`** produces a separate OpenCL kernel [Sto21]. A LIFT program supports multiple **`oclKernel`** instances – the compiler generates an accompanying host code which schedules kernel execution and passes data between kernels.

2.3.1.5 OpenCL Primitives

LIFT IR is defined on two levels: platform-agnostic and platform-specific. In order to support the generation of code for parallel accelerators, LIFT introduces low-level primitives that are tightly coupled with the hardware-specific programming model. This section reviews the main OpenCL primitives used in this work to target a mobile GPU (Figure 2.12).

Scheduled Control Structures While the **`reduce`** pattern is always sequential, LIFT complements the sequential **`mapSeq`** with several parallel variants of **`map`** for OpenCL: `mapGlobal`, **`mapWrg`** and **`mapLcl`**. These variants capture the OpenCL programming model, where work can be parallelised across a flat thread index domain (global), work groups and threads within work groups (local). For each of the three domains, OpenCL permits parallelising in three dimensions. For local and global domains, dimension 0 indexes neighbouring threads.

$$\begin{aligned}
\textbf{mapGlb} &: (dim : int,\ f : T \Rightarrow U,\ x : [T]_n) \Rightarrow [U]_n \\
\textbf{mapWrg} &: (dim : int,\ f : T \Rightarrow U,\ x : [T]_n) \Rightarrow [U]_n \\
\textbf{mapLcl} &: (dim : int,\ f : T \Rightarrow U,\ x : [T]_n) \Rightarrow [U]_n \\
\textbf{mapSeq} &: (f : T \Rightarrow U,\ x : [T]_n) \Rightarrow [U]_n \\
\textbf{reduceSeq} &: (init : U,\ f : (U,T) \Rightarrow U,\ x : [T]_n) \Rightarrow [U]_1
\end{aligned}$$

(a) Scheduled control structures

$$\begin{aligned}
\textbf{toHost} &: (f : T \Rightarrow U,\ x : T) \Rightarrow U \\
\textbf{toGPU} &: (f : T \Rightarrow U,\ x : T) \Rightarrow U \\
\textbf{toGlobal} &: (f : T \Rightarrow U,\ x : T) \Rightarrow U \\
\textbf{toLocal} &: (f : T \Rightarrow U,\ x : T) \Rightarrow U \\
\textbf{toPrivate} &: (f : T \Rightarrow U,\ x : T) \Rightarrow U \\
\textbf{toMem} &: (mem : Memory,\ f : T \Rightarrow U,\ x : T) \Rightarrow U
\end{aligned}$$

(b) Address space patterns and a macro

$$\begin{aligned}
\textbf{asVector} &: (n : int,\ x : [T]_m) \Rightarrow [\langle T \rangle_n]_{m/n} \\
\textbf{asScalar} &: (x : [\langle T \rangle_n]_m) \Rightarrow [T]_{m \times n} \\
\textbf{vectorise} &: (n : int,\ f : T \Rightarrow U,\ x : T) \Rightarrow \langle U \rangle_n
\end{aligned}$$

(c) Vectorisation patterns

Figure 2.12: Low-level OpenCL patterns and macros of the LIFT IR.

Address Space Patterns LIFT captures the OpenCL device memory model using primitives **toGlobal**, **toLocal** and **toPrivate**. Wrapping an expression in these forces the output into global, shared or private memory, respectively. OpenCL runtime subsequently maps these concepts to the memories available on hardware such as off-chip DRAM, on-chip SRAM and registers. Due to the differences in memory bandwidths and latencies, LIFT must be able to distinguish between address spaces.

For host code generation, LIFT provides the **toGPU** primitive to move data from host memory to the global memory on the GPU, and **toHost** to do the reverse. The **toMem** macro is syntactic sugar to generate one of the address space patterns based on the memory name passed as an argument.

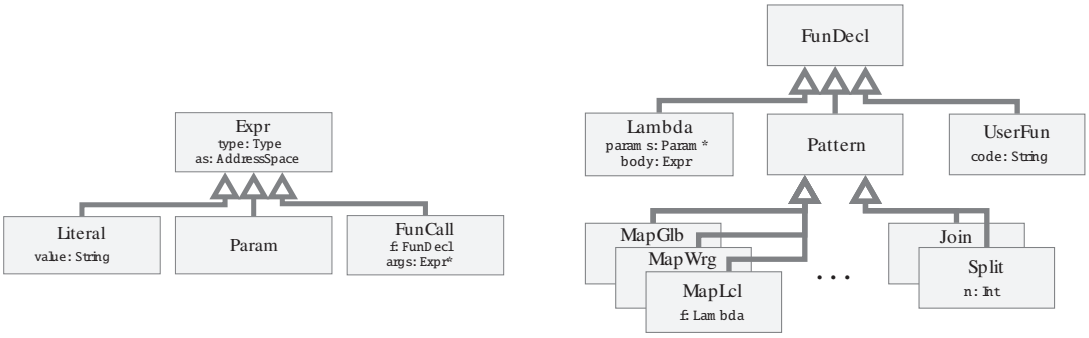


Figure 2.13: Class diagram of the LIFT IR implementation [Ste17].

Vectorisation Patterns LIFT provides **`asVector`** and **`asScalar`** which cast scalar arrays to vector types (*e.g.*, float4) and vice versa. **`vectorise`** is provided to vectorise any scalar operator.

2.3.2 The LIFT Compiler

The LIFT IR is implemented in Scala as an embedded language. The entire IR is captured by the classes in Figure 2.13. The two main entities of the IR implementation are an expression and a function declaration. An expression has a value of a known type; a function declaration can produce a new value when applied to an expression.

An expression is a literal (a constant), a lambda parameter or a function call; a function call is an application of a function declaration on one or more argument expressions. A lambda declaration consists of a list of lambda parameters and a body; a lambda body is an expression that may use lambda parameters. A `Pattern` captures high-level and low-level IR primitives.

The LIFT compiler is implemented in Scala. The compilation flow presented in Figure 2.14 is split into two stages: optimisation and code generation. The optimisation is performed using a system of rewrite rules expressing algorithmic and hardware-specific design choices. The result of rewriting is a LIFT expression that is semantically equivalent to the original expression.

Code generation is performed by walking the AST and iteratively annotating the nodes with information on memory allocation, array indexing and synchronisation requirements. The compiler maintains the rich algorithmic representation of a functional IR until all design decisions are made, and the target AST can be generated and printed. Next, we look at the main LIFT compilation passes individually.

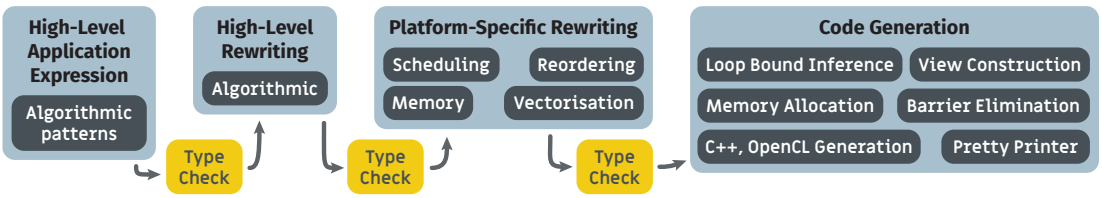


Figure 2.14: Overview of the LIFT compilation workflow.

2.3.2.1 Rewrite Rule System

The LIFT IR yields well to automated program analysis thanks to the lack of side effects and its high-level program representation. The compiler leverages these properties using rewrite rules — local semantics-preserving transformations of the AST defined on IR patterns. A pattern is expressed as a small AST tree restricted to a specific combination of IR primitives and types. A few dozen rewrite rules can be used to create a search space covering design decisions required to achieve high performance [Ste16]. We now look at notable examples of rewrite rules.

Consider `map(f) << x`, where function `f` is applied on each element of `x`. If `x` contains many elements, some target platforms require tiling `x`, so dedicated work groups in local memory can process different chunks. In LIFT, such transformation could be achieved using the *Split-Join* rewrite rule:

$$\texttt{(map(f) << x)} \longmapsto \texttt{join o map(map(f)) o split(chunkSize) << x}$$

Where the long arrow ($\longmapsto$) denotes rewriting an expression into its semantic equivalent. The transformed expression splits `x` into chunks and maps an expression on each chunk, in which a nested **`map`** applies `f` on each element of a chunk. The resulting two-dimensional array is flattened using **`join`** so that the original expression type is preserved.

Tiling the loop is an algorithmic transformation which creates platform-specific rewriting opportunities. To lower an abstract LIFT expression in a platform-specific equivalent, the following rewrite rules are used:

$$\texttt{(map(f) << x)} \longmapsto \texttt{(mapWrg(0)(f) << x)}$$
$$\texttt{(map(f) << x)} \longmapsto \texttt{(mapLcl(0)(f) << x)}$$

These lowering rules apply to the tiled expression above as follows:

```
(join ∘ map(map(f)) ∘ split(chunkSize) << x) ⟼
 join ∘ mapWrg(0)(mapLcl(0)(f)) ∘ split(chunkSize) << x
```

The result is a parallelised expression.

Platforms such as Mali GPU provide vectorised operations such as loads, stores and the built-in dot operator. This fine-grained parallelism is exploited in LIFT using the following rewrite rule.

```
(map(f) << x) ⟼
 asScalar ∘ map(vectorise(f)) ∘ asVector(vectorLength) << x
```

asScalar at the end of the expression preserves the original return type. LIFT code generator produces corresponding OpenCL type cast calls.

Since rewrite rules produce expressions in the same IR, the entire rewrite rule system is decoupled from code generation. Changing or adding new rules to support new optimisations and platforms requires no alteration of the rest of the compiler. Course-grained transformations are achieved by combining rules in chains. The rewriting space can be performed exhaustively, heuristically or using intelligent search methods.

2.3.2.2 View System

Array indexing-based languages express memory access patterns with arithmetic functions defined on loop counters, work group and thread indices and data dimensions. The resulting arithmetic expressions are hard to produce correctly by hand. Fusing multiple data layout transformations in a single expression results in an opaque representation. This strains the static analysis methods used to detect out-of-bounds accesses and determine data dependencies within the same buffer.

The LIFT compiler tracks data layout transformations using an internal representation structure called views [Ste17]. Each transformation produces a separate view; composing data layout patterns produces a view tree capturing the entire layout modification history. This rich representation is preserved for as long as possible: the transformations remain virtual until they are committed to memory by a user function.

View Construction Each view defines how the corresponding primitive affects the array index. The root of the view tree is a `ViewMem`, grounding the tree in a specific memory buffer. Because of tuples, a view tree can have multiple roots. The leaves of

the view tree are user functions accessing one of the memory buffers in the tree roots. Traversing the tree in the bottom-up order produces an array index expression; the root produces a pointer to a specific memory buffer.

For each expression, LIFT produces two views: an input view and an output view. The input view affects how the expression value is read by the subsequent primitives. The output view affects how the expression value is produced by the preceding primitives.

View Consumption When a user function call is generated, views are used to produce array index expressions implementing the transformed data layout. Array access generation starts at the corresponding leaf of the tree and proceeds in the bottom-up order. During tree traversal, the compiler generates an access stack of indexing subexpressions corresponding to array dimensions. Each traversed view modifies the stack, adding or changing the indexing subexpressions. Once a root is reached, the stack is converted into a single index expression by multiplying the subexpressions by the corresponding array dimension sizes.

Accessing an array element through a **`map`** or a **`reduce`** parameter results in a `ViewAccess`, which inserts the corresponding loop variable into the access stack. **`tuple`** and **`zip`** primitives bifurcate the tree above the `ViewTuple` and `ViewZip` nodes, respectively. **`get`** introduces a `ViewGet` tree node, which is used by the compiler to navigate the bifurcated branches.

`ViewTranspose` reorders the loop iterators in the access stack, swapping the dimensions on which the iterators are applied. `ViewJoin` converts a 1D iterator in the array access stack into two iterators over the original two dimensions before the **`join`**. `ViewSplit` does the reverse. `ViewPad` wraps the entire index expression in an inline if-conditional. The conditional returns a constant for the indices pointing to the padded areas; otherwise, array values are returned.

The view tree is consumed during OpenCL AST generation, one of the last steps before pretty-printing. Until then, views are used for static analysis: a view tree can be flattened and filtered by the relevant transformations to extract information quickly. We will see how this rich representation is leveraged in Section 5.4 for synchronisation barrier insertion.

2.3.2.3 Memory Allocation

The LIFT compiler allocates memory by analysing the AST. While the input buffer size is trivial to determine based on the user-defined argument types, the output and intermediate buffers are inferred from the functional expression. Buffers are allocated for the results of each UF used in the expression. A UF defines the type of the result buffer and the size of each scalar or vector element stored within. Based on the position of the UF in the AST, the compiler determines the Address space (AS) and the size of the buffer.

The buffer AS is inferred based on the function argument AS and the encapsulating AS casters such as **`toGlobal`**, **`toLocal`** and **`toPrivate`**. The following aspects of the AST determine buffer size:

- Functional patterns encapsulating the UF. For example, **`map`** produces one output per each element of the argument array; **`reduce`** produces one output for the entire argument.

- Data layout transformations applied on the UF arguments. For example, the **`pad`** primitive changes the size of the output based on the amount of padding.

- Parallel mapping of the IR primitives encapsulating the UF. The buffer size is affected by an encapsulating primitive only if the buffer AS is accessible in the parallel domain of the primitive. For example, **`mapWrg`** affects the size of the global buffers only, since a work group cannot return private (register) or local (per-core) memory. **`mapSeq`** on the other hand affects the size of buffers in all ASs, since all memories support sequential access.

- Function composition. Intermediate buffers are allocated for the subexpressions composed with expressions containing UFs; the UFs whose results are returned by the entire LIFT expression yield output buffers.

The LIFT compiler allocates a buffer for each UF. The buffer information is recorded on multiple AST nodes: the UF, the patterns nesting the UF and the subtrees accessing the results produced by the UF. Next, the compiler uses the propagated memory information to determine data dependencies among the accesses of the allocated buffers and to prevent data races.

Algorithm 1: Example patterns of the pattern matching-based barrier placement approach, where setting a barrier on a primitive forces the code generator to insert a barrier after the primitive implementation in the OpenCL kernel.

```
1  switch (e: expression)
2    case map(mapLcl_A(dim0, f0)) where f0 accesses local memory do
3      set barrier on mapLcl_A
4    case f0 o mapLcl_A(dim0, f1) where f0 transforms data layout do
5      set barrier on mapLcl_A
6    case mapSeq(f0) o mapLcl_A(dim0, f1) do
7      set barrier on mapLcl_A
8    case mapLcl_A(dim0, mapLcl_B(dim1, f0))
9     where barrier is set on mapLcl_B do
10     remove barrier from mapLcl_B
11   ...
```

2.3.2.4 Barrier Elimination

LIFT uses the IR pattern matching-based approach to detect data dependencies, with patterns defined on both the IR primitives, types and address spaces. Algorithm 1 presents example patterns used by the LIFT to determine barrier placement based on the detected data dependencies. The case on lines 2 to 3 inserts a barrier when a work group reads or writes shared data in parallel in a loop, since there might be data dependencies between iterations of the loop. On lines 4 to 5, a barrier is inserted when the layout of the data produced in parallel is transformed; in such case, a subsequent parallel loop might map threads onto the shared data differently, with a data dependency.

The pattern on lines 6 to 7 is based on the parallel mapping of functions `f0` and `f1`: first, each thread applies `f1` on one element of the argument array each; then, each thread applies `f0` on all outputs of **`mapLcl`**$_A$ sequentially. **`mapSeq`** can be executed only after all iterations of **`mapLcl`**$_A$ are completed, hence the barrier is inserted.

The example on lines 8 to 10 pattern-matches a parallel loop nest. To achieve a more efficient barrier placement, the inner barrier is removed under the assumption that a loop is placed on the outer barrier placed to satisfy the same data dependency.

```
1  def stencil2D(weights   : [[float]_3]_3,
2                inputData : [[float]_width]_height
3                ) : [[float]_width-2]_height-2 = {
4    mapWrg(0)(
5      mapLcl(0)(neighbourhood ⇒ toGlobal(id) o
6        reduceSeq(
7          init = toPrivate(id) << value(0, float),
8          f = (acc, (l,r)) ⇒ acc + l * r
9        ) << zip(join(neighbourhood), join(weights))
10     )
11   ) << slideND_2(3,1,3,1) << inputData }
```

Listing 2.2: Example of a 2D Stencil.

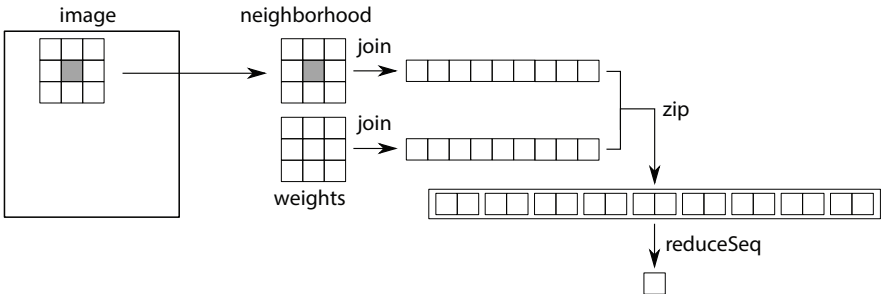


Figure 2.15: Visualisation of the 2D stencil example in Listing 2.2.

2.3.2.5 Code Generation Example

We now look at an example expression illustrating code generation in LIFT. Listing 2.2 encodes a stencil computation, which forms the basis for convolution.

Expression The function `stencil2D` takes a 3×3 array of weights and a 2D image array. The body consists of two parts: data layout transformation and core computation. For data layout, it first creates a 3×3 sliding window using **slideND**$_2$ in line 11, which results in a 2D neighbourhood. Then, two **map**s are used in line 5 to schedule the work to each local thread in each work group running on a GPU.

Each thread performs the core computation on a neighbourhood in lines 9 to 6. This process is visualised in fig. 2.15. First, two instances of **join** are used to flatten the two 2D arrays: the weight array and the sliding window into simpler 1D arrays. Then, the two 1D arrays are zipped into a single array of tuples and reduced sequentially to a single scalar value, representing convolution output for one neighbourhood.

```
1  kernel stencil2D(const global float * weights,
2                   const global float * inputData,
3                   global float * outData,
4                   int width, int height) {
5    for (int wrg_i = get_group_id(0); wrg_i < height;          // mapWrg(0)
6         wrg_i += get_num_groups(0)) {
7      for (int lcl_i = get_local_id(0); lcl_i < width;         // mapLcl(0)
8           lcl_i += get_local_size(0)) {
9        private float acc = 0.0f;                              // toPrivate(id)
10       for (int i = 0; i < 9; i++)                            // reduceSeq
11         acc = acc + weights[i] * inputData[(i%3) + (i/3)*width +
12                                            lcl_i + wrg_i * width];
13       out[lcl_i + wrg_i * width] = acc; }}}                  // toGlobal(id)
```

Listing 2.3: OpenCL code generated with LIFT for the 2D stencil example in Listing 2.2

Generated Code LIFT produces parallel OpenCL code by walking the AST of the example and emits code for each primitive. The exceptions to this process are the primitives that are changing the data layout, such as **join**, **split**, **zip**, **pad** or **slide**. In these cases, the compiler constructs the views representing the layout transformations; the views are used to generate correct accesses for other primitives. Listing 2.3 shows the code produced by the LIFT compiler (with minor cosmetic changes such as naming and indentation) for the example in listing 2.2. First, a for-loop for distributing the work among work groups in the dimension 0 is generated on line 5 corresponding to the **mapWrg**. Then, a second loop for distributing the work among local threads is generated on line 7 corresponding to the **mapLcl**. The reduction accumulator is allocated in private memory and initialised on line 9. Following that is the reduction for-loop, which accumulates the results of multiplying an element of the weight together with the corresponding element of the input data. The array accesses are automatically generated using the information in the view built from the **slide** and **zip** primitives.

2.4 Summary

This chapter has provided the background required for the rest of the thesis. The introduction to convolution has laid out this application's challenges tackled in the later chapters. The direct and GEMM-based convolution methods require different approaches; addressing both within the same framework makes a case for the general-

isability of the optimisation method. The chapter has also introduced GPU programming in OpenCL, the target language of this thesis' code generation efforts. Finally, the chapter has provided an overview of the LIFT IR and the compiler internals, outlining the starting point of this work.

Chapter 3

Related Work

Parallel programming abstractions are broadly categorised into two categories: explicitly and implicitly parallel. Those capturing explicit parallelism are provided either at a low level capturing a particular hardware architecture or a programming model, or at a higher, more structured level. Structured parallel programming organises the code in patterns with well-defined semantics [McC12]. These algorithmic patterns provide code generators with leeway to adapt the algorithm to diverse architectures, enabling performance portability.

Implicitly parallel IRs model dependencies between data and operations without restricting them to a specific ordering or scheduling. These IRs decouple the algorithm from the implementation, shifting the burden of code parallelisation from the user to the code generator. An implicitly parallel IR that captures algorithmic intent yields well to automatic parallelisation [Pre19].

The rest of this chapter is organised in the ascending order of IR abstraction levels. Section 3.1 discusses explicitly parallel programming APIs, which provide fine-grained control over scheduling and memory management in GPUs. The section also covers notable extensions of mainstream sequential languages, which help users parallelise their code by exposing scheduling primitives. Next, optimised kernel libraries are discussed, which provide low-level implementations of domain-specific patterns and coarse-grained primitives.

Section 3.2 covers the techniques of capturing and detecting implicit parallelism in applications. First, we look into the automated approaches to extracting parallelism from sequential IRs. With little or no user input, these techniques find higher-level patterns in low-level sequential code and expose them for automated parallelisation. Next, the chapter covers algorithmic skeletons, which encode parallelisable patterns

and abstract most of the scheduling complexity away from the user. Then, the chapter discusses the two paradigms that are relevant to this thesis: computational graphs and functional IRs. Finally, Section 3.3 covers three specific optimisation techniques: auto-tuning, constraint-based parallelisation and user-guided optimisations.

3.1 Explicitly Parallel Approaches to GPU Programming

Originally designed for graphics processing applications, GPUs expose plenty of power-efficient SIMD processing units tailored for simple floating-point operations. This made GPU an accelerator of choice for data-parallel applications such as machine learning. Harnessing the power of multicore systems required dedicated APIs. We start with explicitly parallel interfaces, relying on the user for task partitioning, scheduling and synchronisation [Sér99]. These low-level APIs provide users with fine-grained control of minute hardware units for optimum resource allocation. To ease framework adoption, most of these languages are embedded in established sequential languages, implemented as annotations or interfaces over libraries.

3.1.1 Low-Level Parallel APIs

Several low-level generic APIs have been released to expose scheduling and memory management functionality to the users. One of the first programming models for General-Purpose computing on Graphics Processing Units (GPGPU) is the streaming model [Buc04]. Data-parallel fine-grained functions called kernels are applied on the sequences of independent data elements in memory called streams. Multi-stage reductions are used to produce a single output from a data stream.

NVIDIA introduced the CUDA language, compiler and runtime system for GPGPU in 2007 [Nic08]. CUDA remains the low-level language of choice for high-performance programming on the NVIDIA GPUs. However, restriction to one brand of GPUs motivated work on open standards to reduce the efforts required to port applications to new platforms.

OpenCL was released in 2009 by a consortium of large organisations as an open standard to support multiple device architectures using one programming model [Khr22]. To date, OpenCL supports CPUs, GPUs, FPGAs and ASICs; a single OpenCL program can manage a heterogeneous system combining multiple devices. However, OpenCL implementations are not directly portable without a loss in performance: optimising a

solution for one platform may reduce its performance on others.

The OpenMP API extends C, C++ and Fortran with compiler directives to expose loop and task-based parallelism [Ope08]. OpenMP uses shared memory and fork-join execution models. These abstractions enable both CPU [Chr11] and GPU [Bey11] code generation. Designed for productivity, OpenMP simplifies access to parallel accelerators; however, optimal scheduling still requires high-performance programming expertise.

OpenACC adds a layer of abstraction over OpenCL and CUDA [Wie12]. The compiler abstracts the details of the target APIs away from the user using loop and code region-based annotations in C/C++ and Fortran. Although this approach simplifies parallel programming, it does so at the expense of performance portability. By representing the shared features across a range of devices, OpenACC misses architectural specifics which are crucial for performance [Ope15]. Both OpenACC and OpenMP have been shown to underperform against the OpenCL-based solutions [She12; Tho12].

The SYCL open standard extends C++ to generate SPIR, SPIR-V and OpenCL kernels [Khr15]. SYCL generates the boilerplate host code and provides type safety across heterogeneous platforms. The same SYCL representation is used across CPUs and GPUs.

The SYCL-like approach of extending a mainstream sequential language or IR retroactively has been adopted by many projects to lower barriers to entry into parallel acceleration. Coarray Fortran (CAF) adds the shared memory programming model to FORTRAN [Num98]. Unified Parallel C (UPC) exposes the shared and distributed memory models of parallel accelerators in C [Car99]. PACXX adopts a multi-stage programming approach to specialise GPU kernels JIT-compiled from extended C++ [Hai16]. The C++17 standard introduced parallel versions of popular STL functions [ISO20]. CPU parallelisation primitives have been added to C++ by the Intel Threaded Building Blocks (TBB) library [Rei07]. Numba [Lam15] and Copperhead [Cat11] extend Python to generate GPU kernels provided that the user limits themselves to a restricted subset of Python.

The LLVM-based NVIDIA's CUDA Compiler (NVCC) translates CUDA, C, C++ and FORTRAN into the NNVM IR, which is compiled into the Parallel Thread Execution (PTX) virtual Instruction Set Architecture (ISA) [NVI18]. The open-source GPUCC compiler improves over NVCC in terms of the compilation speed, generated code efficiency and the range of supported C++11 and C++14 features [Wu16].

The solutions based on explicitly parallel APIs have limited performance porta-

bility. Any one parallel programming model does not capture the full diversity of parallel accelerator architectures. Extending an explicitly parallel implementation for a new platform requires significant engineering effort. We now look at an alternative approach, which abstracts away some of the implementational details from the user.

3.1.2 Kernel Libraries

Statically compiled kernels for popular computation blocks provide robust, predictable performance and are well-optimised by high-performance computing experts. Deep Learning (DL) frameworks in particular often depend on statically compiled kernels within Basic Linear Algebra Subprograms (BLAS) libraries such as Atlas [Wha01], CuBLAS [Nvi07], OpenBLAS [Xia12] and CLBlast [Nug18] to provide a well-optimised and predictable performance.

CUTLASS is a templated library for math applications providing parametrised implementations [NVI]. The C++ templates of CUTLASS are shaped around specific hardware architectures and expose finer-grained design choices than the opaque kernels in the older BLAS libraries. The burden is still on the user to choose optimal parameter values.

Although it takes little effort for the end users to tap into the high performance of the optimised BLAS kernels, these approaches fall short of leveraging the target platform potential fully for a given application. For DL applications in particular, the BLAS-based approach requires that DL units are lowered to corresponding linear algebra operations, which is non-trivial with the multitude of variations across DL model dimensions and hardware specifications. It often comes at the cost of lost opportunities for domain-specific optimisations such as layer fusion.

The Stream-K project tackles this inflexibility in the CUTLASS library [Osa23]. An automated parallelisation pass improves the SM core utilisation in GPUs from 75% to 90% compared to the traditional data-parallel tiling methods. The improvement is achieved by partitioning tasks at the fine-grained scale of Multiply-accumulate operation (MAC) loop iterations. This work is limited to dense GEMM kernels.

Multiple hardware vendors released libraries with optimised DL kernels. This includes Intel oneDNN [Wan14], Nvidia TensorRT [NVI22], cuDNN [Che14], ARM Compute Library [Arm21] and AMD MIOpen [Kha19]. These libraries provide domain-specific optimisations such as TensorRT's layer fusion and low-bit quantisation, convolution kernel auto-tuning in MIOpen and Intel's automatic detection of Vector Neural

Network Instruction (AVX512 VNNI) usage opportunities. Due to the manual development costs, these libraries lag behind the rapid evolution of DL architectures and releases of new hardware models, and cannot provide optimal implementations for new problems.

Other works have recently explored efficient implementations of direct convolution [Geo18; Zha18; And17] but are limited in the scope of their available target platforms. In particular, [Geo18; Zha18] are reliant on the availability of SIMD instructions and are specific to CPUs.

Kernel libraries are constrained in how much they can adapt to the target hardware relying just on tuning and handwritten code selection. Kernels developed for a specific platform do not perform as well on other platforms; this holds even for the cross-platform OpenCL programming model [Rem16]. We now look into the implicitly parallel IRs, which raise the level of abstraction higher.

3.2 Implicit Parallelism for Code Generation

The increased diversity and heterogeneity of accelerator architectures have made it more challenging to achieve high performance through manual implementation. Low-level implementations are not portable without the loss in correctness or performance, incurring high development costs for users. Similarly inflexible are the optimised kernel libraries, where the development burden is shifted onto the library developers.

This rigidity motivated decoupling the problem specification from implementation further. Changing the implementation requires an understanding of the algorithmic intent to ensure that the changes do not break the semantics of the original algorithm. This meta-information is not easily inferred from a low-level IR, where the programming model is moulded around a specific hardware architecture. Data dependencies are obscured through complex index array expressions, thread synchronisation primitives and pointer aliasing.

The rest of this section discusses automated parallelism extraction, the notable IRs capturing implicit parallelism and the optimisation methods they enable.

3.2.1 Automatic Extraction of Parallelism

Loops pervade compute-intensive programs thanks to the succinct representation of repetitive tasks. Automatic parallelisation techniques often use loops as parallelisation

targets and map iterations onto parallel processing units. This requires dependency analysis to detect independent iterations and achieve loop transformations which expose more parallelism. Loop nesting and complex dependencies between loop iterations complicate iteration space analysis and loop optimisations for performance.

Polyhedral approaches use geometric representations of loops, where each iteration is represented as a lattice point on a polyhedron, and dependencies among iterations become apparent. This paradigm shift enables reasoning about loops indirectly in geometric space, which can be leveraged for parallelism detection and loop optimisation.

Polyhedral compilers extract and optimise parallelism through linear programming, affine transformations and data access optimisations. This includes PPCG [Ver13], LetSee [Pou08], Tensor Comprehensions (TC) [Vas18], PlaidML [Zer19] and Tiramisu [Bag19]. TC uses polyhedral IR to create a search space of loop transformations and depends on user-supplied heuristics to prevent search space explosion. The generated kernels perform well on desktop GPUs, but the transformation framework is not flexible enough for low-effort extensions onto other platforms due in part to the hard-coded memory promotion strategies. The user-supplied constraints are too fine-grained and thus do not truncate search space sufficiently; TC's higher-level mapping options are part of the compiler framework and cannot be easily extended.

Pluto is a source-to-source code generator, which parallelises C and FORTRAN loops through polyhedral dependency analysis [Bon08]. Affine loop nests are parallelised efficiently through loop tiling, fusion and unrolling. Non-affine loop parallelisation is not as efficient; the framework also permits code transformations that break program semantics and produce incorrect results [Pre19].

In general, polyhedral optimisations are computationally expensive making polyhedral compilers slower than the alternatives [Dav20]. Extra effort is required to mitigate this aspect, such as offline statement clustering proposed in [Bag20]. Furthermore, loops must possess certain properties to be expressible in a polyhedral model. Specifically, loop bounds must be expressible with linear affine expressions.

AlPyNa [Jac19] parallelises Python code and produces CUDA kernels using the Numba JIT compiler [Lam15]. AlPyNa tackles the ambiguity of Python's dynamic nature using staged dependence analysis. Parallelisable loops are identified at runtime, when the loop bounds and variable types are known.

Several frameworks focus on detecting stencils – grid-based memory access patterns. Stencils are identified through code analysis and exposed for automatic parallelisation. This technique is called lifting, where a high-level algorithm is constructed

from a semantically equivalent low-level implementation. Helium [Men15] applies lifting to x86 binaries and produces equivalent Halide expressions for reparallelisation. The constraint-based Idiom Description Language [Gin18] is used to detect stencils in sequential C/C++ programs, and produce optimised BLAS, Halide and LIFT implementations. Multi-level Tactics [Che21] uses Tactics Description Language (TDL) implemented in the MLIR compiler infrastructure [Lat20] to perform progressive raising of C++ programs through pattern-matching.

Producing efficient implementations through automatic parallelism extraction remains challenging. We now look at the expressive high-level IRs which capture and expose implicit parallelism for automatic parallel code generation.

3.2.2 Algorithmic Skeletons

Low-level parallel IRs such as CUDA and OpenCL are constrained by the hardware architectures they target. They do not require a specific algorithmic structure, which makes them generalisable to a broad range of applications. Optimised kernel libraries trade that generalisability for expressive power by providing course-grained procedures. In terms of the overall application structure though, these libraries are similarly unconstrained: their kernels can be freely combined in any arrangement.

A program expressed using algorithmic skeletons [Col89; Col04] adheres to a more salient and tractable structure than the two approaches above. Skeletons express patterns of computation – high-level properties of control and data flow in a program. In contrast to more monolithic procedures offered by the kernel libraries, skeletons are higher-order procedures that construct the final program based on the user-provided functions. Algorithmic skeletons are not constrained by any one hardware architecture and are therefore a popular choice of representation in the pursuit of performance portability.

Algorithmic skeletons have been used as interfaces between users and parallel accelerators. A notable application of the skeleton-like approach is Google's MapReduce [Dea08]. These frameworks target heterogeneous and distributed systems using an API centred around two algorithmic skeletons: *map* and *reduce*. The former applies a user-provided function on each element of the key-value pair set; the latter applies the user-provided function on all input elements to produce a single output. In this approach, the user describes the computation without declaring a parallelisation strategy explicitly. All design decisions are made by the runtime automatically.

Multiple MapReduce-based solutions have been released. Besides the MapReduce programming model implementation, the open-source Apache Hadoop eco-system integrates MapReduce into several programming languages and provides a data storage system for distributed computing [Whi12; Shv10]. Optimised for large clusters, Hadoop underperforms on multicore nodes due to high start-up overheads and duplicate JVM instantiations [Ste12]. The MapReduce Java framework addresses these shortcomings [Sin11].

Several frameworks expose algorithmic skeletons as C++ templates and target GPUs. SkelCL [Ste11] provides OpenCL implementations of *map*, *reduce*, *scan* and *zip* skeletons. FastFlow [Ald11] implements CUDA implementations of *farm*, *divide and conquer*, *pipeline*, *map*, *reduce* skeletons. SkePU [Enm10] targets multi-GPU systems and generates both CUDA and OpenCL.

3.2.3 Computational Graphs

A computational graph is a Directed Acyclic Graph (DAG) representing relationships between data and operators – coarse-grained algebraic functions. Explicit staging and ordering of computations entail a dependency graph, which is leveraged to map operators onto compute nodes in parallel and distributed platforms. Operator fusion helps saturate the processing units and avoid expensive data movements. Computational graphs are especially popular among DL-specific code generators, where the edges are bidirectional to represent both forward and backward propagations.

Similarly to the LIFT IR, the High-Level Optimizer (HLO) IR of the Tensorflow compiler XLA [Lea17] adopts a functional representation, where most of the primitives have no side effects. For search space exploration, XLA uses an ML-based cost model to predict performance and improve accuracy dynamically. XLA can also apply polyhedral transformations through affine MLIR dialect [Lat19]. Program transformations in XLA are correct by construction thanks to the requirement that the semantics of the language primitives – the high-level operators such as `conv` and `fft` – are fully declared. XLA leverages its well-defined primitives to perform operator fusion and optimise across several operators at once. Extending operators or transformations is costly since the perfect semantics knowledge of both needs to be maintained.

DLVM [Wei17] uses a side-effect-free static single assignment (SSA) IR that yields well to arithmetic simplification, linear algebra fusion and matrix multiplication reordering. The optimised computation graph is lowered into LLVM IR and passed

to the LLVM compiler to generate GPU kernels leveraging the mature optimisation passes of LLVM. The portability of DLVM comes at a high cost since each combination of IR primitives and target hardware needs to be optimised manually; performance evaluation is yet to be provided.

Like DLVM, the Glow compiler [Rot18] uses LLVM as a backend to target new architectures. Glow uses a two-level strongly-typed IR, in which DNN-specific nodes are first lowered into linear algebra operators over tensors, and then converted into an instruction-based representation. The optimisations of the high-level IR include redundant node elimination, tiling and operator stacking, in which fused operators are compiled into efficient kernels automatically without requiring to provide kernels for all permutations of operators. The low-level IR is used to detect opportunities to use hardware intrinsics, apply quantisation and optimise memory through layout transformations, copy elimination and buffer sharing.

Dependence on hard-coded optimisation passes in Glow harms performance portability. Glow is outmatched by other compilers on GPUs [Li20]; the compiler lacks design space exploration required to adapt computation graphs to target hardware perfectly. Adding support for new DNN architectures and hardware platforms requires changing the Glow compiler itself, incurring high development costs.

The nGraph [Cyp18] DL compiler uses its internal stateless DAG IR to perform high-level graph optimisation of an input DNN architecture; the lower-level optimisations are outsourced to Intel oneDNN on CPUs, CuDNN on NVIDIA GPUs or PlaidML on OpenCL GPUs. Similarly to LIFT, PlaidML uses a multi-level IR [Zer19] to separate algorithms from implementations and hardware-specific optimisations, thus preventing search space explosion. The PlaidML compiler uses the nested polyhedral representation to capture parallel execution and memory hierarchies using its Tile and Stripe IRs. However, PlaidML proved to be difficult to extend to new hardware architectures due to the heuristics embedded into its performance model, and strong assumptions made about the target memory structure and ISA [Bar19; Sot19; Pet20].

The Latte [Tru16] compiler supports operator fusion, cross-operator optimisations and auto-tuning. Due to explicit indexing and the lack of type and shape inference, Latte IR is verbose and does not yield well to transformations such as tiling, loop fusion and interchange. This prevents Latte from matching the complex scheduling hierarchy and partitioned memory architectures of GPUs.

3.2.4 Functional IRs

Like computational graph-based approaches, functional IRs benefit from the explicit representation of the composability of the program. Like algorithmic skeletons, data-parallel functional IRs capture computational patterns. Unimpeded by specific workload dimensions or hardware architecture, patterns constitute a fundamental model of the computation, a blueprint. Such representation is scalable which is a useful property for massively parallel and distributed accelerators.

The functional paradigm features extra properties that benefit parallelisation. The lack of explicit control flow gives code generators some leeway in execution scheduling. Stateless functions can be fused or otherwise transformed without an expensive dependency analysis. Due to limited side effects and static typing, functional programs yield well to correctness checks.

TVM builds on two functional IRs: the high-level ML-specific Relay IR [Roe18] and the more generic Halide IR [Rag13]. Halide decouples computation from scheduling by allowing users to specify a set of parametric schedules for exploration. Section 3.3.2 discusses several auto-tuners proposed for schedule exploration. Extensions for new platforms and custom operations are challenging with TVM since schedule design spaces must be implemented for each combination of custom operation and target platform [Zer19]. The usage of explicit indexing in the front-end TVM IR complicates transformations, requiring pattern-matching operations such as matrix multiplication to detect opportunities to use hardware intrinsics and optimised micro-kernels. Such an approach is inflexible on accelerators with deep memory hierarchy and built-in coarse-grained operators.

The functional data-parallel IR of Futhark [Hen17] guarantees race-free semantics. The compiler generates OpenCL and CUDA kernels targeting GPUs. Like LIFT, it provides a strong type system with multidimensional array support. Through uniqueness types, Futhark addresses the common problem in functional approaches of data duplication through in-place updates. Through `map` interchange, distribution and fusion, the compiler brings as much nested parallelism outwards as possible and parallelises the outermost loops only. The loop transformation strategy is hard-coded, which puts pressure on the programmer to choose which `map`s to interchange outwards.

Accelerate [McD13] is an established Haskell DSL for GPU computation. Like Futhark, it focuses on rewriting the AST to expose more parallelism through loop transformations. For scheduling, it relies on template skeletons, which are compiled

into CUDA code through template skeleton instantiation. This puts pressure on the compiler to preserve the skeleton representation, limiting the choice of code transformations.

The functional NOVA compiler [Col14] generates C and CUDA kernels for CPU and GPU respectively. NOVA supports nested parallelism, recursion and type polymorphism. Through the high-level IR and optimisation techniques such as aggressive inlining and loop fusion, the compiler produces high-performance kernels automatically. Parallelisation is achieved through vector flattening and unflattening.

3.3 Parallel Code Optimisation

We now look at four techniques used for parallel code optimisation. Section 3.3.1 discusses the efforts towards minimal, correct and efficient synchronisation primitive placement. Section 3.3.2 covers automated tuning space exploration as a way to make fine-grained design choices. Section 3.3.3 introduces parallelisation techniques which use constraints to truncate the design space. Finally, Section 3.3.4 discusses ways to capture human expertise to help the code generator produce efficient implementations.

3.3.1 Synchronisation Optimisation

Determining the minimal number of synchronisation points is an NP-complete problem [Mid86]. [OBo02] proposed a formalized method of inserting a provably minimal number of barriers in perfect loop nests and some imperfect loop nests. A linear-time algorithm for inserting barriers in general nested loops has been proposed in [Dar05]. [Tse95] uses communication analysis to reduce the synchronisation overhead through superfluous barrier elimination.

The problem of determining the correctness of barrier placements has been proposed in [Aik98]. The forward progress analysis has been applied to the inference of termination guarantees of concurrent programs [Sor21]. The MLIR-based GPU to CPU transpiler [Mos23] tackles synchronisation correctness by defining the barrier operation in terms of its memory side effects.

These techniques have been applied to imperative programming languages such as FORTRAN and CUDA. Complementary to these works, Section 5.4 contributes a discussion of barrier insertion in the context of functional IR.

3.3.2 Auto-Tuning

A low-level implementation is often left parametric to adapt to small differences in the workload dimensions, target platform specification and dynamic resource budgets. The parameters are chosen heuristically or through explorative methods.

OpenTuner is a multi-objective tuner for domain-specific program optimisation [Ans14]. It leverages a suite of search techniques combining random and heuristic exploration to find the best parameter value combination. By running multiple search techniques simultaneously, OpenTuner continuously evaluates the efficiency of each and prioritises those fitting the problem domain. A general approach at its core, OpenTuner performs best on simple applications [Mul15] with no interdependencies among the tuning paramaters [Ras21]. OpenTuner does not automatically detect invalid tuning parameter value combinations.

CLTune [Nug15] and the Auto-Tuning Framework (ATF) [Ras18] use constraints to truncate the search space to valid value combinations only, reducing the search time by up to 9.5× [Ras21]. CLTune provides search techniques specialised for tuning OpenCL kernels. ATF takes into account the interdependencies between parameters to parallelise the search. It also replaces CLTune's search space constraints with parameter constraints, preventing the exploration of entire ranges of invalid values. Both CLTune and ATF depend on the user input and hardware specifications to provide constraints, which presents a challenge for deeply nested applications with dozens of tuning parameters.

PetaBricks [Pho13] goes beyond numeric tuning. Using a hardcoded choice dependency graph, the compiler tunes design choices such as iteration order, data layout and storage. User-specified tuning parameters are also set automatically. The search proceeds using an evolutionary algorithm by testing each new candidate.

When parallelising the kernel, PetaBricks avoids deadlocks and race conditions by analysing the program dependency graphs at compile time. Overall, the compiler does not provide a strong correctness guarantee of the discovered solutions. PetaBricks performs consistency checks by comparing the outputs of candidate implementations to each other, and the user-provided reference implementation.

The Halide Autoscheduler [Ada19] uses a tree search algorithm to explore the design space of Halide schedules. The schedules express parallel mappings optimised towards specific hardware targets. On GPUs, the search is limited by course-grained heuristics such as assigning outermost loops to thread blocks. The cost model-based

AutoTVM [Che18b] achieves good performance on CPUs and GPUs, but the search still requires good initial schedules, placing the burden on the user to provide performance programming expertise [Sot19].

Recent work has focused on the parallelisation of TVM kernels through automatic tuning. Ansor [Zhe20a] and FlexTensor [Zhe20b] provide auto-schedulers to explore the search space of parallel mappings. The search efficiency in both auto-tuners is limited by the user-provided set of handwritten parallel templates. Ansor can generate invalid parallel mappings, resulting in search time wasted on the evaluation of bad candidates. The correctness issue has been addressed through formal correctness proofs of Halide schedules [New20]. However, functional verification is yet to be demonstrated on GPUs with an automated exploration approach like Ansor. The overall search time in Ansor has been improved using transfer-tuning, where the tuning results are reused among applications with similar operations [Gib22].

Bolt takes the auto-tuning approach of TVM programs further than AutoTVM and Ansor by exposing the target hardware details to the auto-tuner [Xin22]. Bolt combines the configurability of CUTLASS hardware-specific templates, profiling and automated search to find optimal template parameter values. Compared with the hardware-agnostic search strategies of AutoTVM and Ansor, Bolt-optimised kernels achieve 2.5× faster inference in CNNs on average.

Numerous other cost model-based approaches have been proposed to navigate the space of tuning parameters and parallel mappings [Mul16; Ada19; Sio18]. These methods use predicted or profiling-based information to find better candidates faster. These techniques are complementary to the contributions of this thesis, which focus on inferring valid choices and incorporating user input effectively.

3.3.3 Constraint-Based Parallelisation

LIFT and Spiral [Fra18] are similar in their use of rewrite rules to transform a functional IR for performance portability. Spiral also uses constraint satisfaction to ensure valid parallelisation. Its IR focuses on macro operators such as Cartesian product, direct sum and Kronecker product; the latter is used to capture parallelism. LIFT operates on more generic low-level primitives such as `map` and `reduce`.

The partially specified implementations IR [Bea19] expresses programs using scalar arithmetic operators and iteration dimensions. The compiler transforms, parallelises and vectorises the implementation through constraint satisfaction. LIFT IR is more

expressive thanks to its data layout transformations such as **transpose**. This allows explicit control over coalescing and creates vectorisation opportunities.

The Sponge compiler solves the problem of portability by using the streaming language StreamIt, thus abstracting the hardware details [Hor11; Udu09]. StreamIt captures parallelism through high-level patterns such as pipeline split-join and feedback-loop. The data flow graph is partitioned across GPU cores using an Integer Linear Program (ILP) solver, which uses profiling to guide the search. The parallelism constraints presented in Chapter 5 are close in spirit to the resource and dependence constraints of StreamIt.

Compared to StreamIt, the LIFT approach benefits from the functional IR in several ways. LIFT imposes parallelisation constraints on functional **map**s, while StreamIt generates constraints on filters, *i.e.*, operations on data. Although the LIFT approach requires more constraints, they are simpler. The data dependencies are captured implicitly by the functional patterns over stateless operations, informing constraint generation only indirectly. The dependencies between the stateful filters of StreamIt are captured explicitly within the constraints, resulting in less tractable constraint predicates. Furthermore, the focus on the streaming programming model limits the range of applications that are well-captured by the StreamIt IR.

3.3.4 User-Guided Optimisation

Human expertise in performance programming should not be completely discarded in automatic code generators lest the compilation times suffer from the search space explosion. Users should be able to communicate promising optimisational targets to the compiler succinctly. At the same time, the IR should expose few implementational details to maintain performance portability.

The TVM [Che18a] compiler achieves state-of-the-art performance across state-of-the-art DNNs using Halide schedules [Li20]. The schedules expose transformations such as tiling, different memory layouts, operator fusion, loop blocking and memory prefetching. Schedules can also invoke tensorisation, instructing the compiler to map operators onto device intrinsics. The intrinsics can be defined manually as part of TVM's larger effort to facilitate extension onto new backend libraries and accelerators through backend API and fallback CPU implementations.

The extensibility of TVM is hindered by the strong coupling between the schedules and the code generator. Adding new schedules requires altering the code generator to

support new transformations. TVM's functional Relay IR [Roe18] is not guaranteed to be compatible with a given schedule, and establishing compatibility is left up to the user. Recent work on sparse tensor optimisations in Sparse Tensor IR suffers from the same drawback [Ye22].

PetaBricks [Pho13] takes the separation of concern principle further: the user is allowed to specify both the algorithm and a set of its valid implementations, leaving it up to the compiler and runtime to choose the best candidate using an auto-tuner. The auto-tuner uses an evolutionary algorithm to find the best solution empirically [Ans09]. The generated code is further optimised using static analysis. While highly extensible, this approach limits the search to a set of manually provided implementations. In LIFT, algorithmic choices remain transparent to the user. The compiler does not depend on static analysis, exploring all optimisations at a higher level through rewrite rules.

Like PetaBricks, the Tangram code generator [Cha16; De 19] allows the user to provide multiple implementations ("codelets") of an algorithm ("spectrum"). The expressive power of the Tangram framework comes in part from the ability of codelets to invoke other codelets. Tuning parameters are inserted automatically by codelets. Search space explosion is prevented using a heuristic pruning policy. The pruning rules prioritise solutions which extract more parallelism and achieve better access locality. Recursive tunable transformations produce kernels that fit multiple diverse GPU architectures. Compared to PetaBricks, Tangram covers larger design space by pursuing architectural optimisations.

This thesis takes a more multi-level approach to code optimisation than Tangram. Tangram achieves task partitioning by expressing parallel mappings through codelets. Since codelets are locally scoped, this approach does not consider parallel mappings benefitting multiple codelets within the same kernel. On the other hand, LIFT splits algorithmic transformations and parallel mapping search into two stages. Rewrite points perform local transformations to tile computation without hard-coding a scheduling policy. A constraint-based parallelisation pass explores the design space of the entire resulting expression, as discussed in Chapter 5.

ELEVATE is an optimisation strategy language for defining program transformation sequences [Hag20a] A functional language itself, ELEVATE complements the functional data-parallel language RISE [Hag20b]. With ELEVATE, users can define composable transformation strategies to lower a high-level RISE expression to a target-specific implementation. The correctness of strategies is proven using Agda. The RISE compiler generates OpenMP code for CPUs and OpenCL for GPUs.

The strategy combinators of RISE are expressive means of controlling the rewrite process and achieving macro-transformations. However, the burden is on the user to define a robust rewriting sequence, where the strategies don't cancel each other by breaking the expected patterns. The compiler tackles this issue through intermediate expression normalisation. By expressing the transformation directives (rewrite points) and the algorithm in a single IR– a single source of truth – this thesis suggests a simpler approach in Chapter 6. Upon application, rewrite points may introduce new rewrite points into the transformed expression, and control their placement exactly. This helps ensure that the new rewrite points pattern-match the transformed expression at no cost to the user by shifting the burden onto the rewrite point designers.

3.4 Summary

This chapter has provided an overview of manual and automated approaches to GPU programming. The IRs based on explicit parallelism allow adapting the implementations to the target platform by exposing the scheduling and memory management primitives. This expressivity comes at the cost of performance portability. More flexible are the high-level IRs capturing implicit parallelism. These IRs decouple the algorithm from implementation, providing code generators with leeway to repurpose the application for a given platform.

This chapter also covered several notable automated optimisation methods used to translate high-level programs into optimised low-level API. However, parallel code generation remains challenging due to the large design space. The next three chapters discuss several issues in parallel code generation in depth and propose methods of tackling the challenges.

Chapter 4

Functional IR for Auto-Tuning

This chapter focuses on the problem of expressing a low-level optimised convolution in a functional IR. This chapter also shows how the functional patterns allow tuning constraints to be inferred automatically. The constraints are useful to truncate the search space to only valid implementations.

4.1 Introduction

CNNs dominate the field of computer vision and image processing [Fuk80]. Due to the availability of parallel accelerators such as mobile GPUs, CNNs can be used to perform these complex tasks on resource-constrained mobile devices. However, modern neural networks are computationally demanding, yielding large memory footprints and slow inference times, which has slowed their adoption in embedded settings.

Typically, CNNs have multiple convolution layers and one or more fully connected layers. Most of their execution time is spent in convolutions – an expensive operation [Lai18]. Convolutions slide multiple kernels across a multi-channel 2D image (*e.g.*, the first input typically has three channels, RGB). The layer configurations vary significantly across networks and even among the layers of the same network. For instance, in VGG architectures [Sim14], the first convolutional layer operates on a 224×224 image with 3 channels while the seventh layer operates on a 112×112 image with 128 channels. Moreover, the size and shape of convolutional kernels might also vary between networks or layers. This diversity in convolution input shapes represents a significant challenge for high-performance software engineers. In fact, obtaining good performance for rapidly evolving networks, hardware and workloads is a significant engineering challenge for library vendors relying on hand-coded solutions.

Most neural network libraries, such as Caffe [Jia14] for CPU and CuDNN [Che14] for Nvidia GPU, solve this issue by expressing convolutions as General Matrix Multiply (GEMM), since heavily optimised implementations are readily available. While this approach leads to higher performance, it significantly increases the required memory footprint, which can be problematic when running on mobile devices. For example, a GEMM-based implementation of the second convolutional layer of VGG requires 116 MB of memory for a single image while the direct convolution requires only 13 MB. In the occasion where a large neural network needs to process multiple images at once (*e.g.*, from a video stream), the device memory can quickly fill up.

Support for direct convolution is uncommon in the high-performance kernel libraries given that it is a specialised operation in comparison with the more generic GEMM. As a result, vendors typically do not invest as much effort in providing a tuned direct convolution implementation. As an example, the ARM Compute Library (v19.02, released in 2019) implementation of direct convolution only supports a handful of convolution shapes and is, in fact, 10× slower than its GEMM counterpart on the ARM Mali GPU. This calls for an automatic approach that produces highly-specialised high-performance code for direct convolutions.

This chapter presents an automatic code generation approach for direct convolution based on LIFT. LIFT expresses algorithms using a functional data-parallel IR which includes primitives such as **`map`**, **`reduce`** and **`zip`**. A system of rewrite rules optimises LIFT expressions to specialise to the target architecture. Similarly to Futhark [Hen17] and Accelerate [McD13], LIFT exploits several properties of a functional paradigm to optimise compilation. The high-level IR abstracts away from the user the implementational details which would otherwise restrict the compiler from making design decisions automatically: explicit array traversal ordering, loop parallelisation and bounds. At the same time, the IR captures plenty of algorithmic meta-information, facilitating static analysis and preserving semantics during program transformations. Finally, the strongly typed functional patterns yield a robust and expressive type system.

The complexity of the convolution algorithm is a good example where strongly typed functional patterns shine. The type systems of LIFT, Futhark and Accelerate capture the shape of multidimensional inputs and naturally propagate the array transformations imposed by the language primitives on respective dimensions. This enables both fine-grained type-checking and program transformations that are correct by design.

This chapter provides a novel account of expressing low-level optimisations in a

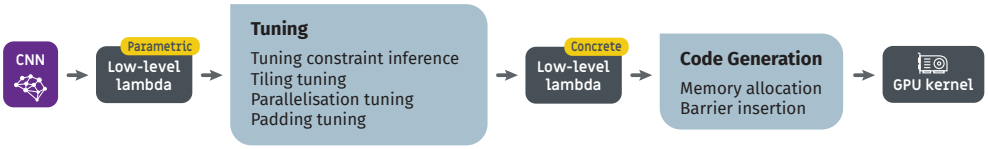


Figure 4.1: The optimisation flow discussed in Chapter 4.

high-level IR such as LIFT. Convolution provides an interesting use case as a compute-intensive problem, where several rounds of tiling and reshaping are required to match the scheduling and memory hierarchies of a target platform. Such a complex use case requires a flexible representation such as LIFT, where functional patterns expose fine-grained parallelism in a high-level IR as opposed to the algorithmic skeletons implemented in the target language as seen in Accelerate [McD13].

More specifically, this chapter shows how CNN convolutions are expressed and optimised in LIFT using the process shown in Figure 4.1. Direct convolution optimisation is achieved by exploring a parametric space which includes tile sizes, amount of padding, amount of data reuse and the number of sequential operations performed by a thread. A series of constraints are automatically produced to restrict the search to valid combinations of tuning parameters (*e.g.*, input size must be divisible by tile size). With an ARM Mali GPU as a use case, LIFT is shown to achieve high-performance direct convolution code that is on average 10× faster than the ARM Compute Library direct convolution implementation while using ×3.6 less space than GEMM-based convolution provided by the same library.

Using LIFT as an example, this chapter discusses several properties of a functional IR that facilitate static analysis during compilation. This includes data dependency analysis for efficient synchronisation barrier insertion, which is aided by LIFT's expressive strong type system and the view system. This also includes memory allocation leveraging functional iterators as a higher level of abstraction over loops.

To summarise, the main contributions are:

- Show how tuning constraints can be automatically inferred from strongly typed functional patterns to tune kernels automatically;
- Show how LIFT is leveraged to express the convolutional layers of neural networks;
- Produce code automatically for direct convolution, exploring a large optimisation space of 1,000 points with LIFT, where the best candidate achieves a

```
1  def conv( inputData       :  [[[float]_inChs]_inW]_inH,
2            kernelsWeights  :  [[[[float]_inChs]_kerW]_kerH]_outChs,
3            padSize         :  (int,int,int,int),
4            kernelStride    :  (int,int) ) :  [[[float]_outChs]_outW]_outH =
5    toHost o oclKernel((slideWindows', kernelsWeights') ⇒
6      mapND_2(slideWin: [T]_(inChs * kerW * kerH) ⇒
7        map(singleK: [T]_(inChs * kerW * kerH) ⇒
8          reduce(0, +) o map(*) << zip(slideWin, singleK)
9        ) << kernelsWeights'
10     ) << slideWindows'
11   ) << ( mapND_2(joinND_2) o
12           slideND_2(kerH, kerW, kernelStride._1, kernelStride._2) o
13           padND_2(padSize, value = 0) o
14           toGPU << inputData,
15         map(joinND_2) o toGPU << kernelsWeights )
```

Listing 4.1: High-level LIFT expression of convolution.

speedup of 10× and memory saving of ×3.6 over the ARM's own high performance library on the ARM Mali GPU.

The rest of the chapter is organised as follows. Section 4.2 discusses how convolution is expressed and optimised in a functional IR. Section 4.3 shows how tuning constraints can be inferred from a functional expression automatically, while Section 4.4 outlines the improvements to memory allocation made within this work. Section 4.5 describes the experimental setup used to evaluate proposed techniques and presents the results of the evaluation. Section 4.6 summarises the chapter.

4.2 Optimising Convolution in LIFT

This section describes how a convolutional layer is expressed functionally in LIFT. We first look at the convolution expression provided in Listing 4.1, which represents the highest possible level in the LIFT IR. The expression is intentionally algorithm-centric – it does not encode any decisions regarding parallelism and memory mappings, order of operations or any other hardware-specific aspects. Also discussed are the low-level optimisations required for high-performance direct convolution on a GPU, and how they are expressed functionally. Chapters 5 and 6 discuss how these design decisions are made automatically in LIFT.

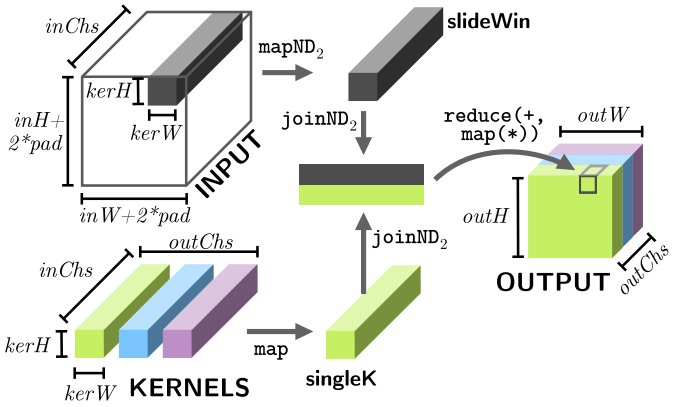


Figure 4.2: Visualisation of direct convolution.

4.2.1 High-level LIFT Expression

Listing 4.1 shows the LIFT expression of a convolution layer with two arguments; Figure 4.2 visualises the algorithm. `inputData` contains the layer's input which is a 3D array (width × height × input channels). `kernelsWeights` contains the weights of all the kernels across the width, height and input channels. *padSize* is a tuple of four values that specifies how much padding is required in each direction by the layer specification. *kernelStride* specifies by how much each kernel is displaced across the input (the step). The output data is a set of feature maps represented as a 3D array with the outer dimension corresponding to the number of kernels.

This LIFT program in Listing 4.1 consists of three steps. First, data is padded with zeros on line 13 as per the configuration of the layer. Then, we slide in 2D across the padded input along the two spatial dimensions (*inW* and *inH*) producing the sliding windows on line 12; the sliding windows are used on line 10. Finally, convolution is performed using a combination of LIFT primitives. First, we map over each sliding window using **mapND**$_2$ on line 6. Then, *kernelsWeights* are mapped over on line 7. On line 8, we finally reduce over the flattened and zipped `slideWin` and `singleK`. The zipping of `slideWin` and `singleK` ensures that the reduction operates on pairs of corresponding elements from both arrays. The reduction operator multiplies the corresponding elements and adds to the accumulator which is initialised with a zero.

The multidimensional operators such as **padND**$_2$, **slideND**$_2$ and **mapND**$_2$ are implemented as macros that are expanded into equivalent LIFT expressions that depend only on one-dimensional primitives. Such design achieves concise representation without overloading the language with redundant primitives. More details on multidimensional

operators are provided in Section 2.3.1.

4.2.2 Optimisations of Convolution on a GPU

Achieving high-performance direct convolution on a GPU comes with a number of unique challenges. From the application side, difficulty is presented by the algorithm being based on stencil computations, resulting in non-contiguous non-regular memory accesses. From the target hardware side, the challenges are caused by the scarcity of the on-chip memory on mobile GPUs and the generic complications of hierarchical parallelism. Addressing these challenges in a functional IR makes for a significant milestone; a gateway to supporting a wide range of simpler algorithms and less constrained platforms are supported. This section provides a list of example low-level optimisations in the context of convolution. Section 4.2.3 discusses how these optimisations are expressed in a functional IR.

4.2.2.1 Data Reuse

Convolutional layers are memory-bound due to the amount of data exceeding the on-chip storage capacity. The input size is consistently large across CNN architectures and specific layers. This is because smaller data sets can usually be parsed by leaner analytical solutions. Similarly large are convolutional weights due to the redundancy inherent to the ensemble Machine Learning method such as CNN, where individual agents' areas of expertise can overlap. The large amount of data requires that data is processed in tiles that fit into the memory.

Beyond splitting the problem in manageable tasks that a mobile GPU can handle and ensuring access locality, tiling helps achieve data reuse on the algorithmic level in local or private memory. Convolution is applied on two arguments – inputs and weights – and both have an independent dimension: different feature maps for inputs and different convolutional kernels for weights. This can be leveraged by reusing a tile of inputs across multiple convolutional kernels, and vice versa.

While data reuse improves kernel performance by reducing the number of memory accesses, care must be taken not to allocate too many registers. High private memory usage can have two negative consequences: register spilling and core undersaturation. Register spilling occurs when a thread allocates more registers than available on the hardware and some of the data is moved to a slower memory incurring a performance overhead. However, even using fewer registers than available is not guaranteed to yield

maximum performance. The number of threads executed in parallel on a compute core is determined by the register consumption of individual threads. The larger the register consumption per thread, the fewer threads are scheduled by the GPU leaving ALUs and memory buses underutilised.

4.2.2.2 Locality of Reference

In a memory-bound application such as convolution run on restricted hardware, reducing global memory usage is a priority. The disparity between the problem size and available cache and private memory can be addressed through memory access locality. By bundling accesses close in time and memory location, the compiler ensures that data is fetched once and reused multiple times, increasing the amount of cheaper compute operations per slow memory access.

Temporal Locality (TL) The number of cache misses can be reduced by optimising access patterns to process horizontally and vertically consecutive sliding windows within the same work group. Since threads within a work group are scheduled to execute in short intervals, the overlapping regions of neighbouring windows are also requested in quick succession. The first load of the overlapped region is most expensive because it accesses global memory directly. The subsequent accesses are cheaper since the data is read from the cache. LIFT achieves such access pattern optimisation through non-continuous input tiling discussed later in Section 4.2.3.1.

Cache reuse can be more effective in direct convolution as opposed to GEMM. The im2col operation of the latter method's preprocessing stage duplicates data in memory, forcing the GPU to load each element anew on every read. Direct convolution depends on the stencil access pattern to simulate overlapping windows, leaving it up to the implementation to choose the number of accesses per element. In a convolutional layer with 3x3 windows and a stride of 1, each element can be accessed up to nine times. Assuming that Mali G72 cache of 512 KB is divided equally among inputs and weights, and that there is only one input channel, the direct convolution method can store up to $88 \times 88 = 7744$ overlapping windows in cache. Conversely, GEMM can cache only up to 910 im2col-processed windows, *i.e.*, $\times 8.5$ fewer windows.

Spatial Locality (SL) The number of memory accesses can be further reduced on the Mali GPU through SL. SL is attributed to memory access patterns which iterate over data elements within relatively small areas of memory. GPUs mask memory

access latency with increased bus bandwidth by fetching the entire cache line for each read: requesting even a single word on a Mali GPU fetches 512 bits. The extra data does not go to waste if all threads in a quad access data within a contiguous 512 bit-wide area of memory. Fitting the data read by a quad in a single cache line improves both performance and energy consumption [Arm20]. SL is achieved through memory access coalescing discussed later in Section 4.2.3.2.

Thus, cache hit rate is improved in two ways. By optimising for TL and SL, the hit rate of the entire cache is improved. Single cache line hit rate is optimised through SL.

4.2.2.3 Cache Line Optimisation

While memory access coalescing reduces the total number of cache lines read, it does not address the problem of cache thrashing. Thrashing occurs when an excessive number of cache loads is requested simultaneously, and request scheduling overheads reduce performance. The 64 byte-wide cache line of a Mali GPU fits 16 floats; even when coalesced, 16 threads issuing scalar float loads from the same cache line produce at least 4 cache requests.

The number of requests to a single cache line can be reduced by **vectorising** the data load, causing the GPU to issue one data request for all elements in a vector. For a preferred size of 4 floats per vector, a vectorised load fetches 16 bytes simultaneously. Combined with coalescing adjusted for vectorisation using a stride of 16 bytes, the size of the atomic read increases fourfold covering the entire cache line. Issuing one memory request per thread quad achieves the best bandwidth while reducing the overhead of memory thrashing.

4.2.2.4 Core Saturation

A GPU must be saturated on two levels: across compute cores and within each core. A kernel executing on N cores should schedule at least N work groups; since synchronisation across cores is usually not supported, work groups are best suited for independent tasks.

Work group size should be a multiple of warp size to ensure that course-grained hardware units are not under-utilised. On a Mali G72, threads execute in quads: all operations across the four threads of a quad are performed at the price of one. The maximum number of threads executed simultaneously by a core is limited by the number of registers used within each thread.

4.2.2.5 Task Granularity

Fine-Grained Parallelism Long-living threads performing a lot of sequential work result in core under-saturation, which necessitates splitting a problem into smaller tasks to be performed in parallel. In convolution, achieving manageable tasks is not always possible through straightforward tiling of independent input and weight dimensions. Producing one output requires reducing an entire sliding window with the corresponding weights; in VGG-16, one output requires processing up to 9.5 MBytes of data. To fit the compute and memory capacity of a mobile GPU core, window reduction must be tiled for threads to reduce each window tile independently, synchronise, then reduce the intermediate results into a single output value.

Thread Coarsening Aggressive parallelisation strategies result in short-lived threads and work groups, harming performance in two ways. First is the context switch overhead associated with each return of a thread or a work group. This overhead can be offset by increasing the amount of sequential work per thread. Second is the low compute-to-memory load ratio – threads perform too few operations per each fetched data element. Balanced threads reuse data in registers and cache by processing multiple inputs per convolutional kernel, and vice versa. Thread coarsening merges the work of multiple threads into one, allowing sharing of the registers across more tasks.

Thread coarsening leads to more sequential work, which can result in core under-saturation if taken to the extreme. The balance between parallelisation and thread coarsening is found for the given problem dimensions and hardware specifications through constrained tuning.

4.2.3 Low-level Optimisations in a Functional IR

This section discusses GPU-specific optimisations from the perspective of a functional IR.

4.2.3.1 Tiling

As discussed in Section 4.2.2.2, horizontal and vertical tiling of inputs – non-continuous tiling – improves cache usage through **spatial and temporal localities**. Support for non-continuous tiling is also important in a functional IR for fine-grained control over tile shape. In direct convolution, sliding windows are not continuous in memory: window rows are located on different rows of the input image. Due to this, tiling data

continuously is undesirable for two reasons. For a tile to cover at least one sliding window completely, it must cover all the image rows that the sliding window is spread across. This results in large tiles that either don't fit into memory at all, or leave no space for caching convolutional weights. Another way to leverage continuous tiling is to include multiple sliding windows partially. Since a single output value of a convolutional layer depends on all values of a sliding window, the thread producing the output value must access multiple tiles; since only one tile at a time is loaded from global memory, the thread needs to be blocked until extra tiles are loaded.

Furthermore, convolution input tiles must overlap to ensure that all sliding windows are fully covered by at least one tile. This is an extra difficulty presented by direct convolution with virtual sliding windows in contrast with the GEMM method, where im2col-processed inputs can be tiled naively without overlapping.

Generally, a non-continuous overlapping tiling can be achieved with a **slideND** pattern. For example, a matrix can be tiled in two dimensions as follows:

$$\textbf{slideND}_2(tileH,\ tileW,\ tileStrideY,\ tileStrideX)\ \texttt{<<}\ (\texttt{matrix:}\ [[T]_M]_N)$$

In a convolutional layer, **slideStrictND** primitive must be used to capture the extra restriction on the tile size and stride requiring that the argument is covered by the tiles fully.

In direct convolution expression which already uses the **slide** primitive to express the stencil pattern, non-continuous overlapping tiling can be achieved using the **split** primitive:

$$\textbf{split}(tileH)\ \circ\ \textbf{map}(\textbf{split}(tileW))\ \circ$$
$$\textbf{slideND}_2(kerH,\ kerW,\ kernelStrideX,\ kernelStrideY)\ \texttt{<<}\ (\texttt{input:}\ [[T]_M]_N)$$

This approach is more straightforward compared to **slideStrictND** since correctness of tiling does not depend on the chosen tiling stride. However, the tile size in **split**-based tiling is expressed not in input values, but in the number of windows stored in a tile. This is because the tiling **slideStrictND** is applied before the stencil sliding, while the tiling **split** is applied after.

Data reuse in private memory is achieved by copying pairs of input and weight tiles into private memory before iterating over them. This is achieved using the **let** primitive:

```
1 let(prefetchedInputTile ->
2   let(prefetchedWeightsTile ->
3     map(x ->
4       map(w -> f(x, w)
5       ) << prefetchedWeightsTile
6     ) << prefetchedInputTile
7   ) o map(toPrivate(id)) << weightsTile
8 ) o map(toPrivate(id)) << inputTile
```

Where `id` is the identity function.

4.2.3.2 Coalescing

For **spatial locality of reference**, a functional IR expresses coalescing using **`transpose`**, **`split`** and **`join`** patterns. Consider the following example:

$$\texttt{mapLcl}(dim, \texttt{mapSeq}(\texttt{f})) \circ \texttt{split}(s) \ll (\texttt{x}: [T]_N)$$

Where parallel accesses occur with a stride of s. If s is larger than $\frac{cacheLineSize}{warpSize \times elementSize}$ where $elementSize$ refers to the number of bytes required to store the scalar or vector element, then each warp accesses more than one cache line underutilising each fetched line. If s is smaller than the preferred vector size, then the compiler is prevented from vectorising the load.

Coalescing can be achieved by transforming the example above as follows:

$$\texttt{join}() \circ \texttt{transposeW}() \circ \texttt{mapLcl}(dim, \texttt{mapSeq}(\texttt{f})) \circ \texttt{transpose}() \circ \texttt{split}(\texttt{s}) \ll (\texttt{x}: [T]_N)$$

Where **`transpose`** transforms the view of `x` such that **`mapLcl`** is applied on the inner (contiguous) dimension and **`mapSeq`** is applied on the outer dimension. **`transposeW`** reverts the transposition such that the output layout matches that of `x`. In this particular transformation, threads access contiguous elements of `x`, guaranteeing no more than one cache line read per thread quad. A further transformation where `x` is vectorised using **`asVector`** would ensure that the entire cache line is used by the quad, and reduce cache thrashing.

4.2.3.3 Padding

The padding expression has a dual purpose. First, it pads the input with zeros along all four edges as per the neural network architecture. Secondly, it zero-pads the input

across the right and bottom edges so that the resulting array can be perfectly tiled. Padding is performed along only two out of four edges to minimise extra fragmentation of input data. The amount of padding ρ is determined automatically by a constraint solver and is explained later.

Consider the following example, where two-dimensional input is padded with ρ zeros along each of the four edges:

$$\texttt{mapSeq(mapSeq(f)) } \circ \texttt{ pad2D(}\rho\texttt{, }\rho\texttt{, }\rho\texttt{, }\rho\texttt{, 0.0f) << (input: } [[T]_M]_N\texttt{)}$$

This expression corresponds to the following OpenCL loop nest, where `x` and `y` are defined on padded ranges, and the inline if-conditionals ensure that out-of-bounds accesses are replaced with a literal zero:

```
for (int y = 0; y < N + 2*ρ; y++) {
  for (int x = 0; x < M + 2*ρ; x++) {
    buf[x + y * (M + 2*ρ)] = f(
        y < ρ ? 0.0f :
          y >= ρ + N ? 0.0f :
            x < ρ ? 0.0f :
              x >= ρ + M ? 0.0f :
                input[x - ρ + (y - ρ) * (M + 2*ρ)]); }}
```

The padding expression is compiled into a separate preprocessing OpenCL kernel that is executed separately from convolution. Since **pad2D** introduces an inline if-conditions, hoisting padding into a separate kernel mitigates the overheads of thread divergence, leaving the most compute-intensive operation – convolution – free of branching.

4.2.4 Low-Level LIFT Expression

As shown in Listing 4.1, convolution is expressed as a set of reductions of sliding windows. However, in popular deep CNNs such as VGG, ResNet and GoogleNet, most convolutional layers are wide to such an extent that the whole input does not fit in the cache (*e.g.*, L2). This issue is addressed by tiling the input and splitting reduction into two steps. In the first stage of partial convolution, input is tiled, each sliding window of each tile is split into chunks, and each chunk is reduced to a single value. In the second stage of final convolution, the resulting vector of values per sliding window is reduced to one final value in the same GPU kernel.

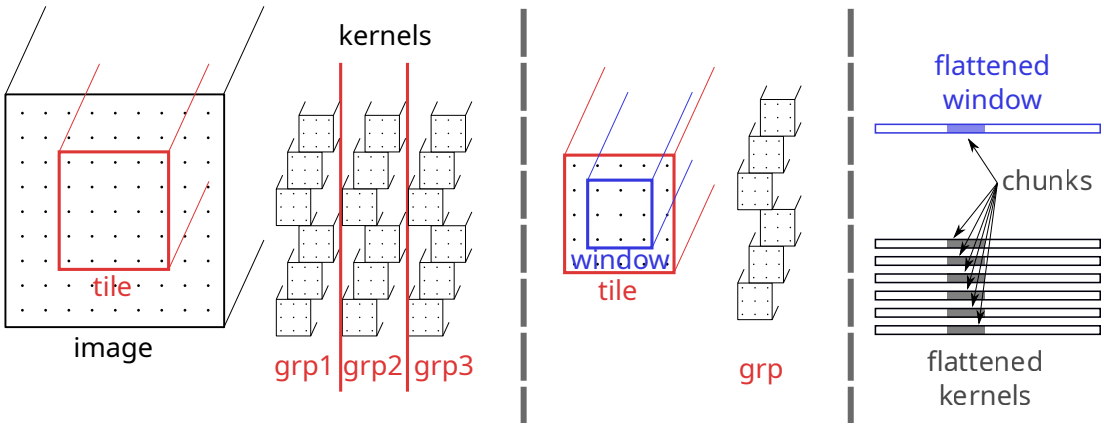


Figure 4.3: Visualisation of the low-level LIFT expression in Listing 4.2.

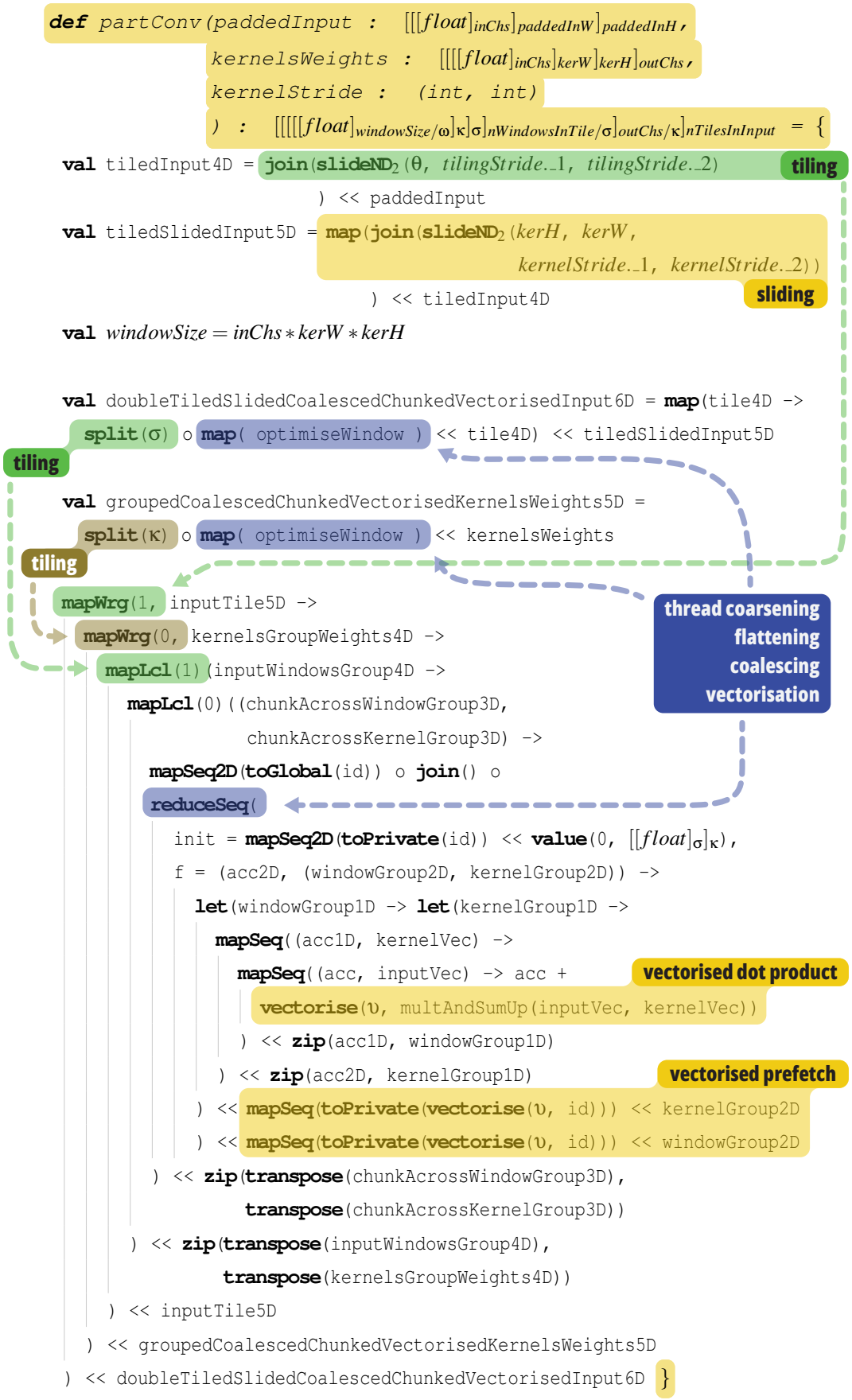
To ensure that the tiles fit perfectly with the input sizes, extra padding might be required on the input using another GPU kernel before processing the data. Conversely, an extra GPU kernel might be required at the end to crop back the output. The four stages are discussed below.

4.2.4.1 Partial Convolution

Figure 4.3 presents an overview of the partial convolution implementation in Listing 4.2. Acquiring input image and a set of convolutional kernels, the expression splits the image into tiles and kernels – in kernel groups. Each combination of a tile and kernel group is processed by a single work group on lines 18 and 19 in Listing 4.2 respectively. Then, a window of the spatial size $kerW \times kerH$ is slided across the tile. This results in a set of sliding windows, which at this point are just virtual views into data, produced using **`slideND`**$_2$ on line 7.

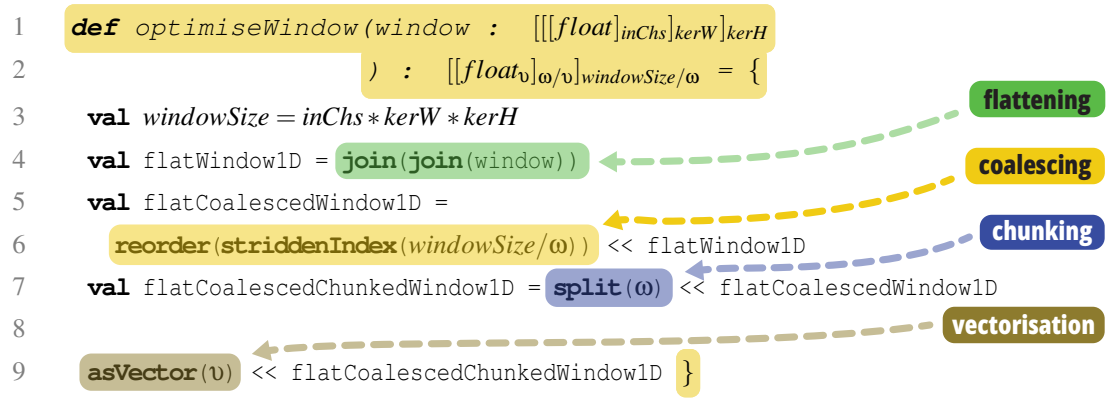
Each sliding window is flattened across two spatial dimensions and input channels, and split into chunks on line 13. Each chunk is reduced sequentially by a single thread on line 24. Each thread can process chunks from more than one sliding window, traversed by a **`mapSeq`** on line 29. Each kernel is split into chunks correspondingly by flattening the three kernel dimensions and splitting with the same chunk size as that of the input chunks on line 16. Each sliding window chunk is coupled with corresponding chunks in each of the kernels in the group. A thread processes each pairing of the input chunk with the kernels in a kernel group.

Processing each input-kernel chunk pair involves multiplying input values and corresponding weights, and summing the resulting vector on line 30. Thus, each sliding window is reduced to a vector of values, corresponding to each chunk in the sliding




```
1  def partConv(paddedInput : [[[float]_inChs]_paddedInW]_paddedInH,
2               kernelsWeights : [[[[float]_inChs]_kerW]_kerH]_outChs,
3               kernelStride : (int, int)
4               ) : [[[[[float]_windowSize/ω]_κ]_σ]_nWindowsInTile/σ]_outChs/κ]_nTilesInInput = {
5    val tiledInput4D = join(slideND_2(θ, tilingStride._1, tilingStride._2)
6                       ) << paddedInput
7    val tiledSlidedInput5D = map(join(slideND_2(kerH, kerW,
8                                                kernelStride._1, kernelStride._2))
9                             ) << tiledInput4D
10   val windowSize = inChs * kerW * kerH
11
12   val doubleTiledSlidedCoalescedChunkedVectorisedInput6D = map(tile4D ->
13     split(σ) o map( optimiseWindow ) << tile4D) << tiledSlidedInput5D
14
15   val groupedCoalescedChunkedVectorisedKernelsWeights5D =
16     split(κ) o map( optimiseWindow ) << kernelsWeights
17
18   mapWrg(1, inputTile5D ->
19     mapWrg(0, kernelsGroupWeights4D ->
20       mapLcl(1)(inputWindowsGroup4D ->
21         mapLcl(0)((chunkAcrossWindowGroup3D,
22                    chunkAcrossKernelGroup3D) ->
23           mapSeq2D(toGlobal(id)) o join() o
24           reduceSeq(
25             init = mapSeq2D(toPrivate(id)) << value(0, [[float]_σ]_κ),
26             f = (acc2D, (windowGroup2D, kernelGroup2D)) ->
27               let(windowGroup1D -> let(kernelGroup1D ->
28                 mapSeq((acc1D, kernelVec) ->
29                   mapSeq((acc, inputVec) -> acc +
30                     vectorise(υ, multAndSumUp(inputVec, kernelVec))
31                   ) << zip(acc1D, windowGroup1D)
32                 ) << zip(acc2D, kernelGroup1D)
33               ) << mapSeq(toPrivate(vectorise(υ, id))) << kernelGroup2D
34               ) << mapSeq(toPrivate(vectorise(υ, id))) << windowGroup2D
35           ) << zip(transpose(chunkAcrossWindowGroup3D),
36                    transpose(chunkAcrossKernelGroup3D))
37         ) << zip(transpose(inputWindowsGroup4D),
38                  transpose(kernelsGroupWeights4D))
39       ) << inputTile5D
40     ) << groupedCoalescedChunkedVectorisedKernelsWeights5D
41   ) << doubleTiledSlidedCoalescedChunkedVectorisedInput6D }
```

Listing 4.2: Low-level LIFT expression of partial convolution. Expression `optimiseWindow` is defined in Listing 4.3.



Listing 4.3: A window preprocessing expression. Applied on both inputs and weights on lines 13 and 16 of Listing 4.2, it transforms data layout of a single window virtually to optimise memory accesses. The window is flattened, reordered for coalesced accesses, tiled and vectorised.

window. This is partial reduction; another expression further reduces the vector to each value resulting in a full convolution of each sliding window to a single output value.

Partial reduction is the main focus of optimisation in this design, since most work is performed during partial reduction if the chunk size is sufficiently large. Small chunk sizes shift the computational burden to the final convolution stage, but at the cost of missed opportunity to reuse input and kernel values. Very large chunk sizes result in too much sequential work and under-saturation of the compute cores.

Listing 4.2 shows the partial convolution algorithm in LIFT. First, the input is tiled using **slideND**$_2$ and the 2D array of tiles is flattened (line 6). The tile size is controlled by the parameter θ and the stride is calculated to minimise the amount of tile overlap:

$$tilingStride = \theta - (kernelWidthHeight - kernelStride)$$

Convolution within each tile is expressed by nesting a second **slide2D** on line 9. This new five-dimensional view of the input data is further transformed using the inner expression on line 1. The 3D sliding window and convolutional kernels are represented as flat vectors; this simpler data layout enables coalescing of data accesses using **reorder**, an important GPU optimisation that improves locality. The elements are virtually reordered with the stride of $windowSize/\omega$, where ω refers to the size of the partial window processed by one thread. The resulting stride is the number of threads processing the same window, ensuring each thread access consecutive elements. The

Table 4.1: OpenCL dimension sizes defined in terms of tuning parameters

Dimension	**Size**
Work group dim. 1	Number of tiles in the input
Work group dim. 0	Number of kernel groups
Thread dim. 1	Number of sliding window groups in a tile
Thread dim. 0	Number of sliding window partitions

window is vectorised with vector length υ which is important for the Mali GPU. Finally, windows are split in groups; each thread will process chunks from the whole group of sliding windows.

Lines 20 to 22 express mapping of parallel computations onto OpenCL threads; for the sizes of the respective work group dimensions, see Table 4.1. In dimension 1, each work group processes one input tile; in dimension 0, each work group is assigned one group of convolutional kernels. The grouping of kernels is expressed on line 16; the size of a kernel group is controlled by the parameter κ.

In local dimension 1, threads are assigned an input window group. In local dimension 0, threads are assigned a chunk of each input window in a window group and a chunk of each kernel in a kernel group. By reading the input window chunk only once and reusing it for κ kernels within the same thread, the number of reads is reduced by a factor of κ. By reading the kernels once and reusing them for σ sliding windows within the thread, the number of reads is further reduced by a factor of σ. By iterating across the fastest changing dimension 0 in the innermost loop, the quad threads are guaranteed to access consecutive window chunks; thanks to the prior coalescing now stored in the view, quad threads access consecutive locations in memory further reducing the number of reads by a factor of four.

The reduction of the partial window across several kernels is expressed on line 24: the accumulator is initialised to a vector of κ zeros on line 25 and the input to **`reduceSeq`** on line 35 is an array of tuples of partial window elements and corresponding elements from kernel weights.

The first **`let`** primitive on line 27 ensures that the input values are fetched into the private memory once on line 34 and are reused across iterations of the sequential loop on line 29. Similarly, a group of kernel values is prefetched on line 33 and reused across the iterations of the loop on line 28.

4.2.4.2 Summing Partial Results

The third expression completes convolution by reducing the partial weighted sums of each window. Each work group processes a single tile for a single kernel group; each thread reduces one or more sliding windows in one output channel.

A straight-forward reduction is applied on the result of the outer **`mapLcl`** on line 20 of Listing 4.2; the barrier insertion pass discussed in Section 5.4 synchronises the two convolution stages if necessary using a barrier. Since summing the intermediate results is not an compute-intensive operation, there is no benefit in tiling the final reduction expression or re-tuning the work group dimensions.

4.2.4.3 Cropping

The final expression reverses the effect of the extra padding performed in the first expression. It crops the output using **`pad2D`** with negative values for padding sizes. The amount of horizontal and vertical cropping is calculated as:

$$cropSize = \frac{\rho}{kernelStride}$$

The *cropSize* is guaranteed to be whole by the **`slide`** constraint discussed later in Section 4.3.

4.2.4.4 Summary

The section has shown how fine-grained hardware optimisations are achieved in a functional IR. It has also presented a low-level LIFT expression combining multiple optimisations. The presented example makes a case for the expressiveness of a functional IR in the context of parallel programming. Despite the focus on algorithm representation, an IR such as LIFT is flexible enough to support fine-grained optimisations of a parallel program. We will see in Chapters 5 and 6 how the application of these optimisations is simplified through automatic parallelism mapping and guided rewriting.

4.2.5 Tuning Parameters

The performance of the low-level expression presented in Listing 4.2 depends on the tuning parameters injected into the expression. These parameters (Table 4.2) represent design choices that can be explored automatically. Before introducing the constraint-based exploration mechanism in Section 4.3, this section discusses how the parameters affect the performance of the generated program.

Table 4.2: Convolution expression tuning parameters

Symbol	Parameter
θ	Input tile size
ρ	Optimisational padding size
κ	Number of kernels per work group
σ	Number of sliding windows per thread
ω	Sequentially processed input elements
υ	Vector size

Input Tiling Splitting the input optimises cache locality by ensuring that adjacent threads process the same neighbourhood. The tile size is explored in the range from the kernel size to double the padded input size.

Padding Changing the input size solves the problem of finding an efficient tile size that both splits the input evenly and can be evenly split by the convolutional kernels. Though time might be wasted on processing dummy data, padding achieves better data alignment and cache locality.

Kernel And Sliding Window Grouping Processing multiple kernels and sliding windows per thread results in data re-usage: input data is fetched once into private memory and is reused during output channel computation; the same for the weight coefficients. The benefit of increased re-usage is a tradeoff since large values of κ increase register pressure.

Sliding Window Chunking Each sliding window and kernel are flattened and split into chunks, processed sequentially within threads. Smaller values for chunk size result in more parallel operations. Varying the amount of sequential work allows to explore work group sizes which influences register consumption and maximum occupancy of the compute cores.

The Need for Constraint Inference Such a large number of tuning parameters presents a challenge, especially if cross-platform and cross-domain performance portability is required. Many tuning parameter value combinations break language semantics, *e.g.*, tile sizes that are not factors of the argument array size. Enumerating these invalid

combinations is not feasible. Similarly costly is hard-coding the constraints on the parameters. Constraints are unique to the low-level LIFT expression, which is domain- and platform-specific; furthermore, the total number of constraints per expression is large. Constraints are also predicated on the rewriting process, which is required for a more comprehensive optimisation workflow. Automatic transformations of an AST such as parallelisation, vectorisation, coalescing and unrolling may introduce, change or remove both constraints and tuning parameters.

Constraint dependence on the input expression and the choice of rewrites, as well as the search space size call for an automatic approach to constraint generation. The next section describes the mechanism of automatic constraint inference. By removing the need to enumerate tuning parameters and constraints manually, this approach allows tuning rewritten expressions, where the number and positions of tuning parameters in the AST are unknown in advance. Chapters 5 and 6 illustrate the radical AST transformations where automatic constraint inference truly shines.

4.3 Constraint Inference

When exploring the search space of possible implementations, the compiler leverages rich algorithmic information captured by the LIFT IR. Type safety and provable correctness of rewrite rules allow to automatically explore structural code transformations that would otherwise require costly static analysis.

LIFT supports symbolic parameter values in the multidimensional array types. Parameter tuning consists of finding valid combinations of symbolic parameter values, replacing them at the type level and generating a specialised implementation. This leads to GPU kernels that are specialised for the given input parameters and tuning values.

The expressiveness of LIFT and the complex search space produced by rewriting results in a high number of dependent and independent parameters which is hard to analyse manually. To address the problem of parameter validation, the compiler infers constraints automatically based on the information encoded in the IR and the type system. By traversing the AST, it collects variables from types and the parameters of the IR primitives, and infers continuous and discrete constraints on the parameter values. A constraint is expressed as a record specifying the condition that must hold true and the list of parameters the condition is imposed upon. The next section presents the novel work of automatically deriving constraints from a LIFT expression.

4.3.1 Constraint Types

This section presents five types of constraints inferred from the expression. Although this looks like a small space, the convolution expression in Listing 4.2 produces 104 instances of the algorithmic and hardware-specific constraints. The power of this approach is simplicity: only a small number of LIFT primitives is used to achieve multiple optimisations.

4.3.1.1 Algorithmic Constraints

Algorithmic constraints are inferred based on the type of an IR primitive and the values of its parameters. Satisfying such constraints is required for producing semantically correct results. For the **`split`** primitive, the first inferred constraint is as follows:

$$\textbf{\textit{split}} : (m : \texttt{int},\ in : [T]_n) \Rightarrow n \,\%\, m = 0 \tag{4.1}$$

This constraint ensures that the **`split`** input is divisible evenly into chunks of m elements. The compiler traverses the arithmetic expression of the condition $n\%m = 0$ and collects all the parameters; they are marked as co-dependent for the sake of establishing parameter evaluation order during constraint satisfaction.

Two more constraints are inferred from the **`split`** primitive to ensure that the chunk size is larger than zero and does not exceed argument size:

$$\begin{aligned} \textbf{\textit{split}} : (m : \texttt{int},\ in : [T]_n) &\Rightarrow m > 0 \\ &\Rightarrow m \leq n \end{aligned} \tag{4.2}$$

Section 4.3.3.2 discusses how inequality constraints can be leveraged to not only ensure valid parameter values, but to optimise search through parameter range reduction.

`asVector` imposes similar constraints to those of **`split`**:

$$\begin{aligned} \textbf{\textit{asVector}} : (m : \texttt{int},\ in : [T]_n) &\Rightarrow n \,\%\, m = 0 \\ &\Rightarrow m > 0 \\ &\Rightarrow m \leq n \end{aligned} \tag{4.3}$$

`slide` comes in two conceptual flavours based on the constraints it imposes on the variables. The **`slideStrict`** requires that the sliding window covers the input perfectly:

$$\textbf{\textit{slideStrict}} : (size : \texttt{int},\ step : \texttt{int},\ in : [T]_n) \Rightarrow \frac{(n - size)}{step} + 1 = 0 \tag{4.4}$$

slideStrict must be used for tiling, when the input data has to be partitioned entirely and without stepping outside of the input boundaries. For kernel sliding, normal **slide** is used since sliding is allowed to produce partial results and the constraint 4.4 is not required. A notable example is the first layer of AlexNet [Kri12].

Finally, both **slide** and **slideStrict** entail the following constraints:

$$\begin{aligned} \textbf{\textit{slide(Strict)}} : (size : \texttt{int},\ step : \texttt{int},\ in : [T]_n) \Rightarrow size > 0 \\ \Rightarrow size \leq n \end{aligned} \tag{4.5}$$

4.3.1.2 Hardware Constraints

The specifications of the target hardware impose constraints on the maximum amount of threads in a single dimension, work group size, total memory allocated and maximum single buffer size. These constraints can be inferred by calculating the minimum resources necessary to compute an expression and matching them against respective OpenCL driver information.

4.3.2 Constraint Solver

When starting from a low-level LIFT expression whose AST is finalised to include tiling, parallelisation, vectorisation, coalescing and unrolling, the compiler uses the following search strategy. Firstly, the expression is traversed to collect tuning parameters and constraints. Next, parameters are sorted in the order of exploration – for example, if the parameter A depends on the parameter B, B needs to be evaluated first. To find this ordering, the compiler represents the collection of constraints as a Directed Acyclic Graph (DAG) and sorts it topologically. The resulting partial sorting order is finalised by imposing a random order on the unsorted groups of parameters. The derived parameter order is used to incrementally generate random combination of parameter values that satisfy all the constraints.

4.3.3 Search Space Simplification

Multiple levels of tiling of inputs, convolutional weights and intermediate results of reduction can result in a large number of tuning parameters, and increase the complexity of constraint satisfaction exponentially. Due to transposition and reordering used to achieve access coalescing and memory locality, the array sizes, and, consequently, the arithmetic expressions in the inferred constraint predicates can become non-linear

and slow to solve. The ranges of predicate expressions are sparse and yield poorly to simplification by the constraint solver. The problem can be alleviated using a few simple techniques to reduce the numbers of parameters and constraints, shrink tuning parameters ranges and simplify constraint predicates for faster constraint satisfaction.

4.3.3.1 Constraint Simplification

LIFT leverages its arithmetic simplification library to prune constraints whose predicates are entailed, *i.e.*, true for all values of tuning parameters. The compiler may perform this simplification thanks to the information inferred from the IR, such as loop iterator ranges and the iteration step.

Examples of entailed predicates encountered among the constraints listed in Section 4.3.1 are following:

$$a + x > 0, \text{ if } a > x \tag{4.6}$$

$$ax \,\%\, b = 0, \text{ if } a \,\%\, b = 0 \tag{4.7}$$

$a > x$ in the predicate 4.6 can be determined by checking the range bounds of a. $a \,\%\, b$ in the predicate 4.7 can be determined if a is provably multiple of b based on their ranges or if they are literals.

The same predicate can be generated from different subexpressions within the same AST. For example, the following expression produces the same constraint $n \,\%\, m = 0$ from both **split** and **asVector**:

```
asVector(m) ∘ join ∘ map(f) ∘ split(m) << (x : [T]_n)
```

Arithmetic simplification is also leveraged to detect duplicate constraints. When equivalent arithmetic expressions in the constraint predicates can be simplified to the same normal form, a trivial equivalence check allows to prune non-unique constraints. Example duplicate predicates are following:

$$((x^2 + 2xy + y^2) \,\%\, a = 0) \text{ and } ((x+y)^2 \,\%\, a = 0) \tag{4.8}$$

$$((\frac{x}{a} + \frac{y}{a}) \,\%\, b = 0) \text{ and } (\frac{x+y}{a} \,\%\, b = 0) \tag{4.9}$$

The symbolic expressions in the constraint predicates and tuning parameter ranges are simplified by substituting convolutional layer configuration parameters early. By restricting the search to one convolutional layer at a time, the compiler can replace the input and weight dimensions with literal values. The simplified constraints either

contain no parameters and can be pruned under the assumption that they are guaranteed to be always true, or contain fewer parameters allowing further pruning through uniqueness and parameter range reduction checks.

4.3.3.2 Tuning Parameter Range Reduction

When choosing tuning parameter values, the constraint solver picks candidate values randomly and checks them against the set of constraints. The range size of the parameters can have a significant effect on the search duration if the ratio of incorrect to correct values covered by the parameter range is large. Thus, care must be taken to choose the minimum range that covers all correct values.

The naive approach of setting the bounds to *(0, maximum parameter type value)* is suboptimal for the GPU kernel tuning parameters that tend to have relatively small ranges: tile sizes, vector lengths and thread configuration. Setting the bounds manually based on the problem size requires hardcoding the parameters to the given problem dimensions, target hardware specifications and the specific optimisation design. Instead, the proposed technique picks the smallest possible range based on the constraints generated from the AST as discussed in Section 4.3.1. More specifically, LIFT pattern-matches the inequality constraints of the following form:

$$t\,[< \mid \leq \mid > \mid \geq]\,x \tag{4.10}$$

Where t is a single tuning parameter, and x is an arithmetic expression that has no tuning parameters and can thus be evaluated statically. Such constraints are further referred to as bound constraints. The compiler uses bound constraints to increase or decrease the lower or the upper range bound of the parameter t and discards the bound constraints. This approach reduces the number of parameter values to be checked by the solver; it also reduces the total number of constraints to check for all candidates since the bound constraints are now enforced through parameter range. The total number of tuning parameters is decreased as well when the corresponding ranges are reduced to single values.

Algorithm 2 presents the full parameter range reduction algorithm. After the initial collecting of bound constraints on line 3, the compiler reduces the ranges of respective parameters on line 9. For each parameter with reduced range t_1, the compiler propagates the changes to all other tuning parameters and constraints that depend on t_1 on lines 14 and 18 respectively. Consider the following initial set of tuning parameters T

Algorithm 2: Tuning Parameter Range Reduction.

input : Set of tuning parameters T
input : Set of constraints C defined on T
output: Reduced set of tuning parameters T' with reduced ranges
output: Reduced set of constraints C' defined on T'

```
1  T' ← T
2  C' ← C
3  B ← bound constraints in C' with at most 1 tuning parameter
4  while B is not empty do
5      foreach constraint b in B do
6          remove b from B
7          remove b from C'
8          t1 ← the tuning parameter bound by constraint b
           // Reduce the range of t1
9          move the lower or upper bound of t1 according to b
10         if size(t1.range) == 1 then
11             remove t1 from T'
           // Reduce the ranges of the parameters that depend on t1
12         foreach parameter t2 in T' where t2 is not t1 do
13             if t2.range is bound by t1 then
14                 simplify the lower or upper bound of t2 with t1
15                 if size(t2.range) == 1 then
16                     Remove t2 from T'
           // Simplify the constraints whose parameter ranges were reduced
17         foreach constraint c in C' do
18             simplify c based on T
19             if c == true then
20                 remove c from C'
21     B ← bound constraints in C' with at most 1 tuning parameter
```

and constraints C:

$$\begin{aligned} T = \{&t_1 : [1..b], \\ &t_2 : [1..\frac{ab}{t_1}], \\ &t_3 : [1..d], d > e\} \\ C = \{&c_1 : t_1 \geq b, \\ &c_2 : t_3 \leq \frac{et_2}{a}\} \end{aligned} \tag{4.11}$$

Where a, b, d, e are non-parametric arithmetic expressions. Initially, C contains only one bound constraint which depends on one tuning parameter: t_1. By using c_1 to update the lower bound of t_1, Algorithm 2 reduces the range of t_1 to one value: $[b..b]$. Now, t_1 and c_1 do not need to be evaluated by the constraint solver and do not have to be included in T' and C' (the updated versions of T and C correspondingly). The range of t_2, which depends on t_1, is also updated:

$$\begin{aligned} T' = \{&t_2 : [1..\frac{ab}{b}], \\ &t_3 : [1..d], d > e\} \\ C' = \{&c_2 : t_3 \leq \frac{et_2}{a}\} \end{aligned}$$

Next, the compiler simplifies the range of t_2 to $[1..a]$. Since t_2 is now no greater than a, c_2 entails that $c_2 : t_3 \leq \frac{et_2}{a} \leq c_2 : t_3 \leq e$. Thus, t_3 range can be reduced from $[1..d]$ to $[1..e]$:

$$\begin{aligned} T' = \{&t_2 : [1..a], \\ &t_3 : [1..e]\} \\ C' = \{&c_2 : t_3 \leq \frac{et_2}{a}\} \end{aligned}$$

c_2 cannot be removed from C' since it depends on t_2, which can take more than one value. Thus, Algorithm 2 eliminates one parameter and one constraint given initially in Equation (4.11), and reduces the ranges of the remaining tuning parameters.

4.4 Memory Allocation

The challenge of tuning is often to find a balance between opposing effects. Reduction tiling comes with a trade-off of memory consumption versus performance. Small

```
mapWrg(0)(mapLcl(0)(
  mapSeq(toGlobal(f)) ∘ mapSeq(toLocal(g))
)) << (x: [[[T]_K]_M]_N)
```

Listing 4.4: Example expression with a shared intermediate buffer

sliding window tiles result in more parallel work at the cost of increased intermediate memory consumption.

For a fair evaluation of tuning effects on memory consumption, the LIFT compiler must allocate the minimum amount of memory required for convolution. The rest of this section presents this thesis' contribution to memory allocation – an intermediate buffer size inference method in functional expressions.

4.4.1 Intermediate Versus Output Buffers

LIFT allocates one buffer for each user function to store its results. Section 2.3.2.3 describes how the position of a user function in the AST determines the size of its buffer, *e.g.*, nesting the user function in a loop increases the size of the output buffer by the number of loop iterations. Special care must be taken for intermediate memory, *i.e.*, the buffers consumed by the user functions and not returned as the output of the entire LIFT program.

Since intermediate results do not need to be preserved beyond the invocation of the consumer user function, the same intermediate buffer can be reused across all invocations of the consumer UF. Although this opportunity to reuse a buffer is straightforward to detect when the intermediate data is produced and accessed sequentially, parallelisation requires a more sophisticated approach. The prior work treats the intermediate and output buffer allocation the same; this results in missed opportunities to reduce memory consumption. We now look at the example illustrating the difference between the intermediate and output buffer allocation.

Example Consider the expression in Listing 4.4, where the UF `g` is applied on each element of the 3D array `x`, the UF `f` is applied on each element of the result of `g` and the result of `f` is returned as the output. The outermost dimension of `x` is mapped onto work groups in dimension 0; the middle dimension of `x` is mapped onto local threads in dimension 0. UF `g` is mapped onto each sub-array of size K sequentially, and the

```
mapWrg(0)(mapSeq(
  mapLcl(0)(toGlobal(f)) ∘ mapLcl(0)(toLocal(g)))
) << (x: [[[T]_K]_M]_N)
```

Listing 4.5: Example expression with a non-shared intermediate buffer

result is put into a local buffer; that local buffer is read sequentially during invocation of UF `f`.

Output buffer size is determined trivially: since a global buffer is accessible by both work groups, local threads and sequentially, the output size of `f` is multiplied by the number of iterations in **`mapWrg`**`(0)`, **`mapLcl`**`(0)` and **`mapSeq`**.

$$size_{mem_f} = N \times M \times K$$

An intermediate local buffer can only be accessed within a single work group, hence there is no need to multiply the buffer size by the number of work groups:

$$size_{mem_g} = M \times K$$

In the expression in Listing 4.5, local threads are sequentially applied on the M chunks of K elements, so the intermediate buffer size can be reduced further. Since mem_g needs to hold only K elements until they are read by `f` and are not needed anymore, the compiler needs to allocate enough space for only one iteration of **`mapSeq`**.

$$size_{mem_g} = K$$

Both Listings 4.4 and 4.5 illustrate the importance of considering parallel mappings of nested loops for memory allocation. Listing 4.5 also shows that function composition needs to be considered to allocate intermediate buffers more sparingly than output buffers. Prior LIFT work allocated intermediate and output buffers in the same way, scaling the buffer sizes based on the parallel mappings of the nested loops and the ASs of the buffers. This approach still results in optimal memory allocation in the Listing 4.4, since considering the work group parallelisation of **`mapWrg`**`(0)` and the local AS of mem_g is sufficient to determine the size of mem_g as $M \times K$. This technique comes up short in Listing 4.5, where the buffer size multiplier of **`mapSeq`** must be propagated only onto the buffer mem_f, not mem_g.

4.4.2 Intermediate Buffer Reuse

The memory allocation pass in LIFT traverses the AST in the top-down order, increasing buffer size multipliers with each level of loop nesting. Three buffer size multipliers are used, one for each of the three memories of the GPU. When a UF is encountered, the size of its result buffer is calculated as the size of a single result times the accumulated multiplier corresponding to the buffer AS. While the size of the output buffers must consider the total number of results produced, the intermediate buffers do not have to store all the results ever produced. This work extends the pass to calculate the intermediate buffer size based strictly on the number of threads and work groups sharing the intermediate buffer.

Specifically, the following approach is used to establish whether to propagate the accumulated multipliers onto function arguments:

- If the function is not concrete – *i.e.*, it contains no UFs and thus doesn't write to memory – propagate the multipliers onto the arguments since they might allocate an output buffer.

- If the function is concrete:

  - Propagate the **global memory multiplier** unconditionally since global memory is shared on all levels of parallelism, thus extra memory needs to be allocated for sequential iterations, threads and work groups.

  - Propagate the **local memory multiplier** if there exists an outer `mapLcl`. In such a case, the function is executed in parallel by local threads; thus all intermediate local buffers need to be large enough to fit the intermediate outputs of all threads.

  - Do not propagate the **private memory multiplier** since registers are not accessible in parallel. An intermediate private buffer can always be reused across sequential and parallel loop iterations.

Effectively, this method helps the compiler detect cases when only a part of the allocated buffer is ever used. Not propagating multipliers implies starting multipliers accumulation from scratch at the current stage of AST traversal. Accumulating fewer multipliers leads to a smaller allocation. Since the unallocated memory was never used, no out-of-bounds accesses are produced.

The intermediate buffer size optimisation is especially useful for convolution on mobile GPUs. Since sliding windows are too large to be processed sequentially, window reduction needs to be split into at least a two-level reduction tree. Thus, an intermediate buffer is required; the small memory size of a mobile GPU calls for extra care when allocating memory.

4.5 Evaluation

This section explores the performance of the automatically generated direct convolution in LIFT. A comparison is given against the best handwritten library for the ARM Mali GPU: the ARM Compute Library [Arm21]. The evaluation uses the version of the LIFT compiler which is extended to include the contributions of this chapter.

While the initial state of the compiler prior to this work could provide an additional evaluation possibility, it is not provided in this chapter for two reasons. Firstly, the type system of LIFT has been extended with more powerful arithmetic simplification patterns to support the complex convolution expressions discussed in Section 4.2.4. While the new simplification patterns do not affect the performance of generated code, they are required to validate the semantics of the provided LIFT expression and generate correct kernels. Secondly, this chapter presents the first efforts in the integration of auto-tuning into the compiler itself. The prior approaches depend on the off-the-shelf tuners such as ATF [Ras18] and OpenTuner [Ans14], and do not leverage domain knowledge or program semantics. The ARM Compute Library provides a more challenging baseline since its tuning is driven by human expert knowledge.

4.5.1 Experimental Methodology

Code generation For each candidate low-level convolution lambda, the LIFT compiler is used to generate a GPU-accelerated program including C++ host code and OpenCL kernels. The host code sets up the device, compiles the GPU code, allocates buffers on the host (Central Processing Unit (CPU)) and the device (GPU), and schedules data transfers and kernel execution; it also measures execution time per kernel. Three OpenCL kernels are used to pad the input in global memory, perform convolution and crop the outputs, respectively. For each layer configuration, 1000 randomly chosen implementations are generated that satisfy all the constraints. The median runtime of 3 runs of each implementation is recorded.

Table 4.3: All unique convolutional layer configurations of VGG-16 and the runtime [ms] evaluated for the ARM Compute Library (Direct and GEMM) and the LIFT-generated code for the HiKey 970 (Kirin 970 processor).

Layer	Input	Conv	ARM Direct	ARM GEMM	Lift
0	3x224x224	64x3x3	38.61	2.98	9.09
2	64x224x224	64x3x3	852.03	80.14	77.08
5	64x112x112	128x3x3	426.22	37.94	40.65
7	128x112x112	128x3x3	906.66	88.09	69.60
10	128x56x56	256x3x3	452.48	23.73	58.90
12 14	256x56x56	256x3x3	975.69	60.45	84.75
17	256x28x28	512x3x3	546.63	22.30	46.07
19 21	512x28x28	512x3x3	1201.93	58.78	94.83
24 26 28	512x14x14	512x3x3	311.04	17.13	19.8

As a baseline to evaluate the performance of LIFT-generated code, the ARM Compute Library (v19.02) with the Graph API is used, implementing the same layers and running these on the GPU by indicating *cl* as the target from the API. All ARM Compute Library results are produced using ARM's built-in auto-tuner. The median runtime of 100 runs of each implementation is recorded.

Benchmarks To evaluate the code generated, all nine unique layer configurations of the VGG-16 model [Sim14] are used. This network is well-studied performance in literature and has higher resource requirements than others such as ResNet and GoogleNet [Che15]. Table 4.3 presents the layer configurations. All results are validated by using a fixed random input and comparing the output with that of PyTorch.

Platform This work targets the ARM Mali-G72 (12 cores) mobile GPU using the HiSilicon Kirin 970 SoC running Debian GNU/Linux 9.8. The highest frequency (767MHz) is used.

GPU Execution Time Measurement For LIFT results, GPU execution time is measured using the `cl_event` associated with the kernel launches. For the ARM Compute Library, GPU execution time is measured by intercepting all OpenCL calls using a

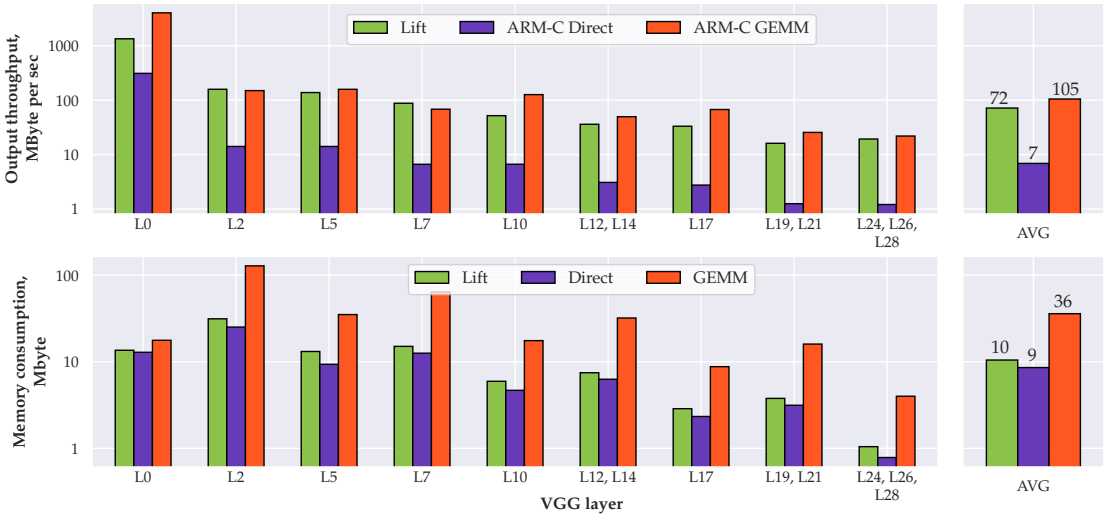


Figure 4.4: Throughput and memory consumption comparison of LIFT-generated kernels versus the direct and GEMM-based convolution methods on VGG-16.

custom profiler, which is an OpenCL wrapper library. The library automatically grabs the `cl_event` associated with each OpenCL kernel launch or creates one on the fly if required. This is done in a fully transparent way and does not influence the application being profiled. This allows us to reuse the exact same methodology for measuring execution time for the LIFT generated GPU code and the ARM Compute Library. The numbers reported are the sum of all the GPU kernels involved in the operations of a convolutional layer, including the time to pad the input and crop the outputs.

4.5.2 Comparison with ARM Compute Library

Table 4.3 shows the execution times of the LIFT-generated OpenCL kernels and the ARM Compute Library direct convolution and GEMM implementation. Both these versions have been auto-tuned using the tools provided by the ARM Compute Library. As evident from the results, the LIFT-generated code is always faster than the ARM Compute Library direct convolution and more space-efficient than its GEMM method. Furthermore, in some cases, it is actually on par or better than the highly tuned GEMM implementation.

Figure 4.4 shows the performance of the LIFT generated code expressed as throughput – the amount of useful outputs generated per second – compared to that of direct and GEMM-based convolution from the ARM Compute Library. For every layer, LIFT is faster than the ARM Compute Library direct convolution and is 10× faster on aver-

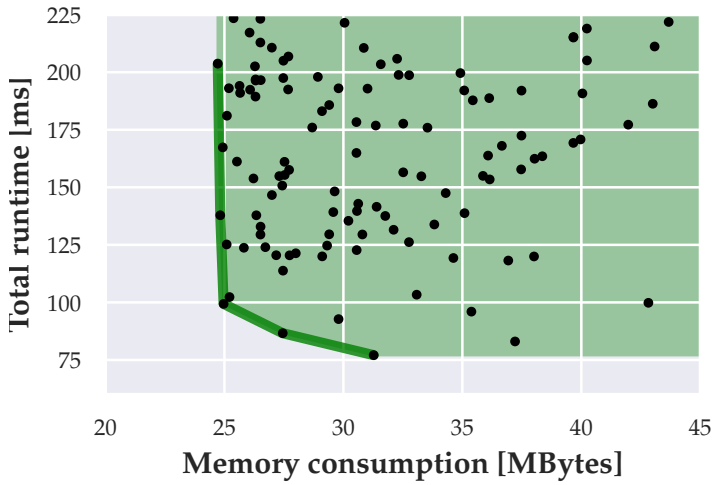


Figure 4.5: The Pareto frontier of time and space efficiency of the search space explored for layer 2 of VGG-16.

age. While LIFT kernels achieve only 0.7× the throughput of the GEMM-based implementation, the memory consumption is 3.6× less and is close to that of the vanilla direct convolution. This demonstrates that an automatic code generation based on LIFT outperforms a human expert.

4.5.3 Multi-objective Optimisation

Depending on the application, priorities in neural network inference optimisation might shift. In a resource-bound system such as a mobile GPU that is shared among multiple tasks, a low memory footprint is required; for time-critical tasks, throughput or latency are to be prioritised. Figure 4.5 demonstrates how search space exploration allows for multi-objective optimisation to cater for various budgets: advancing the Pareto frontier results in a set of implementation candidates to choose from statically or at runtime for specific time and space requirements. In the case of VGG layer 2, the compiler might prioritise space efficiency by using 25 MBytes to compute results in 100 ms; when the memory budget is bigger, the compiler can prefer the 77 ms kernel that uses 31 Mbytes of space.

Populating a sizeable Pareto set is made possible thanks to the exploration of the tuning parameter search space, performed in a safe way thanks to constraint inference. Compared to libraries that depend on sets of handwritten kernels, a compiler can adapt to finer differences in the workload and target hardware.

Table 4.4: Best parameters found for layer 7 of VGG-16.

Parameter	Value
Input tile size	5×5
Number of kernels per work group	4
Number of windows per thread	3
Sequentially processed input elements	144
Optimisational padding size	11
Vector size	4
Unrolling	No
Coalescing	Yes

4.5.4 Analysis of the Best Point

This section analyses one of the best points found using the 7th layer of VGG as an example. Table 4.4 shows the best tuning parameters found together with the thread local sizes for the GPU kernel responsible for performing a partial convolution. These parameters show that a work group processes a tile which can fit 9 sliding windows. 4 out of 128 kernels are processed by a work group, enabling reuse of the input data multiple times, without adding too much register pressure; 3 out of 9 sliding windows are processed by each thread, enabling reuse of the weight data. The amount of padding is also quite minimal, which avoids unnecessary work. This implementation is vectorised which is good for memory loads on the Mali-G72 architecture.

The code produced by the LIFT compiler from Listing 4.2 for this configuration is shown in Listing 4.6 (the code has been slightly adapted for readability). Line 3 allocates the local buffer used to store the intermediate results between the two stages of reduction. Lines 4 to 6 deal with initialising variables related to threads and work group management. Line 7 initialises the accumulators stored in private memory (registers). One accumulator per kernel and per sliding window is allocated. Line 12 is the sequential loop which reduces each chunk; the loop performs 36 iterations since there are 144 elements in each window chunk as per Table 4.4, and vectorised operations process 4 elements at once. This loop first performs several vector loads of input elements from the sliding windows, as well as weight elements from convolutional kernels; this enables data re-usage across the 4 kernels and 3 sliding windows. The

kernels are then convolved with the input data element-wise on lines 23 to 31. After the loop, the accumulator variables holding the partial results of the convolution are written out to local memory. Finally, the intermediate results are reduced in the second stage of the convolution on lines 44 to 46 and written out to global memory on line 47.

4.6 Summary

Compared to handwritten convolution implementations and code generators relying on imperative IRs, a functional IR presents several advantages for performance optimisation. Expressing sophisticated algorithms such as convolution is made safer by a strong type system: semantics-breaking code changes are caught early. The expressivity of a functional IR helps capture and propagate data dimension changes in array types. Expressive types and a view system can be leveraged to generate tuning constraints automatically thus decoupling tuning from structural optimisations such as tiling, coalescing and vectorisation.

Memory allocation is aided by the increased level of abstraction: in the absence of pointer arithmetic and aliasing, buffer sizes are inferred solely from the composition of language primitives applied to the data. Each dimension of an intermediate array is well-characterised by the functional iterators such as **`map`** and **`reduce`** with known interval bounds, strides and parallel mappings.

This project's contribution to barrier insertion described in Section 5.4 shows that a declarative IR such as LIFT facilitates correct and efficient synchronisation through the lack of explicit control flow. Determining sufficient, necessary and reachable barrier placements is valuable for direct convolution, where reduction needs to be parallelised in two stages with synchronisation in between.

This chapter has shown how to achieve high-performance direct convolution on a GPU. This approach leads to a $10\times$ speedup and $3.6\times$ memory saving over the tuned ARM Compute Library implementations. However, these results are limited by the heuristic design decisions made during the manual development of the low-level LIFT expression. Decoupling of tuning from structural optimisations can be leveraged by automating the rewriting process to explore a wider design space. The next chapter discusses an automatic parallelisation method, wherein rewriting the expression affects the tuning space. Since tuning constraint generation is parallelisation-agnostic, a custom tuning space can be explored automatically for each candidate parallel mapping.

```
1  void conv(const global float* restrict kernels, const global float* restrict input,
2            global float* out) {
3    local float local_buf[288];
4    int wg_id_0 = get_group_id(0); int wg_id_1 = get_group_id(1);
5    {
6      int l_id_0 = get_local_id(0); int l_id_1 = get_local_id(1);
7      private float acc_0 = 0.0f; private float acc_1 = 0.0f; ...; private float acc_11 = 0.0f;
8      private float4 inputEl0; private float4 inputEl1; private float4 inputEl2;
9      private float4 weightEl0; private float4 weightEl1;
10     private float4 weightEl2; private float4 weightEl3;
11
12     for (int i = 0; i < 36; i+) { // start reduce_seq
13       inputEl0 = vload4((0  + f(i, l_id_0, l_id_1, wg_id_1), input + g(i, l_id_0));
14       inputEl1 = vload4((32 + f(i, l_id_0, l_id_1, wg_id_1), input + g(i, l_id_0));
15       inputEl2 = vload4((64 + f(i, l_id_0, l_id_1, wg_id_1), input + g(i, l_id_0));
16
17       weightEl0 = vload4(0   + h(i, l_id_0, wg_id_0), kernels);
18       weightEl1 = vload4(288 + h(i, l_id_0, wg_id_0), kernels);
19       weightEl2 = vload4(576 + h(i, l_id_0, wg_id_0), kernels);
20       weightEl3 = vload4(864 + h(i, l_id_0, wg_id_0), kernels);
21
22       // start map_seq_unrolled
23       acc_0  = dotAndSumUp(acc_0,  inputEl0, weightEl0);
24       acc_1  = dotAndSumUp(acc_1,  inputEl0, weightEl1);
25       acc_2  = dotAndSumUp(acc_2,  inputEl0, weightEl2);
26       acc_3  = dotAndSumUp(acc_3,  inputEl0, weightEl3);
27       ...;
28       acc_8  = dotAndSumUp(acc_8,  inputEl2, weightEl0);
29       acc_9  = dotAndSumUp(acc_9,  inputEl2, weightEl1);
30       acc_10 = dotAndSumUp(acc_10, inputEl2, weightEl2);
31       acc_11 = dotAndSumUp(acc_11, inputEl2, weightEl3);
32       // end map_seq_unrolled
33     } // end reduce_seq
34
35     local_buf[0   + l_id_0 + 24*l_id_1] = acc_0;
36     local_buf[72  + l_id_0 + 24*l_id_1] = acc_1;
37     ...;
38     local_buf[232 + l_id_0 + 24*l_id_1] = acc_11;
39   }
40   barrier(CLK_LOCAL_MEM_FENCE);
41   acc_0 = 0;
42   for (int l_id_1 = get_local_id(1); l_id_1 < 4; l_id_1 += get_local_size(1)) {
43     for (int l_id_0 = get_local_id(0); l_id_0 < 9; l_id_0 += get_local_size(0)) {
44       for (int i = 0; i < 8; i++) {
45         acc_0 = acc_0 + local_buf[i + 8*l_id_0 + 72*l_id_1];
46       }
47       out[o(l_id_0, l_id_1, wg_id_0, wg_id_1)] = acc_0;
48     }}}
49 float4 dotAndSumUp(float acc, float4 l, float4 r){ return acc + dot(l, r); }
```

Listing 4.6: Best generated convolution implementation for VGG layer 7. Functions `f` and `g` on lines 13 to 15, `h` on lines 17 to 20, and `o` on line 47 abstract array index expressions for brevity. The indices are functions of loop counters, generated by LIFT based on view transformations and parallel mappings.

Chapter 5

Parallelism Mapping Through Constraint Satisfaction

This chapter tackles the automatic parallelisation of programs for a GPU. The functional IR of LIFT is leveraged in two ways. Firstly, the implicit parallelism of the input program is exposed via the `map` pattern. Based on the positions of `map` instances in the given expression, arithmetic constraints are generated for all loops. The constraints capture the programming model of a target platform and codify valid ways of parallelising each loop.

Secondly, a synchronisation barrier insertion method is proposed, which determines sufficient, reachable and necessary placements of OpenCL barriers. This technique replaces the pattern-matching barrier elimination method described in Section 2.3.2.4 for a more comprehensive approach. The control flow of the program is represented using a Memory Access Graph (MAG) constructed from the functional AST. MAG is used to detect data dependencies between accesses and identify barrier placements to ensure correct access ordering.

5.1 Introduction

Parallelisation presents a unique challenge in automatic code generators. There are many optimisations that need to be considered (*e.g.*, tiling, coalescing, prefetching), and many ways to map data (*e.g.*, shared memory) and computation (*e.g.*, work groups, threads), leading to a large implementation space. Different approaches have been proposed to finding optimal candidates. TVM [Che18a] relies on the user to specify Halide schedules. The *Ansor* [Zhe20a] project provides a TVM auto-scheduler that ex-

plores the search space of parallel templates. However, the scheduler does not prevent generation of invalid parallel mappings thus diminishing search efficiency.

Polyhedral compilers [Zer19; Vas18; Bag19] automate exposing parallelism, but often rely on heuristic scheduling combined with an internal performance model [Cyp18; Zer19] to find an optimal schedule. *Accelerate* [McD13] and *Futhark* [Hen17], two functional approaches, rely on hard-coded heuristics to choose a parallelisation strategy.

LIFT uses a different approach where optimisation choices are expressed via rewrite rules and a search of the space is performed via sampling [Ste16]. To tackle the large search space, LIFT relies on a multi-stage approach where algorithmic optimisations such as tiling are first explored using rewriting. This is followed by hardware-specific optimisations such as using shared memory and mapping parallelism using the same rewriting system.

While the algorithmic exploration phase results in transformed programs that are correct by construction, extra effort is required for the latter phase. Exploiting shared memory, mapping parallelism and ensuring correct synchronisation is a delicate balancing act, and is hard to encode in a rewrite system. Parallelisation involves side-effects that are hard to account for with the fine-grained rewrite rules of LIFT. Most LIFT papers shy away from this problem and, like many, the current LIFT compiler relies on hard-coded heuristics combined with ad-hoc mechanisms to guide this process. Although this practical approach produces high-performance code, it is far from being an ideal state of affairs.

This chapter describes a new approach to mapping parallelism in the context of the data-parallel functional LIFT IR. It reformulates parallelisation mapping as a constraint satisfaction problem encoding most of the restrictions of the programming model. Crucially, rewrite rules are still used to perform the exploration of the space, but the rewrites producing invalid mappings of parallelism can be avoided. By automatically generating parallelisation constraints, the compiler prunes away invalid implementations from the search space. While the invalid mappings can instead be discovered during subsequent compilation using static AST analysis, the constraint-based method detects invalid mappings earlier, avoiding wasted compilation effort.

To evaluate this new approach, the VGG-16 CNN [Sim14] is used as a use-case on a mobile GPU, with convolution as the focus. Convolution implementations require high levels of loop nesting, especially after tiling and thus present an additional challenge to parallelise. Chapter 4 has shown how convolution is expressed and optimised in

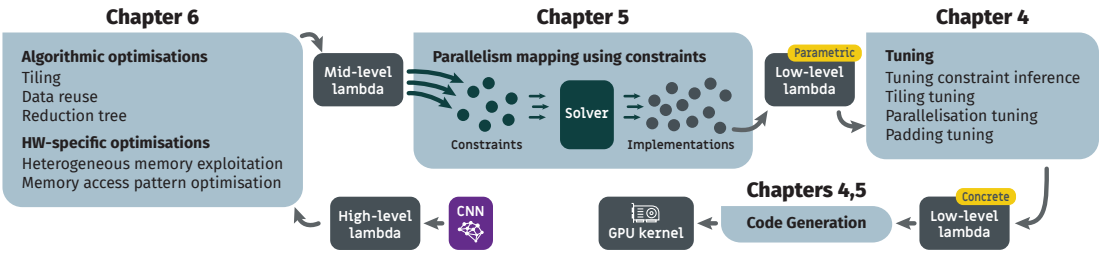


Figure 5.1: The entire optimisation flow in LIFT.

LIFT. In contrast to prior work, parallelism mapping is performed automatically using constraints.

The experimental results collected on ARM Mali GPU show that this new constraint-based approach outperforms the handwritten ARM Compute Library [Arm21] direct convolution kernels by 12× and is on par with its GEMM method while using 3.6× less memory. It also matches the performance of the state-of-the-art TVM [Che18a] code generator while using 2.7× less memory, which is important in the context of mobile GPUs.

The main contributions of this chapter are:

- Parallelisation constraint generation capturing scheduling restrictions specific to individual loops;
- Memory Access Graph-based approach to synchronisation barrier insertion in a functional IR;
- Automatic parallelisation of VGG-16 on ARM Mali GPU, achieving performance on par with TVM and memory savings of more than 2×.

The rest of this chapter is organised as follows. The next section provides motivation, introduces the use case-input program and contextualises the parallelism mapping approach within the entire optimisation flow of LIFT. Section 5.3 presents the core contribution of mapping parallelism in a functional IR with constraints. Section 5.4 describes the novel method of inserting synchronisation barriers in a functional IR. Section 5.5 evaluates the proposed approach and Section 5.6 concludes.

5.2 Overview and Motivation

Figure 5.1 presents the LIFT optimisation flow. Starting from a high-level expression, the input program is first transformed at the algorithm level by applying optimisations

```
1  def conv( inputData      :  [[[float]_inChs]_inW]_inH,
2            kernelsWeights :  [[[[float]_inChs]_kerW]_kerH]_outChs,
3            padSize        :  (int,int,int,int),
4            kernelStride   :  (int,int) ) :  [[[float]_outChs]_outW]_outH =
5    toHost o oclKernel((slideWindows', kernelsWeights') ⇒
6      mapND_2(slideWin: [T]_(inChs * kerW * kerH) ⇒
7        map(singleK: [T]_(inChs * kerW * kerH) ⇒
8          reduce(0, +) o map(*) << zip(slideWin, singleK)
9        ) << kernelsWeights'
10     ) << slideWindows'
11   ) << ( mapND_2(joinND_2) o
12           slideND_2(kerH, kerW, kernelStride._1, kernelStride._2) o
13           padND_2(padSize, value = 0) o
14           toGPU << inputData,
15         map(joinND_2) o toGPU << kernelsWeights )
```

Listing 5.1: High-level LIFT expression of convolution. This expression is identical to the one in Listing 4.1.

such as tiling. Then, hardware-specific optimisations come into play such as optimising memory access patterns or exploiting shared memory and a mid-level lambda is produced. Chapter 6 discusses how this stage is performed automatically in LIFT. The resulting mid-level lambda expresses several structural optimisations and presents multiple opportunities for parallelisation, which are exploited in the second stage.

In contrast to prior LIFT work [Hag18], this chapter separates parallelism mapping into its own stage, which produces a low-level parametric parallelised lambda. In the final stage, the lambda is auto-tuned, using the approach presented in Chapter 4. This results in a low-level expression, which is vectorised and passed to the LIFT OpenCL code generator [Ste17].

5.2.1 The Input Program

The input program presented as a use case in this chapter is the high-level direct convolution expression shown in Listing 4.1 and repeated in Listing 5.1. Since this chapter focuses on modelling the parallelisation restrictions, the problem of navigating the search space of other optimisations is left for Chapter 6. The set of rewrite rules is assumed to be chosen heuristically to optimise the input program, short of mapping

parallelism, which is the focus of this chapter. The optimisations applied are summarised below.

- Tiling input image and weights multiple times allows exploiting *data locality* at multiple levels;
- *Data is reused* by prefetching many input tiles and kernels before entering the loops on lines 6 and 7;
- *Memory hierarchy* is exploited by storing the results of prefetched data in shared or private memory and by accumulating in shared or private memory when reducing;
- *Memory access patterns* are optimised by inserting transposition before and after prefetching as well as across reduction trees;
- *Weight kernel partitioning* is performed to increase locality and, sometimes, improve access patterns;
- *Vectorisation* is applied exhaustively to 1D **map**s with consecutive memory accesses. To establish whether accesses are consecutive, LIFT checks the differences between index expressions at iterations i, $i+1$, .., $i+(vectorLen-1)$.

Applying the optimisations listed above, the rewriting process creates over 200 parallelisable loops (in the form of **map**s), allowing to match the complex thread hierarchy of a GPU. The next sections discuss how the exposed parallelism is automatically exploited and mapped.

5.2.2 Challenge of Mapping Parallelism

LIFT exposes parallelism opportunity through the use of the **map** IR primitive which corresponds to a loop. During rewriting, the **map** primitive can be replaced by a parallel loop implementation, exploiting different levels of parallelism in the architecture (*e.g.*, work groups, local threads, vectorisation) or turned into a sequential implementation. Herein lies the challenge of the search space explosion: treating each loop as an independent parallelisation opportunity results in a large number of invalid mappings of parallelism.

Consider the code in Listing 5.2 that must be mapped onto the OpenCL parallelism hierarchy. As seen in Section 2.2.2, in OpenCL parallelism exists at the global level

```
1  for i in 1 to I do                                          // Loop A
2    for j in 1 to J do                                        // Loop B
3      for k in 1 to K do                                      // Loop C
4        local_buf[j][k] = f( input[i][j][k] );
5    for k in 1 to K do                                        // Loop D
6      for j in 1 to J do                                      // Loop E
7        output[i][j][k] = g( local_buf[j][k] );
```

Listing 5.2: Example parallelisable expression

(G0,G1), or in a combination of work group levels (W0,W1) and local levels (L0,L1). In Listing 5.2, `f` is applied on a global input, the results are stored in a local buffer, which is then read to produce a global result using `g`. Listing 5.2 contains five loops that can be mapped in numerous ways. Table 5.1 provides six (non-exhaustive) possible mappings.

Naive mapping M1 parallelises the outer loop across global threads. Although valid, this might lead to a lot of sequential work or perhaps little parallelism depending on the loop iterations count. Since the iteration number could be a tuning parameter (*e.g.*, dependent on tile size) this could be an interesting design point nonetheless, as certain tuning values could lead to good performance.

Mapping M2 parallelises loops A, B and D across all global OpenCL threads in two dimensions. However, because of a data dependency, a global barrier is required between lines 4 and 5. Since OpenCL does not support global synchronisation, this mapping is invalid.

Mapping M3 produces out-of-scope reads since local data cannot be shared between work groups.

Mapping M4 schedules work groups across the outer loop A, and the compiler can insert a local synchronisation barrier between loops B and D. However, the barrier could impose a performance penalty.

Mapping M5 vectorises loop E. However, since vectorised loads require contiguous data in memory and the threads are accessing non-contiguous elements in line 7, such vectorisation is invalid.

Mapping M6 is valid and ensures that each thread access only the results it produced itself, thus eliminating the inter-thread data dependency and the need for the barrier. Vectorisation is applied on the contiguous access in line 4. This mapping might lead to good performance.

Table 5.1: Example parallel mappings for Listing 5.2. **Gn**, **Wn** and **Ln** stand for global, work group and local parallelisations respectively in the OpenCL dimension `n`. S stands for *sequential* and **V** – for *vectorised*.

	Loop parallelisation					Parallel mapping assessment
	A	B	C	D	E	
M1)	**G0**	S	S	S	S	Under-saturated cores
M2)	**G1**	**G0**	S	**G0**	S	Invalid: cannot synchronise
M3)	S	**W0**	**L0**	**W0**	**L0**	Invalid: out-of-scope reads
M4)	**W0**	**L0**	S	**L0**	S	Synchronisation overhead
M5)	**W0**	**L0**	S	**L0**	**V**	Invalid: unvectorisable
M6)	**W0**	**L0**	**V**	S	**L0**	Might be optimal

This example demonstrates that naive parallelisation strategies can produce invalid code. Manual scheduling of kernels with hundreds of loops – such as convolution – is costly and poorly generalisable. Furthermore, although invalid kernels could be detected during code generation, early detection of invalid parallel mappings is desired to avoid the overhead of tuning invalid kernels. This chapter tackles the problem by modelling parallelisation restrictions in LIFT using constraints and finding valid mappings using a solver.

5.3 Parallelisation Constraint Generation

State-of-the-art heuristic parallelisation methods focus on defining the prospective parallelising strategies. This approach falls short when presented with new parallel architectures and exotic applications. This section discusses an alternative approach of capturing the restrictions of the target. Invalid parallel mappings can be avoided by automatically generating constraints based on a given AST. Valid parallelisations are free of data races, respect the memory scoping rules and the parallelism hierarchy.

The constraints discussed here encode parallelisation restrictions present in many programming models such as OpenCL, CUDA and OpenMP. OpenCL provides an interesting shared-memory execution model as a use case, but the methodology is not restricted to this model.

The parallelism mapping stage begins by traversing the given expression in search

Table 5.2: Encoding of map transformation choices

Code value	Map transformation
0	**mapSeq**
1	Fused with the outer **map**
10, 11, 12	**mapLcl** in dimension 0, 1 or 2 respectively
20, 21, 22	**mapWrg** in dimension 0, 1 or 2 respectively
30, 31, 32	**mapGlb** in dimension 0, 1 or 2 respectively

of **map**s, which are used by LIFT IR to express parallelisation opportunities. Each **map** is associated with a new arithmetic parameter representing the choice of **map** scheduling. Then, the search space is restricted with a set of constraints on the new parameters, built using expression types, views, memory allocation, AST structure and target hardware limitations. Any constraint solver library that supports all predicates described in Section 5.3.2 can be used to generate a restricted search space of implementations to be traversed using established search techniques.

5.3.1 Map Scheduling Choices

Sequential loops are created by replacing **map** with **mapSeq**. They create vectorisation opportunities and extend the lifetime of a thread.

Parallel loops distribute work across all threads (**mapGlb**), work groups (**mapWrg**) or threads within a work group (**mapLcl**). For each of these parallelisation domains, one out of three OpenCL dimensions is chosen. The dimension choice is explored to achieve memory coalescing since threads within a warp are in dimension 0.

Map fusion can be applied on chains of perfectly nested **map**s. parallelising fused maps allows distributing more work across threads. Fusion is achieved by replacing **map**(**map**(f)) with (**split**(..) ∘ **map**(f) ∘ **join**).

The **map** transformation parameters are defined on an integer range representing scheduling choices, enabling the use of well-optimised integer constraint solvers. Table 5.2 provides the parameter encoding scheme. The encoding scheme is chosen such that constraint predicates can be defined concisely. As we will see later, this encoding allows distinguishing between levels of parallelism using division by 10, and between parallelisation dimensions – using modulo 10.

5.3.2 Constraint Generation

In the context of rewriting LIFT programs, a constraint is a predicate restricting the range of values of one or more integer parameters representing design choices:

```
Constraint( parameters: List⟨Parameter⟩,
            predicate: List⟨Int⟩ => Boolean )
```

Where `List⟨T⟩` denotes a list of elements of type `T`, and `=>` denotes a compiler-level function. The emitted predicates are logical conjunctions of quantifier-free equality and inequality constraints over nonlinear integer domain. The supported operators express comparison ($>$, $<$, $\geq$, $\leq$, $=$, $\neq$), integer arithmetic ($+$, $-$, $/$, $\times$, $\%$) and logical operators: `AND` ($\wedge$), `OR` ($\vee$) and `NOT` ($\neg$). Constraints are defined either manually to express OpenCL and hardware limitations and heuristics, or generated automatically to preserve program semantics.

The number of parallelisable **map**s and their positions in the AST are not known in advance since rewriting is performed before scheduling, and rewrite rules can add, relocate and eliminate **map**s in the program candidates. This makes it necessary to collect contextual information from the current expression before constraints can be generated. The contextual functions listed in Table 5.3 are used by the compiler to identify **map**s upon which constraints must be imposed.

Consider an example function f, which multiplies each element of the array `X` by a scalar value `y` and adds the scalar `z` to each element of the resulting array:

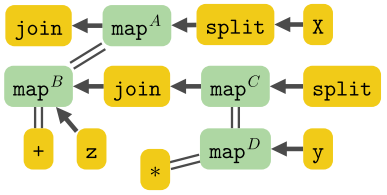
$$f(X : [T]_N,\ \ y : T,\ \ z : T) = X * y + z$$

The expression in Figure 5.2a is one possible implementation of f, in which multiplication is double-tiled and summation is single-tiled. The corresponding **map** nesting tree shown in Figure 5.2c is built from the AST shown in Figure 5.2b. The nesting tree is used to provide the first four functions in Table 5.3. The `MapNestingChain` set, for example, would contain chains ($\mathbf{map}^A, \mathbf{map}^B$) and ($\mathbf{map}^A, \mathbf{map}^C, \mathbf{map}^D$). `ConcreteMaps` is collected by checking the nested user functions; memory usage is inferred from the **toLocal** and **toPrivate** primitives. The three last functions return the dimensions of parallelism for each map construct in a nesting chain.

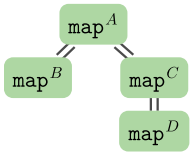
Most contextual functions in Table 5.3 are known to the compiler during constraint generation from just the given lambda and can be used to decide which constraints to generate. The three last functions, however, are based on the chosen **map** parallelisations, and can therefore be evaluated only in the solving phase, once the solver paral-

```
1  join ∘ map^A (
2    map^B (_ + z) ∘ join ∘
3    map^C (map^D (_ * y)) ∘ split(s1)
4  ) ∘ split(s0) << (X: [T]_n)
```

(a)



(b)



(c)

Figure 5.2: An example LIFT expression (a), its AST (b) and the corresponding **map** nesting tree (c). Letters A-D are unique loop identifiers used to refer to the corresponding **map** instances later on in text. The underscore symbol in **map**(_ + z) denotes the **map** lambda parameter. Arrows denote composition, double lines denote nesting.

lelises the relevant **map**s. These functions are expressed in a way a solver can parse, *i.e.*, as logical conjunctions integrated into the constraints themselves. For example, when some constraint C must be enforced for each global dimension used among (**map**^{A}, **map**^{B}), the production rule generating constraint C is equivalent to the following:

$$\forall g \in \texttt{GlbDimsUsedIn}(\textbf{map}^A, \textbf{map}^B) \ : \ \texttt{C}(\textbf{map}^A, \textbf{map}^B)$$

$$\Longleftrightarrow$$

$$\bigwedge_{g \in \{0,1,2\}} \texttt{C}(\textbf{map}^A, \textbf{map}^B) \ \vee \ (\texttt{mapEncoding}(\textbf{map}^A) \neq 30 + g \ \wedge \ \texttt{mapEncoding}(\textbf{map}^B) \neq 30 + g)$$

Where $\bigwedge$ denotes conjunction, and mapEncoding(m) returns the encoding of m according to Table 5.2. This means that for each parallel dimension g, either the constraint C must hold, or the dimension g must not be used in any of the **map**s. Abstracting these extra conjunctions away as stand-alone contextual functions leads to concise production rules.

The contextual information is used to generate six types of constraints that are satisfied only by programs that adhere to the OpenCL programming model and are data race-free. In the absence of a formal definition of a correct OpenCL program, the constraints discussed below attempt to capture the restrictions listed in the OpenCL documentation [Khr22].

5.3.3 Memory Scoping Constraints

Parallel programming models often restrict memory types to specific parallel levels. In OpenCL, private memory is accessible to a single thread, while local memory is

Table 5.3: Contextual functions for constraint generation

Contextual function	Result
`NestedMaps(m)`	All **map**s nested in **map** `m`.
`OuterMaps(m)`	All **map**s wrapping **map** `m`.
`MapNestingChains`	All **map** chains from the outer to the innermost nested **map**s.
`m1.perfectlyNestedIn(m2)`	True if **map** `m1` is perfectly nested in **map** `m2`.
`ConcreteMaps`	All **map**s that write to memory.
`m.usesPrivateMemory`, `m.usesLocalMemory`	True if **map** `m` accesses private or local memory respectively.
`GlbDimsUsedIn(`**maps**`)`, `WrgDimsUsedIn(`**maps**`)`, `LclDimsUsedIn(`**maps**`)`	Global, work group or local dimensions respectively used in **map**s.

shared across threads in a work group, but not across work groups; global memory is accessible on all levels. On a GPU, private, local and global memories can correspond to registers, compute core SRAM blocks and DRAM respectively. Due to differing access speeds and capacities of the three memory types, an optimal memory mapping for a memory-bound application is heterogeneous and specific to the target platform. An automatic parallelisation method must produce valid implementations irrespectively of the memory mapping.

5.3.3.1 Private Memory Scoping

maps that consume or produce private memory cannot be parallelised since private memory is restricted to a single thread. For example in Figure 5.2a, if $\textbf{map}^D$ writes the output into registers, both $\textbf{map}^D$ and its consumer $\textbf{map}^B$ must be executed within the same thread, *i.e.*, only the outer $\textbf{map}^A$ can be parallelised. Thus, if parameter `y` of $\textbf{map}^D$, or $\textbf{map}^D$ output are in private memory, $\textbf{map}^D$ cannot be transformed into **mapGlb**, **mapWrg** or **mapLcl**.

A constraint must be generated for $\textbf{map}^D$ that allows only the mappings where $\textbf{map}^D$ is sequential or fused with an outer **map**. A constraint for such **map**s can be generated

as follows:

$$\begin{gathered}\forall \texttt{m} \in \texttt{ConcreteMaps}, \texttt{m.usesPrivateMemory} \\ \textbf{GEN CONSTRAINT: } \texttt{mapEncoding(m)} \,/ 10 < 1\end{gathered} \tag{5.1}$$

This represents a production rule defining which **map** or combination thereof to generate which constraint for. As per Table 5.3, `ConcreteMaps` is a set of all **map**s in the expression that contain a function writing to memory as opposed to just transforming views. Rule 5.1 generates the constraint if input or output memories of `m` are private, or if `m` accesses a free private variable defined in an outer scope. $\texttt{mapEncoding(m)} \,/ 10 < 1$ requires that such **map**s are not parallel.

5.3.3.2 Shared Memory Scoping

Local memory scoping requires that shared memory is only accessed by threads executed on the same compute core. In Figure 5.2a, the multiplication output could be put in local memory; in that case, a legal parallelisation is following:

```
1  join ∘ mapWrg^A(0)(
2    mapLcl^B(0)(toGlobal(_ + z)) ∘ join ∘
3    mapLcl^C(0)(mapSeq^D(toLocal(_ * y))) ∘ split(s1)
4  ) ∘ split(s0) << X
```

The following parallelisation is illegal:

```
1  join ∘ mapGlb^A(1)(
2    mapGlb^B(0)(toGlobal(_ + z)) ∘ join ∘
3    mapGlb^C(0)(mapSeq^D(toLocal(_ * y))) ∘ split(s1)
4  ) ∘ split(s0) << X
```

A constraint must be produced requiring that local memory is accessed only within local **map**s assigned to a single work group. This constraint is expressed as follows:

$$\begin{gathered}\forall \texttt{m} \in \texttt{ConcreteMaps}, \texttt{m.usesLocalMemory}, \\ \forall \texttt{Chain} \in \texttt{MapNestingChains}, \texttt{m} \in \texttt{Chain}, \\ \forall \texttt{w} \in \texttt{WrgDimsUsedIn(Chain)} \\ \textbf{GEN CONSTRAINT: } \left(\texttt{mapEncoding(m)} \,/ 10 < 2\right) \wedge \\ \bigvee_{\texttt{mOuter} \in (\texttt{Chain} \cap \texttt{OuterMaps(m)})} \texttt{mapEncoding(mOuter)} = 20 + \texttt{w}\end{gathered} \tag{5.2}$$

Where $\bigvee$ denotes inclusive disjunction, $\texttt{m} \in \texttt{Chain}$ restricts the constraint to the chains that include `m`, and `mOuter` is one of the outer **map**s of `m`. The `WrgDimsUsedIn` term can

only be evaluated by the constraint solver once the search begins, so the constraint is supplemented by predicates enumerating all outer **map**s of `m`.

Rule 5.2 ensures that **map**s consuming or producing shared memory are local or sequential. It also ensures that such **map**s are uniquely assigned to a single work group by outer instances of **mapWrg**. In the resulting kernel, the shared memory of a work group is accessed only by the threads of that specific work group.

5.3.4 Hierarchical Parallelism Constraints

The hierarchies of parallelism in parallel hardware present an extra challenge in scheduling computation. The levels of parallelism must be mapped exhaustively, unambiguously, and conforming to the hierarchy. This section focuses on three parallelism levels of OpenCL: global, work group and local.

5.3.4.1 Duplicate Scheduling Constraint

`maps` that are directly or indirectly nested cannot be parallelised in the same domain and dimension. In Figure 5.2a, $\mathbf{map}^C$ and $\mathbf{map}^D$ cannot be parallelised equally, and the same for other **map** nests. Duplicate scheduling is prevented as follows:

$$\forall \texttt{m} \in \texttt{ConcreteMaps}, \forall \texttt{mInner} \in \texttt{NestedMaps(m)}$$
$$\textbf{GEN CONSTRAINT: } (\texttt{mapEncoding(m)}/10 < 1) \vee (\texttt{mapEncoding(m)} \neq \texttt{mapEncoding(mInner)}) \tag{5.3}$$

The disjunction term $\texttt{mapEncoding(m)}/10 < 1$ ensures that the constraint is applied to the parallel **map**s.

5.3.4.2 Local Thread Indexing Dimensionality Constraint

Dimensions used for thread indexing within a work group must match dimensions used for work group indexing and vice versa. For example, if an expression uses **mapLcl**`(0)` and **mapLcl**`(1)` and not **mapLcl**`(2)`, it must also use **mapWrg**`(0)` and **mapWrg**`(1)`, but not **mapWrg**`(2)`. It is evident from the nesting tree in Figure 5.2c that the maximum depth of **map** nesting in the example expression is three; the only allowed parallelisations are therefore a **mapWrg** nesting a **mapLcl** in the same dimension, or up to three nested instances of **mapGlb** using different dimensions. This restriction is expressed as follows:

$$\forall \texttt{Chain} \in \texttt{MapNestingChains}, \forall \texttt{l} \in \texttt{LclDimsUsedIn(Chain)}$$
$$\textbf{GEN CONSTRAINT: } \bigvee_{\texttt{w} \in \texttt{WrgDimsUsedIn(Chain)}} \texttt{w} = \texttt{l} \tag{5.4}$$

$$\forall \texttt{Chain} \in \texttt{MapNestingChains}, \forall \texttt{w} \in \texttt{WrgDimsUsedIn(Chain)}$$
$$\textbf{GEN CONSTRAINT: } \bigvee_{\texttt{l} \in \texttt{LclDimsUsedIn(Chain)}} \texttt{l} = \texttt{w} \tag{5.5}$$

5.3.4.3 Local Thread Indexing Hierarchy Constraint

User functions must be parallelised across work groups before they can be parallelised across work group threads. In LIFT, this means that no **mapLcl** can have a **mapWrg** with the same dimension nested inside, and each **mapLcl** must be nested in a **mapWrg** with the same dimension.

In the Figure 5.2a example, this constraint permits $\textbf{map}^A$, $\textbf{map}^B$ and $\textbf{map}^C$ to be made **mapWrg**(0), **mapLcl**(0) and **mapLcl**(0) respectively, but not **mapLcl**(0), **mapWrg**(0) and **mapWrg**(0).

This constraint is produced as follows:

$$\forall \texttt{m} \in \texttt{ConcreteMaps}, \forall \texttt{mInner} \in \texttt{NestedMaps(m)}$$
$$\textbf{GEN CONSTRAINT:}$$
$$\neg((\texttt{mapEncoding(m)}/10 = 1) \wedge (\texttt{mapEncoding(mInner)}/10 = 2) \wedge (\texttt{mapEncoding(mInner)}\,\%10 = \texttt{mapEncoding(m)}\,\%10)) \tag{5.6}$$

5.3.4.4 Exhaustive Thread Indexing Constraint

All user functions must be parallelised in the same number of dimensions to leave no ambiguity in work distribution among threads. For the two nesting chains in Figure 5.2c, this requires that if $\textbf{map}^A$ and $\textbf{map}^B$ are **mapWrg**(0) and **mapLcl**(0) respectively, there must also be a **mapLcl**(0) among $\textbf{map}^C$ and $\textbf{map}^D$.

$$\forall \texttt{ChainA} \in \texttt{MapNestingChains}, \forall \texttt{ChainB} \in \texttt{MapNestingChains},$$
$$\forall \texttt{l} \in \texttt{LclDimsUsedIn(ChainA)}$$
$$\textbf{GEN CONSTRAINT: } \bigvee_{\texttt{m} \in \texttt{ChainB}} \texttt{mapEncoding(m)} = 10 + \texttt{l} \tag{5.7}$$

$$\begin{aligned}&\forall \texttt{ChainA} \in \texttt{MapNestingChains}, \forall \texttt{ChainB} \in \texttt{MapNestingChains},\\ &\forall \texttt{w} \in \texttt{WrgDimsUsedIn(ChainA)}\\ &\textbf{GEN CONSTRAINT: } \bigvee_{\texttt{m} \in \texttt{ChainB}} \texttt{mapEncoding(m)} = 20 + \texttt{w}\end{aligned} \tag{5.8}$$

$$\begin{aligned}&\forall \texttt{ChainA} \in \texttt{MapNestingChains}, \forall \texttt{ChainB} \in \texttt{MapNestingChains},\\ &\forall \texttt{g} \in \texttt{GlbDimsUsedIn(ChainA)}\\ &\textbf{GEN CONSTRAINT: } \bigvee_{\texttt{m} \in \texttt{ChainB}} \texttt{mapEncoding(m)} = 30 + \texttt{g}\end{aligned} \tag{5.9}$$

All three constraints require that if a parallel dimension of a given domain is used in one **map** nesting chain, it must also be used in all other chains.

5.3.5 Sequential Map Fusion Heuristic

Perfectly nested sequential **map**s can always be fused to reduce search space:

$$\begin{aligned}&\forall \texttt{Chain} \in \texttt{MapNestingChains}, \forall \texttt{m1} \in \texttt{Chain}, \forall \texttt{m2} \in \texttt{Chain},\\ &\texttt{m2.perfectlyNestedIn(m1)}\\ &\textbf{GEN CONSTRAINT: } \neg((\texttt{mapEncoding(m1)} = 0) \wedge\\ &\qquad\qquad (\texttt{mapEncoding(m2)} = 0))\end{aligned} \tag{5.10}$$

This forces all perfectly nested **map** pairs to be either fused or parallelised. Section 5.5.5 analyses how this heuristic affects the search.

5.3.6 Synchronisability

Safe parallelisation requires that interdependent threads are synchronisable. The following expression is an example where this is not always the case:

```
1  mapLcl^A(1)(
2    mapLcl^B(0)(f) ∘ transpose ∘ mapLcl^C(0)(g)
3  ) << (X: [T]_n)
```

transpose introduces an inter-thread dependency between $\textbf{mapLcl}^B$ and $\textbf{mapLcl}^C$, forcing the compiler to insert a barrier between the loops. However, depending on `n` and the work group size, some threads might perform fewer iterations:

```
1 for (int i^A = get_local_id(1);
2       i^A < n / get_local_size(1);
3       i^A += get_local_size(1)) {..}
```

`get_local_id(1)` and `get_local_size(1)` are OpenCL built-in primitives that return local thread index and the work group size respectively in the dimension 1. When `n` is not multiple of the work group size, threads perform differing numbers of iterations. With a barrier inside the loop, some threads get blocked indefinitely.

In LIFT, the synchronisability condition is enforced through a compiler check just before code generation. Barrier locations are determined by analysing loop bounds, computed using tuned parameters such as tile sizes and work group dimensions. Modelling synchronisability as a constraint before tuning would require estimating barrier locations conservatively. This could prevent discovery of good program candidates, so in this case early detection is sacrificed in favour of better search coverage. The next section describes the compiler pass that ensures synchronisability by either determining reachable and sufficient barrier placements, or throwing an error if the expression is not synchronisable.

5.4 Synchronisation Barrier Insertion

The usage of on-chip and off-chip shared memories often necessitates thread synchronisation. In convolution, shared memory usage is required for tiled reduction due to the large size of sliding windows, as discussed in Section 4.2.2.5. Tiled reduction poses the problem of synchronisation between the stages of the reduction tree – an instance of the readers-writers problem [Cou71]. The threads performing final reduction may depend on the results produced by the partial reduction threads; conversely, the latter may depend on the former to finish reading the shared memory region before it can be written to again.

The interdependent memory access operations are ordered using OpenCL barriers. Correct barrier placement requires that the barrier is reachable by all threads within a work group, thus limiting which conditionals can be used in the if-blocks and for-loop stopping conditions. Suspension of barrier-blocked threads can cause under-utilisation of compute cores; hence, barriers need to be used sparingly. Correct and efficient barrier placement satisfies the following conditions:

- *Sufficiency* – enough barriers must be placed to avoid data races and produce a

GPU kernel that is semantically equivalent to the input expression.

- *Reachability* – barriers must be reachable by all threads in a work group for the kernel to return.
- *Necessity* – barriers must be placed only where they are needed so that they are hit the minimum number of times to avoid overheads.

Finding the minimum number of data dependencies to synchronise is an NP-complete problem [Mid86]. Prior to this work, LIFT tackled barrier placement by detecting data dependencies through direct pattern matching of the IR and analysis of inferred address spaces [Hag18]. Section 2.3.2.4 provides an overview of the original approach. The original pattern matching-based approach benefits from the functional IR by avoiding the expensive analysis required for the imperative IRs; there is no need to analyse pointer aliasing and arithmetic, as well as array index expressions. However, the complexity of direct convolution expressions reveals the limitations of this method. Analysing the IR directly requires enumerating many permutations of functional primitives during pattern matching, which does not scale well to the increased complexity of expressions such as convolution. We will see in Section 5.5.6 that in some cases, the pattern match-based approach produces insufficient, unreachable, unnecessary and inefficient barriers.

MAG-based Approach The proposed barrier insertion method shifts focus from localised IR patterns to memory accesses, *i.e.*, the actual events to be synchronised. Barriers are required between pairs of accesses that are interdependent; dependence is determined by the positions of the accesses in the control flow and the overlap in accessed array slices. LIFT captures both control flow positions and overlaps by representing accesses symbolically using arithmetic expressions defined on loop counters. This results in a more comprehensive view of the problem, where the entire control flow between the subsequent pairs of memory accesses is considered.

The proposed method uses two program representations: views and a Memory Access Graph. Views are used to determine pairs of memory accesses that require synchronisation; MAG is used to represent control flow paths between all memory accesses. The graph is used to find intersections between paths that require synchronisation (further referred to as critical paths); the barriers are placed in these intersections to synchronise the maximum number of critical paths with the minimum number of barriers.

Algorithm 3: Memory Access Graph-based Barrier Insertion.

input : AST with allocated memory and built views

output: AST with the `Barrier` IR node or an exception if the AST is not synchronisable

```
// Build MAG
V ←a node per each memory access in the AST
E ←a directed edge for each pair of subsequent accesses in V
foreach edge e in E do
    if (
        barrier is not needed on e  or
        e is not always reachable
    ) then
        mark e as non-synchronisable
M ← ⟨V, E⟩ // Directed unweighted graph
// Identify control flow paths that require synchronisation
I ←pairs of accesses requiring a barrier // Interdependent memory accesses
P ← ∅ // Control Flow paths requiring a barrier
foreach (a1, a2) in I do
    p ←shortest path between a1 and a2
    if ∀edge e ∈ p : e is not synchronisable then
        throw NotSynchronisableProgramException
    insert p into P
// Insert barriers into the critical subpaths
foreach edge e in E : e is synchronisable do
    e.weight ← the number of paths in P that include e
foreach path p in P do
    e ← the edge in p with the maximum weight within p;
        resolve ties in favour of later edges in the control flow
    place a barrier on e
foreach access a in V do
    // Remove the barriers that are hit less often
    remove duplicate barriers on edges starting in a
// Insert barriers into the AST
foreach edge e in E : e contains a barrier do
    insert the barrier IR node into the AST after the node corresponding to e
```

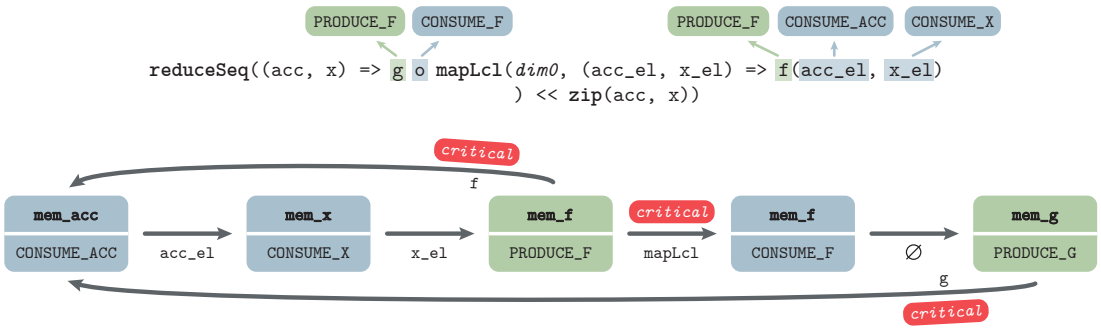


Figure 5.3: An example LIFT expression, and a corresponding Memory Access Graph used to determine optimal barrier placements. Blue and green squares represent memory accesses (reads and writes respectively); edges represent control flow paths between respective memory accesses. Critical paths are the control flow paths that require synchronisation between threads due to an inter-thread data dependency.

The compiler builds the graph by inferring the control flow between memory accesses from the AST. This representation allows placing the minimum amount of barriers efficiently, favouring barriers in the intersections of multiple critical paths with smaller degrees of loop nesting. The next section outlines the steps required to build and leverage MAG for barrier placement.

5.4.1 Memory Access Graph Construction

A MAG is a directed graph where vertices represent memory accesses and edges represent control flow paths between subsequent accesses. The graph can be built as shown on lines 1 to 8 in Algorithm 3 using the AST after memory allocation and view building. An edge between the two nodes corresponding to accesses `a1` and `a2` represents a code block `c` with one or more OpenCL statements, none of which issue memory accesses; a barrier placed anywhere in `c` provides sufficient synchronisation between `a1` and `a2`. In LIFT, memory is accessed only by user functions, so MAG vertices can be produced by collecting all user functions in the AST. The edges of the MAG are determined by function composition and application operators, as well as the control flow primitives enclosing user functions, *e.g.*, **`map`**, **`iterate`**, **`scan`** and **`reduce`**.

Figure 5.3 shows an example MAG and the corresponding expression. Each vertex is annotated with the name of the buffer being accessed: `mem_acc` for the **`reduce`** accumulator, `mem_x` for the input buffer `x`, `mem_f` and `mem_g` for the output buffers of function `f` and `g` respectively. Each vertex is also annotated with the type of access: *produce* or *consume*.

During the MAG construction stage, each edge is annotated with three bits of information: direction, the corresponding AST node and synchronisability. **Edge direction** characterises the control flow between memory accesses: the graph in Figure 5.3 contains an edge from `CONSUME_ACC` to `CONSUME_X` because `x_el` is evaluated after `acc_el` in the left-to-right evaluation of `f(acc_el, x_el)`. The edge from `PRODUCE_F` to `CONSUME_ACC` exists because **`mapLcl`** yields a **`for`**-loop, wherein writing the result of `f` is followed by reading `acc_el` in the next iteration.

The AST node corresponding to an edge `e` is the last function call to be invoked after the edge target memory access is performed; inserting a barrier IR node after that call satisfies the dependency between the two accesses of the edge. For example, in Figure 5.3, `g o` **`mapLcl`**`(f)` produces two edges: from `PRODUCE_F` to `CONSUME_F`, and from `CONSUME_F` to `PRODUCE_G`. The former edge corresponds to the call of **`mapLcl`**`(f)`; the latter represents the control flow transition from reading the result of `f` to producing the result of `g`.

Edge synchronisability defines whether a barrier could be inserted and might be needed in the corresponding code block. An edge from some access `a1` to some access `a2` targeting memories `A` and `B` correspondingly is marked as synchronisable if neither of the following conditions hold:

- `A` and `B` are different memories. In such case, accesses `a1` and `a2` can be performed in any order without the need for synchronisation.
- `a1` and `a2` are performed by user functions nested in a **`mapLcl`** with a variable number of iterations per thread. In such case, a barrier placed between `a1` and `a2` might be unreachable by some threads.

5.4.2 Critical Path Detection

Once a MAG is built, the compiler identifies control flow paths that require synchronisation. This is done in two stages: first, pairs of interdependent accesses are identified (line 9 in Algorithm 3). Second, for each pair `(a1, a2)` a path `p` is chosen such that placing a barrier on `p` synchronises all control flow paths between `a1` and `a2`. This condition is enforced on lines 10 to 15 as follows:

- No node appears in the path more than once.
- The chosen path must have at least one node reachable by all threads equal number of times. This requirement is evaluated by checking loop exit conditions.

- The IR supports only limited branching. Beyond loop exit conditions, LIFT only uses branching in the inline IF expressions produced by the **pad2D** primitive, which requires no synchronisation. For an IR with more sophisticated branching, the MAG-based method can be extended to mark conditional control flow subpaths as ineligible for barrier placements to avoid unreachable barriers.

Determining whether a pair of accesses `(a1, a2)` is interdependent is less trivial. Unless both `a1` and `a2` are reads, the compiler needs to establish whether there exists a thread that accesses data produced by other threads. For example, in Figure 5.3, if a thread t_i executing `g` consumes the output of `f` produced by another thread t_j, access `CONSUME_F` is dependent on access `PRODUCE_F`.

Establishing interdependence of accesses `a1` and `a2` requires analysing the corresponding array index expressions $h_{a1}(t)$ and $h_{a2}(t)$, where t is a variable containing the index of the thread performing the accesses. The array index expression $h_{an}(t)$ is obtained by LIFT using the view built for the LIFT expression e_n performing the access. $h_{an}(t)$ is defined using the counters of the loops encapsulating e_n and takes into account all view transformations performed by e_n. The View System is described in more detail in Section 2.3.2.2.

Access dependence is defined as follows:

Definition 5.4.1 *Assuming that the thread with index t performs access `a1` first and access `a2` second, `a2` is dependent on `a1` if the range of $h_{a2}(t)$ is not fully included in the range of $h_{a1}(t)$ for some value of t.*

If `a2` is dependent on `a1`, `a2` accesses values that were produced or consumed during the access `a1` by threads other than t and thus synchronisation is required. For example, if access `a1` is a write, checking `a2` range inclusion in that of `a1` means answering the following question: *within access `a2`, does thread t access only the data it produces itself within access `a1`?*

The LIFT compiler checks access range inclusion by analysing the ranges of the arithmetic expressions of $h_{a1}(t)$ and $h_{a2}(t)$. Due to the complexity of array index expressions involved in optimised direct convolution, range analysis is costly in terms of the overall compilation time. To avoid this overhead, LIFT performs range analysis only if it fails to prove access *independence* using cheaper checks. These checks focus on access types (read, write), views and parallel mapping.

The **access type**-based check is trivial: if both accesses are reads, no synchronisation is required and further checks are skipped.

$$
\begin{array}{l}
a, b : \textit{memory accesses}, \\
\textit{Iterators}_a : [i_0, i_1, .., i_n], \\
\textit{Iterators}_b : [j_0, j_1, .., j_n], \\
\forall i_k \in \textit{Iterators}_a, \forall j_k \in \textit{Iterators}_b, \\
i_k.min = j_k.min,\ i_k.max = j_k.max, \\
i_k.parallelDomain = j_k.parallelDomain \\
\hline
\therefore \textit{parallelMapping of } a = \textit{parallelMapping of } b
\end{array}
$$

Definition 5.4.2: Equality definition of parallel mappings of accesses a and b. The accesses are represented by the definitions of the loop counters used in the array index expressions of the two accesses. Symbol $\therefore$ means "therefore".

Using the **views** of the two accesses, the compiler detects the absence of data layout transformation. Since view trees capture the entire history of data layout transformations, the absence of the following primitives in the view tree guarantees that the data layout was not changed: **`transpose`**, **`split`**, **`slide`**, **`join`**, **`pad`**, **`asVector`**. If the data layout was changed, index range analysis is required to determine data dependency. If the data layout is preserved, the data dependency might still be presented depending on the parallel mappings of memory accesses.

The comparison of **parallel mappings** of two accesses is straightforward. For each of the accesses, the compiler collects the iterator variables of **`map`**s that traverse the array. For example, an expression such as **`mapSeq`**$_A$ (**`mapLcl`**$_B$ `(d)` (**`mapSeq`**$_C$ `(f))` produces the following list of iterators: `[i`$_A$`, local_index_d`$_B$`, i`$_C$`]`, where `local_index_d`$_B$ is the index of the local thread in dimension d. Since each iterator variable contains both the range information and the order of traversal (sequential and parallel), the full list of iterators of an access captures its entire parallel mapping. The compiler considers two accesses to have the same parallel mapping if they have the same number of iterators, and corresponding iterators have the same ranges. The full definition of parallel mapping equality for accesses is provided in Definition 5.4.2.

5.5 Evaluation

This section evaluates the proposed parallelism mapping technique and the barrier insertion method. First, it looks into the performance and memory consumption of the best parallel mappings found through uniformly random exploration of the constraint space. It then provides the analysis of the generated constraints, compares the search efficiency to the manual and randomised methods and analyses the sequential **`map`** fusion heuristic. Finally, the barrier insertion method is qualitatively evaluated.

5.5.1 Experimental Methodology

Convolutional layers of the VGG-16 are expressed in LIFT. LIFT is compared against TVM and the ARM Compute Library on the HiSilicon Kirin 970 SoC embedded GPU (ARM Mali-G72 with 12 cores) using Debian GNU/Linux 9.8. GPU frequency is fixed to 767MHz, the highest level. Each inference is performed over one image, which is common for streaming applications. All three frameworks produce specialised OpenCL code to run on the GPU. The search and LIFT compilation are timed on an Intel Xeon E5-2630v3 with 8GB RAM.

Lift Convolution is automatically parallelised by constructing a search space through constraint generation using the Choco-solver library [Pru16] v4.10.1, which is a good fit since it supports nonlinear integer constraints. Choco-solver is used in this work as a replacement for the custom constraint solver implemented for the auto-tuning experiments in Chapter 4 due the increased complexity of the parallelisation search space compared to that of the tuning space. Through optimised constraint propagation techniques, Choco-solver delivers faster constraint satisfaction time. Since the search is uniformly random and the constraint DSL is solver-agnostic, the change of the solver does not affect the search strategy and the results reported further in this section.

Values of the tuning parameters such as tile sizes and thread configuration are then chosen heuristically to saturate compute cores and registers. The parallelised expression is vectorised wherever possible by analysing array indices. Each low-level expression is compiled into a C++ host code and a set of OpenCL kernels. The best candidate is chosen through randomised exploration based on time and memory consumption measurements.

For the run time measurements, a custom OpenCL profiler is used — a wrapper that intercepts `cl_event` instances raised on the start and finish of the OpenCL kernel

executions. The timings collected include the time spent on padding and depadding. Each candidate is run 3 times and the median value is reported. In all sets of 3 runs, the first run suffers from the warm-up overhead. In the best-performing candidates across all VGG-16 layers, the relative standard deviation (RSD) of 3 runs is 2.66%, while the RSD across 2 runs without the first warm-up run is 1.12%. Functional correctness is verified by comparing the LIFT kernel outputs against those of the PyTorch-generated solutions.

The parallel mapping search times include the penalty of evaluating the candidates that satisfy the constraints but fail the extra ad-hoc checks. LIFT compiler memory allocation reports are used to calculate the exact memory consumption of the generated programs.

ARM Compute Library ARM Compute Library (v19.02) is used to produce OpenCL implementations of VGG-16 convolutional layers, configured using the ARM Compute Graph API. The implementations are tuned using the ARM Compute auto-tuner to ensure a fair comparison. The library produces both direct and GEMM-based implementations.

The performance is profiled by intercepting the OpenCL events in the same way the LIFT kernels are profiled. Memory consumption is calculated manually based on the stencil and GEMM algorithms. The profiler does not depend on any modifications to the ARM Compute Library or its generated implementations and does not influence the measured run time.

TVM TVM is chosen as the comparison code generator since it generally offers better performance than competing frameworks such as nGraph, Glow, XLA on CPU and GPU across variants of ResNet, VGG, MobileNet, MNASNet networks [Li20]. TVM v0.6 is used, built with OpenCL support generated using LLVM version 4.0.0. For a fair comparison, the Winograd strategy is disabled in the Python wrapper of TVM. Winograd implementations incur a loss of precision and are therefore not semantically equivalent to convolution, while LIFT preserves the semantics of the original problem. TVM is set to use its preferred convolution method for the Mali GPU: Spatial Pack Convolution [Zhe18], which applies im2col and GEMM on the tiled input.

The TVM auto-tuner, with the GATuner, explores 1000 candidates discarding slower implementations that take longer than 100 ms to finish. A median of 30 trials per candidate is recorded. Convolutional layers are constructed using the `Relay` operation,

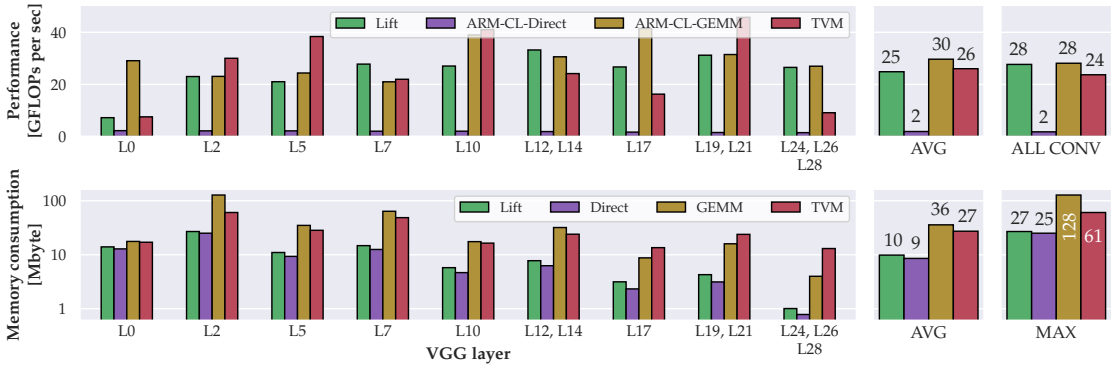


Figure 5.4: Performance and memory consumption comparison of LIFT-generated direct convolution implementations versus the direct and GEMM-based convolution methods in the ARM Compute Library and TVM-generated kernels on the convolutional layers of VGG-16 [Sim14].

`conv2d`, with all weights and input being represented in `float32` format. TVM compiler optimisation is set to level 3.

TVM auto-tuner reports are used to collect run time measurements; memory consumption is calculated based on the intercepted OpenCL memory allocation calls.

5.5.2 Results

Performance and memory consumption of the best implementations are provided in Figure 5.4. The performance of each implementation is measured in Floating-point operations s (FLOPs) per second. To determine the performance of each framework on a given layer, the theoretical number of FLOPs required by the layer configuration is divided by the time spent computing layer outputs. When calculating FLOPs per layer, Multiply-accumulate operation is counted as two separate operations. Compared with the pure run time, the chosen metric is normalised across layers of varying sizes.

Figure 5.4 also provides average per-layer performance and memory consumption. Performance across all convolutional layers (ALL CONV) takes into account duplicate layer configurations; it is calculated by dividing the total number of FLOPs of all convolutional layers by the total run time. The end-to-end run time of VGG can be inferred from ALL CONV since most of the time is spent computing convolution.

On average, LIFT is 17% slower than ARM Compute GEMM method while consuming 3.6× less memory. Across the whole VGG-16, LIFT is on par with ARM Compute GEMM; the maximum memory consumption is 4.7× smaller. LIFT is on par

Table 5.4: Breakdown of parallelisation constraints generated by the LIFT from the mid-level convolution lambda.

Constraint	# of instances
Private memory	32
Shared memory	13
Duplicate scheduling	34
Local thread indexing dimensionality	1
Local thread indexing hierarchy	34
Exhaustive thread indexing	9

with TVM on the average layer and is slightly faster across the whole VGG-16. Average and maximum memory consumption is 2.7× and 2.3× better respectively. Compared to the theoretical minimum memory footprint achieved by the direct method, LIFT requires only 1 Mbyte more on average.

5.5.3 Parallelisation Analysis

LIFT generated 123 parallelisation constraints from the mid-level lambda; the breakdown of the constraint instances is provided in Table 5.4. Of those, most constraints were generated to prevent naive scheduling mistakes such as duplicate scheduling and wrong thread indexing hierarchy. 32 constraints were required to enforce private memory scoping since the mid-level lambda was optimised to use the register memory as much as possible.

Out of 81,000 generated candidates satisfying all constraints, 5% passed the extra compiler checks; 33% were not synchronisable and 62% used too much memory. This suggests that further effort is warranted to model these compiler checks as constraints to accelerate the search further.

5.5.4 Exploration Efficiency

We now turn our attention to the efficiency of the exploration, *i.e.*, the amount of time it takes to find high-performance parallel mappings. Figure 5.5 shows the best throughput achieved as a function of exploration time for the heuristic-based manual approach [Mog20] and the automatic approaches. The constraint solver-based approach outperforms the manual approach after just 88 seconds and reaches peak

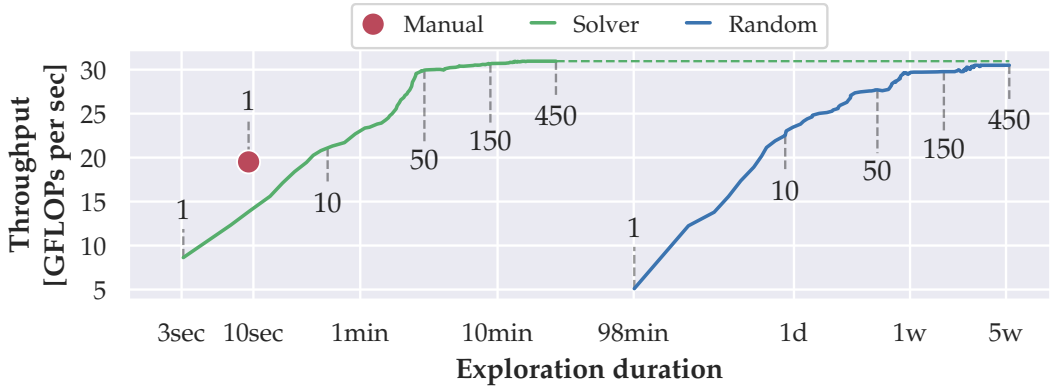


Figure 5.5: Exploration efficiency of three parallelisation approaches: manual, constraint solver-based and random for VGG-16 layers 19 and 21. The curves are annotated with the numbers of evaluated candidates; the horizontal dotted line shows the projected best throughput.

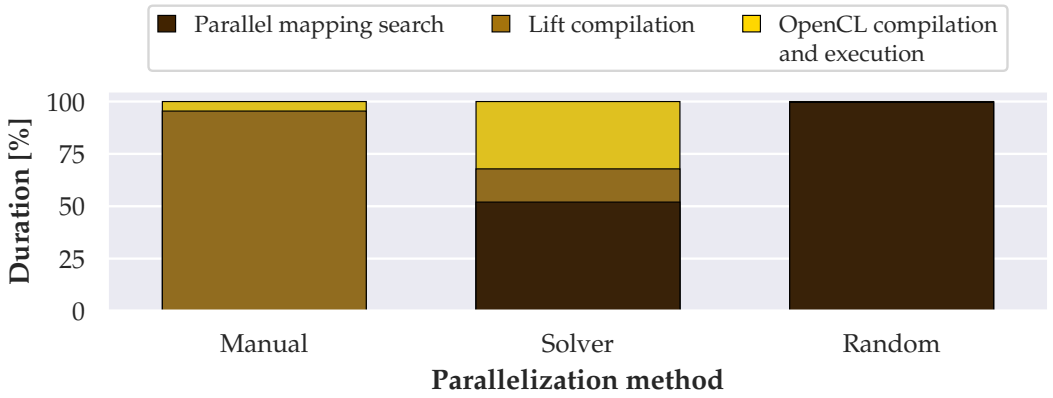


Figure 5.6: Breakdown of time spent in each stage of three parallelisation approaches: manual, constraint solver-based and random for VGG-16 layers 19 and 21.

performance after 95 minutes. With the random approach, only 1 out of 49,000 generated candidates satisfies the constraints. In the 98 minutes it takes to produce 1 random valid candidate (with poor performance), the solver-based method already finishes its search achieving the highest throughput.

The exploration time breakdown shown in Figure 5.6 suggests that the solver-based approach spends half the time searching for valid parallel mappings. This increases the time it takes to generate one valid implementation at least twofold compared to the manual approach. The solver-based approach also takes more time in the execution phase due to having to evaluate slow implementations. However, the manual approach requires more time and human expertise to pick a valid parallel mapping.

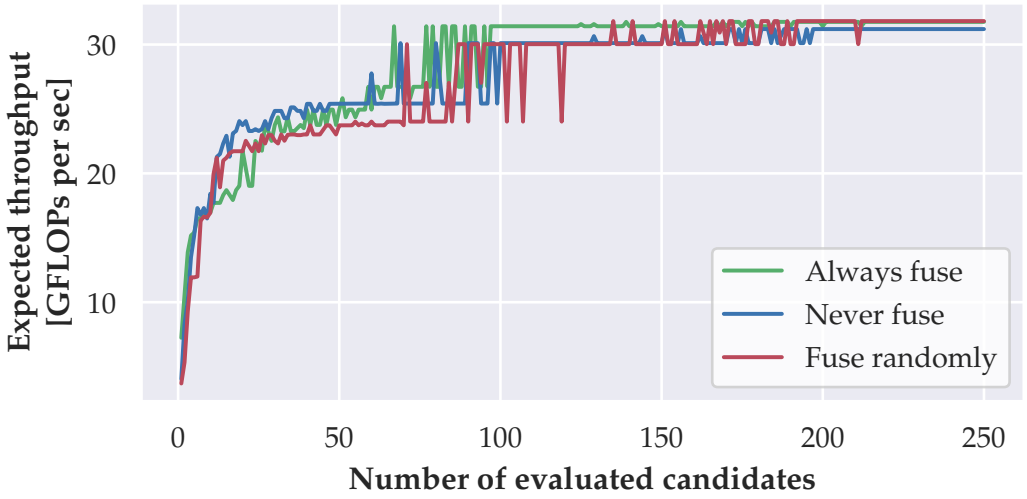


Figure 5.7: Throughput expectancy per number of evaluated candidates with different sequential `map` fusion strategies for VGG-16 convolution layers 19 and 21.

The time spent by the random approach is dominated by the search phase due to the naive search strategy. Both the naive and the solver-based approaches avoid spending time compiling and executing invalid implementations thanks to the parallelisation constraints. Despite that and the randomised search in both cases, the solver-based method iteratively reduces the search space during constraint satisfaction through constraint propagation. Parts of the search space are discarded quickly and less time is spent evaluating invalid candidates. The exploration time could be reduced further by adding more constraints expressing parallelisation heuristics, synchronisability and memory consumption restrictions, and an objective function.

5.5.5 Sequential Map Fusion

Throughput expectancy depending on search duration is shown on Figure 5.7 for three sequential `map` fusion strategies. Expectancy after evaluating N candidates is calculated using a data set of 250 candidates per strategy by uniformly sampling N candidates 100 times and taking a median of 100 maximum throughputs sampled.

The results indicate that always fusing sequential maps does not reduce maximum throughput. The strategy yielded best throughput on VGG-16 layers 19 and 21 after evaluating just 70 candidates compared to 140 for randomised `map` fusion. The difference might be explained by the reduction of search space.

mapSeq(**mapSeq**(g) ∘ **mapLcl**(0)(f)) << (x: $[[T]_M]_N$)

```
1  for (int i = 0; i < N; i++) {
2    for (int j = get_local_id(0); j < M; j += get_local_size(0)) {
3      buf[j + i * M] = f(input[j + i * M]);
4    }
5    barrier(CLK_GLOBAL_MEM_FENCE);
6    for (int j = 0; j < M; j++) {
7      output[j + i * M] = g(buf[j + i * M]);
8    }
9     + barrier(CLK_GLOBAL_MEM_FENCE); }
```

Listing 5.3: A LIFT example where the pattern matching-based method misses a required barrier on line 9.

5.5.6 Barrier Insertion

This section provides a qualitative evaluation of the proposed MAG-based barrier insertion method by illustrating the limitations of the original pattern matching-based approach that the proposed method addresses. Listings 5.3 to 5.5 provide examples where the barriers inserted by the original method are insufficient, unreachable or unnecessary due to missing or conservative IR patterns.

In Listing 5.3, the data dependency between the calls of `f` and `g` requires two barriers: one on line 5 to ensure that all the data read by `g` is produced before reading, and another on line 9 to ensure that `g` has consumed all data before overwriting the buffer. The current barrier placement approach places the barrier correctly on line 5 since the expression matches the pattern on line 6 of Algorithm 1. The method does not detect the write-after-read dependency between the calls of `g` and `f`.

Listing 5.4 is an example of an unreachable barrier. Since in this example N is not a multiple of the work group size in the dimension 1, threads enter the loop on line 1 a different number of times. Placing a barrier inside a loop with a variable number of iterations causes threads that do reach the barrier to wait indefinitely. Both the current and the proposed barrier placement methods detect non-synchronisable expressions by checking the number of iterations of each **mapLcl** containing a barrier.

Listing 5.5 is an example where an unnecessary barrier is placed due to the pattern on line 2 of Algorithm 1. Even though `f` and `g` share a buffer, the two data access patterns are such that there is no data dependency.

In Listing 5.6, the current approach detects that the barrier is needed between `f` and `g`, but chooses an inefficient barrier placement. The same data dependency can be respected by placing a barrier on line 7, resulting in fewer barrier hits. Identifying a better placement requires considering the wider context of a parallel **`map`** – *e.g.*, enclosing sequential loops produced by any of multiple functional primitives – which is non-trivial in a IR pattern matching-based approach.

5.6 Summary

As seen, the LIFT IR exposes plenty of parallelism. This chapter demonstrates how this parallelism is exploited safely by auto-generating parallelisation constraints from the algorithmic representation of the program. Arithmetic constraints capture a parallel programming model such as OpenCL concisely: 10 types of constraints suffice to exclude invalid parallel mappings. However, the overall number of generated constraints – 123 instances – highlights the difficulty of mapping onto the scheduling and memory hierarchies of a GPU.

This chapter also demonstrated a more exhaustive approach to synchronisation barrier insertion. Instead of relying on enumerating functional patterns which require synchronisation, the proposed method finds all memory accesses in the program, and determines synchronisability based on the control flow between interdependent accesses.

Although the proposed parallelism mapping technique still depends on an ad-hoc check of kernel synchronisability, the overall approach is still fully automated. The barrier insertion method could be integrated into the constraint solver to detect non-synchronisable mappings even earlier.

The results presented in this chapter are heavily dependent on algorithmic and hardware-specific optimisations. Tiling, data reuse and cache optimisations are not achievable through parallel mapping space exploration alone; the high-level expression needs to be lowered to express higher degrees of loop nesting and reordered memory accesses. The next chapter discusses how these radical structural transformations are achieved in LIFT, bridging the gap between high-level problem specification and an optimised hardware-specific kernel.

mapLcl(1)(**mapLcl**(0)(**mapSeq**(g)) ∘
mapSeq(**mapLcl**(0)(f))
) << (x: $[[[T]_K]_M]_N$), where N % get_local_size(1) $\neq$ 0

```
1  for (int i = get_local_id(1); i < N; i += get_local_size(1)) {
2    for (int j = get_local_id(0); j < M; j += get_local_size(0)) {
3      for (int l = 0; l < K; l++) {
4        buf[l + j * K + i * K * M] = f(input[l + j * K + i * K * M]);
5      }
6    }
7     - barrier(CLK_GLOBAL_MEM_FENCE);
8    for (int j = 0; j < M; j++) {
9      for (int l = get_local_id(0); l < K; l += get_local_size(0)) {
10       output[l + j * K + i * K * M] = g(buf[l + j * K + i * K * M]); }}}
```

Listing 5.4: A LIFT example with an unreachable barrier on line 7.

mapSeq(**mapLcl**(0)(g)) ∘
mapSeq(**mapLcl**(0)(**toLocal**(f))) << (x: $[[T]_M]_N$)

```
1  for (int i = 0; i < N; i++) {
2    for (int j = get_local_id(0); j < M; j += get_local_size(0)) {
3      local_buf[j + i * M] = f(x[j + i * M]);
4    }
5     - barrier(CLK_LOCAL_MEM_FENCE);
6  }
7  for (int i = 0; i < N; i++) {
8    for (int j = get_local_id(0); j < M; j += get_local_size(0)) {
9      output[j + i * M] = g(local_buf[j + i * M]);  }}
```

Listing 5.5: A LIFT example with two loop nests, where the pattern matching-based method inserts an unnecessary barrier.

mapLcl(0)(**mapSeq**(g)) ∘
mapSeq(**mapLcl**(0)(**toLocal**(f))) << (x: $[[T]_M]_N$)

```
1  for (int i = 0; i < N; i++) {
2    for (int j = get_local_id(0); j < M; j += get_local_size(0)) {
3      local_buf[j + i * M] = f(x[j + i * M]);
4    }
5     - barrier(CLK_LOCAL_MEM_FENCE);
6  }
7  + barrier(CLK_LOCAL_MEM_FENCE);
8  for (int i = get_local_id(0); i < N; i += get_local_size(0)) {
9    for (int j = 0; j < M; j++) {
10     output[j + i * M] = g(local_buf[j + i * M]); }}
11 barrier(CLK_LOCAL_MEM_FENCE);
```

Listing 5.6: A LIFT example with two loop nests with a data dependency, where the pattern matching-based method chooses an inefficient barrier placement on line 5.

Chapter 6

Towards Guided Rewriting

This chapter tackles the problem of balancing user-driven and explorative code generation through Rewrite points. An RP is a programming abstraction used to perform local macro-optimisations of a given expression. Annotating an expression with RPs does not fully define the rewritten expression, since each RP exposes design choices for exploration as parameters. RP parameters create a design space; user-defined RP placements truncate the design space. This chapter shows that these strongly typed pragmas encode a diverse design space without burdening the user with too many implementational details.

6.1 Introduction

We have seen in the previous chapters how a functional IR is leveraged to reduce the gap between an algorithm specification and a custom-tailored implementation. The functional paradigm is well-suited to expose parallelism and implement low-level optimisation methods, with expressive type and view systems preserving the original semantics of the problem specification. The rich algorithmic representation is leveraged to infer arithmetic tuning constraints and tune a low-level expression for a given platform. A large space of valid parallel mappings is constructed automatically for a specific AST, given a well-tiled expression.

This chapter furthers the separation of concern by raising the level of IR abstraction. A performance-portable code generator should create new parallelisation opportunities and optimise memory usage and access patterns. Previous work discusses how dozens of fine-grained generic rewrite rules are combined to transform abstract concise expressions into highly optimised low-level expressions. The rewrite rule-based

approach has been shown to match and outperform state-of-the-art solutions in several domains [Hag18; Piz19; Ste15]. Although this approach eventually finds good implementations, it poses the problem of long exploration times.

This chapter describes the first steps towards a guided rewriting approach combining human expertise and automated exploration. This approach frees the user from encoding low-level optimisations by offering high-level parametric abstractions – Rewrite points (RPs) – representing classical optimisations. The RPs are applied automatically by the compiler to transform parts of the input LIFT program to equivalent expressions. The structure of the transformed expressions is determined by the RP parameters, which can be explored automatically.

The RP parameters expose high-level structural design choices. Since they can be explored automatically, the user does not fully define the final expression. Instead, the user guides the compiler through the design space in broad strokes, leading the search to the subspaces with good implementations. The manual process of annotating the LIFT program with RPs can be considered meta-programming, and RPs themselves – optimisational blueprints.

The RP-based approach spreads the optimisational effort among the user and the compiler engineer. The user encodes high-level optimisation heuristics by annotating a program with RPs. The compiler engineer designs the RPs, aided by the high-level IR of LIFT, where most RPs act on the algorithmic representation and are reusable across applications and platforms. Since the RPs are part of the LIFT IR, they are type-safe and have no side effects, which makes it easier for the user to apply them safely. Incorrectly applied RPs do not pattern-match and get eliminated; the user is notified when the application fails to help improve the rewriting process.

We will see an example where a single RP parameter captures the choice between two different convolution algorithms. The same set of RPs is shown to optimise each implementation differently achieving multiple levels of tiling, data reuse, heterogeneous memory exploitation and memory access pattern optimisation. This diversity is enabled by giving the compiler a degree of freedom in choosing among the alternative designs. The RP application is both optional and parametric.

The focus of this chapter is on expressivity: how can an IR encode a large design space concisely? The evaluation shows that the RP-based approach produces high-performance implementations for two different convolution algorithms from the same annotated high-level LIFT expression. The results indicate that future work could leverage RPs to achieve semi-automated exploration guided by human expertise.

This work follows in the footsteps of PetaBricks [Pho13], which pioneered exposing alternative implementations to the compiler. Compared to the schedules and the Halide IR in TVM [Che18a], as well as the ELEVATE strategies and the RISE IR [Hag20b; Hag20a], the LIFT IR is used to both specify the algorithm and encode the design space. Like Tangram codelets [Cha16; De 19], LIFT RPs may insert other RPs into the transformed expression. This composability comes naturally because RPs are first-class citizens of the LIFT IR. Compound RPs lead to long rewriting sequences expressing macro-optimisations, which would otherwise take a long time to discover through the rewrite rule-based approach.

RPs are completely decoupled from code generation since RP application is a source-to-source translation within the same LIFT IR. By the time compilation reaches code generation, the RP nodes are eliminated from the program. Such an approach results in an extensible compiler, where adding support for new platforms and optimisations is possible at a low cost since the code generation functionality is reused.

This chapter focuses on understanding how good design choices are encoded directly in the IR. This work shows that an RP-annotated high-level expression can express a large design space that includes high-performance direct and GEMM-based convolution implementations without changing the original semantics of the expression. The modular design produces complex optimisations by reusing a small set of rewrite points. Given well-chosen parameter values, the compiler automatically rewrites the expression into the programs matching the performance of a vendor-provided handwritten kernel library and a state-of-the-art code generator. Since the RP mechanism is automatic, the same implementations can be found through automatic exploration.

The main contributions of this chapter are following:

- User-guided parametric rewriting mechanism as a way of injecting loosely-defined heuristics into the search.

- Expressing a design space through RPs that includes two convolution methods, tiling, prefetching, data reuse, memory optimisation, access coalescing and OpenCL kernel fission.

- Extension of the tuning mechanism, in which new tuning opportunities are created by the constraint solver through rewrite points.

$$\texttt{RewritePoint} : (inputFun : T \Rightarrow U,\ x : T) \Rightarrow U$$

Figure 6.1: The type signature of a rewrite point. The blue background is used henceforth to highlight rewrite points.

6.2 Rewrite Points

The system of rewrite rules introduced by LIFT produces very large design spaces which are difficult to explore. Although the compiler eventually achieves non-trivial optimisations that are well-tailored to the target platform, it needs a long time to discover common optimisations that take little effort for the user to define manually: tiling, prefetching and coalescing, etc. This chapter suggests a compromise approach, wherein the user guides the rewriting process by annotating the input expression with predefined RPs, which are used by the compiler to identify the subexpressions to rewrite. RP definitions are partially relaxed: parts of the transformed expressions are defined by rewrite parameters that can be explored automatically by the compiler. The resulting expression is defined both by the manual placement of the RPs, and through the automatic exploration of the RP parameters.

RPs are first-class citizens of the LIFT IR, provided as an extensible standalone library. RPs can be considered strongly typed pragmas. From the compiler's perspective, an RP is an ordinary IR node whose type is fully defined by its input LIFT expression. The code generator is agnostic of RPs since they are either replaced with the transformed versions of their input functions or eliminated during rewriting. Although RP transformations are more coarse-grained than the regular rewrite rules [Ste15], we will later see that the RPs are also generic and reusable. They are similarly composable since an RP can annotate the transformed expression with other RPs.

RPs are a flexible tool to calibrate the ratio of human expertise to automatic exploration in the optimisation process according to the complexity of the input expression and the target platform. The search space of the fully automated blind rewriting is first reduced through the manual placement of RPs in the input expression. Defining heuristic constraints on the RP parameters truncates the space further.

The next sections describe the interface of RPs and their application process carried out by the compiler during rewriting. Section 6.2.4 uses convolution as a case study and shows an example annotation of the high-level convolution expression from Chapters 4 and 5. Section 6.2.5 defines eleven RPs that are sufficient to express both

```
1  interface RewritePoint {
2    skip: Bool; structParams: List⟨V⟩; tuningParams: List⟨I⟩
3    pattern: T ⇒ U
4    def rewrite( inputFunComponents ): T ⇒ U
5
6    def apply( inputFun: T ⇒ U ): T ⇒ U = {
7      if (skip) return inputFun
8      else
9        switch( inputFun ) {
10         pattern:
11           return rewrite( pattern.decompose( inputFun ) )
12         nestedRP( nestedFun ):
13           return nestedRP( apply( nestedFun ) )
14         otherwise:
15           return inputFun } }}
```

Listing 6.1: The rewrite point interface. Formatting differentiates the implementation language of the LIFT compiler and the LIFT IR. The former is highlighted using the yellow background and the slanted font; the latter uses the white background. Blue colour highlights rewrite points. `Bool` represents the Boolean type in the compiler IR.

the optimisations implemented manually in Chapter 4, and to optimise the expression further.

6.2.1 Definition

The type signature of a rewrite point is shown in Figure 6.1. An RP is a higher-order identity function which applies a given function on an argument. During the RP application, the RP is replaced with a transformed version of the input function; the return type is preserved.

In the Object-oriented programming (OOP) terms, `RewritePoint` is implemented as an interface encapsulating common functionality shared by all RPs. The classes extending `RewritePoint` define specific macro-transformations and the patterns where they are legal. An instance of the RP class extending the `RewritePoint` interface represents an IR primitive.

The full RP interface is shown in Listing 6.1. From here on, two IRs are used to define rewrite points: that of the LIFT compiler (highlighted in yellow), and the LIFT language. The compiler IR presented in this thesis is a pseudocode based on

the Scala programming language, which is the implementation language of the LIFT compiler. The white background denotes LIFT expressions; the blue background is used to highlight rewrite points.

Each class of RP extends the `RewritePoint` interface, overriding all members except for `apply`. The `apply` function is invoked by the compiler during the top-down rewriting process explained in Section 6.2.3. `apply` replaces the RP instance with either the rewritten version of the input function on line 11, or the unchanged `inputFun` if it doesn't match `pattern` (line 15). The pattern (Line 3) is defined by each RP class on LIFT IR primitives and types. The `apply` function checks whether the input function matches the pattern using the switch case on line 10. Type equality of the input and return functions of `apply` on line 6 reflects the semantics-preserving property of RPs.

Each RP class defines two types of parameters: structural and tuning. Structural parameters affect the AST transformation performed by `rewrite`; tuning parameters are used as arguments to the IR nodes in the rewritten expression. The first structural parameter that all RPs have is `skip`. It defines whether the RP is applied or eliminated. Structural parameter type `V` is defined on all types of the compiler IR. These are the types the compiler uses inside the `rewrite` function to choose the IR nodes and their positions in the transformed expression; compiler IR types are not part of the LIFT language. Examples of these types in Section 6.2.5 include LIFT address spaces, vectors of integers representing the order of transposed dimensions, and integers. The user can impose heuristic constraints on the structural parameter values to prune the search space.

Tuning parameters expose numerical attributes such as split size. They are injected into the transformed expression to be explored as per Chapter 4. Tuning parameters are restricted by the constraints inferred automatically from the AST as described in Chapter 4. A constraint solver is used to explore both structural and tuning parameters.

Function `rewrite` takes a variable number of parameters collectively referred to as `inputFunComponents` on line 4. These are the components of the input function that `rewrite` depends on to produce a new expression; they include both `inputFun` subexpressions and array sizes extracted from the types inside `inputFun`. Each RP class defines its own `inputFunComponents` that it depends on.

The components are extracted by the compiler function `decompose` from `inputFun` on line 11. The definition of `decompose` is implied here as a utility function that returns the subexpressions and types matched by named subpatterns within `pattern`. We will

see examples of such named subpatterns in Sections 6.2.3 and 6.2.5.

To sum up, the transformation performed by the `rewrite` function can be defined as rearrangement of the components of the input function according to the RP-specific implementation of `rewrite` and the chosen values of structural and tuning parameters.

6.2.2 Nesting

Multiple RPs can be nested around the same input function to apply multiple different transformations in sequence:

`rewritePointA( rewritePointB( f ) )` ⟼ `rewritePointB( f' )` ⟼ `f''`

RPs are nestable thanks to recursive pattern-matching on lines 12 and 13 of Listing 6.1; we will see on the example of convolution in Section 6.2.4 how RP nesting and composition allow designing complex chains of optimisations. During pattern-matching, line 12 captures directly nested rewrite points and line 13 invokes `apply` on the LIFT expression at the bottom of the RP nest. The rewritten expression is wrapped into the nested RPs again to be rewritten further as we will see in Section 6.2.3.1.

As rewriting progresses, nested RPs might become inapplicable and get eliminated since their transformed input function might not match their patterns. To avoid this, RPs can take control over the placement of the nested RPs. Instead of letting `apply` wrap the entire rewritten expression in the nested RPs on line 13, an RP class can capture the nested RPs explicitly within its `pattern` on line 3. Captured nested RPs can then be extracted using `pattern.decompose` on line 11 and provided to the `rewrite` function. The `rewrite` function on line 4 can use the captured RPs to wrap the part of the rewritten expression that is likely to match their patterns. Section 6.2.3.2 shows an example RP class which is expected to transform `inputFun` significantly; the example RP places the nested RPs around the subexpression that resembles the original pattern more than the entire rewritten function.

6.2.3 Application

An expression with RPs is rewritten in the top-down order, in which the RPs that wrap other RPs are applied before their nested RPs. The top-down order gives the outer RPs control over inner RP placement in the transformed expression to maximize the likelihood of inner RPs pattern-matching the transformed subexpression.

Algorithm 4: Recursive top-down rewriting of all rewrite points in a LIFT expression. **yield** indicates that the nested block returns a value of the expression at the end of the nested block.

```
1  function applyRPs(e : expression):
       input  : LIFT expression
       output: Rewritten expression with all RPs applied
2      switch e
3        case Literal | Param do return e
4        case f << args do
5          newArgs ← (foreach arg in args yield applyRPs(arg))
6          newF ← ( switch f
7            case rp: RewritePoint yield
8              valuesOfStructParams ← solve(rp.structParams)
9              newInputFun ← rp(valuesOfStructParams)
10                                .apply(rp.inputFun)
11             if newInputFun == rp.inputFun then rp.inputFun
12             else
13               TypeChecker(newInputFun << newArgs)
14               assert newInputFun.type == rp.inputFun.type
15               (newInputFun.params ⇒ applyRPs(newInputFun.body))
16           case fPattern: ( map | reduce | .. ) yield
17             fPattern.f.body = applyRPs(fPattern.f.body)
18             fPattern
19           otherwise yield f )
20         return newF << newArgs
```

6.2.3.1 Algorithm

Algorithm 4 outlines the recursive RP application process in pseudocode. For each function call matched on line 4, the algorithm is invoked recursively on all function call arguments on line 5. The function itself is rewritten on lines 6 to 19.

Function rewriting proceeds differently depending on the function. If the function is an RP, the RP is applied on its `inputFun` on line 10 using the structural parameter values picked by the constraint solver on line 8. Lines 11 to 14 ensure that the RP either does not pattern-match the input function, or produces a new function with the same type. Line 15 attempts to rewrite `newInputFun` again in case it contains nested RPs, and produces a new function. Lines 16 to 18 recursively rewrite the functions

```
1  splitJoinND ⟼ splitJoin1D | splitJoin2D | ..
2
3  splitJoin2D extends RewritePoint {
4    structParams = List()
5    tuningParams = List(tileSize0: Int, tileSize1: Int)
6    pattern = nestedRPs(map(f))
7    def rewrite( nestedRPs, f ) = {
8      return joinND_2 ∘ nestedRPs(map(map(map(f)))) ∘
9        splitND_2(tileSize0, tileSize1) }}
```

Listing 6.2: Split-join rewrite point definition.

inside functional patterns such as **map** and **reduce**.

6.2.3.2 Example

The rewriting process can be exemplified using the Split-join RP provided in Listing 6.2. This is an RP-based version of the Split-join rewrite rule defined in previous work [Ste15], which replaces **map**(f) with **join** ∘ **map**(**map**(f)) ∘ **split**(s). By virtue of splitting data and increasing the number of **map**s, this RP is useful for data tiling and creating parallelisation, vectorisation and coalescing opportunities.

The dimensionality of Split-join can be varied to achieve higher degrees of tiling and finer-grained parallelisation and coalescing. The choice can be either left to the user, or explored within a heuristic upper limit on the number of dimensions to prevent search space explosion. Listing 6.2 provides a 2D version as an example.

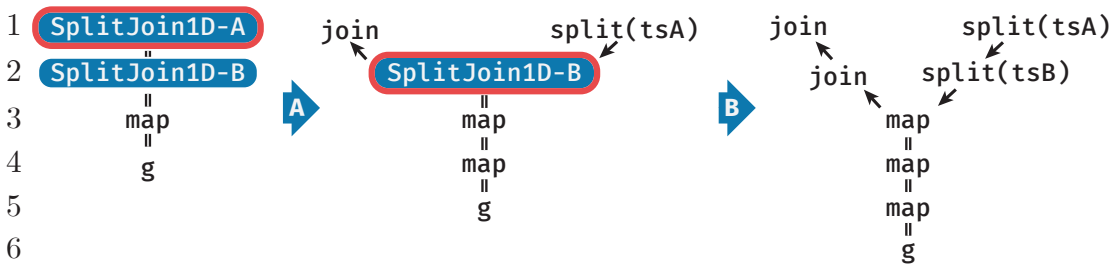
Line 6 defines the RP pattern as a **map** nested in zero or more nested RPs and wrapping some function `f`. Lines 7 to 9 produce a transformed version, in which the argument is tiled twice using the tuning parameters `tileSize0` and `tileSize1`. The original function `f` is wrapped in three **map**s to account for the extra two dimensions.

Function `f` is a component of the input function; it originates from `pattern` on line 6, where it is captured as a named subpattern. Listing 6.1 in Section 6.2.1 shows that the input function decomposition takes place in the `apply` function of the RP interface. Both `f` and `nestedRPs` are provided to `splitJoinND.rewrite` by `decompose` as arguments.

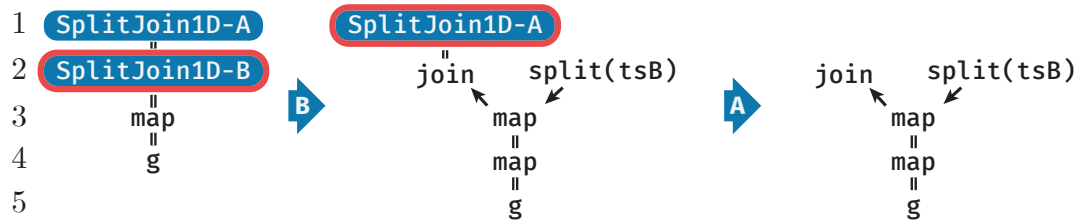
The new placement of `nestedRPs` on line 8 of Listing 6.2 reduces the disruptive effect of this RP application on pattern-matching within the nested RPs. Immediately nested RPs are applied on the same input function, so their patterns can be expected to

```
splitJoin1D_A( splitJoin1D_B( map(g) ) )
```

(a) Example input function.



(b) Top-down rewriting order.



(c) Bottom-up rewriting order.

Figure 6.2: Example rewrite point application sequences of a LIFT function in (a). Double lines denote nesting, black small arrows denote composition and blue large arrows denote rewrite point application. In the top-down rewriting order (b), `splitJoin1D-A` is expanded first, and `splitJoin1D-B` second. In the bottom-up rewriting order (c), `splitJoin1D-B` is expanded first and prevents `splitJoin1D-A` from pattern-matching.

be similar. Meanwhile, the Split-join RP replaces **map** with a composition of **joinND** and a **map** nest, which would prevent nested RPs expecting a **map** on top from pattern-matching the result of the transformation. To avoid this, the Split-join RP overrides the default behaviour of `RewritePoint.apply`: the RP places the nested RPs around the new **map** nest on line 8 of Listing 6.2 instead of placing the nested RPs around the entire transformed function on line 13 of Listing 6.1.

Figure 6.2 shows two example RP application sequences, which illustrate the advantage of top-down rewriting versus bottom-up rewriting. In the top-down order (b), `splitJoin1D-A` is applied first on the function inside `splitJoin1D-B`. **map**`(g)` is replaced with **join** `∘` **map**(**map**`(g)) ∘` **split**`(ts0)`. `splitJoin1D-B` is applied next: since it was placed around the **map** nest by `splitJoin1D-A`, the second pattern-matching is successful and the final expression is produced with a double-tiled **map**.

```
1  def conv( inputData     :  [[[float]_inCh]_inW]_inH,
2            kernelsWeights:  [[[[float]_inCh]_kerW]_kerH]_outCh,
3            padSize       :  (int,int,int,int),
4            kernelStride  :  (int,int) ) :  [[[float]_outCh]_outW]_outH =
5    toHost o oclKernel((slideWindows', kernelsWeights') ⇒
6      mapND_2(slideWin: [T]_(inCh * kerW * kerH) ⇒
7        map(singleK: [T]_(inCh * kerW * kerH) ⇒
8          reduce(0, +) o map(*) << zip(slideWin, singleK)
9        ) << kernelsWeights'
10     ) << slideWindows'
11   ) << ( mapND_2(joinND_2) o
12            slideND_2(kerH, kerW, kernelStride._1, kernelStride._2) o
13            padND_2(padSize, value = 0) o
14            toGPU << inputData,
15          map(joinND_2) o toGPU << kernelsWeights )
```

Listing 6.3: High-level LIFT expression of convolution. This expression is identical to the one in Listing 4.1.

In the bottom-up order (c) `splitJoin1D-B` is applied first. Since `splitJoin1D-B` has no control over the placement of its outer RP, `splitJoin1D-A` is placed around the entire transformed expression by `RewritePoint.applyRPs`. With **`join`** at the top of the transformed function, `splitJoin1D-A` does not pattern-match and gets eliminated, producing only a single-tiled function.

Next, we look at a real-life use case, where eleven RP classes are combined to define and curb design space of convolution implementations.

6.2.4 High-level Convolution, Annotated

Section 4.2.4 illustrates the complexity of the low-level LIFT implementation of convolution even with the final step of the reduction omitted. While the functional paradigm helps by abstracting the array index expressions away, LIFT's strict type checker justifiably makes it hard to express low-level state-of-the-art performance optimisation methods correctly. The automatic parallelisation approach discussed in Chapter 5 alleviates the concern for parallelisation correctness; however, abstracting away parallelisation does not reduce the size of the low-level expression in Listings 4.2 and 4.3.

Listing 6.4 illustrates how RPs abstract the optimisational details further from the

```
1  def conv( inputData      :  [[[float]_inChs]_inW]_inH,
2            kernelsWeights :  [[[[float]_inChs]_kerW]_kerH]_outChs,
3            padSize        :  (int,int,int,int),
4            kernelStride   :  (int,int) ) :  [[[float]_outChs]_outW]_outH =
5    toHost o
6    padExtra2D( kernelFission(
7      joinSplitND( padExtra1D( kernelFission( slideWindows' ⇒
8        kernelFission( kernelsWeights' ⇒
9          oclKernel((slideWindows'', kernelsWeights'') ⇒
10           abTile( abTile(
11             tileNestedMapReduce(reduceFun = +)(
12               tileNestedMapReduce(reduceFun = +)(
13                 abTile(
14                   mapND_2(slideWin: [T]_(inChs * kerW * kerH) ⇒
15                     map(singleK: [T]_(inChs * kerW * kerH) ⇒
16                       privatiseAccumulator(
17                         reduce(0, +) o map(*) << zip(slideWin, singleK) )
18                       ) << kernelsWeights''
19                     ) << slideWindows'' )))))
20           ) << (slideWindows', kernelsWeights')
21       ) o map(joinND_2) o toGPU << kernelsWeights
22     ))) o mapND_2(joinND_2) o
23           slideND_2(kerH, kerW, kernelStride._1, kernelStride._2)
24   )) o padND_2(padSize, value = 0) o toGPU << inputData
```

Listing 6.4: High-level LIFT expression of the convolutional layer with rewrite points.

user. The listing presents an expression which is semantically equivalent to the one in Listing 6.4. This expression has been manually annotated with RPs without changing the high-level definition of convolution. All the added nodes are RPs expressing a specific parametric optimisation strategy – one that automatically produces the low-level direct convolution expression in Section 4.2.4 and GEMM-based convolution. Each of the chosen RPs expresses one of the following:

- Optimisations beneficial to both direct and GEMM-based convolution, *e.g.*, tiling, accumulator privatisation and kernel fission.

- Optimisations beneficial to only one of the convolution methods, *e.g.*, 2D padding for direct convolution and 1D padding for GEMM-based convolution.

Table 6.1: RP instances in the high-level convolution expression in Listing 6.4 and their desired effects. Padding refers to optional zero-padding of data for optimisational purposes, not the padding required by the convolutional layer specification.

Line	Rewrite point instance	Desired application effect
6	`padExtra2D`	Input padding before sliding to expand tile size range
6	`kernelFission`	Input data layout optimisation before sliding
7	`joinSplitND`	Input flattening to simplify rewriting
7	`padExtra1D`	Input padding after sliding to expand tile size range
7	`kernelFission`	im2col-formatted input data layout optimisation
8	`kernelFission`	Weights data layout optimisation
10	`abTile` (outer)	Tiling for parallelisation; loop interchange
10	`abTile` (inner)	Tiling for cache locality
11	`tileNestedMapReduce`	Reduction tiling for parallelisation, thread coarsening
12	`tileNestedMapReduce`	Data reuse tuning
13	`abTile`	Tiled prefetching for data reuse
16	`privatiseAccumulator`	Private memory optimisation

- Expression differentiation into one of the two convolution methods: the materialisation of the slided windows produces the im2col stage of the GEMM-based method.

The RP annotations only add information to the system without changing the overall compilation flow. The expression is type-checked before and after RP application to ensure the preservation of convolution semantics.

The desired effects of all RPs in Listing 6.4 are summarised in Table 6.1; we will see complete definitions of the individual RPs in Section 6.2.5. Although the actual effect of the RP application is ultimately determined by its parameter values and the application of other RPs, the annotations of the high-level expression are designed to achieve most of these effects in the same candidate. Following the top-down rewriting order, the chosen strategy may proceed as follows.

6.2.4.1 Input Pre-Processing

The pre-processing stage opens the data layout up for exploration – array elements can be reordered in memory before the main computation begins. Combined with tiling, an optimal data layout improves cache locality and creates coalescing and vectorisation

opportunities.

This stage also determines the algorithm choice: direct or GEMM-based. The main difference between the two is the im2col operation, which materialises sliding windows in memory. In the convolution expression in Listing 6.4, im2col can be achieved by inserting an identity user function after the **slideND**$_2$ primitive on line 23. The extra copy materialises the slided view of the input data in memory, rendering the rest of the convolution expression an ordinary matrix multiplication.

The following two scenarios describe how rewriting proceeds for direct and GEMM-based convolutions.

Direct Convolution First, input data is zero-padded along all four edges on line 24 as required by the convolutional layer specification. The result is zero-padded again along the right and bottom edges by the `padExtra2D` RP on line 6. Extra padding allows more flexibility when tuning the tile sizes. Both padding operations are virtual since they are applied on the input data view by the **pad** primitive (see Section 2.3.1.2 for more details about **pad**). Instead of padding data in memory, the compiler uses the View System to alter the array access of the user function to simulate padding. In the generated OpenCL kernel, the access is wrapped in an if-conditional checking the access index expression: if a given access points into the virtual padding area, a constant value of zero is returned and no memory access is performed. Section 2.3.2.2 covers the View system in more detail.

On a GPU, branching control flow comes with a performance penalty, so extra optimisation effort is required to achieve efficient virtual padding. The `kernelFission` RP on line 6 reduces the overhead of branching by making a copy of the padded data efficiently in a separate OpenCL kernel. Optionally, it also transforms the data layout during the copy operation to improve reading efficiency during convolution.

The copy is passed to the **slideND**$_2$ primitive on line 23. After the spatial sliding, input data dimensionality can be reduced for easier rewriting. The 3D sliding windows are flattened on line 22 as per the original expression; the `joinSplitND` RP optionally flattens the 2D array of windows into a 1D array.

For direct convolution, the RPs `padExtra1D` and `kernelFission` on line 7 are not needed; since RP application is optional, we can expect these two RPs to get eliminated for the best direct convolution candidate.

im2col For GEMM-based convolution, copying pre-padded data before sliding is unnecessary since we expect a post-slide copy for the im2col operation. Since the application of each RP is optional, there are rewriting sequences where the first `kernelFission` (line 6) is eliminated, avoiding an extra copy.

The input data is pre-padded along the right and bottom edges on line 6; the result is passed directly to **slideND**$_2$ on line 23. Next, we perform finer-grained padding of the input data by adding values "at the bottom right corner". This is achieved by reducing the dimensionality of the slided array from five to two on lines 7 and 22, and padding the result using the `padExtra1D` RP on line 7. All of the above is performed in a separate OpenCL kernel thanks to the `kernelFission` RP on line 7. By materialising the padded and slided result at the end of the pre-processing stage in memory, `kernelFission` achieves the im2col operation.

6.2.4.2 Weights Pre-Processing

The `kernelFission` RP on line 8 improves weights data access patterns for better cache locality, and to create coalescing and vectorisation opportunities. It does so by reordering elements in the flattened weights array on line 21 and materialising results in memory within a separate OpenCL kernel.

6.2.4.3 Spatial Tiling

For both direct and GEMM-based convolution, the annotated expression creates three opportunities for spatial tiling: on lines 10 and 13. Several levels of tiling are performed to achieve several optimisations at once and to provide the subsequent parallelisation pass with enough loop nesting to match the deep scheduling hierarchy of a GPU. The `abTile` RPs tile both inputs and weights: as we will see in Section 6.2.5.2, the RP is defined on a nest of two **map**s traversing independent arrays `A` and `B`, and tiles both.

All three RPs increase **map** nesting to create new parallelisation opportunities. When the outermost **map**s get parallelised, the combined effect of the second `abTile` and the preprocessing RPs is improved cache locality. The innermost `abTile` improves data reuse: each pair of input and weight tiles is prefetched from global to private memory and reused across the iterations of loops generated by **mapND**$_2$ and **map** on lines 14 and 15 respectively.

6.2.4.4 Reduction Tiling

The two `tileNestedMapReduce` RPs on lines 11 and 12 tile the inputs and weights across the dimension being reduced, namely the combined spatial and input channel dimensions of the flattened sliding windows. Up to two extra **`reduce`** primitives are inserted, creating a reduction tree. When both RPs are applied, the outer one creates a parallelisation opportunity, wherein multiple chunks of a sliding window are reduced in parallel by different threads. Since a sliding window must be reduced to a single output, the results of the partial reduction of a sliding window must be aggregated by a single thread. We will see in Section 6.3 that such parallel mapping is indeed found for direct convolution. Using the synchronisation barrier insertion technique described in Section 5.4, the compiler places an OpenCL barrier between the two stages of reduction to ensure that all chunks of a sliding window are partially reduced before one of the threads aggregates them.

The inner `tileNestedMapReduce` adds an extra degree of tuning control over data reuse: the extra tile size controls the portion of each sliding window and convolutional kernel that is prefetched by `abTile` on line 13. This interaction with the inner `abTile` is made possible by defining the reduction tiling RP over a **`reduce`** nested inside **`map`**s. When defined over just **`reduce`** and placed inside the innermost `abTile`, the reduction tiling RP has no control over the total amount of data prefetched by `abTile` for each pair of input and weight tiles.

The tile sizes used by the `tileNestedMapReduce` RPs also affect memory consumption. Small reduction tile sizes achieve higher degrees of parallelism at the expense of the memory required to store the intermediate results. The RP exposes the tile sizes to the arithmetic constraint solver to allow finding the balance exploratively.

6.2.4.5 Private Memory Optimisation

The two RPs on line 16 optimise register usage by **`map`** and **`reduce`**.
`privatiseAccumulator` places the reduction accumulator in private memory to reduce the overhead of repeated accesses to the accumulator.

6.2.4.6 Post-processing

As we will see in Section 6.2.5.10, each instance of `padExtraND` introduces an extra OpenCL kernel at the end of the expression to remove the extra padding from the final result. When multiple instances of `padExtraND` are applied, some depadding OpenCL

```
1  joinSplitND extends RewritePoint {
2    structParams = List(); tuningParams = List()
3    pattern = nestedRPs(mapND_{||N||}(f)): [T]_N -> [U]_N
4    def rewrite( nestedRPs, f, N ) = {
5      return splitND_{||N||}(N) o nestedRPs(map(f)) o join_{||N||-1} }}
```

Listing 6.5: Join-split rewrite point definition.

kernels could be fused to reduce the memory access overheads. However, since only one `padExtraND` RP at a time is expected to benefit a given convolution algorithm, post-processing OpenCL kernel fusion is left for future work.

We now look at the definition of each rewrite point.

6.2.5 Expressing Optimisations Through Rewriting Points

6.2.5.1 Flattening

The opposite of the "tiling" rewrite point `splitJoin` in Listing 6.2 is the "flattening" RP `joinSplitND` in Listing 6.5. Because of its simple function, the RP uses no structural or tuning parameters beyond the `skip` parameter determining whether the whole RP is applied or eliminated.

As per line 3 of Listing 6.5, the RP is defined on an N-dimensional **`map`** nest. The rewritten function on line 5 first flattens the argument completely, then applies the original function `f` within one **`map`** and splits the flattened result to preserve the original return type.

Similarly to `splitJoin`, the inserted **`join`** and **`split`** primitives are expected to alter the overall pattern of the original function significantly. To preserve the applicability of the potential nested RPs, they are captured on line 3 and inserted around the **`map`** on line 5.

This trivial transformation simplifies the rest of the optimisation process. In convolution, `joinSplitND` abstracts away the algorithmic detail that the input has two spatial dimensions, highlighting the traversal of two independent arrays (inputs and weights) by the immediately nested **`map`**s. This simplifies `abTile` and `tileNestedMapReduce` RP design: their patterns do not need to detect and preserve the dimensionalities of the two arrays. Thanks to `joinSplitND`, each application of AB-tiling and nested **`map`**-reduce tiling produces one less **`map`**. Fewer **`map`**s produce fewer tuning constraints,

```
1  abTile extends RewritePoint {
2    structParams = List()
3    tuningParams = List(tileSizeA: Int, tileSizeB: Int)
4    pattern = (a, b) => nestedRPs( map(map(f) << b) << a )
5    def rewrite( nestedRPs, f ) = {
6      return (a, b) => map(join) o join o interchangeMaps(
7        map(tileA => let(tileACopy =>
8          map(tileB => let(tileBCopy =>
9            nestedRPs( map(map(f) << tileBCopy) << tileACopy )
10         ) o materialise(p => p) << tileB) o split(tileSizeB) << b
11       ) o materialise(p => p) << tileA) o split(tileSizeA) << a ) }}
```

Listing 6.6: AB-tiling rewrite point definition.

which simplifies constraint solving. Finally, tuning is also improved: since one tile size is used across several flattened dimensions, the total number of tuning parameters is smaller, and the number of valid tile sizes is increased.

6.2.5.2 AB-tiling

The AB-tiling RP is defined on a generic pattern, wherein some function `f` is applied on all combinations of elements from independent arrays A and B. The functional definition of this pattern is provided on line 4 of Listing 6.6. The `rewrite` function on line 5 performs four crucial transformations. Firstly, it tiles `a` and `b` independently on lines 10 and 11 and flattens the results back on line 6, not unlike the `splitJoin` RP. Secondly, the number of **map**s is doubled to iterate both across tiles and their elements; this creates new parallelisation opportunities. Thirdly, `abTile` optionally prefetches one tile of `a` and `b` each on lines 10 and 11 respectively using the `materialise` RP defined later in Section 6.2.5.5. Each pair of copies is reused inside the original **map** nest inserted on line 9; when copies are placed in a faster memory, access overhead is reduced. Finally, line 6 optionally interchanges the **map**s iterating over the tiles of `a` and `b` using the `interchangeMaps` RP. Section 6.2.5.7 shows how **map** interchange creates a new parallelisation opportunity.

The nested RPs captured on line 4 are inserted around the inner **map** nest on line 9. Such placement of `nestedRPs` allows nesting multiple instances of `abTile` and `tileNestedMapReduce` effectively.

The main purpose of the `abTile` RP is to exploit the opportunity for data reuse

```
1  reduce(0, f) ∘ map(g) ⟼
2  reduce(0, f') ∘ map( reduce(0, f) ∘ map(g) ) ∘ split(..)
```

Listing 6.7: Reduction tiling.

presented by the nested loops over a and b. While the tiling behaviour of abTile resembles that of two instances of splitJoin applied independently on a and b, the results are different. Applying splitJoin on **map**(g) << a would nest the **map** over elements of tileA immediately within the **map** over tiles of a. In abTile, however, it is sunk into the **map** over the tiles of b (line 8). This difference allows prefetching tileA before iterating over the tiles of b.

6.2.5.3 Nested Map-Reduce Tiling

While AB-tiling can be used to partition the independent dimensions of inputs and weights, further tiling is necessary within the scope of a sliding window, *i.e.*, across the reduced dimensions. The sliding window tends to have a wide input channel dimension – up to 512 elements in VGG. Due to the large total size of the sliding window, sequential reduction leads to under-saturation of the GPU cores, while prefetching the entire window results in register spilling. The compiler must be able to fetch one window tile at a time and reduce it independently from the other tiles.

Reduction Tiling As shown in Listing 6.7, the reduced dimension requires a different tiling approach than that of abTile due to the data dependencies of the reduction pattern. After tiling the argument, the original **map-reduce** is applied on each tile. Intermediate results are passed to the additional **reduce**, which produces the final results.

Composability With Prefetching Prefetching only part of a sliding window at a time requires that the prefetching RP (abTile) is applied on a single window tile. Consequently, window tiling must occur before prefetching; in the top-down rewriting order, this is achieved by nesting the prefetching abTile in the tileNestedMapReduce RP. Furthermore, for abTile to remain applicable after tiling reduction, the **map** nest over inputs and weights must be nested inside tileNestedMapReduce together with abTile as per the following pattern:

```
tileNestedMapReduce( abTile(
  map(aEl ⇒ map(bEl ⇒
    reduce(0, f) o map(g) << zip(aEl, bEl)) << b) << a ) )
```

Composability with prefetching motivates defining the tiling RP on **map-reduce** inside a **map** nest instead of just **map-reduce**. Although the latter is easier to pattern-match and transform, combining it with prefetching requires extra RPs and longer rewriting sequences. Coarse-grained RPs such as nested **map-reduce** tiling complement the fine-grained rewrite rules-based approach by truncating the design space more aggressively.

The Final Reduction Operator The final component of the reduction tiling mechanism is determining the final reduction operator. In Listing 6.7, `f'` is not the same function as `f`. Consider the following example:

```
reduce(0, (acc, x) ⇒ acc + c*x) o map(g) ⟼
reduce(0, +) o map( reduce(0, (acc, x) ⇒ acc + c*x) o map(g) ) o split(..)
```

Here, the original reduction operator – addition and multiplication – cannot be used in the final reduction lest the multiplication is performed too many times. Only part of the original function – addition – can be used as a final reduction operator.

Extracting the final reduction operator from the original reduction function in the general case is an open research problem. As a workaround, the `tileNestedMapReduce` RP shifts the burden of determining the operator onto the user. Specifically, the RP takes a LIFT function as an extra argument and inserts it in place of the final reduction operator. In convolution, where input elements are multiplied by weights and the results are added together, the final reduction operation is addition.

The Full Pattern The full definition of nested **map-reduce** tiling is provided in Listing 6.8. The pattern on lines 3 to 7 defines the RP on a function that takes two arguments with at least two dimensions each – an independent dimension and a reduced dimension. The **map** nest on line 4 applies the following on each row of `a` and `b`. Firstly, some function `g` is applied on each pair of corresponding elements of `rowA` and `rowB`. Then, the results of `g` are reduced to one value per row pair using some function `f` as a reduction operator. The entire pattern is bound to the compiler variable `matchedMapReduce` on line 3 to be reused within the rewritten function. The variable

```
1  tileNestedMapReduce( reduceFun: (S,S) ⇒ S ) extends RewritePoint {
2    structParams = List(); tuningParams = List(tileSize: Int)
3    pattern = matchedMapReduce @ ( nestedRPs(
4      (a: [[T]_K]_N, b: [[U]_K]_M) ⇒ map(rowA ⇒ map(rowB ⇒
5                                       reduce(f: ((S,V) ⇒ S)) o
6                                       map(g: ((T,U) ⇒ V)) << zip(rowA, rowB)
7                                     ) << b) << a ) )
8    def rewrite( matchedMapReduce, M ) = {
9      return ((a, b) ⇒ split(M) o
10       interchangeReduceND( privatiseAccumulator(
11         reduce(map( reduceFun ) o join)
12       )) o splitJoinND(map(tileAB ⇒ matchedMapReduce << tileAB)
13       ) << zip(transpose() o map(split(tileSize)) << a,
14                transpose() o map(split(tileSize)) << b)) }}
```

Listing 6.8: Nested Map-Reduce Tiling rewrite point definition. The asperand symbol (@) on line 3 denotes binding the matched expression to a variable. In this example, the entire matched pattern is bound to `matchedMapReduce`.

`matchedMapReduce` is provided to `rewrite` on line 8 by the `pattern.decompose` function discussed in Section 6.2.1.

Divide And Conquer The transformed expression on lines 8 to 14 uses the divide-and-conquer approach. First, `a` and `b` are tiled along the reduced dimension on lines 13 and 14 and transposed; the new types are $[[[T]_{tileSize}]_N]_{\frac{K}{tileSize}}$ and $[[[U]_{tileSize}]_M]_{\frac{K}{tileSize}}$, respectively. With the tiled dimension brought outwards through transposition, the conquer stage begins: the **map** on line 12 applies the original nested **map-reduce** on each tile separately. The intermediate results are passed to the final **reduce** on line 11. Within **reduce**, the two independent dimensions of `a` and `b` (of sizes N and M, respectively) are joined, and the final reduction operator provided by the user is applied. The independent dimensions are reconstructed using the **split** on line 9. The join-split of the last stage helps simplify further rewriting.

Further Rewriting `tileNestedMapReduce` inserts three new RPs to optimise the transformed expression further. The `splitJoinND` RP on line 12 tiles the **map** to create new parallelisation opportunities. `interchangeReduceND` on line 10 sinks the **reduce** primitive deeper into its inner **map** nest: first, it tiles the inner **map** nest to increase the

```
1  kernelFission extends RewritePoint {
2    structParams = List(); tuningParams = List()
3    pattern = f o g
4    def rewrite( f, g ) = {
5      return optimiseDataLayout( f o oclKernel(materialise( g ))) }}
```

Listing 6.9: Kernel Fission rewrite point definition.

number of **map**s, then, it interchanges **reduce** with one of the inner **map**s. This interchange creates new parallelisation opportunities by replacing the sequential **reduce** with a **map** that can be fused with a **map** on line 12. Finally, `privatiseAccumulator` on line 10 allocates the reduction accumulator in the faster private memory.

The nested RPs captured on line 3 are inserted into the transformed expression within `matchedMapReduce` on line 12. Since their input function does not change, the entire transformation does not affect their applicability. This allows nesting multiple instances of `tileNestedMapReduce` to produce a reduction tree of arbitrary depth.

Generalisability Although the presented RP is defined on a two argument-function, generalising the pattern and the transformation to a variable number of arguments and levels of **map** nesting is straightforward. Despite its coarse granularity, the overall functional pattern on lines 3 to 7 is encountered in many applications such as those depending on matrix multiplication.

6.2.5.4 Kernel Fission

Generally, fusing OpenCL kernels is beneficial due to reduced intermediate memory consumption and scheduling overheads. However, kernel fission can be an efficient synchronisation tool for multi-stage computation. It is preferable to the local synchronisation of OpenCL barriers when the two stages require significantly different work group configurations. Performing both stages with the same work group configuration results in idle work groups and undersaturated compute cores.

Thanks to LIFT's host code generation capabilities [Sto21], multi-kernel management is nearly transparent to the user. Assigning a subexpression to a separate OpenCL kernel is performed using the **oclKernel** primitive. **oclKernel** is a first-class citizen of the IR and is therefore rewritable – a property leveraged by the `kernelFission` RP defined in Listing 6.9. The RP is defined on the composition of two functions `f` and `g`

```
1  materialise extends RewritePoint {
2    structParams = List(targetMem: AddressSpace)
3    tuningParams = List()
4    pattern = f: T ⇒ [U]_N
5    def rewrite( f ) = {
6      return joinSplitND( splitJoinND( map(toMem(targetMem)(id)) )) o f }}
```

Listing 6.10: Materialisation rewrite point definition.

on line 3. The first function to be computed – `g` – is wrapped in **`oclKernel`** on line 5. The rewritten expression computes `g` in an OpenCL kernel which is separate from that of `f`; the result of `g` is written to memory using the `materialise` RP.

The `kernelFission` RP creates two new rewriting opportunities to optimise the transformed expression further. As we will see in Section 6.2.5.5, the `materialise` RP uses tiling to help parallelise and vectorise reading and writing. While `materialise` leaves the target address space up for exploration, the OpenCL programming model requires that the results of an OpenCL kernel are placed in the global memory to be accessible by other kernels. This requirement is enforced as a compiler check. It can also be enforced during the structural parameter exploration by inferring a constraint on the target memory choice.

The `optimiseDataLayout` RP reorders the results of `g` in memory for a more optimal reading pattern in `f`. We will see in Section 6.2.5.6 how the `optimiseDataLayout` RP performs reordering by inserting **`transposeWND`** to change the writing pattern of `g` and a **`transposeND`** to adapt the subsequent reading pattern of `f` to the transformed data layout in memory. This reordering of an OpenCL kernel outputs is a crucial part of the input and weights preprocessing in convolution since it improves cache locality and achieves access coalescing.

6.2.5.5 Materialisation

The RP in Listing 6.10 optionally inserts a write operation after the input function of any type with at least one output dimension. The target memory address space is chosen through a structural parameter. Application of this RP could benefit the implementation in three cases:

- When there is no preceding write operation, *i.e.*, when the input function and its argument only transform the virtual data layout (View) of the argument to the

entire LIFT program. The newly inserted write operation commits the accumulated virtual data layout transformations to memory. The new layout might yield a better reading pattern in the subsequent memory accesses.

- When the input function writes in a slow memory address space. Transferring data to a faster memory (*e.g.*, registers) – prefetching – improves overall performance if the data is accessed repeatedly later.

- When there is no subsequent write operation until the end of the OpenCL kernel. When the `targetMem` parameter is set to global memory, the outputs of the LIFT program are made available beyond the lifetime of the OpenCL kernel.

The RP is defined on a function `f` of any type with at least one output dimension on line 4 of Listing 6.10. On line 6, the output of `f` is copied using the scalar identity function `id`. The **`toMem`** macro expands to either **`toGlobal`**, **`toLocal`** or **`toPrivate`**. The choice of the target memory is predicated on the value of the `targetMem` parameter.

Even though the output of `f` can have any non-zero number of dimensions, the RP inserts only one **`map`** in the transformed expression. Such brevity is made possible by the `joinSplitND` RP, which flattens the output of `f` to one dimension, and restores the dimensionality afterwards. To create parallelisation and vectorisation opportunities, `splitJoinND` splits the argument and increases the depth of the **`map`** nest. The new number of dimensions and their sizes are both parametric within `splitJoinND`.

Discarding the original dimensionality of the output of `f` and splitting it anew gives the compiler a fine-grained control over parallelisation of the potentially expensive read and write operations of the identity function. Thanks to this design, the parallelisation opportunities are decoupled from the number and sizes of the original dimensions of the input data.

Prefetching a subset of data into a faster memory before reusing it is invaluable for a memory-bound problem such as convolution-based image recognition [Siu18]. Expressing the choice of the target memory as a parameter allows the constraint solver to explore the trade-off between memory optimisation and register spilling.

6.2.5.6 Data Layout Optimisation

An optimal memory access pattern depends on multiple factors, including parallelisation strategy, cache size and cache line size. The rewrite point in Listing 6.11 opens the space of memory layouts up for exploration by reordering the final write of the

```
1  optimiseDataLayout( n: Int ) extends RewritePoint {
2    structParams = List(newDimOrder: Vector_(n+1)⟨Int⟩)
3    tuningParams = List(factor1: Int, .., factorN: Int)
4    pattern = f ∘ (g: T ⇒ [U]_M⃗)
5    def rewrite( f, g, M⃗ ) = {
6      reorderOnWrite = transposeWND_n(newDimOrder) ∘
7                       splitND_n(factor1, .., factorN) ∘ joinND_(‖M⃗‖−1)
8      restoreOrderOnRead = splitND_‖M⃗‖(M⃗) ∘ joinND_n ∘
9                           transposeND_n(newDimOrder⁻¹)
10     return f ∘ restoreOrderOnRead ∘ reorderOnWrite ∘ g }}
```

Listing 6.11: Data Layout Optimisation rewrite point definition. Parameter `n` determines reordering granularity. `n` is chosen by the user.

first function in a pair of composed functions. The subsequent read is also reordered to achieve the original order with a new access pattern. The composite function on line 10 inserts the two reordering operations in-between two composed functions. In the simplest case, the transformation is following:

$$\texttt{f} \circ \texttt{g} \longmapsto \texttt{f} \circ \textbf{transpose} \circ \textbf{transposeW} \circ \texttt{g}$$

Although the two transpositions seem to cancel each other and do not change the reading order within `f`, the memory access patterns are transformed. **transposeW** changes the layout of the data written by `g`; **transpose** adapts the reading pattern of `f` to the new data layout. The transformed reading pattern might be coalesced and more cache-friendly.

Reordering The reordering is performed on lines 6 and 7 as follows. First, the argument – the result of `g` – is flattened into a 1D array. Then, it is split into `n+1` dimensions, where the sizes of the inner `n` dimensions are determined by the tuning parameters on line 3. Finally, the new dimensions are transposed according to the new order determined by the structural parameter `newDimOrder`. The transposition is performed using **transposeWND**, meaning the write performed by `g` materialises it.

In the combination of **splitND** and **transposeWND** on lines 6 and 7, the parameters of **splitND** act as reordering factors. Larger chunk sizes lead to coarser-grained reordering. The user-chosen parameter `n` determines the depth of splitting; larger values of `n` create more axes of transformation at the cost of increased search space.

Parameter `newDimOrder` takes a vector of integers corresponding to dimensions zero to `n` (inclusive), where the position of the dimension id in the vector determines its position in the transposed array on line 6. Vector `[0,1,..,n]` corresponds to the original order; `[1,0,..,n]` transposes the two outermost dimensions, and so on.

Order Restoration The inverse dimension order is denoted as $\overrightarrow{\texttt{newDimOrder}}^{-1}$; it is used to negate the write transposition and achieve the original reading order. If the new order of dimension `d` is denoted as $\overrightarrow{\texttt{newDimOrder}}(\texttt{d})$, its inverse order is determined using the following inverse function:

$$\overrightarrow{\texttt{newDimOrder}}^{-1}(\texttt{d}) = \overrightarrow{\texttt{newDimOrder}}.\texttt{indexOf}(\texttt{d}) \tag{6.1}$$

Where `indexOf` returns the position of the argument in the vector. The inverse order vector is used on line 9, where **`transposeND`** reorders the read performed by `f` to achieve the original reading order. To restore the original shape, the result of transposition is flattened and split again using the original shape of the input function result ($\overrightarrow{M}$).

Detection of Reads and Writes An extra check is required to confirm whether `f` and `g` refer to a subsequent read and a preceding write operations, respectively. Reordering elements without a preceding write breaks the semantics of the program since **`transposeW`** only applies to writing operations. Similarly, the absence of a subsequent read renders **`transpose`** useless.

Checking whether `f` and `g` are concrete – *i.e.*, write to memory – is straightforward in LIFT. The ASTs of both are traversed, and if each contains a user function, the check is successful. Since the RP itself could be composed with a preceding write and a subsequent read, it might be applicable even if `f` or `g` are abstract. In such case, the validity of the RP application can be established by analysing the view trees of `f` and `g`.

6.2.5.7 Map Interchange

Interchanging directly nested maps creates new parallelisation opportunities. Specifically, it addresses the parallelisation restriction discussed in Section 5.3.4.3: a **`mapLcl`** cannot contain a **`mapWrg`** inside. For example, the following parallel mapping is not allowed:

```
mapLcl(0)( mapWrg(0)(f) << b ) << a
```

```
1  interchangeMaps extends RewritePoint {
2    structParams = List(); tuningParams = List()
3    pattern = (a, b) ⇒ nestedRPs(map(map(f) << b)) << a
4    def rewrite( f ) = {
5      return (a, b) ⇒ transposeW o nestedRPs(map(map(f) << a)) << b }}
```

Listing 6.12: Map Interchange rewrite point definition.

However, when the array `b` contains significantly more elements than can be processed in parallel by a single work group, it is be better to parallelise `b` across multiple work groups. **map** interchange allows achieving this by swapping the loops:

```
mapWrg(0)( mapLcl(0)(f) << a ) << b
```

The Map Interchange RP is presented in Listing 6.12. In addition to swapping the loops, a **transposeW** is inserted after the **map** nest to preserve the original result ordering in memory. For **transposeW** to take effect, function `f` needs to be concrete. This can be confirmed by checking whether `f` or the arguments of RP contain user functions.

6.2.5.8 Reduce Interchange

Interchanging reduction with one of its inner **map**s allows parallelising the reduction operator and accumulator initialisation without a synchronisation overhead. This transformation is similar to ROW-TO-COLUMN REDUCE transformation in DMLL [Bro16].

The **reduce** primitive takes two expressions as arguments: the accumulator initialiser and the array to reduce. Consider the following example, where the accumulator is initialised to an array of zeros:

```
reduce(map(f), init = map(id) << value(0, [T]N))
```

There are two ways to parallelise reduction in this expression. The first is to parallelise the inner **map**s:

```
reduceSeq(mapGlb(0)(f), init = mapGlb(0)(id) << value(0, [T]N))
```

This approach requires a synchronisation barrier between the accumulator initialiser and the reduction operator. Furthermore, since the private memory is not shareable, a

```
1  interchangeReduceND ⟼ interchangeReduce2D | interchangeReduce3D
2                      | interchangeReduce4D | ..
3
4  interchangeReduce4D extends RewritePoint {
5    structParams = List(newReduceDim: Int)
6                   0 > newReduceDim < 4
7    tuningParams = List(factor0: Int, factor1: Int)
8    pattern = nestedRPs(reduce(map(f) ∘ g))
9    def rewrite( nestedRPs, f, g ) = {
10     interchangedReduce = (switch (newReduceDim) {
11       1: map(nestedRPs(reduce(map(map(f))))) ∘ transposeND4(1, 0, 2, 3)
12       2: map(map(nestedRPs(reduce(map(f))))) ∘ transposeND4(1, 2, 0, 3)
13       3: map(map(map(nestedRPs(reduce(f))))) ∘ transposeND4(1, 2, 3, 0) })
14     return joinND2 ∘ interchangedReduce ∘
15            map(splitND2(factor0, factor1) ∘ g) }}
```

Listing 6.13: Reduce Interchange rewrite point definition.

parallelised **map** cannot return a private array; hence, the accumulator has to be placed in the slower local or global memory.

An alternative approach is to interchange **reduce** with its inner **map**, enabling a more efficient parallel mapping:

$$\texttt{reduce}(\texttt{map}(f),\ \texttt{init} = \texttt{map}(\texttt{id}) \mathbin{<<} \texttt{value}(0,\ [T]_N)) \longmapsto$$
$$\texttt{map}(\texttt{reduce}(f,\ \texttt{init} = \texttt{id} \mathbin{<<} \texttt{value}(0,\ T))) \circ \texttt{transpose} \longmapsto$$
$$\texttt{mapGlb}(0)(\texttt{reduceSeq}(f,\ \texttt{init} = \texttt{id} \mathbin{<<} \texttt{value}(0,\ T))) \circ \texttt{transpose}$$

Now, the same thread applies `g`, initialises the accumulator and applies the reduction operator. No synchronisation is required, and the accumulator can be placed in a register.

A Deeper Map Nest The rewrite point in Listing 6.13 interchanges reduction with its inner **map**s. While the pattern on line 8 matches only a one-dimensional inner **map**, the RP splits it into multiple **map**s before choosing how deep to sink **reduce** into the inner **map** nest.

A deeper **map** nest allows finer-grained control over the new memory access pattern and creates more parallelisation opportunities. The number of **map**s to create is determined by the dimensionality of the RP, which is chosen by the user or explored. A 2D RP adds no **map**s to **reduce**(**map**(`f`)); a 3D RP adds one **map**, and so on. The

number of iterations within each new **map** is controlled by the tuning parameters on line 7.

Interchange The interchange proceeds as follows. As per the general case defined in the pattern on line 8, each element of the reduced dimension is processed by some function `g` before being reduced within the original inner **map**`(f)`. The rewritten expression applies `g` on each reduced element on line 15. The result is split using the tuning parameters to create new array dimensions for the interchanged **map** nest in `interchangedReduce` on line 14. Finally, the extra dimensions are flattened to preserve the input function type.

The new position of **reduce** in the **map** nest is controlled by the structural parameter `newReduceDim`. The value of one corresponds to sinking **reduce** down one **map**; the value of two corresponds to sinking down two **map**s, and so on. The new **map** nest with the interchanged **reduce** is built on lines 10 to 13.

Based on `newReduceDim`, the array is first transposed to sink the new reduced dimension inwards. The reduced dimension is originally the outermost according to the input function pattern; therefore, it corresponds to zero in the arguments to $\texttt{transposedND}_4$ on lines 11 to 13. After transposition, the reduce is placed inside the **map** nest accordingly and wrapped in the captured nested RPs since they are likely to be defined on a **reduce**. No extra transposition is required to restore the original data layout since only the reduced dimension has been affected, and that dimension is eliminated by **reduce**.

Impact Reduction is often best performed with the accumulator placed in a private memory and the final result copied to the global or local memory. We will see in Section 6.2.5.9 how the accumulator memory is optimised within another RP and an extra copy inserted after the reduction. By interchanging **reduce** with its inner **map**s, `interchangeReduceND` lessens the amount of intermediate memory required for the accumulator, relieving register pressure.

The argument to the input function could already be transposed across multiple dimensions through the RPs such as `optimiseDataLayout`. Since the extra **map**s brought outside **reduce** are parallelised independently, the **map** nest can match the new data layout and achieve coalesced memory access pattern.

```
1  privatiseAccumulator extends RewritePoint {
2    structParams = List(resultMem: AddressSpace)
3    tuningParams = List()
4    pattern = nestedRPs(reduce(init, g))
5    def rewrite( nestedRPs, init, g ) = {
6      return toMem( resultMem )(id) o
7             nestedRPs(reduce(toPrivate(id) << init, g)) }}
```

Listing 6.14: Accumulator Privatisation rewrite point definition.

6.2.5.9 Accumulator Privatisation

The reduction pattern depends on an accumulator to store the intermediate results. The accumulator is read and updated once per each element in the reduced array. Such a high access rate warrants placing the accumulator in the register memory.

The RP defined in Listing 6.14 inserts a private copy operation into accumulator initialisation on line 7. Private initialisation forces the compiler to allocate the accumulator in registers. Once the reduction is completed, the result is copied again on line 6. The result memory is chosen through exploration using the structural parameter `resultMem`. An optimal choice depends on how the result is accessed elsewhere in the program and the tile sizes.

6.2.5.10 Extra Padding

The one-dimensional array indexing approach of OpenCL is flexible when it comes to tile sizes. Picking a tile size to fit the target platform – its memory and parallel computation capabilities – requires considering only one array dimension, and tiles do not have to cover it precisely. If the last tile spans beyond the array bound, the illegal access can be skipped using an `if`-conditional or a loop terminating condition.

The functional patterns and the multidimensional arrays of LIFT impose more restrictions on tiling. Each array dimension has to be covered by tiles without spanning beyond the array bounds. This restriction limits tile sizes to factors of the corresponding array sizes and ties the success of tiling to the input data dimensions.

Padding breaks the link between tiling and the input data dimensions. By adding zeros at the end of an array dimension, the `padExtraND` rewrite point in Listing 6.15 creates additional tile size choices. The amount of padding is determined by a tuning parameter on line 4, leaving it up to the constraint solver to find the balance between

```
1  padExtraND ⟼ padExtra1D | padExtra2D | ..
2
3  padExtra1D extends RewritePoint {
4    structParams = List(); tuningParams = List(p: Int)
5    pattern = matchedFun @ {
6      (depaddablePattern: [T]_N ⇒ [U]_M )
7      val depaddablePattern = {
8          join ∘ depaddablePattern
9        | split ∘ depaddablePattern
10       | slideND_K ∘ depaddablePattern
11       | map(f) ∘ depaddablePattern
12       | nestedRPs(depaddablePattern) ∘ depaddablePattern
13       | oclKernel(depaddablePattern) ∘ depaddablePattern
14       | ε }}
15   def rewrite( matchedFun, M ) = {
16     prepad = pad((0, p), 0)
17     depad = oclKernel(map(id)) ∘ take(M)
18     return depad ∘ matchedFun ∘ prepad }}
```

Listing 6.15: Extra padding rewrite point definition. The vertical bar symbol (|) on lines 9 to 14 denotes alternative patterns (logical disjunction). Epsilon on line 14 denotes an empty pattern equivalent to an identity function. The subpattern `depaddablePattern` is bound to a variable on line 7 to enable recursive calls on lines 9 to 13.

optimal array size for tiling and extra memory consumption.

The RP pattern definition on lines 5 to 14 encodes four restrictions on patterns where padding is possible and depadding is straightforward.

1. The input function argument must be an array to be paddable and depaddable.
2. The input function return type must be an array to be paddable and depaddable.
3. The input function must preserve the ordering within the padded dimension.
4. The input function result must not change when the argument array is padded with zeros.

The overall pattern type $[T]_N \Rightarrow [U]_M$ on line 5 encodes the first two restrictions. The third restriction requires the padding and its derivative values must remain on the right edge of the array. Reordering the argument or the result through **scatter**, **gather**

and **transpose** complicate removing the extraneous elements – depadding – from the result. By restricting the pattern, the RP allows a straightforward depadding approach.

Finally, the pattern excludes expressions in which the result changes by padding the argument array with zeros. Reduction, for example, is excluded in the general case: although zero-padding does not affect addition or subtraction, it does change the result of multiplication or division.

The recursive subpattern on lines 7 to 14 encodes restrictions three and four. Only the primitives that do not reorder the array are allowed on the padded dimension; line 11 allows all primitives on other dimensions. Line 12 captures the potential nested RPs within the overall expression bound to `matchedFun` on line 5 to give the `rewrite` function control over the new placement of `nestedRPs`.

The composite expression on line 18 has three stages: prepadding, the matched input function and depadding. Prepadding is defined on line 16 using the **pad** primitive adding no elements on the left, and `p` elements on the right of the argument. Depadding on line 17 removes the output elements produced by the application of `matchedFun` on the padding within the argument.

The number of elements to remove depends on `matchedFun` and may not be equal to the number of elements added (`p`). For example, a **join** in `matchedFun` might increase the amount of depadding required. It is easier to infer the number of elements to preserve – `M` – as it is captured in the overall type of the input function on line 5. Line 17 truncates the result to `M` elements using the **take** macro defined in Section 2.3.1.3.

By default, the padding introduced within this RP is virtual. The inserted **pad** primitive transforms the view so that the next read is performed through an inline if. The conditional expression returns either an array element or a zero based on the access index. However, since branching is expensive on a GPU, `padExtraND` is best combined with `kernelFission`, which materialises a padded view in memory within a separate OpenCL kernel. Then, subsequent repeated accesses do not have to be performed through an if-conditional. The convolution expression in Listing 6.4 combines the padding and kernel fission RPs on lines 6 and 7.

Similarly to **pad**, **take** is a view transformation. As seen in Section 2.3.1.3, **take** (`m`) preserves the first `m` elements of the argument and discards the rest. The truncated result is materialised on line 17 using an identity user function. Depadding is performed in a separate OpenCL kernel to reduce the negative impact of branching introduced by the **take** macro. While the main computation is expected to take place within `matchedFun`, depadding can be performed separately using a better-suited work group

configuration.

Higher-dimensional versions of `padExtraND` are defined similarly. The pattern prohibits reordering on N outer dimensions; the transformed expression uses the N-dimensional versions of **`pad`** and **`take`**.

6.2.6 Summary

While the arrangement of RPs in Listing 6.4 enables a specific set of optimisations, the compiler still has a degree of freedom in choosing alternative designs since the application of RPs is optional and parametric. The desired strategy can be further reinforced by imposing manual constraints on structural parameters: for example, the two `abTile` instances on line 10 can be prohibited from issuing memory copy operations, while the copies in the inner `abTile` can be restricted to private memory only. The manual constraints provide means of fine-grained user control over the optimisation process.

As a part of the LIFT IR, RPs are type-safe and have no side effects, which reduces the expertise required to apply them safely. Since RPs are based on pattern-matching, incorrect placement can be detected and reported as a warning, thus helping to use them efficiently. Finally, as first-class citizens of the LIFT IR, RPs yield to rewriting themselves like the rest of the language primitives, allowing complex optimisations to be achieved automatically through composition and nesting of RPs. This property has proven useful during the design stage of this chapter where a set of diverse optimisation techniques across both the direct and GEMM-based convolution turned out to be expressible with a small set of generic RPs as building blocks. The examples include `splitJoinND` used in both `materialise` and `tileNestedMapReduce`, and `materialise` used in `kernelFission` and `abTile`.

6.3 Evaluation

This chapter evaluates the expressivity of rewrite points by analysing the alternative implementations generated by applying rewrite points automatically. First, it presents the best performance and memory consumption achieved through the RP approach and compares them with the alternative code generation methods. Then, the chapter breaks down the runtime of generated kernels in stages to see where the time is spent. Next, the chapter examines the design choices leading to the best performance and

memory consumption, and analyses the rewriting sequences performed by the compiler to achieve these choices. Finally, the size of the search space is briefly analysed, highlighting the challenge of automatic exploration.

This chapter makes a case for the expressive power of the functional rewrite points approach. We will see that the compiler can rewrite the single high-level expression into both direct and GEMM-based convolution. The same set of RPs optimises both methods and achieves performance comparable to that of a vendor-provided handwritten kernel library and a state-of-the-art code generator.

6.3.1 Experimental Methodology

Convolution implementations of LIFT, ARM Compute Library and TVM are measured across the nine unique layer configurations of the VGG-16 model [Sim14]; the layer configurations are provided in Table 4.3. The implementations are run on the ARM Mali-G72 (12 cores) mobile GPU of the HiSilicon Kirin 970 SoC using Debian GNU/Linux 9.8. The GPU frequency is set to 767Mhz, the highest level. The inference is performed over one image, as seen in streaming applications. This setup is the same as in previous chapters.

Performance is measured in the number of floating point operations (FLOPs) per second. Framework performance on a given VGG-16 layer is calculated as the theoretical number of FLOPs required by the layer configuration divided by the time spent computing layer outputs. In the number of FLOPs per layer, multiplication-addition is counted as two separate operations.

LIFT The high-level LIFT convolution expression in Listing 6.3 is annotated with RPs as per Listing 6.4. The LIFT compiler is used to generate one set of OpenCL kernels and a corresponding C++ host code per each layer of VGG as follows. First, the parameter values expressing the configuration of a given VGG layer are substituted into the high-level annotated expression. Early parameter substitution simplifies further transformations and constraints. Then, the annotated expression is iteratively rewritten to apply all RPs; for each RP, the compiler uses a constraint solver to pick structural parameter values that drive RP application. The solver is provided with manually-defined constraints to pick a specific combination of structural parameter values. The values encode design decisions known to achieve good performance on a given VGG layer.

Applying all RPs produces an intermediate LIFT expression with no RPs, but with a high level of **map** nesting and a variable number of tuning parameters. The next stage is parallelisation and **map** fusion. As discussed in Chapter 5, each **map** is associated with an arithmetic parameter encoding parallelisation choices as integers. The compiler also generates a set of integer constraints encoding parallelisation restrictions specific to a given intermediate LIFT expression and a given platform. A constraint solver is then used to pick a valid parallel mapping. The solver is provided with manually-defined constraints to pick a specific parallel mapping informed by the exploration done within the experiments of Chapter 5.

Next, the compiler tunes the simplified parallelised expression as per Chapter 4. The expression is traversed to collect the tuning parameters inserted during RP application; the compiler also generates constraints encoding valid tuning parameter values. The solver picks tuning values using an additional set of manually defined constraints, restricting choices to the combination known to achieve good performance. The picked values are substituted into the LIFT expression.

The tuned expression is vectorised whenever possible as follows. A potential vectorisation opportunity is created by each sequential **map** which contains an identity function, accesses global or local memory and writes into private memory. Among such **map**s, those with a contiguous memory access pattern are vectorised.

The final design decision before code generation is the number of work groups. The compiler picks the minimum number of work groups required to perform all work in parallel. For each indexing dimension of OpenCL work groups, the compiler picks the highest number of iterations performed by the corresponding instances of **mapWrg** in the expression. The approach is similar for **mapGlb**, where the highest number of iterations performed by instances of **mapGlb** is divided by the work group size in the corresponding dimension.

Finally, code generation is performed. The compiler checks memory consumption and inserts synchronisation barriers as discussed in Section 5.4. A single C++ host program is generated, scheduling data transfers and the generated OpenCL kernels. The program is run on the HiKey 970 board with random inputs, and the outputs are validated using the PyTorch implementation of VGG. Each LIFT-generated program is run three times; the median performance of 3 runs is reported. Memory consumption is calculated using the memory allocation reports of the LIFT compiler.

Constraint Solver The parameter values are picked by hand because the search space is too large to explore automatically, given the limited time for experimentation. Using a constraint solver to pick predetermined values ensures that good parameter values can be found through automatic exploration given enough time. The library used for constraint solving is the Choco-solver [Pru16] v4.10.1.

ARM Compute Library ARM Compute (v21.11) is used to produce direct and GEMM-based implementations of the convolutional layers of VGG. Chapters 4 and 5 use an earlier version of ARM Compute (v19.02) since those experiments were run prior to the results of this chapter. Due to this chapter's greater focus on GEMM-based convolution – the most performant approach among all those evaluated – the evaluation uses a more recent version (v21.11). This version achieves higher top performance and therefore provides a compelling baseline.

ARM Compute VGG-16 convolutional layer implementations are produced using the ARM Compute auto-tuner run in the exhaustive mode with 100 iterations per layer. Runtime is measured using OpenCL event profiling; the median runtime across all iterations is reported. Memory consumption is calculated manually based on the direct and GEMM algorithms.

TVM TVM (v0.6) is built with OpenCL support generated using LLVM version 4.0.0. The Winograd strategy is disabled for fairness since it changes the nature of the computation and uses a different convolution algorithm. TVM uses the Spatial Pack Convolution [Zhe18] applying im2col and GEMM on the tiled input. TVM compiler optimisation level is set to 3; the kernels are auto-tuned using GATuner. 1000 candidates per VGG layer are explored, and a median runtime of 30 trials per candidate is reported by the TVM auto-tuner. Memory consumption is calculated based on the intercepted OpenCL memory allocation calls.

6.3.2 Performance and Memory Consumption

While the platform's capabilities determine the actual number of operations per second, the performance metric used hereon represents the number of convolution-related operations per second. The lower values of the metric indicate that the time is spent doing work beyond convolution – synchronising threads, performing unrelated computations – or waiting for blocking reads and writes. Since the minimum number of operations required for convolutional inference by VGG is predetermined, all useful

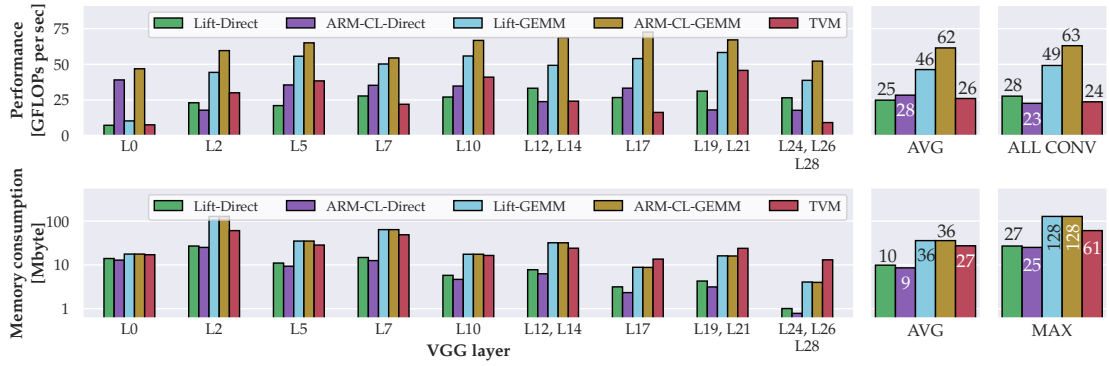


Figure 6.3: Performance and memory consumption comparison of LIFT-generated programs versus ARM Compute- and TVM-generated kernels on VGG-16 layers. AVG refers to the mean performance or memory consumption across all unique convolutional layer configurations. ALL CONV refers to the performance across the entire convolution of VGG-16 taking into account duplicate layer configurations.

operations beyond that number are performed to map computation onto the platform. A well-optimised implementation does the minimum amount of work beyond the convolution itself to leverage hardware capabilities.

The results presented in Figure 6.3 illustrate how well the three frameworks map convolution onto a Mali GPU. In direct convolution, the performance of LIFT-generated programs is on par with those of the ARM Compute and TVM. In GEMM-based convolution, LIFT achieves 68% and 78% of hand-optimised ARM Compute performance on average and across all VGG-16 convolutional kernels, respectively. Both code generators lag behind the ARM Compute on the edge case of the first layer, where the sliding windows and convolutional kernels have dimensions of $3 \times 3 \times 3$.

These results show that the RP approach is expressive enough to transform a single hardware-agnostic convolution expression into two distinct low-level implementations and achieve a good performance on both. We will see in section 6.3.4.2 that GEMM-based implementations on a Mali GPU require further diversification of input preprocessing depending on the layer configuration; the same set of RPs produces both implementations.

The achieved results are especially promising considering that the ARM Compute kernels were written manually by experts. As an indirect indication of the complexity of a well-optimised GEMM-based convolution, we look at the number of code lines in the ARM Compute OpenCL kernels. Across the input preprocessing, weight preprocessing, im2col and GEMM stages, the respective OpenCL kernels use at least

711 lines of code which depend on upwards of 126 code lines implementing C preprocessor macros. This includes the macros that adapt the implementation to a given platform, which corresponds to the rewrite point and parallelism mapping functionalities in LIFT. The size of the ARM Compute implementation of convolution showcases the complexity of the problem successfully handled by RPs.

Memory consumption of LIFT programs is on par with those of the ARM Compute irrespectively of the convolution method. LIFT-Direct requires 11% more memory than ARM Compute-Direct on average since it relies on local buffers to store intermediate results across the two reduction stages of convolution. The amount of extra padding performed by LIFT-GEMM is so small that the memory consumption is not impacted. TVM uses more memory than the direct implementations of LIFT and ARM Compute since it depends on the im2col operation. Its Spatial Tiling approach allows it to use less memory than the GEMM-based implementations.

The difference of $3.6-4\times$ in average memory consumption between the direct and GEMM methods makes the former suitable for low memory budgets, and the latter – for high-bandwidth requirements. A code generator capable of producing both implementations can adapt to changing priorities with less effort than a handwritten kernel library.

We now focus on the breakdown of runtime across different stages of convolution implementations (preprocessing, GEMM, postprocessing) in the RP-based approach and ARM Compute and identify further optimisation opportunities.

6.3.3 Runtime Breakdown

The high-level expression in Listing 6.4 might apply at most six kernel fission RPs. This means the generated convolution is broken down into at most six OpenCL kernels: preprocessing of inputs (two kernels) and weights (one kernel), the main computation (GEMM or direct convolution) and postprocessing of inputs (two kernels). However, the best candidates use no more than four kernels at once since only one type of pre and postprocessing is beneficial for a given convolution method.

Input preprocessing includes the padding required by the layer configuration, extra padding to enable more tiling opportunities, reshaping and im2col. Postprocessing includes depadding the outputs. The RPs introduce two types of extra padding: before and after sliding. Padding before sliding is performed in memory along the right and bottom edges of the input image: it is beneficial for direct convolution, where mate-

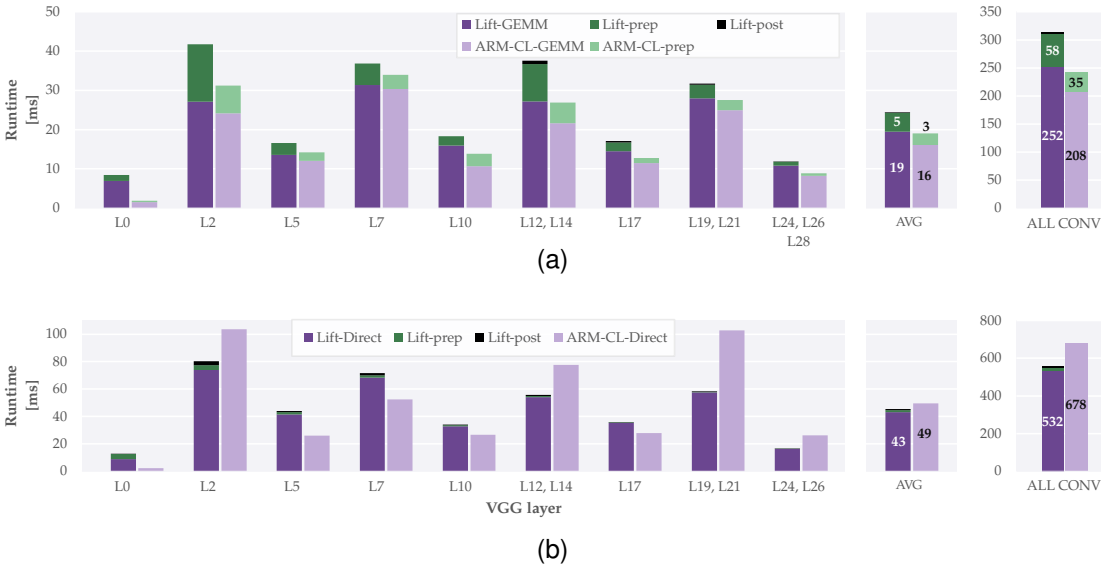


Figure 6.4: Breakdown of runtime across OpenCL kernels in the best-performing LIFT and ARM Compute implementations of GEMM-based (a) and direct (b) convolution. The preprocessing stage includes input reshaping, padding and im2col and does not include weight reshaping. The postprocessing stage includes depadding.

rialising data before sliding is preferable to keep memory consumption low. Padding after sliding is performed in memory at the corner of the image by adding extra empty sliding windows. Corner padding is beneficial with the GEMM-based method since it already requires materialising data after sliding for im2col. Padding at the corner instead of along the edges uses less extra memory, but it is only possible after the 2D sliding is performed and the two spatial dimensions can be fused. Each of the two types of padding requires a corresponding depadding OpenCL kernel, which removes the extra values from the output after the main computation is finished.

Figure 6.4 shows how much runtime LIFT and ARM Compute spend in each stage. In GEMM-based convolution (a), LIFT spends 19% of runtime across the entire VGG-16 convolution on preprocessing (padding, im2col, reshaping), 80% on GEMM and 1% on postprocessing (depadding). In direct convolution, the main computation takes 96% of runtime. On average, LIFT-GEMM takes 18% more time than ARM Compute-GEMM, while LIFT pre and postprocessing stages take 72% more time than ARM Compute preprocessing. LIFT preprocessing in GEMM leaves room for improvement since the main optimisational focus of this work has been on GEMM and not on the pre- and postprocessing kernels.

ARM Compute does not perform extra padding and depadding: splitting data in

uneven tiles is achieved by checking the thread index and returning early when the buffer bound is reached. LIFT's type system requires splitting data in even chunks, but the IR could be extended to allow uneven tiling. Extra padding for optimisation purposes is not expensive in LIFT since convolutional layers already require padding as part of the layer definition, and branching overheads are already incurred. In GEMM, LIFT spends 1.7% of runtime on depadding across the entire VGG-16 convolution and uses 0.01% extra memory.

Both LIFT and ARM Compute reshape weights in separate OpenCL kernels as part of the preprocessing stage. Reshaping optimises the data access pattern in the GEMM kernel for coalescing, vectorisation and cache locality. LIFT's weight preprocessing stage adds 9% to the total convolution runtime of VGG-16. Weight optimisation can be performed offline in advance of the inference. Since the overhead is amortised across multiple inference runs, weight preprocessing is not included in Figures 6.3 and 6.4.

Next, we look at how the proposed set of RPs supports a range of efficient design choices covering two convolution algorithms and all convolutional layer configurations of the VGG network.

6.3.4 Design Choices

Despite the radical difference in memory access patterns and memory requirements of direct and GEMM-based convolution methods, LIFT IR represents both similarly. The stencil memory access pattern is at the core of both methods: it drives the im2col operation in the GEMM-based method and matrix multiplications in direct convolution. Thanks to the concise expression of the stencil pattern in LIFT– the **`slideND`** primitive [Hag18] – inserting a copy operation after a **`slideND`** is enough to transform direct convolution into im2col and GEMM. The RP-based approach expresses this choice through the `kernelFission` RP, which optionally moves the **`slideND`** in a separate OpenCL kernel and commits the stencil pattern to memory using `materialise`.

Of the many RP application sequences made possible by the RP arrangement in Listing 6.4, Tables 6.2 and 6.3 present three that achieve high performance across two convolution methods. It is worth highlighting that all three designs are expressed purely through the structural parameter values, while the RP arrangement in the expression is the same for all three. In other words, the user sees only one high-level program, while the compiler generates the three designs automatically. Although these parame-

ter values are set manually in this work, the process uses the same mechanism which performs automatic exploration of parallel mappings and tuning values. The compiler is capable of finding these implementations automatically.

We now look at the best-performing implementations of direct and GEMM-based convolution. Direct convolution is produced by applying the input preprocessing `kernelFission` RP before $\mathbf{slideND}_2$ in Table 6.2. GEMM-based convolution is produced by applying `kernelFission` after $\mathbf{slideND}_2$ in Table 6.3. Depending on the layer configuration, one of two GEMM implementations is produced: with or without input reshaping. We now look at how rewriting differs across the convolution methods.

6.3.4.1 Direct Convolution

A more complex operation than GEMM, direct convolution requires more effort to optimise; hence, the rewriting sequence is longer in Table 6.2. While the two methods still use the same set of RPs, direct convolution requires more levels of tiling to enable more complex parallel mappings. This is achieved by nesting the same RPs several times. In total, the rewriting sequence in Table 6.2 creates 33 tuning parameters.

Input Pre and Postprocessing To enable a tile size of 64 in the first `abTile` instance (RP #10 in Table 6.2), `padExtra2D` #1 pads the 28×28 input with extra four elements along the bottom edge. Padding is performed together with input reshaping in a separate OpenCL kernel created by `kernelFission` #2. Reshaping is achieved using the `optimiseDataLayout` RP #3: the input is split into 8 tiles of 48 subtiles of 4 elements, where the innermost dimension is left as-is for vectorisation. The other two dimensions are swapped to achieve coalesced access during prefetching performed by `materialise` #20. The transformed input view is materialised in global memory; the copy operation is vectorisable thanks to `splitJoinND` #5.

The postprocessing stage is performed by a separate OpenCL kernel created by `padExtra2D` to remove the extra #4 elements.

Weight Preprocessing Weights are reshaped similarly to inputs: `optimiseDataLayout` #7 leaves 4 elements in the innermost dimension for access vectorisation; the next two dimensions are swapped for coalescing. `materialise` inserts the copy operation and `splitJoinND` tiles it.

Table 6.2: Rewrite point application sequence producing high-performance direct convolution on VGG-16 layers 19 and 21. Arrows denote the RPs inserted by other RPs. The RPs that were not applied and the Join-Split RP are excluded for brevity. The "Line" column refers to Listing 6.4. The row with **slideND**$_2$ separates RPs into those applied on the original input and those applied on the slided input.

#	Rewrite Point	Struct and tune params	Result	Line
1	padExtra2D	**ps**=[4,0]	Extra edge padding, depadding	**6**
2	kernelFission		Input preprocessing OpenCL kernel	**6**
3	↳ optimiseDataLayout	**newDimOrder**=[0,2,1,3], **factors**=[8,48,4]	Reshaped input for coalescing	
4	↳ materialise	**targetMem**=global	Input reshaping materialised	
5	↳ splitJoinND	**tileSizes**=[1,1,4]	Parallelisable vectorisable loops	
		slideND$_2$		
6	kernelFission		Weight preprocessing OpenCL kernel	**8**
7	↳ optimiseDataLayout	**newDimOrder**=[0,1,3,2,4], **factors**=[1,8,48,4]	Reshaped weights for coalescing	
8	↳ materialise	**targetMem**=global	Weight reshaping materialised	
9	↳ splitJoinND	**tileSizes**=[1,1,4]	Vectorisable loop nest	
10	abTile	**tileSizeA**=64, **tileSizeB**=4	Work groups mapped over input & weight tiles	**10**
11	↳ interchangeMaps		Local threads mapped over input tiles	
12	abTile	**tileSizeA**=4, **tileSizeB**=4	Work groups mapped over weight subtiles	**10**
13	tileNestedMapReduce	**tileSize**=192	Parallelisable reduction tree	**11**
14	↳ interchangeReduceND	**newReduceDim**=3, **factors**=[4,4]	Parallelisable final reduction	
15	↳ privatiseAccumulator	**resultMem**=global	Faster accumulator access	
16	↳ splitJoinND	**factors**=[3,8]	Local threads mapped over input & weight subtiles	
17	tileNestedMapReduce	**tileSize**=4	Sliding window tiled for prefetching	**12**
18	↳ privatiseAccumulator	**resultMem**=local	Shared partial reduction results	
19	abTile	**tileSizeA**=4, **tileSizeB**=4	Subtiled inputs and weights for prefetching	**13**
20	↳ materialise	**targetMem**=private	Prefetched a 4x4 tile of inputs	
21	↳ splitJoinND	**tileSizes**=[1,4,4]	Vectorisable loop	
22	↳ materialise	**targetMem**=private	Prefetched a 4x4 tile of weights	
23	↳ splitJoinND	**tileSizes**=[1,4,4]	Vectorisable loop	
24	privatiseAccumulator	**targetMem**=private	Faster accumulator access	**16**

Stencil Computation Inputs and weights are tiled three times in the spatial and output channel dimensions, respectively. This is achieved through the `abTile` instances #10, #12 and #19. The first two tiling levels create parallelisation opportunities. The parallelisation pass maps work groups over 14 tiles of 64 windows in the OpenCL dimension 1, and 128 tiles of 4 output channels in the OpenCL dimension 0. The loop interchange performed by `interchangeMaps` #11 allows mapping local threads over input tiles, and work groups – over output channel tiles (Section 6.2.5.7 discusses this mechanism).

The second `abTile` (#12) creates only one output channel tile; the new **map** is then parallelised across one work group in the OpenCL dimension 2. While no extra parallel

work is achieved, the addition of **mapWrg**(2) "unlocks" the usage of **mapLcl**(2) inside the **map** nest, an extra parallelisation flexibility. Section 5.3.4.3 discusses why LIFT IR requires all instances of **mapLcl** to be wrapped by a **mapWrg** of the same OpenCL dimension.

Reduction is tiled twice using `tileNestedMapReduce` #13 and #17. This time, the compiler is tiling the joint three dimensions of sliding windows, which are being reduced. The first tiling level splits reduction in partial and final reductions, which are parallelised independently across local threads. Intermediate results are exchanged between threads via a shared buffer; the computation is synchronised using a barrier inserted automatically as per Section 5.4. Since the reduce primitive cannot be parallelised, `interchangeReduceND` #14 sinks the **reduce** into its inner **map**s, which are then parallelised.

The second level of reduction (#17) and the final `abTile` (#19) determine the number of prefetched elements. The final `privatiseAccumulator` (#24) keeps the intermediate results in private memory until they are further reduced and placed in the shared memory.

6.3.4.2 GEMM-Based Convolution

At the cost of increased memory usage, im2col simplifies the example rewriting sequence to 18 RP instances, fewer levels of tiling and a more straightforward parallel mapping without local synchronisation. Table 6.3 presents two GEMM-based convolution rewriting sequences: with and without input reshaping (RPs #1 and #3). In total, the rewriting sequence in Table 6.3 creates up to 26 tuning parameters.

Input Pre- and Postprocessing The im2col operation is created by materialising the slided inputs in a separate OpenCL kernel inserted by `kernelFission` #2. Depending on the layer configuration, input reshaping is also performed as follows. Rewrite point `optimiseDataLayout` flattens the array of sliding windows and splits it into dimensions A, B, C, D and E, where E is the innermost dimension. The sizes of D and E are set to 1152 and 4, respectively; in VGG-16 layers 19 and 21, D and E cover one sliding window of 4608 elements. This means that dimensions A, B and C span over different sliding windows. Then, dimension D is transposed with outer dimensions twice: the order of dimensions is now A, D, B, C and E.

The effect of reshaping is a coalesced memory access pattern and a vectorisation opportunity. Dimensions D and E are traversed sequentially during reduction; dimen-

sions A, B and C can be traversed in parallel. Thanks to the new data layout, the threads in dimension C access elements that are close in memory, improving cache line usage. Leaving dimension E innermost ensures that there are always tuples of four consecutive elements to be fetched in the same vector.

The input preprocessing stage also includes corner padding, wherein 6 sliding windows are added at the end of the array before reshaping. Extra elements allow splitting data in chunks of 5 (RP #13). Together with depthwise tiling into chunks of 4 (RP #11) and weight tiling into chunks of 4 (RP #13), the prefetching stage (RP #13) moves 5×4 inputs and 4×4 weights into the private memory. With 64 float registers of the Mali GPU, prefetching 36 elements leaves 28 registers for accumulators and for the other threads (to increase compute core occupancy).

Weight Preprocessing Similarly to input preprocessing, weight reshaping moves the sequentially traversed array dimensions apart. This spreads interdependent elements across memory so that independent elements are placed close to each other for concurrent access.

The `splitJoinND` RP #5 tiles the weight preprocessing OpenCL kernel to create a vectorisation opportunity; the extra **maps** are fused in the parallelisation pass to simplify the search space.

Matrix Multiplication The AB-Tiling RP #10 splits the input into 158 tiles of 5 sliding windows and weights into 128 tiles of 4 output channels. The global threads process the tiles in the OpenCL dimensions 0 and 1. Each thread, therefore, produces 20 outputs and requires 20 registers for the reduction accumulators. The prefetching buffer size is 36 elements, leaving 8 registers free.

Prefetching is performed by `materialise` instances #14 and #16 inserted by `abTile` #13. `splitJoinND` instances #15 and #17 ensure that the vectorisation opportunities left during reshaping (RPs #3, #7) are preserved. Due to reshaping, an innermost dimension of more than 4 elements would be detected as non-contiguous and therefore non-vectorisable.

6.3.5 Rewrite Point Generalisability

Direct and GEMM convolution use cases provide several examples where the same RPs are reused for different purposes. `splitJoinND` is used to create parallelisation and vectorisation opportunities for partial reduction by `tileNestedMapReduce`

Table 6.3: Rewrite Point application sequence producing high-performance GEMM-based convolution on VGG-16 layers 19 and 21. The first and the third RPs (the blue rows) are applied on all VGG-16 layers except for 0, 2, 5 and 7.

#	Rewrite Point	Struct and tune parameters	Result	Line
		`slideND`$_2$		
1	padExtra1D	**p**=6	Corner padding, depadding	**7**
2	kernelFission		Input preprocessing OpenCL kernel	**7**
3	↳ optimiseDataLayout	**newDimOrder**=[0,3,1,2,4], **factors**=[2,5,1152,4]	Reshaped input for coalescing	
4	↳ materialise	**targetMem**=global	Input reshaping materialised, im2col	
5	↳ splitJoinND	**tileSizes**=[9,32,4]	Parallelisable and vectorisable loops	
6	kernelFission		Weight preprocessing OpenCL kernel	**8**
7	↳ optimiseDataLayout	**newDimOrder**=[0,3,2,1,4], **factors**=[16,4,1152,4]	Reshaped weights for coalescing	
8	↳ materialise	**targetMem**=global	Weight reshaping materialised	
9	↳ splitJoinND	**tileSizes**=[1,1,4]	Vectorisable loop nest	
10	abTile	**tileSizeA**=5, **tileSizeB**=4	Global threads mapped over input & weight tiles	**10**
11	tileNestedMapReduce	**tileSize**=4	Each thread reduces 4 interleaved windows	**12**
12	↳ privatiseAccumulator	**resultMem**=global	Faster accumulator access	
13	abTile	**tileSizeA**=5, **tileSizeB**=4	Subtiled inputs and weights for prefetching	**13**
14	↳ materialise	**targetMem**=private	Prefetched a 5x4 tile of inputs	
15	↳ splitJoinND	**tileSizes**=[1,5,4]	Vectorisable loop	
16	↳ materialise	**targetMem**=private	Prefetched a 4x4 tile of weights	
17	↳ splitJoinND	**tileSizes**=[1,4,4]	Vectorisable loop	
18	privatiseAccumulator	**targetMem**=private	Faster accumulator access	**16**

and view materialisation by `materialise`. The materialisation RP is inserted by the `kernelFission` RP to ensure that the OpenCL kernel results are committed to memory, and by `abTile` to optionally move a tile into a faster memory. `padExtraND` is applied on two-dimensional data and a one-dimensional slided view, achieving different padding patterns with the same RP.

RPs are coarser-grained than rewrite rules in Lift by design. Macro transformations truncate the search space to find high-performance implementations faster. However, two aspects make RP generalisable. Firstly, RP patterns are defined on an algorithm-centred IR; therefore, they are reusable across hardware platforms and applications. Secondly, macro transformations are achieved by composing multiple RPs together.

Combining RPs is facilitated by strong typing, where RPs are designed to preserve the type of the original expression. Instead of re-implementing the same transformation repeatedly, each RP only implements one optimisation and inserts other RPs in the transformed expression to rewrite it further. In this regard, RPs are similar to rewrite

rules; however, RPs are more proactive in shaping subsequent transformations. Since RP application is driven by structural parameters, this proactive approach does not overconstrain rewriting.

6.3.6 Search Space

The search space created by RPs, parallelisation and tuning is large. The exact size is non-trivial to estimate, considering that RP application alters the design choices across all three stages.

The GEMM-based convolution solution provided in Table 6.3 contains up to 7 structural parameters with 1.05×10^7 combinations of values. Disregarding the parallelisation constraints, the average number of 40 `map`s per rewritten expression increases the number of candidates to 1.04×10^{47}. The 26 tuning parameters increase the number of candidates further to 1.53×10^{95}, assuming the range of 32 values per parameter. Overall, the high-level annotated convolution expression can insert up to 51 RPs with up to 16 structural parameters and 59 tuning parameters.

As seen, the design space is large, calling for an efficient search strategy such as an auto-tuner, evolutionary algorithm or a machine learning-based approach. The goal of this chapter has been to demonstrate how to encode the design choices at a very high level to create a search space with high-performance candidates. The best candidates were generated by manually fixing the parameter values to demonstrate that the RP-based approach can produce high-performance kernels. The parameter values are informed by the exploration performed in the previous chapters. However, the system supports the automatic optimisation of parameter values.

6.4 Summary

This chapter shows that a small set of RPs applied on a single high-level expression can express a large design space with multiple good solutions. The search space includes two convolution methods and method-specific optimisations; structural and tuning parameters support multiple convolutional layer configurations. The automatically created OpenCL kernels produce pre and postprocessing stages when required.

The examples make a case for the generalisability of the suggested RP designs. The same RPs are inserted manually or through other RPs to achieve different goals. The convolution algorithm-specific manual choices are restricted to the specific placements

or dimensionalities of generic RPs.

The RPs are more coarse-grained than the traditional rewrite rules, but modularity keeps their design simple. RPs inserting other RPs to transform the expression further preserves the extensibility of the rewrite rule-based systems while creating a smaller design space.

Although the parameters are chosen manually in the experiments, they are set through the same mechanism as exploration: arithmetic constraints. Automatic exploration does present a challenge due to the size of the design space, and the next step is to apply an intelligent search strategy. However, this work is a necessary step towards automated exploration, since it demonstrates that high-performant candidates are indeed supported by the proposed code generation technique.

Chapter 7

Conclusions

This thesis has proposed methods for addressing the programmability of parallel accelerators using a functional representation of programs. The thesis has advocated that accelerator diversity necessitates performance-portable solutions to employ a multi-level approach with universal Intermediate Representation and automated code transformation techniques. The combinatorial explosion problem in explorative code generation was tackled using automatically generated constraints and loosely-defined heuristics.

This chapter provides a summary of the work presented in this thesis. Section 7.1 recapitulates the contributions of this work in the context of the GPU programmability challenge. Section 7.2 provides a critical analysis of the contributions of this thesis. Section 7.3 suggests avenues of future work based on the proposed techniques. Section 7.4 concludes.

7.1 Summary of Contributions

This section summarises the contributions of the previous three chapters.

7.1.1 Functional IR for Auto-Tuning

Chapter 4 introduced two fully functional implementations of convolution: high-level and platform-specific expressions written manually. This work showcased the expressiveness of a functional IR; it supports both the algorithm-centred representation and low-level optimisations with fine-grained control over memory and scheduling. Furthermore, the chapter described how strongly-typed functional patterns yield arithmetic constraints on numeric tuning parameters. Automatic constraint generation ad-

dresses the problem of combinatorial explosion of the tuning space by avoiding the evaluation of invalid candidates.

The automated method has two advantages: it removes the need for the user to enumerate the constraints manually and produces the constraints even when the tuning parameters are not known in advance. The latter truly shines when combined with the guided rewriting method proposed in Chapter 6, where tuning parameters are inserted into an expression automatically using rewrite points. The tuning parameter positions and their entailed constraints depend on the rewrite point parameters controlling the structural transformations.

The chapter showed that constrained exploration makes the tuning space more tractable. With the memory consumption further reduced through the improved memory allocation method, the chapter demonstrated performance on par or better than a handwritten kernel library and a state-of-the-art code generator. The chapter also showed that the Pareto front of the performance and memory consumption trade-off can be explored to provide solutions based on the given latency and memory budgets.

7.1.2 Parallelism Mapping Through Constraint Satisfaction

Chapter 5 addressed the need for automatic parallelisation as a bridge between high-level universal IR and a low-level parallel programming model. Specifically, the chapter described a way to model the target parallel architecture with arithmetic constraints and explore the space of valid parallel mappings. On the one hand, this approach explores fine-grained variations of each parallel mapping by focusing on loop-level parallelism. This provides extensive design space coverage. On the other hand, search time is not wasted on evaluating invalid candidates. The technique was shown to find high-performant solutions before even one valid solution is found through unconstrained random exploration.

Furthermore, Chapter 5 presented a synchronisation barrier insertion method. The chapter demonstrated how a Memory Access Graph is constructed from a functional IR to represent the control flow between memory accesses. The MAG was shown to reveal data dependencies and help identify control flow paths that require synchronisation. Compared to the original pattern matching-based barrier insertion method, the proposed technique was shown to find more efficient barrier placements and prevent more data races.

7.1.3 Guided Rewriting

Chapter 6 tackled algorithmic optimisations through a combination of explorable parameters and loosely-defined heuristics. The chapter showed that using the same IR to represent both the application and optimisations leads to composability – rewrite points may insert other rewrite points into the transformed expression to reuse their functionality. The chapter also showed that exposing macro design decisions as rewrite point parameters achieves large design space coverage. This was demonstrated by producing two convolution algorithms with algorithm-specific optimisation chains from the same annotated expression. By defining rewrite points on a universal algorithmic IR, the chapter presented a generic optimisation method not limited to domain- or platform-specific transformations.

7.2 Critical Analysis

This section critically analyses several aspects of the proposed techniques, highlighting the issues identified retrospectively.

7.2.1 Redundant Space Pruning

Early detection of invalid implementations reduces the search space considerably: Section 5.5 shows that only 1 out of 49,000 parallel mappings is valid. Although high-performance candidates are found quickly enough, search times could be reduced further. Currently, extra time is spent evaluating design decisions which produce equivalent programs or achieve the same performance. For example, consider a loop nest iterating over a single multidimensional array, where tile sizes determine the number of iterations in the corresponding loops. When only some of the loops are parallelised, changing the tile sizes affects the memory access pattern, thread coarsening and compute core saturation. When all loops are sequential, changing the tile sizes does not affect the memory access pattern and performance. The tuning stage does not consider the effect of the tuning parameter values and still evaluates all tile sizes of sequential loops.

Exploration of the parallel mapping space also has redundancies that could be optimised. For example, OpenCL thread indexing is performed in three dimensions; the first dimension enumerates threads in the same warp. While the other two indexing dimensions can be mapped to different arrays and thus exploit more parallelism, the

two dimensions are interchangeable. However, LIFT still explores their permutations across the same sets of loops.

During algorithmic rewriting, some functional patterns cancel each other. Two rewrite points might compose an even number of transpositions, which is semantically equivalent to not transposing. Another example is splitting an array and flattening it again. Although a trivial pass of the AST detects and simplifies these patterns, the resulting simplified expression is still evaluated.

Avoiding redundant exploration could be achieved through uniqueness constraints. In auto-tuning, the constraint inference mechanism could be reused to enumerate the parameter value equivalences based on the AST. In parallelisation mapping, some equivalences could be expressed as simple constraints; others require the constraint solver to delegate partial evaluation of parallel mappings to the LIFT compiler as part of constraint satisfaction.

Rewrite point application could be optimised automatically for a given program by enumerating the equivalent transformations before exploration and pruning these redundancies with constraints. While enumerating the equivalences across the entire expression might be too expensive, many redundancies can be detected locally by analysing the output of a single rewrite point, two nested rewrite points, and so on.

7.2.2 Synchronisability-Based Space Pruning

Synchronisability of the LIFT programs is determined by parallel mapping and tuning parameter values. Section 5.3.6 discusses the example where a parallel mapping might be synchronisable depending on the distribution of work among threads. The current approach depends on the proposed barrier insertion technique implemented as a compiler check to detect whether a kernel is synchronisable. Although this approach is still fully automated and finds solutions quickly enough through uniformly random exploration. However, the compiler still evaluates all tuning combinations for non-synchronisable parallel mappings. Furthermore, the compiler does not consider that some parallel mappings require tuning parameters to satisfy extra constraints to produce synchronisable kernels.

Parallel mapping search could be improved by performing the synchronisability analysis during the parallelisation stage. The proposed barrier insertion method could be extended to evaluate a parallel mapping with unresolved tuning parameters and determine whether the mapping is (1) always, (2) never or (3) sometimes synchronisable.

In (1), tuning may proceed normally; in (2), tuning of the parallel mapping is aborted; in (3), tuning proceeds with extra constraints limiting the parameters to those producing synchronisable kernels. This way, the parallelisation stage would feed the tuning stage with additional constraints truncating the search further.

7.2.3 Multi-Stage Rewrite Point Application

The composability of the RP approach allows reusing RPs for macro-transformations. The efficiency of RP nesting is predicated on the ability of RPs to control the placement of the nested RPs in the transformed expression. Section 6.2.2 discusses how this control is achieved through pattern-matching nested RPs in the input function. This approach requires top-down rewriting order, where the outer RP is applied before the nested RPs; the outer RP may move the nested RPs as required to ensure that the nested RPs remain applicable on the transformed expression.

Sometimes, top-down rewriting order results in overly coarse-grained RPs, missing opportunities to reuse finer-grained RPs. The design of some RPs would be simplified if some of their nested RPs were applied first, creating a pattern for further transformation. This flexibility could be enabled by applying RPs in stages based on their transformations. For example, the compiler could first apply all the RPs, which insert extra `map`s into the expression. Then, the RPs interchanging or eliminating `map`s would be applied. Automated analysis of all possible RP transformations could be used to determine the order of application statically.

7.2.4 Rewrite Point DSL

The current RP implementation uses the full expressive power of Scala, the implementation language of the LIFT compiler. While Scala's pattern-matching mechanism is indispensable for RP application, a general-purpose programming language yields verbose implementations. This presents an opportunity to improve the approach.

Although the proposed RP interface outlines the mechanism in brush strokes, the work on Chapter 6 identified several common patterns in RP implementations warranting a rewrite point DSL. The pattern-matching mechanism of Scala could be used to simplify matching the LIFT IR. Example "meta-patterns" include matching a pattern on each LIFT function in a composition chain individually, which could be used to find the first concrete function in a chain; matching a pattern on each LIFT function in a nest of functional patterns, which could be used to find perfectly nested `map`s.

Section 6.2.1 describes the decomposition of the matched expression, where the components of the matched expression are exposed to the rewriting method for reuse in the transformed expression. The current definition of expression components is untyped: it includes both LIFT expressions, function declarations and LIFT types. Formalising decomposition would simplify RP implementation and prevent defects.

7.3 Future Work

This section suggests directions for extending the work presented in this thesis.

Informed Search Although the proposed techniques truncate the search space, much time is spent evaluating inefficient solutions. This thesis' contributions could be complemented with informed search methods to find high-performance implementations faster.

The constraints mechanism could be reused to express heuristics for tuning, parallelisation and algorithmic optimisations. The heuristic rules could be derived from static analysis of the AST to predict performance based on the design decisions. Several properties of a functional IR could be used as a proxy for performance. The numbers of fetched cache lines and cache misses could be calculated based on the scheduling policy, distances between array accesses and cache specifications. The amount of memory consumed in each address space could be derived from the LIFT memory allocator; the candidates depending on slower memories could be penalised. Functional patterns such as **`slide`** could be used to detect and avoid data duplication. The suggested heuristics could be used both to truncate the space further based on hard cut-offs and as objective functions to optimise.

Reinforcement Learning (RL) techniques [Sut18] could be applied to drive rewriting. The rewrite point approach is well-represented in the domain of RL. With the compiler as the agent, RP applications as actions and the target hardware as the environment, the compiler could be rewarded for the RP applications which improve performance or memory consumption. Since the optimisations used in this thesis are not application- or platform-specific, the RL agent could be reused across compilations, with transfer learning used to improve sample efficiency. Alternatively, an evolutionary algorithm could be used to optimise the search.

Cross-Domain Optimisation The proposed techniques could be applied to other applications benefitting from parallel acceleration, such as computer vision and scientific computing. Rewrite points could be used to implement domain-specific optimisations such as quantisation and the Coppersmith-Winograd algorithm [Cop87]. Some of these methods break the semantics of the input program. The LIFT type system could be extended to capture approximate types and quantify the loss of precision similarly to how the type system has been extended to represent sparse matrices [Piz20]. The RP mechanism is decoupled from the type system, so the other domains would still benefit from the guided rewriting approach.

Further extension of this work would be to provide DSLs based on the LIFT IR. Raised level of abstraction would shift the burden of providing efficient RP annotations from the user to the compiler engineer.

Distributed and Heterogenous Platforms Although the contributions of this thesis were demonstrated on the example of GPU, the proposed techniques are not specific to the platform. The LIFT compiler has been shown to generate efficient code across a range of GPUs and CPUs [Ste15; Ste17], including using the OpenMP programming model [Piz16]. The portability of the framework could be further extended using the proposed techniques on distributed and heterogeneous platforms.

The MapReduce project [Dea08] has demonstrated the power of functional patterns in a distributed setting. The guided rewriting method could optimise distributed programs with low manual effort. The kernel fission rewrite point could be used to split the application into distributed kernels; LIFT host code generation could be extended to support distributed address spaces. The hierarchical and memory-scoping parallelisation constraints could be adapted to the heterogeneous programming model.

7.4 Summary

This chapter has summarised the contributions of this thesis, critically analysed the limitations of the work presented and suggested avenues for future work. Overall, the thesis attempted to bridge the gap between the two poles of optimisation: expertise-based heuristics and mechanical exploration. With one providing custom-tailored solutions based on experience and the other achieving great design space coverage, a hybrid of the two is needed to tackle the challenges of parallel programming.

This work focused on tuning, parallelism mapping and algorithmic optimisations.

The first two stages were tackled exploratively yet safely; the latter stage incorporated user expertise into the code generation process while maintaining a high level of separation of concern.

Acronyms

ALU Arithmetic Logic Unit. 1, 14, 15, 59

API Application Programming Interface. vii, 37, 38, 39, 43, 50, 52, 82

AS Address space. 32, 79, 80

ASIC Application-Specific Integrated Circuit. 1, 16, 38

AST Abstract Syntax Tree. 25, 28, 29, 31, 32, 35, 46, 71, 73, 74, 75, 78, 80, 89, 90, 95, 96, 97, 98, 106, 107, 108, 121, 126, 146, 172, 174

BLAS Basic Linear Algebra Subprograms. 11, 13, 40

CNN Convolutional Neural Network. 4, 6, 9, 10, 12, 13, 49, 53, 55, 58, 64, 90

CPU Central Processing Unit. 2, 11, 14, 16, 38, 39, 41, 45, 47, 49, 50, 54, 81, 112, 175

DAG Directed Acyclic Graph. 44, 73

DL Deep Learning. 40, 41, 44, 45

DRAM Dynamic random-access memory. 14, 15, 16, 17

DSL Domain-Specific Language. x, 46, 111, 173, 175

FLOP Floating-point operations. 113

FPGA Field Programmable Gate Arrays. 1, 16, 38

GEMM General Matrix Multiply. iii, 6, 7, 11, 12, 13, 35, 40, 54, 55, 59, 62, 82, 83, 84, 91, 112, 113, 123, 132, 133, 134, 135, 153, 154, 156, 157, 158, 159, 160, 161, 163, 164, 165, 166

GPGPU General-Purpose computing on Graphics Processing Units. 38

GPU Graphics Processing Unit. iii, iv, vii, viii, 1, 2, 4, 5, 6, 9, 11, 12, 13, 14, 15, 16, 17, 19, 26, 27, 30, 34, 36, 37, 38, 39, 40, 42, 44, 45, 46, 47, 48, 49, 50, 52, 53, 54, 55, 56, 58, 59, 60, 61, 64, 65, 67, 68, 71, 75, 80, 81, 82, 83, 84, 85, 86, 89, 90, 91, 93, 99, 105, 111, 112, 118, 134, 135, 139, 152, 154, 157, 164, 169, 175

ILP Integer Linear Program. 50

im2col Image To Column. 11, 12, 13, 59, 62, 112, 133, 134, 135, 156, 157, 158, 159, 160, 163

IR Intermediate Representation. iii, viii, x, 2, 3, 5, 6, 7, 9, 19, 21, 22, 25, 26, 27, 28, 29, 30, 32, 33, 36, 37, 38, 39, 41, 42, 43, 44, 45, 46, 47, 49, 50, 51, 52, 53, 54, 55, 56, 58, 61, 63, 69, 71, 72, 74, 86, 89, 90, 91, 105, 106, 108, 109, 117, 118, 121, 122, 123, 124, 125, 126, 142, 160, 163, 165, 169, 170, 171, 173, 174, 175

ISA Instruction Set Architecture. 39, 45

MAC Multiply-accumulate operation. 40, 113

MAG Memory Access Graph. 89, 105, 107, 108, 109, 117, 170

ML Machine Learning. 1, 19, 44, 46, 58

OOP Object-oriented programming. 125

RL Reinforcement Learning. 174

RP Rewrite point. 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 157, 158, 160, 161, 162, 163, 164, 165, 166, 167, 173, 174, 175

SIMD Single Instruction/Multiple Data. 14, 17, 38

SL Spatial Locality. 59, 60

SM Streaming Multiprocessor core. 1, 40

SRAM Static random-access memory. 14, 15, 17

TL Temporal Locality. 59, 60

UF User function. 26, 32, 78, 79, 80

Bibliography

[Ada19] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, et al. "Learning to optimize halide with tree search and random programs". In: *ACM Transactions on Graphics (TOG)* 38.4 (2019), pages 1–12. DOI: 10.1145/3306346.3322967 (cited on pages 4, 48, 49).

[Aik98] Alexander Aiken and David Gay. "Barrier inference". In: *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1998, pages 342–354 (cited on page 47).

[Ald11] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. "Accelerating code on multi-cores with fastflow". In: *European Conference on Parallel Processing*. Springer. 2011, pages 170–181 (cited on pages 3, 44).

[And17] Andrew Anderson, Aravind Vasudevan, Cormac Keane, and David Gregg. "Low-memory gemm-based convolution algorithms for deep neural networks". In: *arXiv preprint arXiv:1709.03395* (2017) (cited on page 41).

[Ans09] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. "Petabricks: A language and compiler for algorithmic choice". In: *ACM Sigplan Notices* 44.6 (2009), pages 38–49 (cited on page 51).

[Ans14] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. "Opentuner: An extensible framework for program autotuning". In: *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 2014, pages 303–316 (cited on pages 4, 48, 81).

[Arm20] Arm. *Arm Mali Bifrost and Valhall OpenCL Developer Guide: Developer Guide*. 4.1. Arm, 2020 (cited on page 60).

[Arm21] Arm Ltd. *Arm Compute Library*. en. 2021. URL: https://developer.arm.com/ip-products/processors/machine-learning/compute-library (visited on 07/02/2021) (cited on pages 2, 40, 81, 91).

[Asa06] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. "The landscape of parallel computing research: A view from berkeley". In: (2006) (cited on page 1).

[Bag19] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. "Tiramisu: A polyhedral compiler for expressing fast and portable code". In: *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2019, pages 193–205. DOI: 10.1109/CGO.2019.8661197 (cited on pages 3, 42, 90).

[Bag20] Riyadh Baghdadi and Albert Cohen. "Scalable Polyhedral Compilation, Syntax vs. Semantics: 1–0 in the First Round". In: (2020) (cited on page 42).

[Bar19] Paul Barham and Michael Isard. "Machine learning systems are stuck in a rut". In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. 2019, pages 177–183 (cited on page 45).

[Bea19] Ulysse Beaugnon, Basile Clément, Nicolas Tollenaere, and Albert Cohen. "On the Representation of Partially Specified Implementations and its Application to the Optimization of Linear Algebra Kernels on GPU". In: *arXiv preprint arXiv:1904.03383* (2019). DOI: 10.48550/arXiv.1904.03383 (cited on page 49).

[Bey11] James C Beyer, Eric J Stotzer, Alistair Hart, and Bronis R de Supinski. "OpenMP for accelerators". In: *OpenMP in the Petascale Era: 7th International Workshop on OpenMP, IWOMP 2011, Chicago, IL, USA, June 13-15, 2011. Proceedings 7*. Springer. 2011, pages 108–121 (cited on page 39).

[Bon08] Uday Bondhugula, Albert Hartono, J Ramanujam, and P Sadayappan. "Pluto: A practical and fully automatic polyhedral program optimization system". In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer. 2008 (cited on page 42).

[Bos11] Pradip Bose. “Power Wall”. In: *Encyclopedia of Parallel Computing*. Edited by David Padua. Boston, MA: Springer US, 2011, pages 1593–1608. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_499. URL: https://doi.org/10.1007/978-0-387-09766-4_499 (cited on page 1).

[Bro16] Kevin J Brown, HyoukJoong Lee, Tiark Rompf, Arvind K Sujeeth, Christopher De Sa, Christopher Aberger, and Kunle Olukotun. “Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns”. In: *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. 2016, pages 194–205 (cited on page 147).

[Buc04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. “Brook for GPUs: stream computing on graphics hardware”. In: *ACM transactions on graphics (TOG)* 23.3 (2004), pages 777–786 (cited on page 38).

[Car99] William W Carlson, Jesse M Draper, David E Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. *Introduction to UPC and language specification*. Technical report. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999 (cited on page 39).

[Cat11] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. “Copperhead: compiling an embedded data parallel language”. In: *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*. 2011, pages 47–56 (cited on page 39).

[Cha16] Li-Wen Chang, Izzat El Hajj, Christopher Rodrigues, Juan Gómez-Luna, and Wen-mei Hwu. “Efficient kernel synthesis for performance portable programming”. In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2016, pages 1–13 (cited on pages 4, 51, 123).

[Che14] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. “cudnn: Efficient primitives for deep learning”. In: *arXiv preprint arXiv:1410.0759* (2014) (cited on pages 2, 40, 54).

[Che15] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems". In: *arXiv preprint arXiv:1512.01274* (2015) (cited on page 82).

[Che18a] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. "{TVM}: An automated end-to-end optimizing compiler for deep learning". In: *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 2018, pages 578–594. DOI: 10.5555/3291168.3291211 (cited on pages 5, 50, 89, 91, 123).

[Che18b] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. "Learning to optimize tensor programs". In: *Advances in Neural Information Processing Systems* 31 (2018) (cited on pages 4, 49).

[Che21] Lorenzo Chelini, Andi Drebes, Oleksandr Zinenko, Albert Cohen, Nicolas Vasilache, Tobias Grosser, and Henk Corporaal. "Progressive raising in multi-level ir". In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2021, pages 15–26 (cited on page 43).

[Chr11] Matthias Christen, Olaf Schenk, and Helmar Burkhart. "Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures". In: *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE. 2011, pages 676–687 (cited on page 39).

[Col04] Murray Cole. "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming". In: *Parallel computing* 30.3 (2004), pages 389–406 (cited on pages 3, 43).

[Col11] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. "Natural language processing (almost) from scratch". In: *Journal of machine learning research* 12.ARTICLE (2011), pages 2493–2537 (cited on page 1).

[Col14] Alexander Collins, Dominik Grewe, Vinod Grover, Sean Lee, and Adriana Susnea. “NOVA: A functional language for data parallelism”. In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. 2014, pages 8–13 (cited on page 47).

[Col89] Murray I Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989 (cited on pages 3, 43).

[Cop87] Don Coppersmith and Shmuel Winograd. “Matrix multiplication via arithmetic progressions”. In: *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. 1987, pages 1–6 (cited on page 175).

[Cou71] Pierre-Jacques Courtois, Frans Heymans, and David Lorge Parnas. “Concurrent control with “readers” and “writers””. In: *Communications of the ACM* 14.10 (1971), pages 667–668 (cited on page 104).

[Cyp18] Scott Cyphers, Arjun K Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, et al. “Intel ngraph: An intermediate representation, compiler, and executor for deep learning”. In: *arXiv preprint arXiv:1801.08058* (2018) (cited on pages 3, 45, 90).

[Dar05] Alain Darte and Robert Schreiber. “A linear-time algorithm for optimal barrier placement”. In: *Proceedings of the tenth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. 2005, pages 26–35 (cited on page 47).

[Dav20] Shail Dave, Riyadh Baghdadi, Tony Nowatzki, Sasikanth Avancha, Aviral Shrivastava, and Baoxin Li. “Hardware acceleration of sparse and irregular tensor computations of ml models: A survey and insights”. In: *arXiv preprint arXiv:2007.00864* (2020) (cited on page 42).

[De 19] Simon Garcia De Gonzalo, Sitao Huang, Juan Gómez-Luna, Simon Hammond, Onur Mutlu, and Wen-mei Hwu. “Automatic generation of warp-level primitives and atomic instructions for fast and portable parallel reduction on GPUs”. In: *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2019, pages 73–84 (cited on pages 4, 51, 123).

[Dea08] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters”. In: *Communications of the ACM* 51.1 (2008), pages 107–113 (cited on pages 3, 43, 175).

[Den09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. “Imagenet: A large-scale hierarchical image database”. In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pages 248–255 (cited on page 10).

[Dos14] Cicero Dos Santos and Maira Gatti. “Deep convolutional neural networks for sentiment analysis of short texts”. In: *Proceedings of COLING 2014, the 25th international conference on computational linguistics: technical papers*. 2014, pages 69–78 (cited on page 1).

[Enm10] Johan Enmyren and Christoph W Kessler. “SkePU: a multi-backend skeleton programming library for multi-GPU systems”. In: *Proceedings of the fourth international workshop on High-level parallel programming and applications*. 2010, pages 5–14 (cited on page 44).

[Fra18] Franz Franchetti, Tze Meng Low, Doru Thom Popovici, Richard M Veras, Daniele G Spampinato, Jeremy R Johnson, Markus Püschel, James C Hoe, and José MF Moura. “SPIRAL: Extreme performance portability”. In: *Proceedings of the IEEE* 106.11 (2018), pages 1935–1968. DOI: 10.1109/JPROC.2018.2873289 (cited on pages 3, 49).

[Fuk80] Kunihiko Fukushima. “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position”. In: *Biological cybernetics* 36.4 (1980), pages 193–202 (cited on page 53).

[Geo18] Evangelos Georganas, Sasikanth Avancha, Kunal Banerjee, Dhiraj Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. “Anatomy of high-performance deep learning convolutions on simd architectures”. In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2018, pages 830–841 (cited on page 41).

[Gib22] Perry Gibson and José Cano. “Transfer-Tuning: Reusing Auto-Schedules for Efficient Tensor Program Code Generation”. In: *31st International Conference on Parallel Architectures and Compilation Techniques (PACT). Chicago*. 2022 (cited on page 49).

[Gin18] Philip Ginsbach, Toomas Remmelg, Michel Steuwer, Bruno Bodin, Christophe Dubach, and Michael FP O'Boyle. "Automatic matching of legacy code to heterogeneous APIs: An idiomatic approach". In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 2018, pages 139–153 (cited on page 43).

[Hag18] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. "High performance stencil code generation with lift". In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 2018, pages 100–112. DOI: 10.1145/3168824 (cited on pages 19, 20, 92, 105, 122, 160).

[Hag20a] Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Sergei Gorlatch, and Michel Steuwer. "A language for describing optimization strategies". In: *arXiv preprint arXiv:2002.02268* (2020) (cited on pages 51, 123).

[Hag20b] Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. "Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies". In: *Proceedings of the ACM on Programming Languages* 4.ICFP (2020), pages 1–29 (cited on pages 51, 123).

[Hai16] Michael Haidl, Michel Steuwer, Tim Humernbrum, and Sergei Gorlatch. "Multi-stage programming for GPUs in C++ using PACXX". In: *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit*. 2016, pages 32–41 (cited on page 39).

[Hen17] Troels Henriksen, Niels GW Serup, Martin Elsman, Fritz Henglein, and Cosmin E Oancea. "Futhark: purely functional GPU-programming with nested parallelism and in-place array updates". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2017, pages 556–571. DOI: 10.1145/3062341.3062354 (cited on pages 3, 46, 54, 90).

[Hor11] Amir H Hormati, Mehrzad Samadi, Mark Woh, Trevor Mudge, and Scott Mahlke. "Sponge: portable stream programming on graphics engines". In: *ACM SIGPLAN Notices* 46.3 (2011), pages 381–392 (cited on page 50).

[Hu18] Jie Hu, Li Shen, and Gang Sun. “Squeeze-and-excitation networks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pages 7132–7141. DOI: 10.1109/CVPR.2018.00745 (cited on page 11).

[ISO20] ISO Central Secretary. *Programming languages — C++*. en. Standard ISO/IEC 14882:2020. Geneva, CH: International Organization for Standardization, 2020. URL: https://www.iso.org/standard/79358.html (cited on page 39).

[Jac19] Dejice Jacob, Phil Trinder, and Jeremy Singer. “Python programmers have GPUs too: Automatic python loop parallelization with staged dependence analysis”. In: *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages*. 2019, pages 42–54 (cited on page 42).

[Jia14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. “Caffe: Convolutional architecture for fast feature embedding”. In: *Proceedings of the 22nd ACM international conference on Multimedia*. 2014, pages 675–678 (cited on page 54).

[Kal14] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. “A convolutional neural network for modelling sentences”. In: *arXiv preprint arXiv:1404.2188* (2014) (cited on page 1).

[Kha19] Jehandad Khan, Paul Fultz, Artem Tamazov, Daniel Lowell, Chao Liu, Michael Melesse, Murali Nandhimandalam, Kamil Nasyrov, Ilya Perminov, Tejash Shah, et al. “MIOpen: An open source library for deep learning primitives”. In: *arXiv preprint arXiv:1910.00078* (2019) (cited on pages 2, 40).

[Khr15] SYCL subgroup Khronos OpenCL Working Group et al. *SYCL Specification, SYCL integrates OpenCL devices with modern C++.(May 2015)*. 2015 (cited on page 39).

[Khr22] The Khronos Group. *OpenCL Reference Pages*. en. 2022. URL: https://www.khronos.org/registry/OpenCL/sdk/2.2/docs/man/html/ (visited on 02/02/2022) (cited on pages 2, 9, 16, 17, 38, 98).

[Kri12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pages 1097–1105 (cited on page 73).

[Lai18] Liangzhen Lai, Naveen Suda, and Vikas Chandra. "Not all ops are created equal!" In: *arXiv preprint arXiv:1801.04326* (2018). DOI: 10.48550/arXiv.1801.04326 (cited on page 53).

[Lam15] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. "Numba: A llvm-based python jit compiler". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 2015, pages 1–6 (cited on pages 39, 42).

[Lat19] Chris Lattner and Jacques Pienaar. *MLIR primer: A compiler infrastructure for the end of Moore's law*. 2019. URL: https://storage.googleapis.com/pub-tools-public-publication-data/pdf/1c082b766d8e14b54e36e37c9fc3ebbe8b4a72dd.pdf (visited on 11/06/2023) (cited on page 44).

[Lat20] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. "MLIR: A compiler infrastructure for the end of Moore's law". In: *arXiv preprint arXiv:2002.11054* (2020) (cited on page 43).

[Lea17] Chris Leary and Todd Wang. "XLA: TensorFlow, compiled". In: *TensorFlow Dev Summit* (2017) (cited on pages 3, 44).

[Li20] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. "The deep learning compiler: A comprehensive survey". In: *IEEE Transactions on Parallel and Distributed Systems* 32.3 (2020), pages 708–727. DOI: 10.1109/TPDS.2020.3030548 (cited on pages 45, 50, 112).

[McC12] Michael McCool, James Reinders, and Arch Robison. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012 (cited on page 37).

[McD13] Trevor L McDonell, Manuel MT Chakravarty, Gabriele Keller, and Ben Lippmeier. "Optimising purely functional GPU programs". In: *ACM SIGPLAN Notices* 48.9 (2013), pages 49–60. DOI: 10.1145/2500365.2500595 (cited on pages 3, 46, 54, 55, 90).

[Men15] Charith Mendis, Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao, and Saman Amarasinghe. "Helium: Lifting high-performance stencil kernels from stripped x86 binaries to Halide DSL code". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2015, pages 391–402 (cited on page 43).

[Mid86] Samuel Pratt Midkiff. *Automatic generation of synchronization instructions for parallel processors*. Technical report. Illinois Univ., Urbana (USA). Center for Supercomputing Research and Development, 1986 (cited on pages 47, 105).

[Mog20] Naums Mogers, Valentin Radu, Lu Li, Jack Turner, Michael O'Boyle, and Christophe Dubach. "Automatic generation of specialized direct convolutions for mobile GPUs". In: *Proceedings of the 13th Annual Workshop on General Purpose Processing using Graphics Processing Unit*. 2020, pages 41–50. DOI: 10.1145/3366428.3380771 (cited on pages vi, 114).

[Mog22] Naums Mogers, Lu Li, Valentin Radu, and Christophe Dubach. "Mapping parallelism in a functional IR through constraint satisfaction: a case study on convolution for mobile GPUs". In: *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*. 2022, pages 218–230 (cited on page vi).

[Mos23] William S Moses, Ivan R Ivanov, Jens Domke, Toshio Endo, Johannes Doerfert, and Oleksandr Zinenko. "High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs". In: *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 2023, pages 119–134 (cited on page 47).

[Mul15] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. "Polymage: Automatic optimization for image processing pipelines". In: *ACM SIGARCH Computer Architecture News* 43.1 (2015), pages 429–443 (cited on page 48).

[Mul16] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. "Automatically scheduling halide image processing pipelines". In: *ACM Transactions on Graphics (TOG)* 35.4 (2016), pages 1–11 (cited on page 49).

[Mun11] Aaftab Munshi, Benedict Gaster, Timothy G Mattson, and Dan Ginsburg. *OpenCL programming guide*. Pearson Education, 2011 (cited on page 18).

[New20] Julie L Newcomb, Andrew Adams, Steven Johnson, Rastislav Bodik, and Shoaib Kamil. "Verifying and improving halide's term rewriting system with program synthesis". In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020), pages 1–28 (cited on page 49).

[Nic08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. "Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for?" In: *Queue* 6.2 (2008), pages 40–53 (cited on pages 2, 38).

[Nug15] Cedric Nugteren and Valeriu Codreanu. "CLTune: A generic auto-tuner for OpenCL kernels". In: *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*. IEEE. 2015, pages 195–202 (cited on page 48).

[Nug18] Cedric Nugteren. "CLBlast: A tuned OpenCL BLAS library". In: *Proceedings of the International Workshop on OpenCL*. 2018, pages 1–10 (cited on page 40).

[Num98] Robert W Numrich and John Reid. "Co-Array Fortran for parallel programming". In: *ACM Sigplan Fortran Forum*. Volume 17. 2. ACM New York, NY, USA. 1998, pages 1–31 (cited on page 39).

[NVI] NVIDIA. *GitHub - NVIDIA/cutlass: CUDA Templates for Linear Algebra Subroutines — github.com*. `https://github.com/NVIDIA/cutlass` (cited on page 40).

[Nvi07] CUDA Nvidia. "Cublas library programming guide". In: *NVIDIA Corporation. edit* 1 (2007), page 206 (cited on pages 2, 40).

[NVI18] NVIDIA. *CUDA LLVM Compiler — developer.nvidia.com*. `https://developer.nvidia.com/cuda-llvm-compiler`. 2018 (cited on page 39).

[NVI22] NVIDIA. *NVIDIA TensorRT*. Dec. 2022. URL: `https://developer.nvidia.com/tensorrt` (cited on page 40).

[OBo02] Michael O'Boyle and Elena Stohr. "Compile time barrier synchronization minimization". In: *IEEE Transactions on Parallel and Distributed Systems* 13.6 (2002), pages 529–543 (cited on page 47).

[Ope08] ARB OpenMP. “Openmp application program interface v3. 0”. In: *OpenMP Architecture Review Board* (2008) (cited on page 39).

[Ope15] OpenACC. “OpenACC programming and best practices guide”. In: (2015) (cited on page 39).

[Osa23] Muhammad Osama, Duane Merrill, Cris Cecka, Michael Garland, and John D Owens. “Stream-K: Work-centric Parallel Decomposition for Dense Matrix-Matrix Multiplication on the GPU”. In: *arXiv preprint arXiv:2301.03598* (2023) (cited on page 40).

[Pet20] Ivan Petrov, Daiheng Gao, Nikolay Chervoniy, Kunlin Liu, Sugasa Marangonda, Chris Umé, Jian Jiang, Luis RP, Sheng Zhang, Pingyu Wu, et al. “Deepfacelab: A simple, flexible and extensible face swapping framework”. In: *arXiv preprint arXiv:2005.05535* (2020) (cited on page 45).

[Pho13] Phitchaya Mangpo Phothilimthana, Jason Ansel, Jonathan Ragan-Kelley, and Saman Amarasinghe. “Portable performance on heterogeneous architectures”. In: *ACM SIGARCH Computer Architecture News* 41.1 (2013), pages 431–444. DOI: 10.1145/2451116.2451162 (cited on pages 4, 48, 51, 123).

[Piz16] Federico Pizzuti. “Implementing an OpenMP backend for the Lift compiler”. 2016 (cited on page 175).

[Piz19] Federico Pizzuti, Michel Steuwer, and Christophe Dubach. “Position-dependent arrays and their application for high performance code generation”. In: *Proceedings of the 8th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing*. 2019, pages 14–26 (cited on page 122).

[Piz20] Federico Pizzuti, Michel Steuwer, and Christophe Dubach. “Generating fast sparse matrix vector multiplication from a high level generic functional IR”. In: *Proceedings of the 29th International Conference on Compiler Construction*. 2020, pages 85–95 (cited on page 175).

[Piz22] Federico Pizzuti, Michel Steuwer, and Christophe Dubach. “Generating Work Efficient Scan Implementations for GPUs the Functional Way”. In: *European Conference on Parallel Processing*. Springer. 2022, pages 335–349 (cited on page 19).

[Pou08] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. "Iterative optimization in the polyhedral model: Part II, multidimensional time". In: *ACM SIGPLAN Notices* 43.6 (2008), pages 90–100. DOI: 10.1145/1379022.1375594 (cited on page 42).

[Pre19] S Prema, Rupesh Nasre, R Jehadeesan, and BK Panigrahi. "A study on popular auto-parallelization frameworks". In: *Concurrency and Computation: Practice and Experience* 31.17 (2019), e5168 (cited on pages 3, 37, 42).

[Pru16] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. "Choco solver documentation". In: *TASC, INRIA Rennes, LINA CNRS UMR* 6241 (2016) (cited on pages 111, 156).

[Rag13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines". In: *Acm Sigplan Notices* 48.6 (2013), pages 519–530 (cited on page 46).

[Ras18] Ari Rasch and Sergei Gorlatch. "ATF: A generic auto-tuning framework". In: *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. 2018, pages 3–4 (cited on pages 48, 81).

[Ras21] Ari Rasch, Richard Schulze, Michel Steuwer, and Sergei Gorlatch. "Efficient auto-tuning of parallel programs with interdependent tuning parameters via auto-tuning framework (ATF)". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 18.1 (2021), pages 1–26 (cited on page 48).

[Rei07] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007 (cited on page 39).

[Rem16] 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*. 2016, pages 22–31 (cited on page 41).

[Roe18] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. "Relay: A new ir for machine learning frameworks". In: *Proceedings of the 2nd ACM SIGPLAN international workshop on machine learning and programming languages*. 2018, pages 58–68 (cited on pages 46, 51).

[Rot18] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, Jack Montgomery, Bert Maher, Satish Nadathur, Jakob Olesen, Jongsoo Park, Artem Rakhov, Misha Smelyanskiy, and Man Wang. "Glow: Graph lowering compiler techniques for neural networks". In: *arXiv preprint arXiv:1805.00907* (2018) (cited on pages 3, 45).

[Sér99] Jocelyn Sérot. "Explicit parallelism". In: *Research Directions in Parallel Functional Programming*. Springer, 1999, pages 379–396 (cited on page 38).

[She12] Jie Shen, Jianbin Fang, Henk Sips, and Ana Lucia Varbanescu. "Performance gaps between OpenMP and OpenCL for multi-core CPUs". In: *2012 41st International Conference on Parallel Processing Workshops*. IEEE. 2012, pages 116–125 (cited on page 39).

[Shv10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. "The hadoop distributed file system". In: *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee. 2010, pages 1–10 (cited on page 44).

[Sim14] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014). DOI: 10.48550/arXiv.1409.1556 (cited on pages 1, 13, 53, 82, 90, 113, 154).

[Sin11] Jeremy Singer, George Kovoor, Gavin Brown, and Mikel Luján. "Garbage collection auto-tuning for java mapreduce on multi-cores". In: *ACM SIGPLAN Notices* 46.11 (2011), pages 109–118 (cited on page 44).

[Sio18] Savvas Sioutas, Sander Stuijk, Henk Corporaal, Twan Basten, and Lou Somers. "Loop transformations leveraging hardware prefetching". In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 2018, pages 254–264 (cited on page 49).

[Siu18] Kevin Siu, Dylan Malone Stuart, Mostafa Mahmoud, and Andreas Moshovos. "Memory requirements for convolutional neural network hardware accelerators". In: *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2018, pages 111–121 (cited on page 144).

[Sor21] Tyler Sorensen, Lucas F Salvador, Harmit Raval, Hugues Evrard, John Wickerson, Margaret Martonosi, and Alastair F Donaldson. "Specifying and testing GPU workgroup progress models". In: *Proceedings of the ACM on Programming Languages* 5.OOPSLA (2021), pages 1–30 (cited on page 47).

[Sot19] Matthew Sotoudeh, Anand Venkat, Michael Anderson, Evangelos Georganas, Alexander Heinecke, and Jason Knight. "ISA mapper: a compute and hardware agnostic deep learning compiler". In: *Proceedings of the 16th ACM International Conference on Computing Frontiers*. 2019, pages 164–173 (cited on pages 4, 45, 49).

[Ste11] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. "Skelcl-a portable skeleton library for high-level gpu programming". In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. IEEE. 2011, pages 1176–1182 (cited on pages 3, 44).

[Ste12] Robert Stewart and Jeremy Singer. "Comparing fork/join and MapReduce". In: *Department of Computer Science, Heriot-Watt University* (2012) (cited on page 44).

[Ste15] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. "Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code". In: *ACM SIGPLAN Notices* 50.9 (2015), pages 205–217 (cited on pages 122, 124, 129, 175).

[Ste16] 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*. 2016, pages 1–10. DOI: 10.1145/2968455.2968521 (cited on pages 29, 90).

[Ste17] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. "Lift: a functional data-parallel IR for high-performance GPU code generation". In: *2017 IEEE/ACM International Symposium on Code Generation and Op-*

timization (CGO). IEEE. 2017, pages 74–85. DOI: 10.1109/CGO.2017.7863730 (cited on pages 3, 19, 28, 30, 92, 175).

[Sto21] Larisa Stoltzfus, Brian Hamilton, Michel Steuwer, Lu Li, and Christophe Dubach. "Code Generation for Room Acoustics Simulations with Complex Boundary Conditions". In: *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2021, pages 485–496 (cited on pages 19, 26, 142).

[Sut05] Herb Sutter et al. "The free lunch is over: A fundamental turn toward concurrency in software". In: *Dr. Dobb's journal* 30.3 (2005), pages 202–210 (cited on page 2).

[Sut18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018 (cited on page 174).

[Tho12] Krishnahari Thouti and SR Sathe. "Comparison of OpenMP & OpenCL parallel processing technologies". In: *arXiv preprint arXiv:1211.2038* (2012) (cited on page 39).

[Tru16] Leonard Truong, Rajkishore Barik, Ehsan Totoni, Hai Liu, Chick Markley, Armando Fox, and Tatiana Shpeisman. "Latte: a language, compiler, and runtime for elegant and efficient deep neural networks". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2016, pages 209–223 (cited on page 45).

[Tse95] Chau-Wen Tseng. "Compiler optimizations for eliminating barrier synchronization". In: *ACM SIGPLAN Notices* 30.8 (1995), pages 144–155 (cited on page 47).

[Udu09] Abhishek Udupa, R Govindarajan, and Matthew J Thazhuthaveetil. "Software pipelined execution of stream programs on GPUs". In: *2009 International Symposium on Code Generation and Optimization*. IEEE. 2009, pages 200–209 (cited on page 50).

[Vas18] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions". In: *arXiv preprint arXiv:1802.04730* (2018). DOI: 10.48550/arXiv.1802.04730 (cited on pages 3, 42, 90).

[Ver13] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. "Polyhedral parallel code generation for CUDA". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 9.4 (2013), pages 1–23. DOI: 10.1145/2400682.2400713 (cited on pages 3, 42).

[Wan14] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. "Intel math kernel library". In: *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 2014, pages 167–188 (cited on page 40).

[Wan18] Zheng Wang and Michael O'Boyle. "Machine learning in compiler optimization". In: *Proceedings of the IEEE* 106.11 (2018), pages 1879–1901 (cited on page 2).

[Wei17] Richard Wei, Lane Schwartz, and Vikram Adve. "DLVM: A modern compiler infrastructure for deep learning systems". In: *arXiv preprint arXiv:1711.03016* (2017) (cited on pages 3, 44).

[Wha01] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. "Automated Empirical Optimization of Software and the ATLAS Project". In: *Parallel Computing* 27.1–2 (2001). Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000, pages 3–35 (cited on pages 2, 40).

[Whi12] Tom White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012 (cited on page 44).

[Wie12] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. "OpenACC—first experiences with real-world applications". In: *Euro-Par 2012 Parallel Processing: 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012. Proceedings 18*. Springer. 2012, pages 859–870 (cited on page 39).

[Win08] Winram, Laurence. *The Informatics Forum*. [Copyright: The Univesity of Edinburgh]. 2008 (cited on page 10).

[Wu16] Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng, and Robert Hundt. "gpucc: an open-source GPGPU compiler". In: *Proceedings of*

the 2016 International Symposium on Code Generation and Optimization. 2016, pages 105–116 (cited on page 39).

[Xia12] Zhang Xianyi, Wang Qian, and Zhang Yunquan. "Model-driven level 3 BLAS performance optimization on Loongson 3A processor". In: *2012 IEEE 18th international conference on parallel and distributed systems*. IEEE. 2012, pages 684–691 (cited on pages 2, 40).

[Xin22] Jiarong Xing, Leyuan Wang, Shang Zhang, Jack Chen, Ang Chen, and Yibo Zhu. "Bolt: Bridging the Gap between Auto-tuners and Hardware-native Performance". In: *Proceedings of Machine Learning and Systems* 4 (2022), pages 204–216 (cited on page 49).

[Ye22] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. "SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning". In: *arXiv preprint arXiv:2207.04606* (2022) (cited on page 51).

[Zer19] Tim Zerrell and Jeremy Bruestle. "Stripe: Tensor compilation via the nested polyhedral model". In: *arXiv preprint arXiv:1903.06498* (2019). DOI: 10.48550/arXiv.1903.06498 (cited on pages 42, 45, 46, 90).

[Zha18] Jiyuan Zhang, Franz Franchetti, and Tze Meng Low. "High performance zero-memory overhead direct convolutions". In: *Proceedings of the 35th International Conference on Machine Learning* (2018) (cited on page 41).

[Zhe18] Lanmin Zheng and Tianqi Chen. "Optimizing deep learning workloads on ARM GPU with TVM". In: *Proceedings of the 1st on Reproducible Quality-Efficient Systems Tournament on Co-Designing Pareto-Efficient Deep Learning*. ACM New York, NY, USA, 2018, page 1. DOI: 10.1145/3229762.3229764 (cited on pages 112, 156).

[Zhe20a] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. "Ansor: Generating high-performance tensor programs for deep learning". In: *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 2020, pages 863–879. DOI: 10.5555/3488766.3488815 (cited on pages 4, 49, 89).

[Zhe20b] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. "Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system". In: *Proceedings*

of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. 2020, pages 859–873 (cited on page 49).