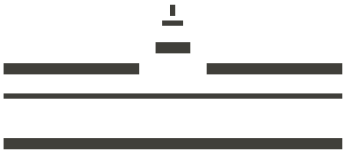


Westfälische
Wilhelms-Universität
Münster

AN EXTENSION OF A FUNCTIONAL INTERMEDIATE LANGUAGE FOR PARALLELIZING STENCIL COMPUTATIONS AND ITS OPTIMIZING GPU IMPLEMENTATION USING OPENCL

BASTIAN HAGEDORN

Masterthesis

Computer Science Department
Parallel and Distributed Systems
University of Münster

FIRST SUPERVISOR: Prof. Dr. habil. Sergei Gorlatch
(University of Münster)

SECOND SUPERVISOR: Dr. Michel Steuwer
(University of Edinburgh)

September 2016

Bastian Hagedorn: *An Extension of a Functional Intermediate Language for Parallelizing Stencil Computations and its Optimizing GPU Implementation using OpenCL,* Masterthesis, © September 2016

Dedicated to my wife.

Another lesson we should have learned from the past is that
the development of "richer" or "more powerful" programming languages
was a mistake in the sense that these baroque monstrosities,
these conglomerations of idiosyncrasies,
are really unmanagable, both mechanically and mentally.
I see a great future for very systematic
and very modest programming languages.

— Edsger W. Dijkstra [18]

ACKNOWLEDGMENTS

First of all, I thank my supervisor Sergei Gorlatch for giving me the freedom to pursue my research interest. I'm especially thankful for my research visit at the University of Edinburgh where I met numerous inspiring and supportive people. I have greatly benefited from collaborating with Michel Steuwer, Christophe Dubache and Toomas Remmelg. I thank my second supervisor Michel Steuwer for his continuous support and encouragement and I am thankful for the numerous inspiring discussions with Christophe Dubache. I appreciate their support and I am looking forward to continue our collaboration on the work described in this thesis. I'm grateful for the financial support received from the EuroLab-4-HPC which made my research visit to Edinburgh possible.

I thank Michael Haidl and Ari Rasch for the fruitful discussions during my work in Münster. I thank my family and my sister for their support and interest in my research. Last, but not least I thank my wife for her never-ending support, love and encouragement.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

1 INTRODUCTION AND BACKGROUND

Nowadays, Graphics Processing Units (GPUs) are an inherent part of high-performance computing. GPUs are used to accelerate compute-heavy parts of applications, they speed up computations of many scientific applications like stencil computations which appear in applications like digital signal processing or computer simulations. In this thesis, we introduce a functional approach to parallelize stencil computations for modern GPUs. We provide a high-level functional language that enables the expression of stencil computations by composing functional primitives. Based on the ideas of algorithmic reasoning, these expressions are rewritten into a functional Intermediate Language (IL) and eventually used to automatically generate high-performance OpenCL code for GPUs.

1.1 MOTIVATION

Using a functional approach to generate high-performance code for modern GPUs has already been proven to be successful with the introduction of LIFT. [53, 60]. The main goal of this thesis is to extend the LIFT framework to support the generation of high-performance OpenCL stencil kernels. Inspired by the ideas of Backus [3], Bird [7], Cole [15] and many others from the 70's and 80's, we use a functional approach to generate high-performance code for modern GPUs. In our approach, computations are expressed as compositions of intuitive built-in primitives. A formal rewrite system allows to systematically transform high-level expressions written by a programmer into efficient low-level expressions written in LIFT's functional IL (Intermediate Language). This IL is closely related to the OpenCL programming model and bridges the gap between the high-level functional expression and the imperative OpenCL kernel. By extending LIFT's collection of high-level primitives as well as its IL, we provide a powerful functional approach to express and optimize stencil computations for GPUs. Multidimensional stencil computations are expressed using our functional language without the use of specialized hard-coded solutions found in existing library approaches [36, 59]. Well-known stencil optimizations are formalized and applied as sequences of rewrite rules. Therefore, our functional IL allows to systematically (and in the future possibly automatically) apply device-specific low-level stencil optimizations. Thus, instead of providing one specific tuned OpenCL kernel for a certain class of stencil computations, we are able to automatically generate device-specific optimized kernels.

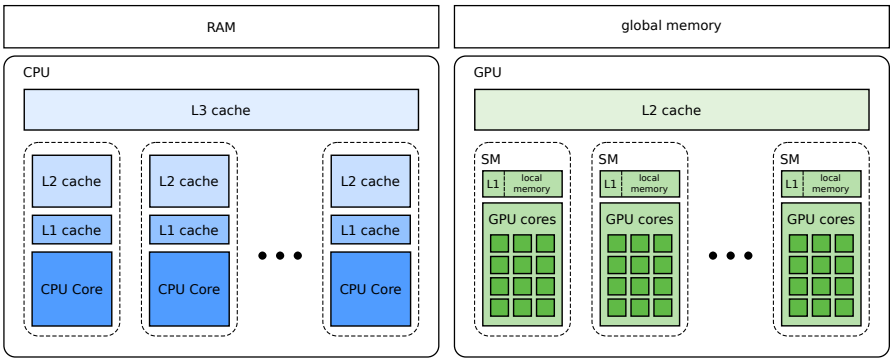


Figure 1: Overview of a CPU and GPU architecture (inspired by [57])

1.2 BACKGROUND

In this section, we introduce all concepts and related work required to understand the rest of this thesis. We start by introducing Graphics Processing Units (GPUs) and how to program them using OpenCL. Afterwards, we introduce high-level abstractions that aim to simplify the programming of GPUs. Then, we introduce the LIFT framework which is extended in this thesis. Finally, we introduce stencil computations, discuss related work and end this chapter with a list of contributions made in this thesis.

1.2.1 *GPU Programming in OpenCL*

Nowadays, *Graphics Processing Units (GPUs)* are an inherent part of high performance computing. Although originally designed to specifically accelerate the rendering of complex graphics in computer simulation or 3D-Games, they are now used to perform a far wider range of computations. This is emphasized by the term *General-Purpose computation on Graphics Processing Units (GPGPU)* coined by Mark Harris in 2002. For example, GPUs are heavily used as *accelerators* in many scientific applications where compute intensive parts of an application are offloaded to the GPU.

Central Processing Units (CPUs) are designed to achieve low instruction latency at the cost of having low instruction throughput. They use large cache hierarchies and cores that are able to prefetch instructions or execute instructions out-of-order in order to minimize latency. In contrast, a GPU is designed to achieve high instruction throughput at the cost of having a higher instruction latency. Therefore, the architecture of a GPU is significantly different compared to the architecture of a typical CPU as depicted in Figure 1. A GPU consists of thousands of lightweight cores grouped together in so-called *Streaming Multiprocessors (SM)* which is the term used by Nvidia. Furthermore, each SM has its own control units, registers, execution pipelines and small caches. These SMs execute instructions in a *Single*

Instruction Multiple Data (SIMD) manner. A SM schedules threads in groups of 32 called *warps* for execution. Every warp in a SM executes the same instructions in each step on different data. Modern GPUs feature four warp schedulers per SM which allows to execute four warps concurrently.

A GPU contains a large but slow *off-chip* memory also called *global memory* with a capacity of several gigabytes. It is accessible from both the GPU and CPU, shared among all SMs and analogous to the traditional RAM (Random Access Memory). Each SM contains a small amount of fast *on-chip* memory that is accessible to all cores in a SM with a capacity of several kilobytes. On modern GPUs, the on-chip memory is partitioned into a hardware-managed Level-1-Cache and a programmer-managed *local* memory. The latency of the on-chip memory is roughly 100× lower than the latency of the global memory that is shared among all SMs. Therefore, utilizing the local memory to exploit data locality is mandatory in high performance GPU programs. In this thesis, we specifically focus on Nvidia GPUs which are divided into categories called *compute capabilities*. Compute capabilities designate a certain GPU architecture and specify supported features.

OpenCL (Open Computing Language) developed by the Khronos Group in 2008 is an open standard for programming multi-core CPUs, accelerators like the Intel Xeon Phi or GPUs. OpenCL is defined in terms of a hierarchy of four different models: the *Platform Model*, the *Execution Model*, the *Memory Model* and the *Programming Model*. We briefly discuss every model in the following.

PLATFORM MODEL The *Platform Model* consists of a *host* which is connected to several *devices*. In this thesis, the host is always the CPU and the device is the GPU. An OpenCL device is divided into one or more smaller elements called *compute units (CUs)* which are themselves divided into even smaller elements called *processing elements (PEs)*. Note that this platform model is similar to the architecture of a GPU: The SMs correspond to the CUs and the lightweight cores inside a SM correspond to the PEs.

An OpenCL application is divided into the *host code* and the *device kernel code*. In this thesis, we investigate how to generate efficient device kernel code that is offloaded to the device to compute stencil computations.

EXECUTION MODEL The *Execution Model* consists of the *kernels* that are executed on the device and the *host program* that is executed on the host. The communication between a host and a device is realized by using a *command queue*. This queue is used to submit commands that either exchange data between host and device, or to submit a synchronization command that enforces a certain execution of commands or to submit a kernel execution. A *work-item* is a thread executing a kernel. Work-items are exe-

cuted by one ore more PEs and are structured in groups called *work-groups*. Work-groups share the on-chip local memory and local threads inside a work group can be synchronized. Work-items of different work-groups can not be synchronized. Every work-item executes a single kernel instance in a SPMD (Singe Program Multiple Data) manner. Work-items and work-groups can be structured in a three dimensional grid for execution.

MEMORY MODEL The *Memory Model* is divided into two parts: The *host memory* and the *device memory* The host memory is the memory directly available to the host whereas the device memory is the memory directly available to the kernels executing in devices. The device memory is further divided into four *address spaces* or *memory regions*: The *global memory* is accessible by all work-items that can arbitrarily read from or write to this region. The *constant memory* is a read-only memory region inside the global memory accessible by all work-items. The *local memory* is a memory region local to a work-group. This memory region is shared by all work-items of a single work-group. Finally, the *private memory* is private to a work-item and not visible to other work-items. Again, note the similarity between OpenCL's memory model and the architecture of a GPU: The main GPU memory corresponds to the global memory, the fast on-chip memory corresponds to the local memory and registers correspond to the private memory.

PROGRAMMING MODEL The *Programming Model* is divided into the *data parallel model* and the *task parallel model*. We focus on the data parallel model where kernels are executed by many work-items organized in work-groups as explained above. In the task parallel model, kernels are executed by a single work-item and parallelism is exploited by launching multiple kernels that can be executed concurrently.

Programming GPUs using OpenCL is error-prone and cumbersome because it exposes many low-level hardware details. The programmer has to manually manage the memory hierarchy while being careful about memory accesses in order to gain high performance. Memory accesses need to be aligned in order to avoid uncoalesced global memory accesses and the local memory must be utilized efficiently to hide the high latency of the global memory. Furthermore the kernel code needs to manage multiple levels of parallelism for example at work-group and work-item level and is itself written in a low-level C-like language.

1.2.2 *Structured Parallel Programming*

There have been several approaches that aim to simplify the programmability of GPUs. These approaches can be summarized using the term Structured Parallel Programming. In the late 80's, Cole coined the term *Algorithmic Skeletons* [15] in his PhD thesis and proposed to use a structured approach to the management of parallel computation [16]. Cole identified several parallel patterns which he called algorithmic skeletons [47] like *Divide & Conquer*, that often appeared in code written using simple parallel programming frameworks like Pthreads or MPI (Message Passing Interface). These patterns can be extracted to libraries that provide parallel building blocks that can be composed to write parallel programs on a higher level of abstraction. An algorithmic skeleton is basically a higher order function that captures a specific parallel pattern, thus specifies the overall structure of a computation without specifying the details of how to compute the exact result. The computation is described in so called *user functions* or *customizing functions* that are passed to a skeleton as arguments. Based on Cole's ideas, many others argued to raise the abstraction level of parallel programs using structured parallel patterns instead low-level programming models. For example Gorlatch [24] argues to avoid low-level control structures like `send` and `recv` in MPI code and replace them with collective operations which behave like Cole's idea of algorithmic skeletons. Since then, different skeleton libraries have been developed that all aim to simplify parallel programming using high-level abstractions. Examples for these libraries are eSkel [5], Eden [39], SkePU [20], MUESLI [36], Accelerate [13] or SkelCL [59]. The idea to provide high-level building blocks to simplify parallel programming is nowadays also adapted in many different standards like the parallel STL (Standard Template Library) of C++ or Intel's TBB (Thread Building Blocks). Our work directly extends Cole's ideas of algorithmic skeletons as high-level building blocks and is introduced in the next section.

1.2.3 *The LIFT Framework*

LIFT is a novel approach to achieve a high level of programming and performance portability [60]. It compiles a high-level functional expression to high-performance device-specific OpenCL code. A key idea in the compilation process of LIFT is to express the algorithmic structure of a program as well as device-specific optimizations using functional primitives. These primitives are divided into two categories: *high-level functional algorithmic primitives* and *low-level functional OpenCL primitives*. LIFT provides a small collection of *high-level functional algorithmic primitives* (in the following simply called high-level primitives) as a high-level interface to the programmer. Simi-

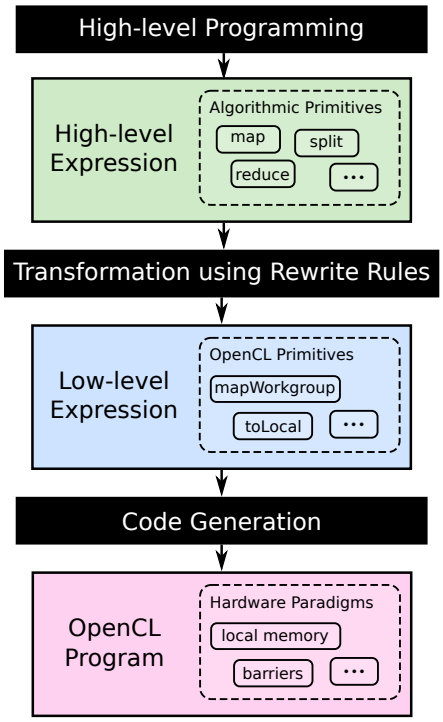


Figure 2: Overview of the compilation flow of the LIFT Framework (inspired by [57])

lar to existing skeleton libraries like SkelCL [59] or Accelerate [13], a programmer uses these high-level building-blocks to write a functional program that expresses the computation. Similar to the work of Backus on functional programming systems [3], our high-level building blocks are either *built-in primitives* or *defined* functions composed of primitives or other defined functions. A high-level expression is then systematically rewritten to another functional representation consisting of *low-level functional OpenCL primitives* (in the following simply called low-level primitives) using rewrite rules. These rewrite rules are based on the concepts of algorithmic reasoning pioneered by Bird and Meertens [7]. A low-level functional expression closely represents an OpenCL program and is eventually used to generate imperative OpenCL code. The compilation process of the LIFT framework is illustrated in Figure 2. Most of the high-level primitives like *map* or *reduce* are defined as higher order functions and are already well known to functional programmers as they are part of most functional programming languages. LIFT's high-level primitives do not capture a complex algorithmic structure like stencil computations, but are rather designed to capture basic fundamental algorithmic structures. Expressing complex problems is achieved by nesting and composing these high-level primitives.

While high-level primitives capture fundamental algorithmic patterns, LIFT's low-level primitives combined with the rewrite rules

are designed to express common optimization patterns. Therefore, instead of applying optimizations ad-hoc using a rule of thumb, rewrite rules are used to systematically apply common optimizations formalized as functional expressions. Unlike library approaches, where high-level programs are usually implemented by low-level loop based code [11, 20, 36], LIFT's low-level primitives act as an intermediate language (IL). Therefore, they fill the gap between the high-level algorithmic representation of a program and its low-level hardware-specific implementation.

In this thesis, we extend the collection of high-level primitives in order enable the expression of stencil computations. Moreover, we introduce well-known optimization for stencil computations on GPUs by using and extending LIFT's IL.

1.2.4 *Stencil Computations*

Stencil computations (also known as *stencil codes* [72]) are a typical algorithmic pattern arising in many scientific application domains like digital signal processing [12] or time-intensive simulations [9]. They are considered as one of the original seven *dwarfs* or *motifs* of high performance computing [1].

In a stencil computation, elements of a multidimensional structured grid are (iteratively) updated. A single element is updated by performing a *stencil operation* which applies a so-called *stencil function* to a neighborhood of elements. This function describes how to update each element of the grid by taking the current value of an element and the values of its neighboring elements into account. The *stencil* or *stencil shape* defines which neighboring elements are used by the stencil operation. We call this group of elements the *neighborhood*. A stencil with a neighborhood of n elements is called a n-point stencil. Border elements of the grid lack some neighboring elements. Therefore, when updating border elements, a so-called *boundary handling* is required that specifies how to replace these missing elements (also called *halo elements*). Stencils often contain coefficients to compute a weighted average of a neighborhood. These specific stencil computations are called *stencil convolution* or simply convolution in the following. We mainly focus on this subclass of stencil applications because of its broad use in high performance computing. In case of a convolution, the stencil that contains numerical values is also called *convolution kernel* or simply kernel.

In this thesis, we consider stencil computations where the result of updating the grid is stored in a secondary grid instead of overwriting the values of the input grid. These are known as *Jacobi*-like stencils, instead of *Gauss-Seidel*-like stencil computations which update element in place.

Although conceptually stencils are easy to implement using nested loops, it is difficult to write efficient stencil code for GPUs. This is emphasized by the fact that Nvidia provides a guide on how to optimize a 17×17 convolution [51] which we examine in the following chapters more extensively.

1.3 RELATED WORK

In this section, we discuss how the work of this thesis fits in the context of existing work. We summarize already mentioned related work and put it into the context of this work.

STENCIL-SPECIFIC HIGH-LEVEL PROGRAMMING APPROACHES There exist a broad collection of high-level approaches that aim to simplify the programming of stencil applications on multi and many-core processors such as GPUs. These include several stencil-specific DSLs (Domain Specific Languages) or EDSLs (Embedded DSLs) like HLSF [19], and others [2, 14, 29, 32, 46, 65]. Furthermore, there exist multiple frameworks that support the development of stencil applications like skeleton libraries that provide a specific stencil skeleton like SkePU [20], MUESLI [36] PASTHA [37] and SkelCL [59] and others [6, 33, 43, 49, 50, 55, 63]. In pragma-based approaches, source code is annotated using pragmas, examples are Mint [67] or PADS [34]. Finally there exist domain specific high-level approaches for domains in which stencil computations occur like solving PDEs (Partial Differential Equations) [4, 10] or image processing [21, 52].

However, none of these approaches decomposes the stencil pattern into fundamental primitives that act as building blocks to express different kinds of stencil computations. By decomposing the stencil pattern we are able to use the power of function composition to express multidimensional stencil computations without providing specialized implementations for each dimension. Furthermore, we are able to define functions that are composed of these basic primitives to provide the same level of abstraction as the mentioned existing high-level programming approaches.

Previous work has already proven that the decomposition of computations into LIFT's fundamental building blocks is beneficial in case of sparse linear algebra [28] and GEMM (General Matrix Matrix Multiplication) [53].

OPTIMIZATIONS FOR STENCIL COMPUTATIONS Besides high-level programming frameworks that aim to simplify the programmability of stencil applications, there exist many different strategies to optimize stencil code to improve performance. These

include different blocking [48, 68, 70] and tiling approaches [25–27, 35, 40, 54], and other collections of optimizations [17, 22, 42, 62]. Furthermore, there exist multiple auto-tuning frameworks that aim to automatically optimize stencil computations [23, 30, 31, 41].

However none of these approaches formalized these optimizations as we do using a core dependently-typed λ-calculus along with a denotational semantics and a set of rewrite rules. By formalizing optimizations this way, we enable to apply them systematically by an optimizing compiler, instead of applying them ad-hoc using imprecise rules of thumb. Moreover, we are able to prove that applying optimizations does not change the semantics of programs.

Previous work [58, 60] already showed how a formal rewrite system can be used to systematically apply optimizations to achieve high performance.

HIGH PERFOMANCE CODE GENERATION High-level languages like Accelerate [44], Delite [64], StreamIt [66] or Halide [52] aim to simplify the programming of GPUs by providing several parallel patterns or algorithmic skeletons. The semantics of algorithmic skeletons offer unique opportunities for compilers to optimize a given program. However, all of these approaches are compiled to low-level loop based code at an early stage of the compilation process. Using a functional data parallel IL (Intermediate Language) prevents us from losing this information which is used multiple times in our code generation process.

1.4 CONTRIBUTIONS

This thesis makes the following three main contributions:

A NEW HIGH-LEVEL APPROACH TO PROGRAM STENCIL COMPUTATIONS We extend the LIFT framework with two new high-level primitives *slide* and *pad*, inspired by decomposing the algorithmic stencil pattern into its fundamental algorithmic building blocks. Multidimensional stencil computations are then defined as compositions of LIFT's primitives.

FORMALIZATION OF OPTIMIZATIONS FOR STENCIL COMPUTATIONS We express well-known stencil optimizations by using and extending LIFT's IL. Two new low-level primitives, *mapSeqUnroll* and *increase*, enable the specification of stencil-specific optimizations using our functional approach This enables the systematic (and in the future possibly automatic) application of stencil-specific optimizations instead of optimizing applications ad-hoc using imprecise rules of thumb.

GENERATION OF HIGH-PERFORMANCE STENCIL CODE FOR GPUS

By improving LIFT's existing OpenCL code generator, we are able to generate high performance stencil kernels that are competitive with hand-tuned stencil code from Nvidia.

2

EXPRESSING STENCIL COMPUTATIONS USING HIGH-LEVEL FUNCTIONAL PRIMITIVES

Using a functional programming language to express computations allows to exploit unique opportunities when generating code for high performance applications. Our goal is to design a language consisting of a few small but powerful built-in primitives. By the power of function composition, we are able to express a wide range of applications. These functional programs tend to be short and have advantages compared to programs written in conventional low-level imperative programming languages (for example like OpenCL C) as already mentioned in [3]. These programs are hierarchically structured while containing simple primitives combined by function composition and do not contain loops nor control statements. One of the most important advantages compared to imperative programs is that functional programs do not unnecessarily constrain the order of computations. This allows to execute parts of a functional program in parallel. Thus, they are inherently parallel which perfectly fits high-performance computing and makes them a natural candidate to express stencil computations. Finally, using a formal set of rewrite rules, functional programs can be rewritten while provably preserving the semantics into more efficient programs as we observe in the following chapters. LIFT uses the old and well-known ideas of expressing and rewriting computations using functional primitives already studied by Backus [3], Bird [7] and Cole [15], to generate code for modern accelerators like GPUs.

The remainder of this chapter is structured as follows. We begin by introducing the formalism used in this thesis and formally define stencil computations. Afterwards, we introduce how we decompose the stencil pattern into its fundamental algorithmic building blocks using the formal definition of stencil computations. These fundamental building blocks are implemented as new functional high-level primitives in the LIFT framework. We then show how to compose and nest the new primitives with existing ones to express arbitrary and multidimensional stencil computations. We focus on one and two dimensional stencil computations in this thesis although the primitives might also be used to express higher dimensional stencil computations using the techniques introduced in the following sections. We express the stencil pattern as a composition of the fundamental algorithmic building blocks. Using the advantages of a functional programming language, we are able to define a high-level `stencil` function composed of built-in primitives. Instead of manually com-

posing these primitives, a programmer can use these high-level functions that offer the same level of abstraction as existing high-level approaches.

NOTATION We use the same notation as in [57] which is similar to the Bird-Meertens formalism [8] to define the semantics of our primitives. Specifically this means that we write function application using a space between the function and its argument: $f\ x$. Functions are curried and function application is left associative i. e. $f\ x\ y$ means $(f\ x)\ y$. For an array xs with n elements x_i we write $[x_1, \ldots, x_n]$. To increase readability in the examples, we sometimes write $[abcdef]$ instead of $[a, \ldots, f]$, denoting an array with six elements. For sequential function composition we use the $\circ$ operator, e. g. $(f \circ g)\ x = f(g\ x)$. We are especially interested in nesting and composing high-level primitives to express more complex computations. In order to define how primitives can be combined, we specify the *type* for each primitive: We write $e : \sigma$ to denote that expression e has the type σ. Functions that map elements from type α to type β are denoted as: $\alpha \to \beta$. Since functions are curried, we write $f : \alpha \to \beta \to \gamma$ for the type of a function that takes elements of type α and β and produces an element of type γ. Tuple types are denoted as $\langle \alpha, \beta \rangle$ and the type of an array with elements of type α and length n is written as $[\alpha]_n$.

2.1 FORMAL DEFINITION OF STENCIL COMPUTATIONS

In this section, we formally define stencil computations to observe their fundamental algorithmic parts. Our goal is to express stencil computations using LIFT's high-level primitives. We formally define a one dimensional stencil computation [57]:

DEFINITION 2.1: *Let xs be an array of size n with elements x_i where $0 < i \leqslant n$. Let f be a unary function, l and r be positive integer values, and b be an out-of-bound handling function. A stencil computation on array xs is formally defined as follows:*

$$\begin{aligned}
&\textit{stencil}\ f\ l\ r\ b\ [x_1, x_2, \ldots, x_n] \overset{def}{=} [y_1, y_2, \ldots, y_n] \\
&\textit{where} \\
&\qquad y_i = f\ [x_{i-l}, \ldots, x_{i+r}] \qquad \forall\, i : 0 < i \leqslant n \\
&\textit{and} \\
&\qquad x_j = b\ j \qquad \forall\, j : -l < j \leqslant 0 \ \vee\ n < j \leqslant n + r
\end{aligned}$$

The *stencil* function f is applied to a neighborhood of size $l + r + 1$ for every element of the input xs.

As an example, we examine how to express a 17-point convolution with a convolution kernel ws containing the weights, and a boundary function b that returns the border values on out-of-bound accesses:

EXAMPLE 2.1:

$$\begin{aligned}&\mathtt{convolution17\ b\ ws\ xs} \stackrel{def}{=} \mathtt{stencil\ (f\ ws)\ 8\ 8\ b\ xs}\\&\textit{where}\\&\mathtt{f}\ [w_1,\ldots,w_{17}]\ [x_1,\ldots,x_{17}] = w_1\cdot x_1+\cdots+w_{17}\cdot x_{17}\end{aligned}$$

Since we define multidimensional data structures as nested arrays, we use the terms *array of arrays* and *matrix* interchangeably. Thus, instead of writing

$$[[x_{1,1},\ldots,x_{1,m}],\ldots,[x_{n,1},\ldots,x_{n,m}]]$$

we may also write

$$\begin{bmatrix}[x_{1,1} & \cdots & x_{1,m}]\\ \vdots & & \\ [x_{n,1} & \cdots & x_{n,m}]\end{bmatrix} \text{ or } \begin{bmatrix}x_{1,1} & \cdots & x_{1,m}\\ \vdots & & \vdots\\ x_{n,1} & \cdots & x_{n,m}\end{bmatrix}$$

The formal definition of a stencil computation on matrices similar to Definition 2.1 and [57] is given in the following:

DEFINITION 2.2: *Let* $\mathtt{xs}$ *be an array of size* m *whose elements are arrays of size* n*. Let* $\mathtt{f}$ *be a unary function and* $\mathtt{h}$ *be a function specifying the boundary handling. Let* $\mathtt{t}$, $\mathtt{b}$, $\mathtt{l}$ *and* $\mathtt{r}$ *be four positive integer values defining the stencil shape in two dimensions. Let* $u = \mathtt{l}+\mathtt{r}+1$ *and* $v = \mathtt{t}+\mathtt{b}+1$*. A stencil computation on matrices is then defined as follows:*

$$\textit{stencil2d}\ \mathtt{f\ t\ b\ l\ r\ h}\ \begin{bmatrix}x_{1,1} & \cdots & x_{1,m}\\ \vdots & & \vdots\\ x_{n,1} & \cdots & x_{n,m}\end{bmatrix} \stackrel{def}{=} \begin{bmatrix}y_{1,1} & \cdots & y_{1,m}\\ \vdots & & \vdots\\ y_{n,1} & \cdots & y_{n,m}\end{bmatrix}$$

where

$$y_{i,j} = \mathtt{f}\begin{bmatrix}x_{i-l,j-t} & \cdots & x_{i+r,j-t}\\ \vdots & & \vdots\\ x_{i-l,j+b} & \cdots & x_{i+r,j+b}\end{bmatrix}$$

$$\forall i,j:\ 0<i\leqslant n \quad 0<i\leqslant m$$

and

$$\begin{aligned}&x_{i,j} = \mathtt{h\ i\ j}\\&\quad \forall i: -l<i\leqslant 0\vee n<i\leqslant n+r,\\&\quad \forall j: -t<j\leqslant 0\vee m<j\leqslant m+b,\end{aligned}$$

As another example we define a two dimensional convolution in terms of this stencil function. Let $\mathtt{b}$ be an arbitrary boundary function. A 17×17 convolution as in [51] can then be defined as follows:

EXAMPLE 2.2:

$$\mathtt{conv17x17\ b\ ws\ xs = stencil2d\ (f\ ws)\ 8\ 8\ 8\ 8\ b\ xs}$$

where f *is a function that pairwise multiplies all elements of the neighborhoods and its weights and sums up the results.*

In the following section, we analyze stencil computations using these definitions and extract their fundamental parts as primitives.

2.2 HIGH-LEVEL ALGORITHMIC FUNCTIONAL PRIMITIVES

We express stencil computations by extending and using LIFT's high-level primitives [60] which are defined as higher order functions. Higher order functions are a well-known concept in functional programming which describes functions that take one or more functions as arguments, so called *procedural parameters* or *customizing functions*. In fact, most of these high-level primitives, like *map* or *reduce* already exist in functional programming languages.

Following the ideology of LIFT's high-level primitives, we want to decompose the algorithmic structure of stencil computations and express these using the most basic fundamental algorithmic building blocks. Therefore, instead of defining a single high-level stencil primitive, we express stencil computations by composing small generic primitives. Looking at the formal definitions of a stencil computations given in Definition 2.1 and Definition 2.2, we can see that algorithmically, a stencil computation consists of three fundamental parts that are specified as arguments to the different stencil functions:

A. For every element, consider a neighborhood of a given size (specified by the shape of the stencil which is specified by parameters l and r in Definition 2.1)

B. Apply a function to every neighborhood to compute a single output element (specified by the stencil function f in Definition 2.1)

C. In case of border elements apply boundary handling using a given function (specified by parameter b in Definition 2.1)

Hence, we use three high-level primitives that each captures exactly one of these algorithmic parts. Part *B*, *apply a function to every ...*, might already sound familiar to a functional programmer. This is exactly what the *map* primitive does which we formally introduce shortly. Furthermore, we introduce two new primitives which capture the other two parts *A* and *C*.

Decomposing stencil computations allows us to focus on each fundamental part separately instead of providing a solution that covers all parts at once. By composition we are able to combine these building blocks to express stencil computations in arbitrary dimensions. In the course of this chapter we define several functions composed of our built-in high-level primitives. These allow to avoid repeating often occurring combinations of specific primitives. We consider these

functions as *zero-cost abstractions* because they raise the abstraction level to the same level of existing library approaches without providing a fixed solution for specific dimensions. Thus, without being a built-int primitive of the functional language.

By defining simple and stencil independent primitives, we are able to use them in various ways. For example to express optimizations as we will see in the following chapters. Moreover, they might also be used to express computations of different domains for example convolution networks used in machine learning.

In the following, we give formal definitions for a subset of the already existing high-level primitives which we use in the course of this thesis to express stencil computations. We use the definitions given in [57]:

MAP *map* is a well-known element in many functional programming languages. The *map* primitives applies a given unary function f to every element of an array. We formally define *map* as follows:

DEFINITION 2.3: *Let* xs *be an array of size* n *with elements* x_i *where* $0 < i \leqslant n$. *Let* f *be a unary customizing function defined on elements. The map primitive is then defined as follows:*

$$map\ f\ [x_1, x_2, \ldots, x_n] \stackrel{def}{=} [f\ x_1, f\ x_2, \ldots, f\ x_n]$$

The type of map is defined as follows:

$$map : (\alpha \to \beta) \to [\alpha]_n \to [\beta]_n$$

REDUCE The *reduce* primitive, also known as *fold* or *accumulate*, combines every element of a given array using a binary operator. We formally define *reduce* as follows:

DEFINITION 2.4: *Let* xs *be an array of size* n *with elements* x_i *where* $0 < i \leqslant n$. *Let* $\oplus$ *be an associative and commutative binary customizing operator with the identity element* $id_\oplus$. *The reduce primitive is then defined as follows:*

$$reduce\ (\oplus)\ id_\oplus\ [x_1, x_2, \ldots, x_n] \stackrel{def}{=} [id_\oplus \oplus x_1 \oplus x_2 \oplus \cdots \oplus x_n]$$

The type of reduce is defined as follows:

$$reduce : (\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha]_n \to [\alpha]_1$$

Note that we define the *reduce* primitive to return an array containing a single element instead of the element itself.

ZIP The *zip* primitive transforms the shape of the data. Given multiple arrays of the same length, it creates a single array of tuples. We formally define *zip* for two input arrays as follows:

DEFINITION 2.5: *Let* xs *and* ys *be arrays of size* n *with elements* x_i *and* y_i *where* $0 < i \leqslant n$. *The zip primitive is then defined as follows:*

$$zip\ [x_1, x_2, \ldots, x_n]\ [y_1, y_2, \ldots, y_n] \stackrel{def}{=} [\langle x_1, y_1\rangle, \langle x_2, y_2\rangle, \ldots, \langle x_n, y_n\rangle]$$

The type of zip is defined as follows:

$$zip : [\alpha]_n \to [\beta]_n \to [\langle \alpha, \beta\rangle]_n$$

Extending the definition for more than two input arrays is obvious.

SPLIT AND JOIN The *split* and *join* primitives increase and decrease the dimension of a given array respectively. We start by defining the *split* primitive which divides a given array in an array of chunks: We formally define *split* as follows:

DEFINITION 2.6: *Let* xs *be an array of size* m *with elements* x_i *where* $0 < i \leqslant m$. *Let* n *be an integer value such that* m *is evenly divisible by* n. *The split primitive is then defined as follows:*

$$split\ n\ [x_1, x_2, \ldots, x_m] \stackrel{def}{=} [[x_1, \ldots, x_n], [x_{n+1}, \ldots, x_{2n}], \ldots, [x_{m-n+1}, \ldots, x_m]]$$

The type of split is defined as follows:

$$split : (n : int) \to [\alpha]_m \to [[\alpha]_n]_{\frac{m}{n}}$$

The corresponding *join* primitive does exactly the opposite by flattening a given array:

DEFINITION 2.7: *Let* xs *be an array of size* $\frac{m}{n}$ *whose elements are arrays of size* n. *We denote the elements of the* i*th inner array as* $x_{((i-1)\times n)+j}$ *where* $0 < i \leqslant \frac{m}{n}$ *and* $0 < j \leqslant n$. *The join primitive is then defined as follows:*

$$join\ [[x_1, \ldots, x_n], [x_{n+1}, \ldots, x_{2n}], \ldots, [x_{m-n+1}, \ldots, x_m]] \stackrel{def}{=} [[x_1, x_2, \ldots, x_m]$$

The type of join is defined as follows:

$$join : [[\alpha]_n]_{\frac{m}{n}} \to [\alpha]_m$$

REORDER The *reorder* primitive changes the order of elements in a given array using a specific permutation. We formally define *reorder* as follows:

DEFINITION 2.8: *Let* xs *be an array of size* n *with elements* x_i *where* $0 < i \leqslant n$. *Let* σ *be an arbitrary permutation of* $[1, \ldots, n]$. *The reorder primitive is then defined as follows:*

$$reorder\ \sigma\ [x_1, x_2, \ldots, x_n] \stackrel{def}{=} [x_{\sigma(1)}, \ldots, x_{\sigma(n)}]$$

The type of reorder is defined as follows:

$$\mathit{reorder} : (\mathtt{int} \to \mathtt{int}) \to [\alpha]_n \to [\alpha]_n$$

TRANSPOSE *transpose* is the first example of a *defined* function which is not a built-in primitive, but a function composed of primitives. It transposes a given matrix by flattening it, reordering the elements using a transposition permutation and splitting it to create a matrix again. The *transpose* function is defined as follows:

DEFINITION 2.9: *Let* xs *be an array of size* m *whose elements are arrays of size* n*. The transpose function is then defined as follows:*

$$\mathit{transpose} \overset{def}{=} \mathit{split}\ n \circ \mathit{reorder}\ \sigma_{\mathit{transpose}} \circ \mathit{join}$$

The type of transpose is as follows:

$$\mathit{tranpose} : [[\alpha]_n]_m \to [[\alpha]_m]_n$$

In the following, we introduce two new high-level primitives *slide* and *pad* in more detail.

2.2.1 *Creating Neighborhoods Using the Slide Primitive*

We introduce the *slide* primitive to LIFT which is used to create an array of neighborhoods for a given array. It corresponds to the fundamental part *A* of stencil computations identified in the beginning of Section 2.2. Neighborhoods are created by sliding a window of a specific size and step over a given array. This results in an extra dimension in the output array.

DEFINITION 2.10: *Let* xs *be an array of size* m *with elements* x_i *where* $0 < i \leqslant m$*. Let* n *and* s *be integer values such that*

$$(m - n + s) \bmod s = 0 \tag{1}$$

The slide *primitive is then defined as follows:*

$$\mathit{slide}\ n\ s\ [x_1, x_2, ..., x_m] \overset{def}{=} [Y_1, Y_2, ..., Y_k]$$

where

$$Y_i = [y_{i,1}, y_{i,2}, ..., y_{i,n}]$$

$$y_{i,j} = x_{(i-1)s+j}$$

and

$$k = \frac{m - n + s}{s} \tag{2}$$

The type of slide *is defined as follows:*

$$\mathit{slide} : (n : \mathtt{int}) \to (s : \mathtt{int}) \to [\alpha]_m \to [[\alpha]_n]_{(\frac{m-n+s}{s})}$$

To illustrate the behavior of the *slide* primitive, consider the following example. We want to group each element of the input array with its left and right neighbor. Thus, we want to create neighborhoods for a one dimensional 3-point stencil, which we express using the *slide* primitive.

EXAMPLE 2.3:

$$slide\ 3\ 1\ [abcdefg] = [[abc][bcd][cde][def][efg]]$$

For a 3-point stencil, the size n of the sliding window has to be 3. The step s with which the window is moved over the array has to be 1 because we want to create neighborhoods for every element. However, note that there is no neighborhood for elements a or g, hence, no window where a or g are the center element. This is because they do not have a left or right neighbor. Therefore, the size of the outermost array has decreased compared to the size of the input exactly as defined in Equation 2 of Definition 2.10: $k = (7 - 3 + 1)/1 = 5$. Obviously, when thinking about stencil computations, we want to create neighborhoods for border elements as well. We introduce means to do this later in this chapter. However, the *slide* primitive is defined to place its first window at the beginning of its input. Thus, the first element of the input array always is the first element of the first window as depicted in Example 2.3. Note that Equation 1 restricts the semantics of the *slide* primitive such that the last element of the input has to be the last element in the last window as well. This way, we ensure that every window has the same size specified by parameter n. Thus, the expression $slide\ 2\ 2\ [abcdefg]$ is not defined. Note that when n equals s and the input size is evenly divisible by n, *slide* has the same semantics as the *split* primitive introduced earlier:

$$slide\ 2\ 2\ [abcdef] = split\ 2\ [abcdef] = [[ab][cd][ef]]$$

Finally, we look at two other examples of how to use the *slide* primitive:

EXAMPLE 2.4:

$$\begin{aligned} slide\ 1\ 2\ [abcdefg] &= [[a][c][e][g]] \\ slide\ 4\ 2\ [abcdef] &= [[abcd][cdef]] \end{aligned}$$

If the step s exceeds the size n, as depicted in the first line of Example 2.4, the resulting array does not contain every element of the input. The second example shows how to create overlapping windows with a step that is bigger than one. We use the *slide* primitive this way to express tiling optimizations, discussed in the next chapter.

To summarize, *slide* is used to shape an array in different ways: If the step exceeds the size, it creates windows with a gap in between. If the step equals the size it behaves like the *split* primitive and evenly divides an array into chunks. And finally, if the size exceeds the step it creates an array of overlapping windows.

2.2.2 *Boundary Handling Using the Pad Primitive*

Now that we specified means to generate an array of neighborhoods using the *slide* primitive, we define another primitive which we use for the boundary handling in stencil computations. We already defined *slide* expressing the fundamental part *A* of stencil computations as identified in the beginning of Section 2.2 and are able to use *map* to express part *B*. In this section, we introduce a new primitive expression the last fundamental part *C* of stencil computations. Intuitively, we want to append the elements that are required to create neighborhoods for border elements to the input array. Thus, before creating an array of neighborhoods, we want to add extra elements on both sides of the array. The elements that are added are defined by the boundary handling of the specific stencil application. This way, we transform the input array which lacks neighbors for the outermost elements, to an array where each element of the original array has its required neighbors next to it.

The *pad* primitive adds elements to the left and right end of a given array. Thus, it increases the size of the input array. It is formally defined as follows:

DEFINITION 2.11: *Let* xs *be an array of size* n *with elements* x_i *where* $0 \leqslant i < n$. *Let* l *and* r *be positive integer values and* b *a binary function defined on integers. The pad primitive is then defined as follows:*

$$\begin{aligned} &\mathit{pad}\ l\ r\ b\ [x_0, x_1, \ldots, x_{n-1}] \stackrel{\mathit{def}}{=} \\ &[y_{-l-1}, \ldots, y_{-1},\ x_0, \ldots, x_{n-1},\ y_n, \ldots, y_{n-1+r}] \end{aligned}$$

where

$$y_i = x_{b(i,n)}$$

The type of pad is as follows:

$$\mathit{pad} : (l : \mathtt{int}) \to (r : \mathtt{int}) \to (\mathtt{int} \to \mathtt{int} \to \mathtt{int}) \to [\alpha]_n \to [\alpha]_{(n+l+r)}$$

The first two arguments specify the amount of elements to be added on both sides. The boundary function b is used to determine which elements are added. Note that elements that are added to the array are always elements of the input array. Specifically, the boundary function maps indices which would exceed the size of the input array to in-bound indices. Therefore, this definition does not support to *pad* constant values to a given array. Three popular boundary functions are: `clamp`, `mirror` and *wrap*. `clamp` repeats the left- and rightmost element. *wrap* adds elements from the opposite side of the input array which mimics a circular buffer in case of a one dimensional input. Finally, `mirror` repeats elements in reverse order of the corresponding side.

To illustrate how the different boundary functions affect the result of applying the *pad* primitive, we look at an example. Here, we pad a

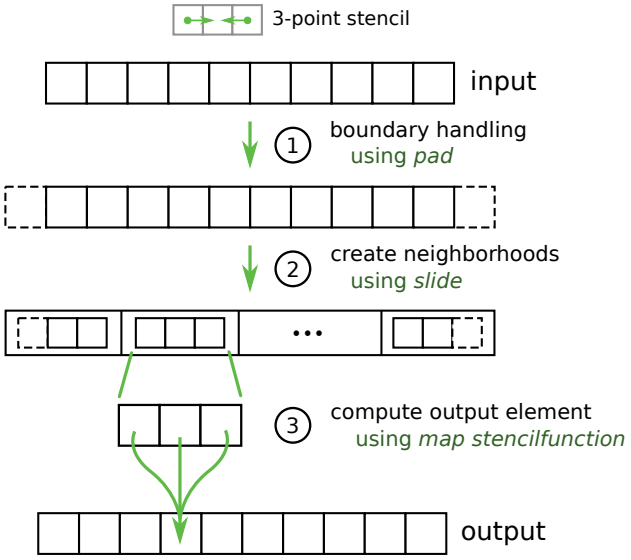


Figure 3: Conceptual structure of a functional program describing a stencil computation

given input array with an extra element on the left-hand side and two elements on the right-hand side by using all three different boundary functions:

EXAMPLE 2.5:

$$\begin{aligned} &\textit{pad}\ 1\ 2\ \mathrm{clamp} && [abcdefg] = [aabcdefggg] \\ &\textit{pad}\ 1\ 2\ \mathrm{mirror} && [abcdefg] = [aabcdefggf] \\ &\textit{pad}\ 1\ 2\ \mathit{wrap} && [abcdefg] = [gabcdefgab] \end{aligned}$$

These boundary functions are defined as follows:

DEFINITION 2.12:

$$\mathrm{clamp}\ i\ n = \begin{cases} 0 & \textit{if}\ i < 0 \\ n-1 & \textit{if}\ i \geqslant n \\ i & \textit{otherwise} \end{cases}$$

$$\mathrm{mirror}\ i\ n = \begin{cases} -1-i & \textit{if}\ i < 0 \\ 2n-i-1 & \textit{if}\ i \geqslant n \\ i & \textit{otherwise} \end{cases}$$

$$\mathit{wrap}\ i\ n = ((i \bmod n) + n) \bmod n$$

2.2.3 *Stencils as Composition of Pad, Slide and Map*

Using *pad*, *slide* and *map*, we are able to express stencil computations on one dimensional arrays. Each of these primitives corresponds to one of the fundamental parts of stencil computations identified in the

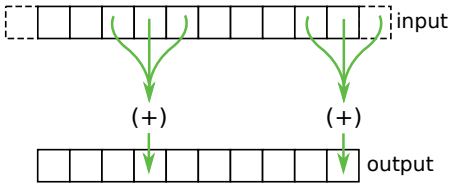


Figure 4: 3-point Jacobi stencil illustration

beginning of Section 2.2. The functional program which describes the stencil computation can always be separated into three main parts as depicted in Figure 3:

1. *Boundary Handling* - The first part of the functional program describes the boundary handling for border elements of the input. This depends on the application and is specified using the *pad* primitive corresponding to part *C*.

2. *Create Neighborhoods* - The second part of the functional program describes how to gather all elements required by a given stencil. We use the *slide* primitive to create neighborhoods for the stencil corresponding to part *A*.

3. *Apply Stencil Computation* - The last part of the functional program describes the application of the computation and corresponds to part *B*. The stencil function is executed for all neighborhoods which were created before. It might be built of high-level primitives itself and describes how to compute a single output element for a given neighborhood.

Figure 3 depicts a 3-point stencil computation. Starting from the top, we add elements on the left and right-hand side of the input using the *pad* primitive. Afterwards, we create neighborhoods using the *slide* primitive and apply the stencil function to all neighborhoods using the *map* primitive. This results in the following structure:

$$\underbrace{\mathit{map}\ \texttt{stencilFunction}}_{\text{apply computation}} \circ \underbrace{\mathit{slide}\ \texttt{n}\ \texttt{s}}_{\text{create neighborhood}} \circ \underbrace{\mathit{pad}\ \texttt{l}\ \texttt{r}\ \texttt{b}}_{\text{boundary handling}}$$

Note that the structure of the functional program exactly matches the algorithmic structure of a stencil computation as identified in the introduction of Section 2.2. Now, we examine how to use our high-level primitives to express some examples of stencil computations functionally.

2.2.3.1 *3-Point Jacobi*

To begin with, consider a simple example of applying a 3-point Jacobi stencil which sums up all elements as illustrated in Figure 4. For boundary handling, we use the `clamp` function to repeat the border elements. This stencil is expressed using the new primitives:

EXAMPLE 2.6 (3 POINT JACOBI):

$$\texttt{3PointJacobi} \stackrel{def}{=} \mathit{map}\ \texttt{f} \circ\ \mathit{slide}\ 3\ 1 \circ\ \mathit{pad}\ 1\ 1\ \texttt{clamp}$$

where the stencil function f *is defined as*

$$\texttt{f} = \mathit{reduce}\ (+)\ 0$$

Now consider to apply this function to the array $[1,2,3,4,5]$*:*

$$\begin{aligned}
&(\mathit{map}\ \texttt{f} \circ\ \mathit{slide}\ 3\ 1 \circ\ \mathit{pad}\ 1\ 1\ \texttt{clamp})\ [1,2,3,4,5] \\
&= (\mathit{map}\ \texttt{f} \circ\ \mathit{slide}\ 3\ 1)\ [1,1,2,3,4,5,5] \\
&= \mathit{map}\ \texttt{f} \begin{bmatrix} [1,1,2] \\ [1,2,3] \\ [2,3,4] \\ [3,4,5] \\ [4,5,5] \end{bmatrix} = \begin{bmatrix} \texttt{f}\ [1,1,2] \\ \texttt{f}\ [1,2,3] \\ \texttt{f}\ [2,3,4] \\ \texttt{f}\ [3,4,5] \\ \texttt{f}\ [4,5,5] \end{bmatrix} = [4,6,9,12,14]
\end{aligned}$$

2.2.3.2 *1D Heat Diffusion*

As another example, we consider the computation of a one dimensional heat flow simulation as explained in [61] and [69]. The distribution of heat in an object e. g. a ring or a bar is described by a parabolic Partial Differential Equation (PDE). To calculate the variation of temperature over time, a linear combination is built of the current temperature at a given point and its surrounding temperatures. To compute the temperature for the next time step at a given point in a one dimensional object, we take the left and right temperature values into account. The equation to calculate the temperature for timestep $k+1$ at position i for a given diffusion coefficient c is given by

$$\begin{aligned}
T_{k+1,i} &= T_{k,i} + c \cdot (T_{k,i-1} - 2T_{k,i} + T_{k,i+1}) \\
&= c \cdot T_{k,i-1} + (1-2c) \cdot T_{k,i} + c \cdot T_{k,i+1}
\end{aligned}$$

Therefore, we need to compute a weighted average of the current and surrounding temperatures using the given diffusion coefficients. Let us assume the coefficients are themselves stored in a one dimensional array $cs = [c,\ 1-2c,\ c]$ and the current temperatures are stored in an array $ts = [t_1, \ldots, t_n]$. The following example illustrates how to express the 1D heat diffusion computation using our high-level primitives:

EXAMPLE 2.7:

$$\texttt{heatDiff cs ts} \stackrel{def}{=} (\mathit{map}\ (\texttt{heat cs}) \circ\ \mathit{slide}\ 3\ 1 \circ\ \mathit{pad}\ 1\ 1\ \texttt{clamp})\ \texttt{ts}$$

where the stencil function heat *is defined as*

$$\texttt{heat cs nbh} = (\mathit{reduce}\ (+)\ 0 \circ \mathit{map}\ (*) \circ \mathit{zip})\ \texttt{cs nbh}$$

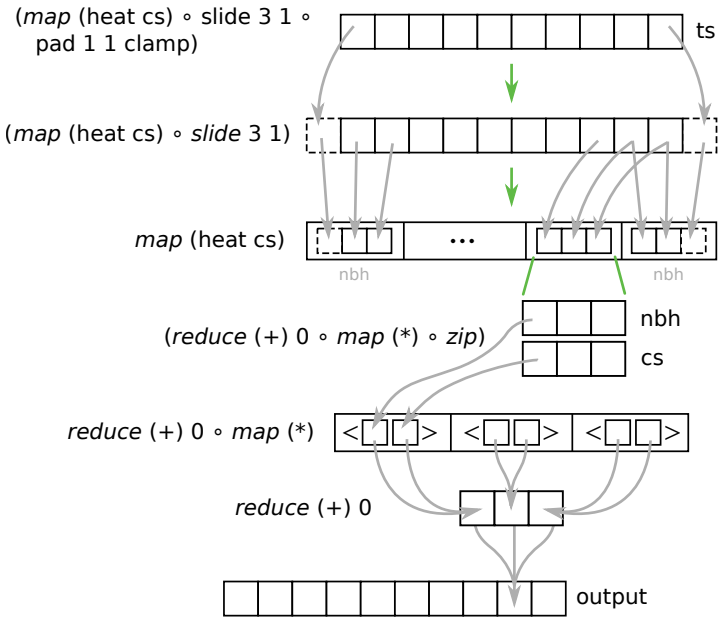


Figure 5: Computation steps for a high-level expression describing a one dimensional heat diffusion application

Figure 5 visualizes the computation of Example 2.7 by showing the intermediate results after evaluating each primitive. Starting from the top, we use *pad* and *slide* to create a 3-point neighborhood for every element of the input array. The `heat` function is mapped to every neighborhood and its evaluation is shown for a single neighborhood. First, the neighborhood (`nbh`) and the convolution kernel (`cs`) are fused to an array of tuples using *zip*. Second, both elements of every tuple are multiplied using *map* and the results are added using *reduce* to compute the single output element.

2.2.3.3 *1D Convolution*

The heat diffusion is a concrete example for a general class of stencil computations called *convolution*. We already briefly discussed convolutions in the previous chapter and formally define them in this section. Convolutions are used in a wide range of applications for example in machine learning or image processing. In one dimension, the convolution operator $*$ is defined as in [38]:

DEFINITION 2.13: *Let Ω be an array of size n and r be a positive integer value. Let K be an array of size $2r+1$. The one dimensional convolution operator is then defined as:*

$$(\Omega * K)(x) = \sum_{i=-r}^{r} \Omega(x+i) \cdot K(i)$$

Ω denotes the grid which is updated during the stencil computation (xs in Example 2.7). K denotes the *convolution kernel* which is a

small array that contains the weights or coefficients (cs in the same example). Finally, r denotes the radius of the stencil shape, thus, for a typical three point stencil r equals 1 because the stencil takes one neighbor to the left and one to the right into account. We generalize the heat diffusion example by defining a convolution1d function that uses our high-level primitives:

DEFINITION 2.14: *Let* Ω *be an array of length* n *and* r *be a positive integer denoting the radius of the stencil. Let* K *be an array of length* $2r+1$ *denoting the convolution kernel. The* convolution1d *function is defined as follows:*

$$\begin{aligned}&\text{convolution1d } b\ K\ \Omega = \Omega\ *_b\ K \stackrel{def}{=}\\ &\quad (\textit{map}\ (\text{conv } K) \circ\ \textit{slide}\ (2r+1)\ 1 \circ\ \textit{pad}\ r\ r\ b)\ \Omega\\ &\textit{where}\\ &\quad \text{conv } K\ nbh = (\textit{reduce}\ (+)\ 0 \circ \textit{map}\ (*) \circ \textit{zip})\ K\ nbh\end{aligned}$$

where $*_b$ *is the convolution operator using boundary function* b.

Note that we do not have to introduce a new primitive to express convolutions. Instead we are able to use our existing primitives to define a convolution1d function composed of primitives. Now we can use the new convolution1d function to define the heat diffusion from the previous section:

EXAMPLE 2.8:

$$\text{heatDiffusion } c\ xs \stackrel{def}{=} \text{convolution1d clamp } [c,\ (1-2c),\ c]\ xs$$

A generic 17-point convolution can also be easily expressed by providing an array containing 17 weights to the convolution1d function. For example:

$$\text{convolution1d mirror } [w_1, \ldots, w_{17}]\ xs$$

2.3 MULTIDIMENSIONAL STENCILS

One dimensional stencil computations are of limited use in real life applications. Therefore, we are especially interested in how to express multidimensional stencil computations, i. e., stencil computations operating on higher dimensional data structures like matrices or cubes. In this section, we focus on two dimensional stencil computations and explain how to use our high-level primitives to express them without introducing new specialized primitives that are defined on higher dimensional data structures. By applying the same techniques which we use to express 2D stencil computations, one could also use our high-level primitives to express ever higher dimensional stencil computations.

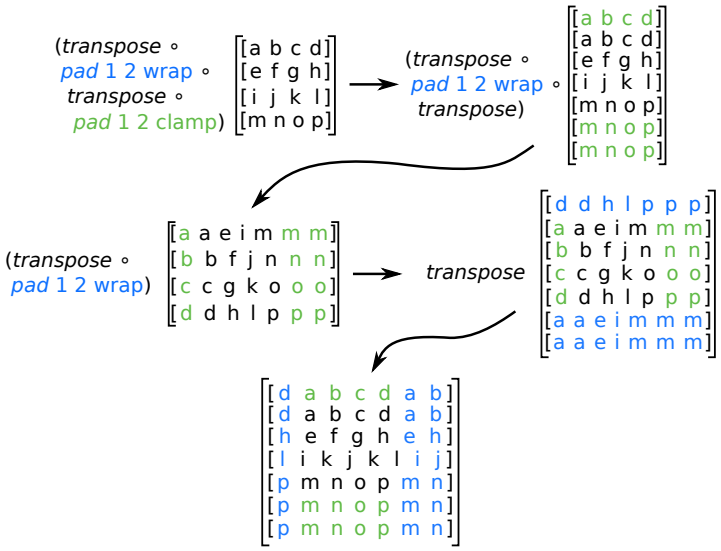


Figure 6: Padding a matrix using *pad* and *transpose*

To be able to express multidimensional stencil computations, we need to express each fundamental part of stencil computations identified in Section 2.2 using our high-level primitives. Therefore, we discuss how to express boundary handling and neighborhood creation in two dimensions and how to apply the stencil function to these two dimensional neighborhoods. The exact same strategies can be used to express stencil in arbitrary dimensions which emphasizes the power of composing fundamental building blocks.

2.3.1 *Two Dimensional Boundary Handling Using Pad*

We start by examining how to pad a matrix in both dimensions using the *pad* primitive which is defined on arrays. As an example we pad a two dimensional array of size 4×4 in every dimension as depicted in Figure 6. In this example, we add an extra row on top, an extra column on the left-hand side while adding two rows at the bottom and two columns at the right-hand side. Moreover, we use two different boundary functions in each dimension to emphasize the flexibility obtained by using a compositional approach.

The *pad* primitive is defined to extend its input array by repeating elements on each side. Since we represent a matrix as an array of arrays, the elements which the *pad* primitive repeats are rows of the given matrix, thus arrays itself. In the first step depicted in Figure 6, the first row is added on top of the matrix and the last row is added twice on the bottom of the matrix. By transposing the matrix, we convert rows into columns and vice versa. By applying the *pad* primitive again we pad the columns of the original matrix. Since both dimensions are now padded we only need to transpose the matrix again to restore the original orientation.

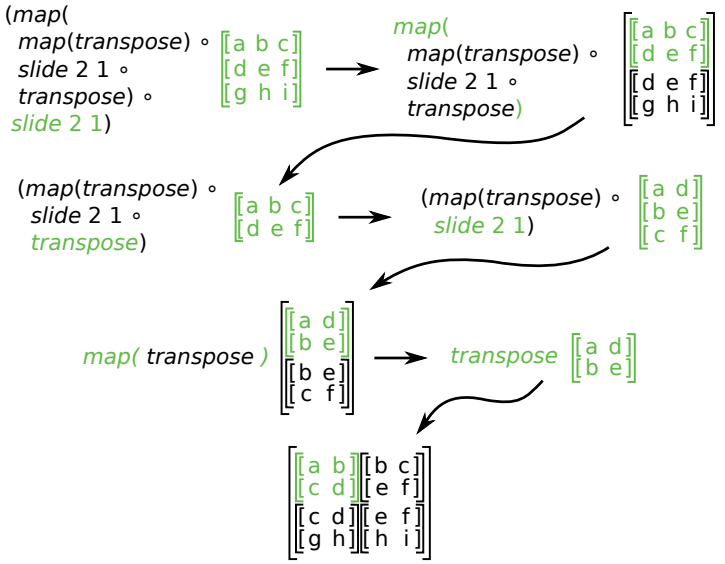


Figure 7: Using *slide* to create neighborhoods for a 2×2 4-point stencil

To increase readability, we define a function *pad2d* which we use in the following examples.

DEFINITION 2.15: *Let xs be an array of size* m *whose elements are arrays of size* n*. Let* top, bottom, left, right *be positive integer values and* b_1 *and* b_2 *two boundary functions. The pad2d function is then defined as:*

$$\begin{aligned} &\mathit{pad2d}\ \mathtt{top}\ \mathtt{bottom}\ \mathtt{left}\ \mathtt{right}\ b_1\ b_2 \stackrel{def}{=} \\ &\quad \mathit{transpose} \circ \mathit{pad}\ \mathtt{left}\ \mathtt{right}\ b_2 \circ \\ &\quad\quad \mathit{transpose} \circ \mathit{pad}\ \mathtt{top}\ \mathtt{bottom}\ b_1 \end{aligned}$$

If top = bottom, left = right *and* $b_1 = b_2$*, we also write*

$$\mathit{pad2d}\ \mathtt{top}\ \mathtt{left}\ b_1\ xs$$

instead of

$$\mathit{pad2d}\ \mathtt{top}\ \mathtt{top}\ \mathtt{left}\ \mathtt{left}\ b_1\ b_1\ xs$$

Note that *pad2d* is a defined function composed of primitives which allows us to avoid repetition instead of being a specialized built-in primitive for higher dimensional data structures.

2.3.2 *Creating Two Dimensional Neighborhoods Using Slide*

In this section, we explain how we create two dimensional neighborhoods using our high-level primitives. The creation of two dimensional neighborhoods in a matrix is depicted in Figure 7. We start in the top left corner. Similar to *pad*, the *slide* primitive works with elements of its input array independent of their type. Since these elements are now arrays itself, *slide* creates windows of rows as depicted

in the second step. After creating these windows, *map* is the next primitive to be applied. Although *map* applies a function to every element, we only depict the application to the first element for simplicity. In the subsequent steps, *slide* is applied to the transposed elements to gather neighboring columns. Finally, every element needs to be transposed again to restore the original orientation. The final result is a four dimensional array: a matrix which contains the two-dimensional neighborhoods.

Again, we define a function *slide2d* which allows to omit repeating this exact combination of high-level primitives:

DEFINITION 2.16: *Let xs be an array of size* m *whose elements are arrays of size* n*. Let* $size_X$, $step_X$, $size_Y$, $step_Y$ *be positive integer values. The slide2d function is then defined as:*

$$
\begin{aligned}
&\mathit{slide2d}\ size_y\ step_y\ size_x\ step_x\ xs \stackrel{def}{=} \\
&\quad \mathit{map}(\mathit{map}\ \mathit{transpose} \circ \\
&\qquad \mathit{slide}\ size_x\ step_x \circ \\
&\qquad\quad \mathit{transpose}) \circ \\
&\quad \mathit{slide}\ size_y\ step_y
\end{aligned}
$$

If $size_x = size_y$ *and* $step_x = step_y$*, we also write*

$$\mathit{slide2d}\ size_x\ step_x\ xs$$

instead of

$$\mathit{slide2d}\ size_x\ step_x\ size_x\ step_x\ xs$$

2.3.3 *Multidimensional Stencil Examples*

Now that we examined how to reuse our high-level primitives for two dimensional data structures by composing our high-level primitives, we examine how to express real life stencil applications using our high-level primitives. Again every example has the following structure:

$$\underbrace{\mathit{map}(\mathit{map}\ f)}_{\text{apply computation in 2D}} \circ \underbrace{\mathit{slide2d}\ n_y\ s_y\ n_x\ s_x}_{\text{create 2D neighborhoods}} \circ \underbrace{\mathit{pad2d}\ t\ b\ l\ r\ h}_{\text{2D boundary handling}}$$

Higher dimensional stencil computations are expressed in a similar fashion.

2.3.3.1 *9-Point Jacobi*

To begin with, consider a simple two-dimensional 9-point Jacobi stencil. To express this stencil, we use the new functions *slide2d* and *pad2d* and the `mirror` boundary condition

EXAMPLE 2.9 (9-POINT JACOBI):

$$\texttt{9PointJacobi} \stackrel{def}{=} \textit{map}(\textit{map}\ \texttt{f}) \circ \textit{slide2d}\ 3\ 1\ \circ \textit{pad2d}\ 1\ 1\ \texttt{mirror}$$

where the stencil function f *is defined as*

$$\texttt{f} \stackrel{def}{=} \textit{reduce}\ (+)\ 0 \circ \textit{join}$$

Let us examine why we define the stencil function using a *join* as the first primitive. Consider mapping the function to a single 9-point neighborhood:

EXAMPLE 2.10:

$$\begin{aligned}
&(\textit{reduce}\ (+)\ 0 \circ \textit{join}) \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \\
&= \textit{reduce}\ (+)\ 0\ [abcdefghi] \\
&= [0 + a + b + c + d + e + f + g + h + i]
\end{aligned}$$

As shown in the previous example, flattening the neighborhood using *join* allows to accumulate all values using a single *reduce* primitive. Another possibility would be to map a *reduce* onto every row and flatten the result afterwards. However, this is a slightly more complex stencil function computing the exact same result. Therefore, we flatten multidimensional neighborhoods in all following examples.

2.3.3.2 *Cellular Automaton*

To emphasize that the high-level primitives might also be used to express non-convolution stencil computations, we express a well-known cellular automaton example: *Conway's Game of Life*. Every element of a two dimensional grid can be in two states: It can either be alive, represented by a 1, or it can be dead represented by a 0. In this example, we express the *B2S23* version of this application. In this version, an element is *born* (its state changes from 0 to 1) if exactly two of its direct neighbors including diagonals are alive. An element survives if either two or three of its neighbors are alive. In all other cases the state changes from 1 to 0 or stays 0.

EXAMPLE 2.11 (GAME OF LIFE):

$$\texttt{GameOfLife} \stackrel{def}{=} \textit{map}(\textit{map}\ \texttt{evolution}) \circ \textit{slide2d}\ 3\ 1 \circ \textit{pad2d}\ 1\ 1\ \texttt{mirror}$$

where the evolution *is defined as*

$$\texttt{evolution} \stackrel{def}{=} \texttt{survive} \circ \textit{reduce}\ (+)\ 0 \circ \textit{join}$$

and survive *is defined as*

$$\texttt{survive}\ x \stackrel{def}{=} \begin{cases} 1 & \textit{if}\ 1 < x < 4 \\ 0 & \textit{otherwise} \end{cases}$$

This example emphasizes the flexibility of our functional approach to express stencil computations. The stencil-function that is mapped onto the neighborhoods is an arbitrary function which can compute anything as long as its output is a single element. In this example we showed how to express the famous Game of Life example but there are many more non-convolution examples that are expressible using our functional approach.

2.3.3.3 *2D Convolution*

In this section we define a 2d equivalent of the `convolution1d` function defined earlier. We start by defining the convolution operator $*$ in 2D as in already defined in [38] and a `convolution2d` function similar to the `convolution1d` function defined in Section 2.2.3.3

DEFINITION 2.17: *Let Ω be an array of size m whose elements are arrays of size n. Let r_x and r_y be positive integer values denoting the radius of the stencil shape in X and Y dimension. Let $u = 2r_x + 1$ denote the width of the stencil and $v = 2r_y + 1$ the height of the stencil. Let K be an array of size $u \times v$ denoting the convolution kernel. Let $-r_x \leqslant i \leqslant r_x$ and $-r_y \leqslant j \leqslant r_y$. The convolution operator $*$ for 2D arrays is the defined as follows*

$$(\Omega * K)(x, y) = \sum_i \sum_j \Omega(x + i, y + j) \cdot K(i + j \cdot r_x)$$

We express the convolution computation functionally using our high-level primitives:

DEFINITION 2.18: Ω, m, n, r_x, r_y, *and* K *are defined as in Definition 2.17 Let* b *be a boundary function as defined in Section 2.2.2 Let* $n_x = 2r_x + 1$ *denote the size of the stencil shape and* $n_y = 2r_y + 1$ *denote the size of the stencil shape.*

$$\begin{aligned} &\texttt{convolution2d}\ b\ K\ \Omega = \Omega\ *_b\ K \overset{def}{=} \\ &\quad (\mathit{map}(\mathit{map}\ (f\ K)) \circ \mathit{slide2d}\ n_y\ 1\ n_x\ 1 \circ \mathit{pad2d}\ r_x\ r_y\ b)\ \Omega \\ &\textit{where} \\ &\quad f\ K\ nbh = (\mathit{reduce}\ (+)\ 0 \circ \mathit{map}\ (*) \circ \mathit{zip})\ K\ (\mathit{join}\ nbh) \end{aligned} \tag{3}$$

Note that we defined the convolution kernel to be a flattened one dimensional array instead of a matrix. This has rather pragmatic reasons and is not necessarily required. By using a flattened version of the convolution kernel, we are able to flatten the neighborhoods as well. This is done by applying the *join* primitive to the neighborhoods nbh as depicted in the stencil function in Equation 3. Since both, the neighborhoods and the convolution kernel, are one dimensional now, we are able to apply *reduce* once. In the two dimensional case, we would have to reduce each row separately and reduce the temporary results. This way, the functional expression would only be more complex while computing the same result.

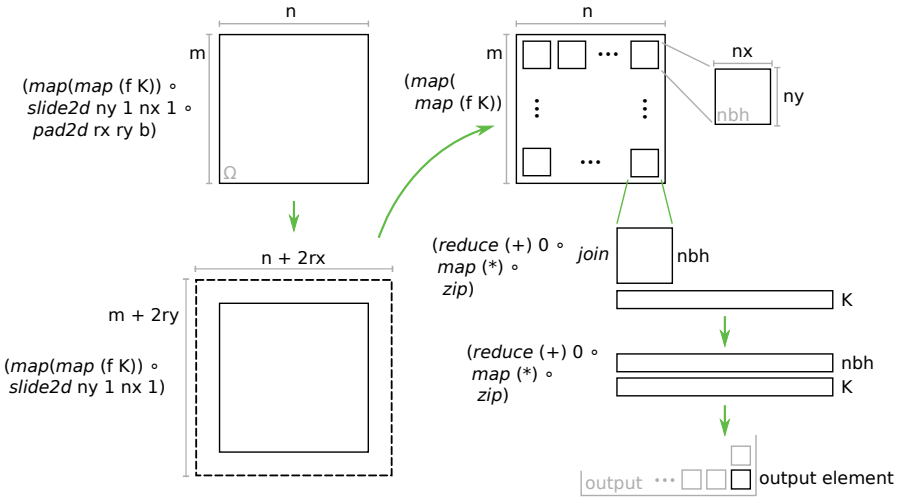


Figure 8: Computation steps for a high-level expression describing a two dimensional convolution

The computation of a two dimensional convolution is depicted in Figure 8. The conceptual structure of the functional program has not changed compared to the structure we identified in the introduction of this section. This is emphasized by the use of the functions *pad2d* and *slide2d* as depicted in Equation 3. As a first step, the input matrix is padded with the appropriate rows and columns specified by the boundary function b. Second, *slide2d* is used to create two dimensional neighborhoods in the padded input matrix. Finally, the stencil function f_K is mapped onto every neighborhood. Since the input is a matrix, thus two dimensional we need to use two *map* primitives to apply the function to every neighborhood.

2.3.3.4 *Gaussian Blur*

A well-known example of a stencil computation (specifically a two dimensional convolution) in image processing is the *Gaussian Blur* filter, also known as *Gaussian Smoothing*. The gaussian blur image filter is usually applied as a pre-processing stage in computer vision algorithms to reduce noise and detail. An example of applying the gaussian blur to reduce noise of an image is depicted in Figure 9. Since it is an image filter, it is usually applied to two dimensional data structures. The gaussian blur uses a gaussian function to compute the convolution kernel that is applied to the pixel values of the input image to compute a weighted average of a neighborhood. Thus, the convolution kernel for the gaussian blur application is defined as follows:

DEFINITION 2.19: *Let* rx *and* ry *denote the radius of the stencil in both dimensions. Let* $n = 2rx + 1$ *and* $-rx \leqslant i \leqslant rx$ *and* $-ry \leqslant j \leqslant ry$.

$$K_{gauss}(i + n * j) = \frac{1}{2\pi\sigma^2} e^{-\frac{i^2+j^2}{2\sigma^2}}$$

(a) Original image

(b) Image after applying the Gaussian-Blur filter

Figure 9: Applying the Gaussian Blur filter to the Lena image often used as an example in image processing

for a given standard derivation σ *used in the gaussian function.*

Using a standard derivation $\sigma = 1$, a 3×3 gaussian convolution kernel, which we use in the flattened one dimensional version as described in the beginning of this chapter, looks like:

$$\begin{bmatrix} 0.077847 & 0.123317 & 0.077847 \\ 0.123317 & 0.195346 & 0.123317 \\ 0.077847 & 0.123317 & 0.077847 \end{bmatrix}$$

The gaussian blur image filter using the `clamp` boundary condition can now be expressed using our `convolution2d` function that utilizes our high-level primitives as follows:

DEFINITION 2.20:

$$\texttt{gaussianBlur}\ \Omega \stackrel{def}{=} \texttt{convolution2d}\ \texttt{clamp}\ \mathsf{K}_{gauss}\ \Omega$$

2.3.3.5 17×17 *Convolution*

As a last example, we define a generic 17×17 convolution in terms of our `convolution2d` function. We specifically focus on optimizing this example in the following chapter. This stencil computation can be used to apply a gaussian blur using a bigger convolution kernel or any other arbitrary convolution.

EXAMPLE 2.12 (17×17 CONVOLUTION): *Let* Ω *be an arbitrary two dimensional grid. Let* K *be an array of size* $17 \cdot 17 = 289$ *with elements* k_i, $0 < i \leqslant 289$ *denoting the convolution kernel. Let* $\mathfrak{b}$ *be an arbitrary boundary function. A* 17×17 *convolution is then expressed as follows:*

$$\texttt{convolution2d}\ \mathfrak{b}\ \mathsf{K}\ \Omega =$$

$$(map(map\ (\mathrm{f}\ \mathrm{K}))\ \circ\ slide2d\ 17\ 1\ \circ\ pad2d\ 8\ 8\ \mathrm{b})\ \Omega \tag{4}$$

where

$$\mathrm{f}\ \mathrm{K}\ \mathrm{nbh} = (reduce\ (+)\ 0\ \circ\ map\ (*)\ \circ\ zip)\ (join\ \mathrm{nbh})\ \mathrm{K}$$

This is the high-level expression which we use in the following chapters as a running example.

2.4 SUMMARY

In this chapter, we identified the fundamental parts of stencil computations and explained how to express them using high-level primitives by defining two new primitives *slide* and *pad*. Using several examples for one and multidimensional stencil computations, we explained how to apply and combine these high-level primitives to express stencil computations in a purely functional manner. Furthermore we defined several functions composed of built-in primitives (for example *slide2d*, *convolution1d*, and more), which allowed to omit repeating specific combinations of primitives. Moreover, these *zero-cost abstractions* raised the abstraction level making the usage of our high-level primitives as expressive and easy to use as existing skeleton libraries or domain specific languages.

3

OPTIMIZING STENCIL COMPUTATIONS USING LOW-LEVEL PRIMITIVES

In this chapter, we analyze how to optimize stencil codes for GPUs using well-known optimizations and formalize them using our functional approach. By formalizing these optimizations and encoding them in specific combinations of our low-level primitives, we are able to apply them systematically rather than ad hoc. We are also able to reason about the correctness of optimizations using semantics preserving rewrite rules. Furthermore, we evaluate handwritten OpenCL kernels that each implement an optimization we observe in this chapter to emphasize the importance of optimizing stencil codes in order to gain high performance kernels.

3.1 LOW-LEVEL OPENCL-SPECIFIC FUNCTIONAL PRIMITIVES

In the following, we formally define a subset of the already existing low-level primitives which we use in the course of this thesis to express optimizations for stencil computations. To do this, we use the definitions given in [57]:

PARALLEL MAP Low-level OpenCL-specific *map* primitives describe different ways to exploit OpenCL's thread-level parallelism. All primitives map computations in different ways to the hardware. The semantics and types of all primitives are the same as of the high-level *map* primitives introduced in Section 2.2:

- The *mapGlobal* primitive assigns work all work-items independent of work-groups.
- The *mapWorkgroup* primitive assigns work to a work-group and the *mapLocal* primitive is used to assign work to work-items inside a work-group. Therefore, the *mapLocal* primitive can only be used nested inside a *mapWorkgroup* primitive.

Since OpenCL supports thread hierarchies in three dimension we write $mapGlobal_0$ to assign work to global work-items in the first dimension or $mapWorkgroup_1$ to assign work to work-groups in the second dimension respectively.

SEQUENTIAL MAP AND REDUCE The *mapSeq* and *reduceSeq* perform a sequential map and reduction using a single work-item. The semantics and type of the *mapSeq* primitive is the same as of the high-level *map* primitive. Since we do not require an associative and commutative operator for the sequential reduction any

longer, we can relax the requirements for the *reduceSeq* primitive and define it as in [57]:

DEFINITION 3.1: *Let* xs *be an array of size* n *with elements* x_i *where* $0 < i \leqslant n$. *Let* $\oplus$ *be a binary customizing operator with the identity element* $id_\oplus$. *The reduceSeq primitive is then defined as follows:*

$$reduceSeq\ (\oplus)\ id_\oplus\ [x_1, x_2, \ldots, x_n] \stackrel{def}{=} [(\ldots((id_\oplus \oplus x_1) \oplus x_2) \cdots \oplus x_n)]$$

The type of reduceSeq is defined as follows:

$$reduceSeq : (\alpha \to \beta \to \alpha) \to \alpha \to [\beta]_n \to [\alpha]_1$$

TOLOCAL AND TOGLOBAL The *toLocal* and *toGlobal* primitives allow to exploit OpenCL's memory hierarchy. Both primitives specify where the result of a given function is stored. In case of the *toLocal* primitive, the result is stored in the fast on-chip local memory. In case of the *toGlobal* primitive, the result is stored in the slower off-chip global memory. First, we define the *toLocal* primitive:

DEFINITION 3.2: *Let* f *be a function. The toLocal primitive is then defined as follows:*

$$toLocal\ f \stackrel{def}{=} f', \text{ where } f'\ x \stackrel{def}{=} f\ x, \forall x \text{ and } f' \text{ is guaranteed to store its result in local memory}$$

The type toLocal is defined as follows:

$$toLocal : (\alpha \to \beta) \to (\alpha \to \beta)$$

The definition for the *toGlobal* primitive is correspondent:

DEFINITION 3.3: *Let* f *be a function. The toGlobal primitive is then defined as follows:*

$$toGlobal\ f \stackrel{def}{=} f', \text{ where } f'\ x \stackrel{def}{=} f\ x, \forall x \text{ and } f' \text{ is guaranteed to store its result in global memory}$$

The type of toGlobal is defined as follows:

$$toGlobal : (\alpha \to \beta) \to (\alpha \to \beta)$$

3.2 OPTIMIZING STENCIL CONVOLUTION

Convolutions are a class of stencil computations found in many applications like digital signal processing, acoustics, electrical engineering, physics or artificial intelligence. Because of its broad use, we decide to specifically examine how this subclass of stencil computations is

optimized in order to gain high performance on GPUs. Nvidia published a guide [51] on how to optimize convolution computations. We use this as an orientation for systematically optimizing our convolution expression. We evaluate handwritten OpenCL kernels that each implement an optimization to observe the differences in performance after applying a specific optimization. These kernels are shown in Appendix A.1. To measure performance, we executed each kernel 100 times on an Nvidia Kepler K20c (Compute Capability 3.5) using the Nvidia OpenCL 1.2 CUDA 8.0.20 platform and the driver version 361.42. We compute the 17×17 convolution using 4096×4096 input elements and the `clamp` boundary handling. We present the median of the kernel runtimes omitting data transfer times. The example in this chapter shows that performance increases by a factor of 41. This shows the importance of optimizations and motivates us to represent these optimizations in our system.

Although we are investigating the optimization of a convolution example, most of the optimizations introduced in the following can be applied to a broader class of stencil computations. However, if an optimization is specific for this particular example we explicitly indicate this and explain why it is not generally applicable. Otherwise, all following optimizations are applicable to every stencil computation that can be expressed using the high-level primitives introduced in the previous chapter.

We begin the following sections with explaining and evaluating the performance benefits of a specific optimization using the handwritten OpenCL kernels. Afterwards, we introduce how to functionally express these optimizations using LIFT's IL.

3.2.1 *Naive Version*

PERFORMANCE To obtain a performance baseline, we measure the runtime of a naive handwritten 17×17 convolution kernel shown in Appendix A.1 in Listing 24. This kernel does not implement any optimization. Every global work-item sequentially computes a single output element. It takes 123.652 ms to execute and achieves a bandwidth of 0.543 GB/s. We use these results throughout this chapter to compare if the discussed optimizations indeed improve performance. In the following we examine how to express this naive version using low-level primitives.

```
1 conv = λ b weights input .
2   (map(map( λ nbh .
3     (reduce (+) 0 ∘ map (∗) ∘ zip) (join nbh) weights)) ∘
4   slide2d 17 1 ∘ pad2d 8 8 b) input
```

Listing 1: High level expression for a 17×17 convolution

```
1 conv = λ b weights input .
2   (mapGlobal1(mapGlobal0( λ nbh .
3     (reduceSeq (+) 0 ∘ mapSeq (∗) ∘ zip) (join nbh) weights)) ∘
4   slide2d 17 1 ∘ pad2d 8 8 b) input
```

Listing 2: Naive mapping of a 17×17 convolution to low-level primitves

REPRESENTATION IN LIFT We examine the high-level expression for the 17×17 convolution example introduced in Section 2.3.3.5:

$$
\begin{aligned}
&\mathtt{convolution2d\ b\ K\ \Omega} = \\
&(\mathit{map}(\mathit{map}\ (\mathtt{f\ K}))\ \circ\ \mathit{slide2d}\ 17\ 1\ \circ\ \mathit{pad2d}\ 8\ 8\ \mathtt{b})\ \Omega \\
&\mathit{where} \\
&\qquad \mathtt{f\ K\ nbh} = (\mathit{reduce}\ (+)\ 0\ \circ\ \mathit{map}\ (*)\ \circ\ \mathit{zip})\ (\mathit{join}\ \mathtt{nbh})\ \mathtt{K}
\end{aligned}
$$

In the previous chapter, we did not inline the stencil function to increase readability and emphasize the structure of a functional expression for stencil computations. In this chapter however, we write our functional programs as shown in Listing 1. This way, the functional program is depicted as one complete expression.

A naive and straightforward way to lower the high-level expression into an expression using low-level primitives is to replace the *map* and *reduce* primitives with their low-level parallel or sequential versions as shown in Listing 2, using the rewrite rules we introduce in the following. The *slide* and *pad* primitives are used in the high-level expressions of the previous chapter as well as the low-level expression used in this chapter. This is because equivalent to *split* and *join* they only modify the type and the structure of the data without describing computation that need to be parallelized or executed sequentially. Thus, they define how to access the data and not how to use the elements to compute the results. These primitives are also called *data layout primitives*. This enables us to reuse them in the exact same way in the low-level expressions without introducing low-level versions of these primitives.

The computation described by this expression is illustrated in Figure 10. Since input is a two dimensional matrix, we structure the global work-items two dimensionally using the low level primitives $\mathit{mapGlobal}_0$ and $\mathit{mapGlobal}_1$ in line 2 of Listing 2. In this version, every global work-item is responsible for computing one element of the output matrix. Thus, every work-item accesses elements of the input

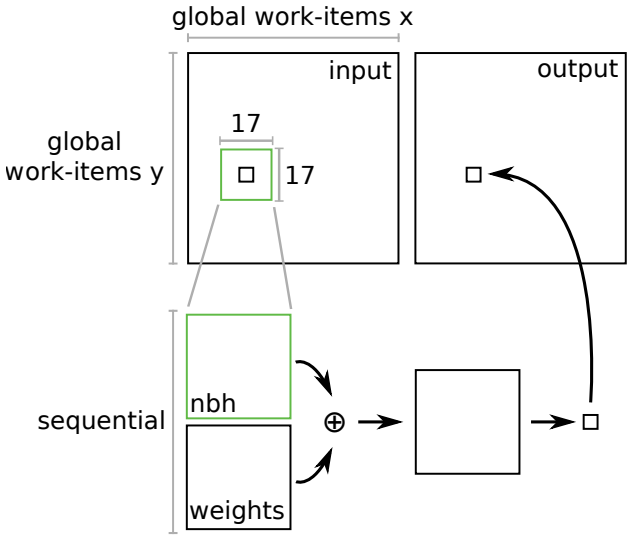


Figure 10: Naive computation of a 17×17 convolution using global work-items to sequentially compute a single output element

matrix in global memory to compute a single output element sequentially.

The transformation from the original high-level expression to the naive low-level expression is semantics preserving. To rewrite the high-level expression, we use two rewrite rules already introduced and proved correct in [57].

REWRITE RULE 1 (LOW-LEVEL OPENCL-SPECIFIC MAP RULE):

$$\begin{aligned} \mathit{map} \quad &\to \mathit{mapWorkgroup} \quad | \mathit{mapLocal} \\ &\quad | \mathit{mapGlobal} \quad\quad | \mathit{mapSeq} \end{aligned} \tag{5}$$

REWRITE RULE 2 (LOW-LEVEL REDUCE RULE):

$$\mathit{reduce}\ (\oplus)\ \mathtt{id}_{\oplus} \to \mathit{reduceSeq}\ (\oplus)\ \mathtt{id}_{\oplus} \tag{6}$$

In the following sections, we introduce optimizations to utilize the fast local memory of a GPU using our functional primitives.

3.2.2 *Applying Tiling to Utilize Local Memory*

In order to use the fast local memory of a GPU we need to divide the input in several *tiles*. Afterwards, a tile is assigned to a work-group which copies its tile to local memory and computes several output elements using local work-items. However, classical tiling approaches, for example used in matrix-matrix multiplication [45, 71], are not applicable in stencil computations. In these tiling approaches, the input is divided into fully separated tiles. This can easily be expressed using the *split* primitive as demonstrated in [53]. However, in stencil computations, the computation of a single output element requires

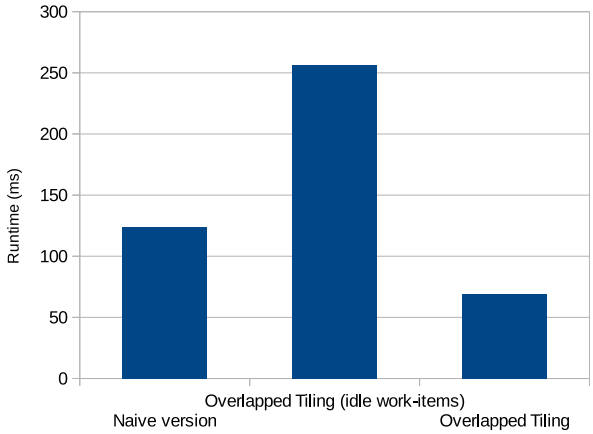


Figure 11: Performance of a naive 17×17 convolution compared to a version that applies overlapped tiling with and without idle work-items

access to surrounding elements. Thus, tiles in stencil computations have to overlap. This optimization is well-known and referred to as *Overlapped Tiling* [25, 27, 73].

PERFORMANCE The performance of handwritten kernels which implement overlapped tiling and use local memory compared to the performance of the naive version is visualized in Figure 11. The kernels evaluated are shown in shown in Appendix A.1 in Listing 25 and Listing 26. Although both kernels apply overlapped tiling and use the fast local memory, the first kernel is significantly slower compared to the naive version. This is because in this version, the work-groups are exactly as big as the overlapping tiles which results in idle work-items during the computations of output elements as we explain later in this chapter. The second version avoids idle threads during computation and achieves a speedup of 1,78 compared to the naive version.

REPRESENTATION IN LIFT In order to achieve a performance benefit using overlapped tiling, we need to examine how to express it functionally. More specifically, we examine the following points which motivate the structure of the next sections:

1. divide the input into overlapping tiles (Section 3.2.2.1)
2. assign a tile to a work-group and copy it to local memory (Section 3.2.2.2)
3. avoid idle threads during computation of output elements (Section 3.2.2.3)

We discuss each of these points in the following sections and start by describing how to express overlapped tiling for one and two dimensional inputs.

```
1 3PointJacobi = λ b input . (join ∘
2   map(λ nbh . reduce (+) 0 nbh ) ∘
3   slide 3 1 ∘ pad 1 1 b) input
```

Listing 3: Low-level expression for a 3-point Jacobi example

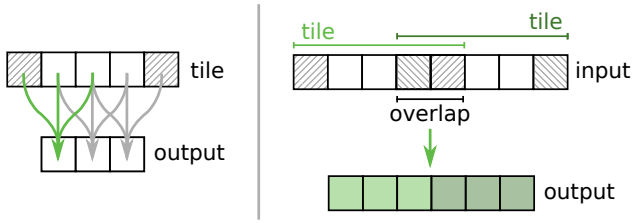


Figure 12: Overlapped Tiling for a 3-point stencil in one dimension

3.2.2.1 *Overlapped Tiling*

APPLYING OVERLAPPED TILING IN LIFT IN ONE DIMENSION To explain the concept of overlapped tiling and how we realize it using our functional primitives, we first consider a one dimensional example before we explain how to apply this to the 17×17 convolution example. Consider the 3-point Jacobi example shown in Listing 3. In the course of this section we systematically rewrite this high-level expression to a low-level expression that applies overlapped tiling. Eventually, we want every work-group to compute multiple (e. g. 3) elements using its local work-items. We can not simply divide the input using *(split 3)*. Instead, each tile needs to contain five elements in order to be able to compute three output elements as shown on the left side of Figure 12. If we divide the input into multiple tiles, these tiles overlap as shown on the right side of Figure 12. Thus, they share certain elements depending on the shape of the stencil.

We reuse the *slide* primitive to create overlapping tiles. Afterwards, we use *slide* again to create the original neighborhoods in each tile. To

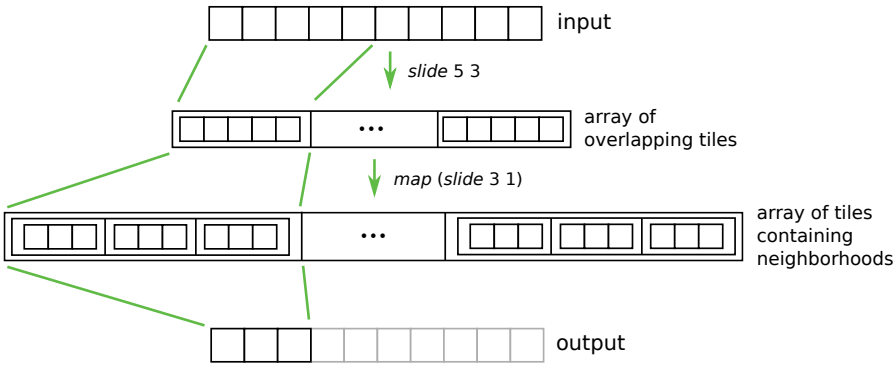


Figure 13: Expressing Overlapped Tiling using *slide*: Applying *slide* to the input creates overlapping tiles. Applying *slide* to every tile creates the required neighborhoods

```
1 3PointJacobi = λ b input . (join ∘
2   mapWorkgroup(λ tile .
3     (mapLocal(λ nbh . reduceSeq (+) 0 nbh ) ∘
4       slide 3 1) tile) ∘
5     slide 5 3 ∘ pad 1 1 b) input
```

Listing 4: Low-level expression for a 3-point Jacobi applying Overlapped Tiling

create the tiles depicted in Figure 12, we need to use the *slide* primitive with a size of five and a step of three. By sliding twice, first the tiles are created and then the neighborhoods inside the tiles are created, as shown in Figure 13. The first *slide* creates an array of tiles. Mapping the second *slide* onto each tile creates neighborhoods in each tile. The functional program expressing the 3-point Jacobi computation using overlapped tiling is shown in Listing 4. We use λ-functions to name the arguments of each function. We use the *mapWorkgroup* and the *mapLocal* primitives to assign each tile to a work-group (line 2 in Listing 4). Inside every work-group, each local work-item is responsible for computing a single output element for a given neighborhood (line 3).

SYSTEMATICAL REWRITING TO APPLY OVERLAPPED TILING

Now we show how to transform the Listing 3 to Listing 4 using provably correct, hence semantics preserving, rewrite rules. We do this by providing the following overlapped tiling rule.

REWRITE RULE 3 (OVERLAPPED TILING RULE):

$$slide\ n\ s = join \circ map(slide\ n\ s) \circ slide\ u\ v \tag{7}$$

This rule states that the exact same elements that are grouped together using *slide* with parameters n and s end up in the same neighborhoods when using *slide* with the tiling parameters u and v first while sliding afterwards using the original parameters n and s again. Obviously, n and s as well as u and v need to be valid parameters for the *slide* primitive. Thus, applied to an array of length m, $m - n + s \bmod s$ needs to equal 0. The same needs to be true for u and v. The correctness proof for the overlapped tiling rule is given in Appendix A.2, Proof A.2.1. To be able to systematically rewrite the high-level 3-point Jacobi to a low-level version that applies overlapped tiling we need to provide one more rewrite rule:

REWRITE RULE 4 (MAP-JOIN REORDER RULE):

$$map\ f \circ join = join \circ map(map\ f) \tag{8}$$

The proof for this rule is given in Appendix A.2, Proof A.2.2. Now we are able to systematically rewrite the high-level stencil expression

to an equivalent low-level version that applies overlapped tiling. Besides the two previously introduced rewrite rule we use the following rule already introduced and proved correct in [57]:

REWRITE RULE 5 (MAP FUSION RULE):

$$map\ f \circ map\ g = map(f \circ g)$$

EXAMPLE 3.1 (APPLYING OVERLAPPED TILING (1D)):

$$map(reduce\ (+)\ 0) \circ slide\ 3\ 1\ \circ pad\ 1\ 1\ b$$

(using the overlapped tiling rule - Rewrite Rule 3)

$$map(reduce\ (+)\ 0) \circ join \circ map(slide\ 3\ 1) \circ slide\ 5\ 3\ \circ pad\ 1\ 1\ b$$

(using the map-join reorder rule - Rewrite Rule 4)

$$join \circ map(map(reduce\ (+)\ 0)) \circ$$
$$map(slide\ 3\ 1) \circ slide\ 5\ 3\ \circ pad\ 1\ 1\ b$$

(using the map-fusion rule - Rewrite Rule 5)

$$join \circ map(\ \ map(reduce\ (+)\ 0) \circ slide\ 3\ 1\ ) \circ$$
$$slide\ 5\ 3\ \circ pad\ 1\ 1\ b$$

(using λ-functions to name arguments)

$$join \circ map(\lambda\ \mathtt{tile}.$$
$$(map(\lambda\ \mathtt{nbh}\ .\ reduce\ (+)\ 0\ \mathtt{nbh}) \circ slide\ 3\ 1)\ \mathtt{tile}) \circ$$
$$slide\ 5\ 3\ \circ pad\ 1\ 1\ b$$

(using the low-level map and reduce rules - Rewrite Rules 1 and 2)

$$join \circ mapWorkgroup(\lambda\ \mathtt{tile}.$$
$$(mapLocal(\lambda\ \mathtt{nbh}\ .\ reduceSeq\ (+)\ 0\ \mathtt{nbh}) \circ slide\ 3\ 1)\ \mathtt{tile}) \circ$$
$$slide\ 5\ 3\ \circ pad\ 1\ 1\ b$$

This is exactly the expression discussed previously and shown in Listing 4. By applying simple rewrite rules, we were able to systematically rewrite the simple 3-point Jacobi example to an expression which applies overlapped tiling. The original and the resulting expression have the same semantics since we only applied semantics preserving rewrite rules.

In the following section, we examine how to express overlapped tiling in two dimensions and apply that optimization for the 17×17 convolution example.

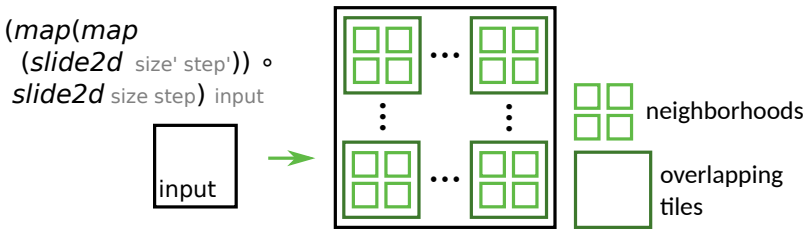


Figure 14: Applying overlapped tiling in two dimensions

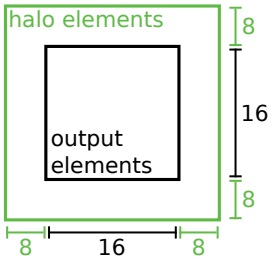


Figure 15: Tile shape for a 17×17 convolution

APPLYING OVERLAPPED TILING IN LIFT IN TWO DIMENSIONS

Given the previous definition of overlapped tiling using low-level primitives, applying this optimization in 2D is straightforward. Instead of using *slide* twice, we are using the function *slide2d* twice. Using *slide2d* with the tiling parameters creates overlapping tiles in two dimensions. Mapping the second *slide2d* creates the original neighborhoods inside each tile as depicted in Figure 14.

Now consider we want every work-group to compute 16×16 output elements in our 2D convolution example. The shape of the tile each work-group processes is illustrated in Figure 15. Since the radius of the convolution stencil equals eight, we need to consider eight extra elements on both sides of each dimension. These elements are also called *halo-elements*. The resulting tile therefore contains 32×32 elements and each work-group is able to compute 16×16 output elements.

The expression that introduces overlapped tiling to the convolution example is depicted in Listing 5.

Changing the shape of the tile is now just a matter of changing the numerical parameters for the first *slide2d* function in line 11.

In the previous sections, we introduced how to divide the input into several overlapping tiles. In order to achieve the speedup observed in the beginning Section 3.2.2, we need to examine how to copy these tiles to the fast local memory.

3.2.2.2 *Utilizing Local Memory*

Now that we divided the input into overlapping tiles, we are able to use the fast local memory to accelerate our computation. Every GPU

```
1  conv = λ b weights input .
2    // assign tiles to work-groups
3    (mapWorkgroup_1 (mapWorkgroup_0 (λ tile .
4      // assign neighborhoods to local work-items
5      (mapLocal_1 (mapLocal_0 (λ nbh .
6        // stencil function
7        (reduceSeq (+) 0 ∘ mapSeq (∗) ∘ zip) (join nbh) weights)) ∘
8          // create neighborhoods in each tile
9          slide2d 17 1) tile)) ∘
10   // create overlapping tiles
11   slide2d 32 16 ∘ pad2d 8 8 b) input
```

Listing 5: Low-level expression for a 17×17 convolution applying Overlapped Tiling

contains a small amount of fast *on-chip memory* often referred to as *shared* or *local memory* which is the term we use in the following.

In order to utilize local memory, we use the *toLocal* primitive which takes a function as an argument and guarantees that its output is stored in local memory. We apply the identity function, nested inside the *toLocal* primitive, to all elements of the tile as shown in Equation 9, to copy an entire tile into local memory.

$$toLocal(mapLocal_1\,(mapLocal_0\ \mathtt{id}))\ \mathtt{tile} \tag{9}$$

By applying the identity to every element, the shape of the tile remains unchanged. The result is guaranteed to be stored in local memory because the computation is nested inside a *toLocal* primitive. Every computation following the expression works with elements residing in local memory. To store the output element in global memory again, we apply the *toGlobal* primitive in a similar fashion:

$$toGlobal(mapSeq\ \mathtt{id})\ \mathtt{element} \tag{10}$$

Every work-item executes this expression after computing the output element in local memory. This ensures that every work-item eventually copies its result back to global memory. The complete expression describing the 17×17 convolution using overlapping tiles and local memory is shown in Listing 6.

To systematically rewrite this expression we need to introduce another two rewrite rules already discussed in [57]:

REWRITE RULE 6 (IDENTITY):

$$\mathtt{f} \rightarrow \mathtt{f} \circ map\ \mathtt{id} \mid map\ \mathtt{id} \circ \mathtt{f}$$

REWRITE RULE 7 (LOCAL AND GLOBAL MEMORY RULE):

$$\begin{aligned} mapLocal\ \mathtt{f} &\rightarrow toGlobal(mapLocal\ \mathtt{f}) \mid toGlobal(mapSeq\ \mathtt{f}) \\ mapLocal\ \mathtt{f} &\rightarrow toLocal(mapLocal\ \mathtt{f}) \mid toLocal(mapSeq\ \mathtt{f}) \end{aligned}$$

```
1  conv = λ b input weights .
2    // assign tiles to work-groups
3    (mapWorkgroup_1 (mapWorkgroup_0 (λ tile .
4      // assign neighborhoods to local work-items
5      (mapLocal_1 (mapLocal_0 (λ nbh .
6        // store result in global memory
7        (toGlobal (mapSeq id) ∘
8        // stencil function
9        reduceSeq (+) 0 ∘ mapSeq (*) ∘ zip) (join nbh) weights)) ∘
10       // create neighborhoods in each tile
11       slide2d 17 1 ∘
12     // copy tile to local memory
13     toLocal (mapLocal_1 (mapLocal_0 id))) tile)) ∘
14   // create overlapping tiles
15   slide2d 32 16 ∘ pad2d 8 8 b) input
```

Listing 6: Low-level expression for a 17×17 Convolution applying Overlapped Tiling and using local memory

```
1  conv = λ b weights input .
2    mapWorkgroup_1 (mapWorkgroup_0 (λ tile .
3      (mapLocal_1 (mapLocal_0 (λ nbh .
4        (toGlobal (mapSeq id) ∘
5        // ...
```

Listing 7: Functionally assigning work to work-groups and work-items using low-level primitives

Listing 5 is then rewritten to Listing 6 as shown in Appendix A.3. Again, all rules applied are provably semantics preserving which causes the rewritten expression which uses local memory to compute the same result as the original expression.

It obviously would be more intuitive to use the *toLocal* and *toGlobal* primitives as in the following expressions:

toLocal(*slide* tilesize tilestep) input

toGlobal(stencilFunction)

However, this is currently not possible in the practical implementation of LIFT, but should be possible in future versions of this work.

Dividing the input into overlapping tiles and loading them to local memory is not necessarily achieving a performance benefit. As observed in the beginning of Section 3.2.2, having idle work-items inside a work-group during the computation of output elements causes a significant performance drawback. Therefore, we examine how to avoid idle-work items in the next section.

3.2.2.3 *Avoiding Idle Work-Items*

An important factor for performance is how many work-items load data to local memory and how many work-items are active during

the computation of output elements. Note that none of our expressions considers the amount of work-items and work-groups started for execution. Instead we use the low-level primitives *mapWorkgroup* and *mapLocal* shown in Listing 7 to assign work to all existing work-items. In the previous section we assumed work-groups to be as big as the tile size such that every work-item loads exactly one element to local memory. In this case the $mapLocal_0$ and $mapLocal_1$ primitives map each work-item to exactly one element of the tile. However, after loading to local memory, only a quarter of the work-items are computing an output element which forces most of the work-items of this work-group (exactly 768 of 1024 in the previous example) to be idle during computation.

To address this, we could also launch work-groups that contain less work-items which leads to multiple loads per work-item. For example, every work-item loading four elements to local memory. Afterwards, all work-items of this work-group compute an output elements without any idle work-items in a work-group. This optimization is independent of our expression but starting fewer work-items per work-group changes the distribution of work.

Applying all the optimization discussed in the previous sections leads to the speedup of 1.78 as discussed in the beginning of Section 3.2.2. These results emphasize the fact that the performance of OpenCL programs is highly sensitive to applied optimizations including its parameters like tile sizes. Choosing the wrong parameters or configurations for a specific optimization leads to a significant performance drawback. Therefore, it is highly desirable to specify means to systematically apply optimizations like tiling or the usage of local memory by using a formal system instead of applying them using a trial and error approach.

3.2.3 *Increasing Efficiency by Separating Convolution*

Sometimes, the convolution computation can be split into two separate computations. Specifically, a row and a column convolution that are applied consecutively. The computation using two separated convolutions is shown in Figure 16. We first apply a row convolution and write an intermediate result to global memory, shown on the left side. Afterwards, we apply the column convolution using the intermediate result, shown on the right side. This optimization is not generally applicable for every stencil application. However, it significantly increases the performance for convolution computations when applicable as shown in Figure 17 (using a logarithmic scale). We examine three different versions of separated convolution. The first version is only separating the convolution into a row and a column convolution without implementing overlapped tiling or the usage of local memory.

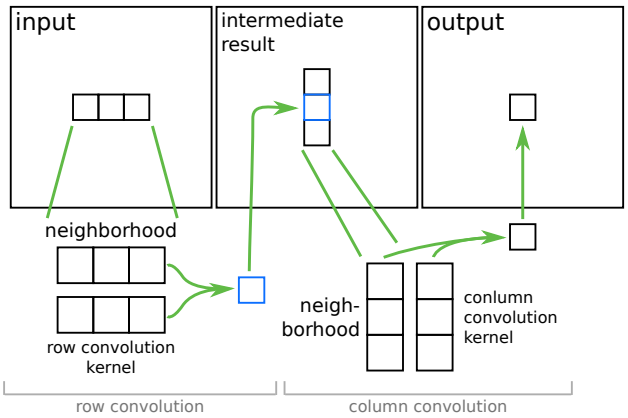


Figure 16: Illustration of a two dimensional convolution using separated convolution kernels

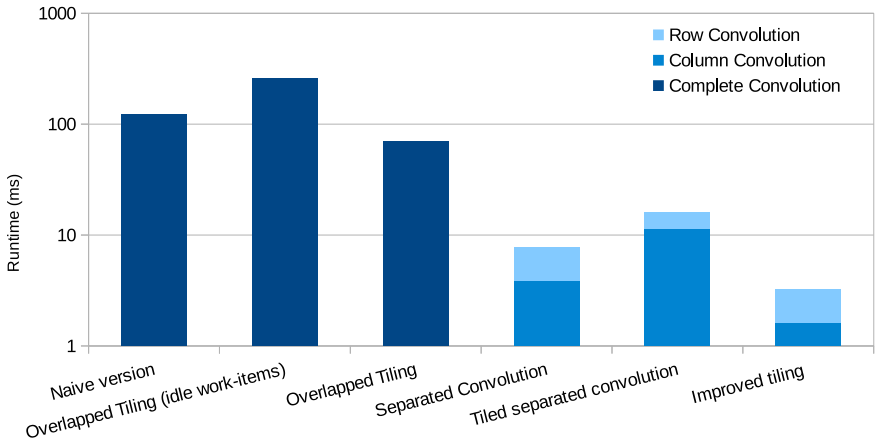


Figure 17: Performance of an improved tiled and separated 17×17 convolution compared to convolutions implementing other optimizations

PERFORMANCE The kernels implementing the row and column convolution are shown in Appendix A.1 in Listing 27 and Listing 28. Separating the convolution like this results in a speedup of 8.89 compared to the previous version and to a speedup of 15.84 compared to the naive version. Reintroducing tiling and local memory does not necessarily improve the performance. Choosing the wrong tile sizes leads to a significant performance drawback. The kernels implementing the tiled row and column convolution are shown in Appendix A.1 in Listing 29 and Listing 30. However, choosing appropriate parameters for all optimizations, we gain a speedup of 38.41 compared to the naive version. The kernels implementing the improved tiled row and column convolution are shown in Appendix A.1 in Listing 31 and Listing 32. These numbers emphasize that separating the convolution significantly improves the performance and should always be applied if possible.

```
1  conv = λ b weights_X weights_Y input .
2    (convColumn b weights_Y ∘ convRow b weights_X) input
3
4  convRow = λ b weights:[float]_17 input:[[float]_n]_m .
5    (mapGlobal_1 (mapGlobal_0( λ nbh:[[float]_17]_1 .
6      (reduceSeq (+) 0 ∘ mapSeq (*) ∘ zip) (join nbh) weights)) ∘
7    slide2d 1 1 17 1 ∘ pad2d 0 0 8 8 b) input
8
9  convColumn = λ b weights:[float]_17 input:[[float]_n]_m .
10   (mapGlobal_1 (mapGlobal_0( λ nbh:[[float]_1]_17 .
11     (reduceSeq (+) 0 ∘ mapSeq (*) ∘ zip) (join nbh) weights)) ∘
12   slide2d 17 1 1 1 ∘ pad2d 8 8 0 0 b) input
```

Listing 8: 17×17 convolution separated into a row and column convolution

REPRESENTATION IN LIFT Separating convolutions into a row and column convolution (also called X and Y convolution) is beneficial because of two reasons: First, it reduces the amount of operations required to compute a single output element. Depending on the size of the convolution kernel, computing a single output is costly. For a 17×17 convolution, it takes 289 multiplications and additions to compute a single output element. If we can separate the convolution into a 17×1 and a 1×17 convolution, we only need $17 + 17 = 34$ multiplications and additions to compute the same output element. This reduces the amount of operations required by a factor of 8.5. Second, it allows a better utilization of the limited local memory when using tiling.

In the following, we examine how to express this optimization in LIFT's IL by discussing how to

1. functionally express a separated convolution (Section 3.2.3.1)
2. reintroduce overlapped tiling and local memory (Section 3.2.3.2)
3. improve the tile sizes for separated convolutions (Section 3.2.3.3)

We start by examining how to express a separated convolution in its simplest form using functional primitives. Thus, without tiling and without using local memory. Afterwards, we proof that this separation is valid given a separable convolution kernel. Then, we examine different ways to use overlapped tiling on separated convolution and explain more advanced optimizations.

3.2.3.1 *Simple Separation*

A separated convolution that is composed of a row convolution and a column convolution is shown in Listing 8 using low-level primitives. The separation is naturally expressed using function composition (line 2). In this version, every global work-item computes a single

output element and accesses global memory. Note that we first apply $convRow$ which writes its intermediate result into global memory. $convColumn$ uses this intermediate result to compute the result of the convolution. Since there is no way to globally synchronize work-groups, $convRow$ and $convColumn$ need to be executed as two separate kernels that are executed consecutively on the GPU. This way, we ensure that the row convolution has finished before we use the results in the column convolution. Therefore, we need to generate a kernel for both convolution functions (lines 4 and 9).

Both functions are similar to the naive versions introduced in Section 3.2.1. In fact, they only differ in numerical parameter for the *pad2d* and *slide2d* functions. This is because from a functional perspective only the shape of the tiles, neighborhoods and convolution kernel changed. Whereas the computation performed, a convolution computation, is still the same. The change from conventional 2D convolution to row and column convolution leads to the changes in the *pad2d* and *slide2d* functions that create the neighborhoods according to the stencil shape. Note that we use *slide2d* and *pad2d* as a convenient way to create the 17×1 neighborhoods for the row convolution (line 7) and the 1×17 neighborhoods for the column convolution (line 12). It is convenient because we only change the parameters instead of changing the composition of *map*, *slide* and *transpose* to create the row and column neighborhoods. Although one dimension equals 1 in each neighborhood, *slide2d* creates matrices in both cases which are flattened as before using *join* in lines 6 and 11.

Now we proof mathematically that convolutions using specific convolution kernels can be separated into a row and column convolution: For completeness we repeat the definitions for one and two dimensional convolution from Sections 2.2.3.3 and 2.3.3.3:

DEFINITION 3.4 (CONVOLUTION): *Let Ω be an array of size n whose elements are arrays of size m. Let r_x and r_y be positive integer values denoting the radius of the stencil shape in X and Y dimension. Let $u = 2r_x + 1$ denote the width of the stencil in X dimension and $v = 2r_y + 1$ the height of the stencil in Y dimension. Let K be an array of size $u \times v$ denoting the convolution kernel. Let $-r_x \leqslant i \leqslant r_x$ and $-r_y \leqslant j \leqslant r_y$. One dimensional convolution is then defined as follows:*

$$(\Omega * K)(x) = \sum_{i=-r_x}^{r_x} \Omega(x+i) \cdot K(i) \tag{11}$$

The convolution operator $*$ for 2D arrays is the defined as follows

$$(\Omega * K)(x,y) = \sum_{i} \sum_{j} \Omega(x+i, y+i) \cdot K(i+j \cdot r_x)$$

which equals

$$(\Omega * K)(x,y) = \sum_{i} \sum_{j} \Omega(x+i, y+i) \cdot K(i,j) \tag{12}$$

(a) Original image

(b) Image after applying the Sobel edge detection filter

Figure 18: Applying the Sobel edge detection filter to the Lena image

if we assume that K *is a two dimensional array of size* $(2r_x + 1) \times (2r_y + 1)$ *again.*

PROOF 3.2.1: *Since multiplication is commutative, we can rewrite Equation 12 to the following equation:*

$$(\Omega * K)(x, y) = \sum_i \sum_j K(i, j) \cdot \Omega(x + i, y + i)$$

If the $u \times v$ *matrix* K *can be decomposed into a* $u \times 1$ *matrix* K_x *and a* $1 \times v$ *matrix* K_y *such that:*

$$K(i, j) = K_y(i) * K_x(j) \tag{13}$$

we call this matrix separable.

$$(\Omega * K)(x, y) = \sum_i \sum_j K_x(i) \cdot K_y(j) \cdot \Omega(x + i, y + i)$$

$$=(\Omega * K)(x, y) = \sum_j K_y(j) \cdot \left(\sum_i K_x(i) \cdot \Omega(x + i, y + j) \right)$$

This shows that it is valid to first perform the convolution in X dimension using K_x *and do the column convolution afterwards using* K_y*, if* K *is separable.* □

The key property that allows to separate the convolution into two parts is given in Equation 13. To understand this property we look at a well-known example of a separable convolution kernel used in image processing. The *Sobel Edge Detection Filter* [56] is a convolution operation frequently used in image processing algorithms. It uses two 3×3 convolution kernels to detect vertical and horizontal edges in a given picture as shown in Figure 18. Since both kernels are separable, we only use one of them to give an example for the property given in Equation 13:

EXAMPLE 3.2:

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

In the remainder of this section we assume that the convolution kernel K is separable.

The applicability of this optimization depends on the values of the convolution kernel. Since we are unable to detect the separability property of a convolution kernel automatically in the current version, we provide a high-level function that allows to specify a column and a row convolution kernel. This allows the programmer to indicate that the convolution can be separated:

DEFINITION 3.5: *Let Ω be an array of size m whose elements are arrays of size n. Let b be a boundary function as defined in Section 2.2.2 Let r_x and r_y be positive integer values. Let $n_x = 2r_x + 1$ denote the size of the stencil shape in X dimension and $n_y = 2r_y + 1$ denote the size of the stencil shape in Y dimension. Let K_x be an array of size n_x and K_y be an array of size n_y. The* separableConvolution2d *function is then defined as:*

$$\begin{aligned}
&\mathtt{separableConvolution2d}\ b\ K_x\ K_y\ \Omega \stackrel{def}{=} \\
&\quad (\mathtt{convColumn}\ b\ K_y \circ \mathtt{convRow}\ b\ K_x)\ \Omega \\
&\textit{where} \\
&\quad \mathtt{convRow}\ b\ K_x = \\
&\quad \textit{map}(\textit{map}\ (f\ K_x)) \circ \textit{slide2d}\ 1\ 1\ n_x\ 1 \circ \textit{pad2d}\ r_x\ r_x\ 0\ 0\ b) \\
&\textit{and} \\
&\quad \mathtt{convColumn}\ b\ K_y = \\
&\quad \textit{map}(\textit{map}\ (f\ K_y)) \circ \textit{slide2d}\ n_y\ 1\ 1\ 1 \circ \textit{pad2d}\ 0\ 0\ r_y\ r_y\ b) \\
&\textit{and} \\
&\quad f\ K\ nbh = (\textit{reduce}\ (+)\ 0 \circ \textit{map}\ (*) \circ \textit{zip})\ (\textit{join}\ nbh)\ K
\end{aligned} \tag{14}$$

Now we defined means to express separated convolutions. In the next sections, we reintroduce tiling and discuss how to improve tiling for separated convolution computations.

3.2.3.2 *Tiling Separated Convolution*

In the previous section we described how to use the separation of the convolution computation to reduce the amount of operations required to compute a single output element. In this section we reintroduce overlapped tiling and local memory usage to the separated convolution computation. To rewrite the simply separated convolution expression introduced in the previous chapter, to an expression that uses tiling and local memory eventually, the same techniques are applied as in the non-separated convolution. Therefore, we do not

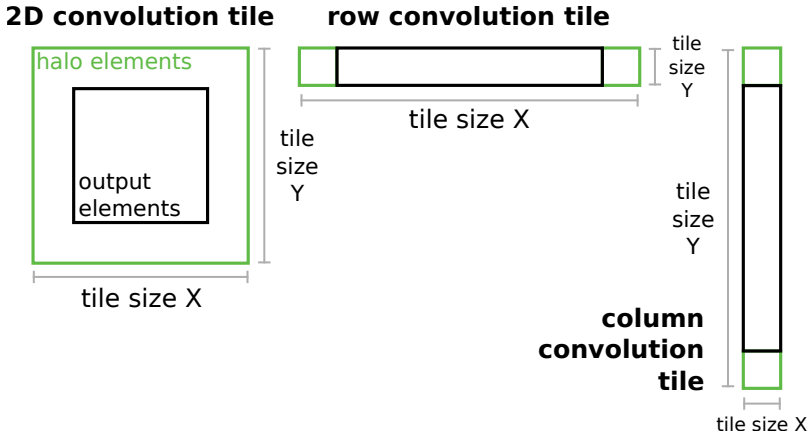


Figure 19: Comparison of tiles for convolutions in one and two dimensions

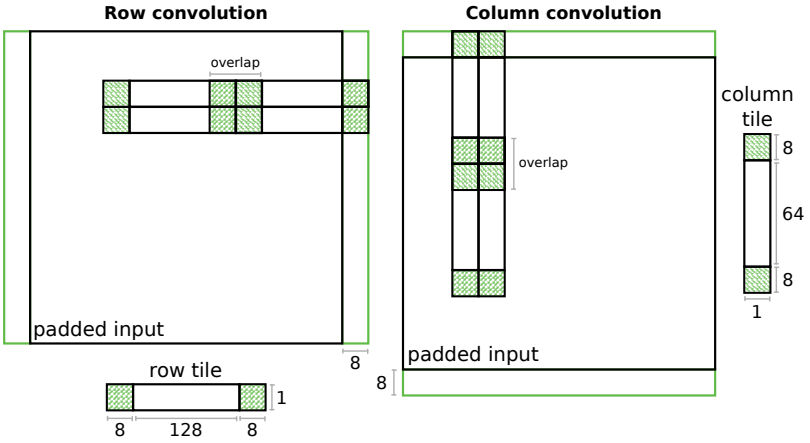


Figure 20: Arrangement of overlapping tiles in separate convolution

repeat the formal rewrites for these optimizations in the following sections again.

Separating the convolution computation into a row and a column convolution reduces the amount of halo elements when loading a tile into the small local memory. For the column convolution we are able to ignore the horizontal halo elements which we had to include in the conventional 2D convolution as depicted in Figure 19. Therefore, the ratio between halo elements and output elements in a tile significantly decreases. Since we do not need to load halo elements in one dimension, the tiles do not overlap in this dimension too.

Creating the overlapping tiles for the row and column convolution is again realized by using the *slide2d* function on a padded input. Intuitively, we divide each row into overlapping tiles for the row convolution and each column into overlapping tiles for the column convolution. Thus, we express the computation using the expression shown in Listing 9.

The arrangement of the tiles used in Listing 9 is illustrated in Figure 20. The tile sizes were chosen according to the Nvidia Toolkit

```
1  conv = λ b weights_X weights_Y input .
2    (convColumn b weights_Y ∘ convRow b weights_X) input
3
4  convRow = λ b weights input .
5    (mapWorkgroup_1(mapWorkgroup_0(λ tile .
6      (mapLocal_1(mapLocal_0((λ nbh .
7        (toGlobal(mapSeq id) ∘
8        reduceSeq (+) 0 ∘ mapSeq (*) ∘ zip) (join nbh) weights)) ∘
9        slide2d 1 1 17 1 ∘
10     toLocal(mapLocal_1(mapLocal_0 id))) tile))) ∘
11     // divide each row into tiles that contain 80 elements
12     // the input only needs to be padded on the left and right
13   slide2d 1 1 144 128 ∘ pad2d 0 0 8 8 b) input
14
15 convColumn = λ weights input .
16   (mapWorkgroup_1(mapWorkgroup_0(λ tile .
17     (mapLocal_1(mapLocal_0((λ nbh .
18       (toGlobal(mapSeq id) ∘
19       reduceSeq (+) 0 ∘ mapSeq (*) ∘ zip) (join nbh) weights)) ∘
20       slide2d 17 1 1 1 ∘
21     toLocal(mapLocal_1(mapLocal_0 id))) tile))) ∘
22     // divide each column into tiles that contain 80 elements
23     // the input only needs to be padded on the top and bottom
24   slide2d 80 64 1 1 ∘ pad2d 8 8 0 0 b) input
```

Listing 9: Applying Overlapped Tiling and using local memory in the row and column convolution

example [51]. In the row convolution, a tile contains 144 elements in total and covers a single row while overlapping 16 elements with neighboring tiles on each side. The column convolution tile contains 80 elements, covering a single column. Now we successfully separated the convolution computation into two distinct computations and decreased the amount of computations needed to compute a single output element. We also decreased the ratio of halo elements to output elements in a tile which allows to compute more output elements with fewer memory accesses.

We discussed how to functionally express separate convolution computations that use a separable convolution kernel. Furthermore, we reintroduced overlapped tiling and local memory to these expressions. The performance of the handwritten OpenCL kernels shown in Figure 17 in Section 3.2.3 revealed that these tile sizes lead to a significant performance drawback compared to the separated convolution that does not apply tiling. In the next section, we discuss the problem with the current tile sizes and examine how to improve tiling in separated convolutions using LIFT's IL.

3.2.3.3 *Improve Tiling in Separated Convolution*

In this section, we specifically improve the tiling in the column convolution. Memory accesses to global memory are costly but if consec-

```
19  ...
20    toLocal(mapLocal_1(mapLocal_0 id)) tile))) ∘
21  slide2d 80 64 1 1 ∘ pad2d 8 8 0 0 b) input
```

Listing 10: Low-level expression creating tiles that enfore uncoalesced global memory access in column convolution

```
19  ...
20    toLocal(mapLocal_1(mapLocal_0 id)) tile))) ∘
21  slide2d 80 64 16 16 ∘ pad2d 8 8 0 0 b) input
```

Listing 11: Low-level expression creating tiles that enable coalesced global memory access in column convolution

utive work-items access consecutive elements in global memory the accesses are *coalesced*. This means that multiple accesses to the DRAM of the GPU are performed in a single transaction. In OpenCL, two dimensional data is stored in a row-major order on the device. This means that consecutive work-items of a work-group should access elements of the same row to achieve coalesced accesses.

Work-items access the tiles created in the expression shown in Listing 10. Local work-items, thus work-items in the same work-group, are mapped onto the column tile whose shape is illustrated in Figure 20. The first work-item of a work-group accesses the first element of the tile which is the uppermost element of the column tile. The second work-item accesses the second element of the tile in row major order and so on. Since the column tile has width of 1, the next element in row major order is the element in the next row directly underneath the first element. Thus, every work-item in a work-group accesses another row and memory accesses of a work-group can not be coalesced. This results in a significant performance drawback of the generated kernel for this expression as observed previously.

In order to address this problem, we need to coalesce the memory accesses of local work-items to global memory. The general idea is to increase the width of the column tile. This way, the second work-item accesses the element next to the first work-item and so on. The choice of how much we increase the column tile width depends on the hardware.

On Nvidia GPUs, work-items of a work-group are divided for execution into groups of 32 called *warps* (or *wavefronts* on AMD GPUs). The optimal size of the column tile width depends on how many work-items are scheduled to access the global memory at the same time. On older Nvidia GPUs (Compute Capability < 2), global memory accesses are coalesced for a half warp (16 work-items). Therefore, we should increase the width of the tile to 16 elements such that every work-item accesses the same row at the same time. This is exactly the tile size we find in the Nvidia guide to optimize separable con-

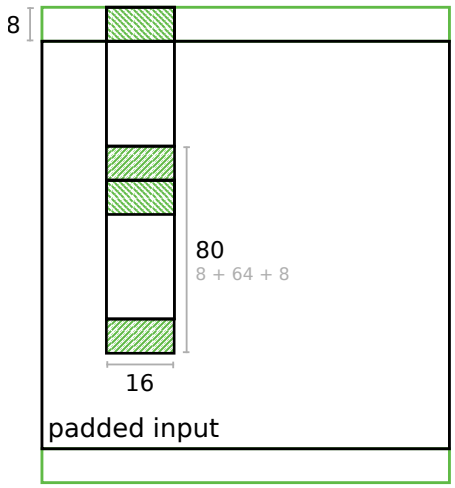


Figure 21: Arrangement of overlapping tiles in the column convolution to coalesce global memory accesses

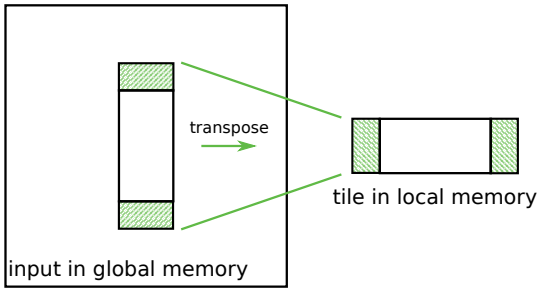


Figure 22: Column convolution: Copying transposed tile to local memory

volution [51]. This guide is from 2007 and newer generations of GPU cores have been released since. Nowadays, global memory accesses are coalesced for a complete warp of 32 work-items.

This way of tiling the input can be expressed using *slide2d* as shown in Listing 11. This arrangement of tiles for the column convolution is visualized in Figure 21.

To summarize, in the previous sections we discussed why it is beneficial to separate convolution computations in a row and column convolution if possible. We evaluated the performance benefit using handwritten OpenCL kernels and introduced how to express this optimization using LIFT's IL. Furthermore, we reintroduced overlapped tiling and local memory and discussed how to improve tile sizes for separated convolution computations. In the following sections we discuss more advanced optimizations specifically for the column convolution.

3.2.4 *Transposing the Local Memory Tile in the Column Convolution*

On some Nvidia GPUs it might be beneficial to transpose the tile before loading it into local memory to change the memory access

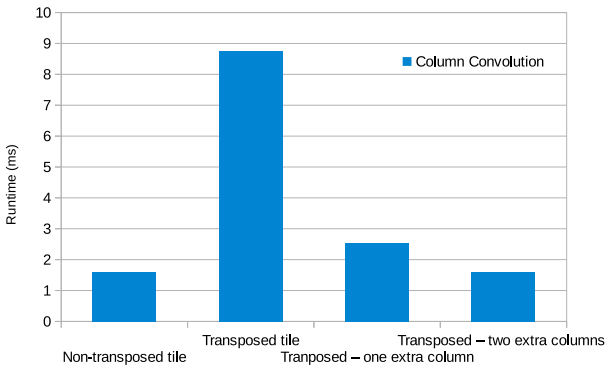


Figure 23: Performance of column convolutions when transposing tiles and adding extra columns

pattern in local memory. This optimization is applied in the example kernel code of the optimization guide for separable convolution by Nvidia [51]. This optimization is shown in Figure 22. Implementing this in low-level programming languages like OpenCL or CUDA requires to change memory allocation and accesses throughout the kernel.

PERFORMANCE The performance compared to the column convolution without transposing the local memory tile is illustrated in Figure 23. The kernel implementing the transposed tile column convolution is shown in Appendix A.1 in Listing 33. Unfortunately, transposing the tile before loading it into local memory is not enough to speed up the computation. By transposing the tile, we introduced so called *bank conflicts* which we discuss later when accessing the local memory. Resolving these bank conflicts eventually achieves a speedup of 1.01 compared to the column convolution without transposing the column convolution tile. The kernels implementing the tiled column convolution with resolved bank conflicts are shown in Appendix A.1 in Listing 34 and Listing 35. Although this optimization does not significantly improve the performance on the GPU we used to conduct our experiments, it is suggested by Nvidia to apply this optimization to the column convolution and we discuss how to functionally express the transposition of the tile before copying it to local memory. In Section 3.2.4.1, we discuss the concept of bank conflicts and how to resolve them using low-level primitives.

REPRESENTATION IN LIFT Using our low-level primitives and the *transpose* function, we are able to apply the transposition of the tile intuitively as shown in Listing 12. Instead of making multiple changes in the kernel which include error-prone index computations, we simply add *transpose* twice directly indicating the original intend of transposing the tile. The first *transpose* (line 22) transposes the tile as shown in Figure 22. After transposing, the tile is loaded into local memory

```
18  ...
19  slide2d 17 1 1 1 ∘
20    transpose ∘
21      toLocal(mapLocal1(mapLocal0 id)) ∘
22    transpose) tile))) ∘
23  slide2d 80 64 16 16 ∘ pad2d 8 8 0 0 b) input
```

Listing 12: Column convolution: Storing transposed tile in local memory

by every work-group as before. By transposing we change the type of the tile from a $n \times m$ matrix to an $m \times n$ matrix. However, the rest of the expression expects a matrix of the original type. The *slide2d* in line 19 creates the original neighborhoods in every column. Applying this function unchanged on the transposed tile obviously returns the result also transposed. Therefore, we apply *transpose* again in line 20 after loading it into local memory. This restores the orientation of the tile. The semantics of the expression remains unchanged because transposing a matrix (line 22), applying the identity (line 21) and transposing it again (20) returns the original matrix.

The only new rewrite rule that is needed to introduce the transposed tile is defined as follows:

REWRITE RULE 8 (TRANSPOSITION IDENTITY):

$$f \rightarrow f \circ \mathit{transpose} \circ \mathit{transpose} \mid \mathit{transpose} \circ \mathit{transpose} \circ f$$

The application of the identity function and the *transpose* shown in Listing 12 can commute without changing the semantics of the functional programming. Since the proof and steps to rewrite the expression to use store a transposed tile are obvious, we do not discuss them here.

Next, we discuss the concept of banks, bank conflicts and how to avoid them using our low-level primitives.

3.2.4.1 *Avoiding Bank Conflicts*

Local memory is organized in so-called *banks* of equal size. These banks are accessed simultaneously by multiple work-items of the same work-group. If two different work-items load from or store to two different banks, these memory transactions are parallelized achieving high memory bandwidth. However, if two different work-items access elements stored in the same memory bank, these accesses are executed sequentially. This is called a *two-way bank conflict*. Generally, n work-items of a warp accessing the same bank simultaneously is called a n-way bank conflict. The amount of banks and their size depend on the specific device. Devices with a compute capability 1.X contain 16 banks whereas devices with higher compute capability contain 32. Usually, a bank contains 32 bit words although

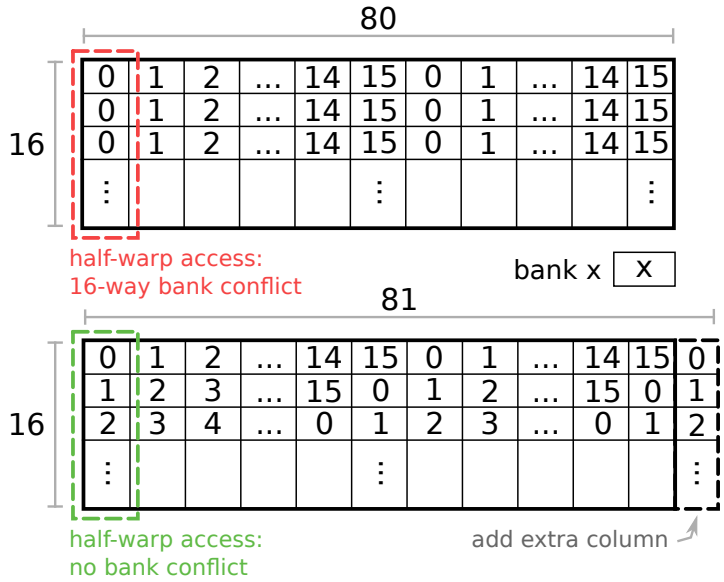


Figure 24: Column convolution: Arrangement of banks for transposed tile on GPUs with 16 banks

the size of a bank can be manually increased to 64 bit words on devices with compute capability 3.X to avoid performance pitfalls using double precision data. Another important factor is how local memory accesses are scheduled. On devices with compute capability 1.X, a memory request for a warp (32 work-items) is split in two separate requests for each half warp. On modern GPUs a complete warp simultaneously accesses the shared memory. When storing data to local memory, consecutive 32 bit words are stored in consecutive banks. Since we use single precision floating point numbers in our evaluation, every `float` is stored into a single bank. For example when storing a buffer containing 16×4 floating point numbers into local memory on a device with 32 banks, the floats in the first two rows (32 floats) are each stored in one of the 32 available banks consecutively. The first float in the third row is then again stored in the first bank and so on.

AVOIDING BANK CONFLICTS FOR HALF WARP ACCESSES Executing our column convolution on an older GPU of compute capability 1.X with 16 banks and shared memory requests for each half warp results in bank conflicts on local memory accesses. We discuss how to avoid bank conflicts for these devices as introduced in [51]. Afterwards we show that this solution is not sufficient for modern GPUs and discuss how to avoid bank conflicts on newer architectures as well.

The arrangement of banks for our transposed buffer containing 80×16 floats is shown in Figure 24. Each row contains 80 floating point numbers. The first consecutive 16 numbers are stored in the 16

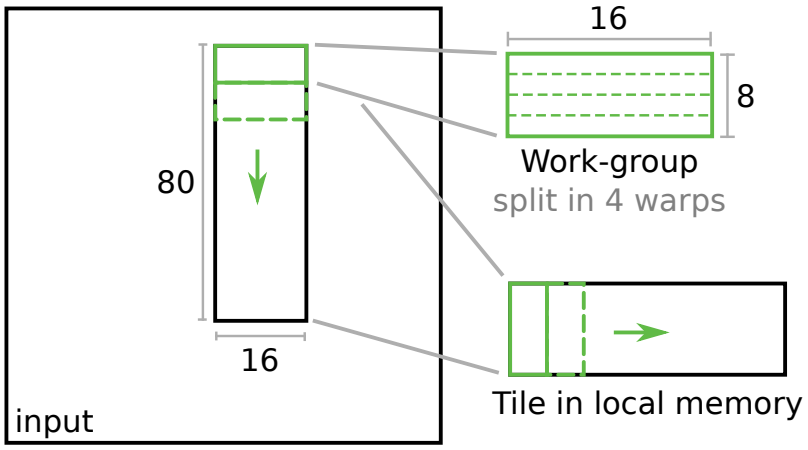


Figure 25: Column convolution: Memory access of a single work-group when transposing the tile before copying to local memory

banks 0 to 15. The 17th number is then again stored in bank 0 and so on. Since 80 is evenly divisible by 16, the last element of the first row is stored in bank 15. Therefore, the arrangement of banks for the second row is exactly the same as for the first row as visualized in Figure 24. The crucial question now is how these elements are accessed. In the hand-tuned version by Nvidia a work-group with 16×8 work-items is used. This work-group is split into four warps for execution. The global and local memory accesses for these work-groups are shown in Figure 25. A 16×80 tile is assigned to a 16×8 work-group. This work-group accesses the first eight rows of the tile and stores them in a transposed manner into local memory. Afterwards the same work-group accesses the next 8 rows and stores them into local memory until the whole buffer is copied. Global memory accesses are coalesced but the shared memory is accessed column wise since we transpose the tile before storing it. Assuming that memory requests of a warp are divided into half warp requests, the first half warp of the work-group accesses the first column of the tile in shared memory. Comparing this access pattern to the arrangement of banks depicted in Figure 24 reveals that every work-item of a half warp accesses the same bank. This results in a 16-way bank conflict, and all 16 accesses are executed sequentially. To resolve these bank conflicts, each work-item of the same half warp has to access another bank. A simple solution is to increase the local memory buffer by adding an extra column which remains unused during computation. However, this additional column changes the arrangement of the local memory banks as shown on the right side of Figure 24. Now all 16 work-items of each half warp access different banks, thus we avoided all bank conflicts.

In LIFT, every array xs has a dedicated size (for example n) encoded in its type: e. g. $xs : [\alpha]_n$. This size is used to determine the size of the OpenCL buffer that eventually contains the elements in

the kernel code. To express this optimization using our primitives, we first need to distinguish the *size* of a buffer and its *capacity*. The size specifies the amount of elements it contains while the capacity specifies the maximum number of elements which can be stored in this buffer. Thus, to avoid bank conflicts we want to increase a buffers capacity without changing its size. To functionally express this optimization, we introduce a new primitive similar to the high-level *pad* primitive. The low-level *increase* primitive is used to increase the capacity of a buffer without adding elements that should be included in the computation.

DEFINITION 3.6: *Let* xs *be an array of size* n *with elements* x_i *where* $0 < i \leqslant n$. *Let* l *and* r *be two positive integer values. The increase primitive is then defined as follows:*

$$\textit{increase}\ l\ r\ [x_1, \ldots, x_n] \overset{def}{=} [x_1, \ldots, x_n]$$

while it is guaranteed that the capacity of the buffer used to store xs *is increased by* l *elements on its left and* r *elements on its right end*

The type of increase is defined as follows:

$$\textit{increase} : \mathtt{int} \to \mathtt{int} \ \to [\alpha]_n \to [\alpha]_n$$

The computation of the stencil does not depend on the capacity of the buffer used to store the elements as long as the correct elements are loaded and stored. Therefore, the *increase* primitive indicates our code generator, which is discussed in the next chapter, to increase the capacity of the buffer in such a way that it does not affect the result of an output element. However functionally, the *increase* primitives equals the identity function. Hence, $\mathtt{id} = \mathtt{increase}\ l\ r$. Note that since we define $\mathtt{increase}$ as the identity function, LIFT's type system should be extended in future work introducing a *capacity* for arrays xs which might differ compared to its *size*. This would allow to functionally encode this optimization in the types of data structures.

Similar to the already introduced *pad2d* function, we introduce a *increase2d* function which is used in our column convolution to increase the size of the tile in shared memory:

DEFINITION 3.7: *Let* xs *be an array of size* m *whose elements are arrays of size* n. *Let* top, bottom, left, right *be positive integer values. The increase2d function is then defined as:*

$$\begin{aligned}&\textit{increase2d}\ \mathtt{top}\ \mathtt{bottom}\ \mathtt{left}\ \mathtt{right}\ xs \overset{def}{=}\\ &\quad(\textit{transpose} \circ \textit{increase}\ \mathtt{left}\ \mathtt{right} \circ\\ &\qquad \textit{transpose} \circ \textit{increase}\ \mathtt{top}\ \mathtt{bottom}\ )\ xs\end{aligned}$$

If top = bottom, left = right *we also write*

$$\textit{increase2d}\ \mathtt{top}\ \mathtt{left}\ xs$$

```
19  ...
20    transpose ∘
21      toLocal(mapLocal_0(mapLocal_1 id)) ∘
22    transpose ∘ increase2d 0 0 0 1) tile))) ∘
23  slide2d 80 64 16 16 ∘ pad2d 8 8 0 0 b) input
```

Listing 13: Column convolution: Avoiding bank conflicts by artificially increasing the buffer capacity

instead of

increase2d top top left left xs

The functional expression which applies this optimization using our new low-level primitive is shown in Listing 13.

AVOIDING BANK CONFLICTS FOR COMPLETE WARP ACCESSES

Finally, we examine if *increase* resolves bank conflicts on modern GPUs as this version of the column convolution is optimized for Nvidia devices with compute capability 1.X. Modern GPUs have 32 banks which are accessed by a complete warp. The arrangement of banks without adding a column using a 80×16 buffer in shared memory is shown in Figure 26. Every work-item accesses the same bank as eight other work-items. Thus without adding an extra column we have bank conflicts too, however this time an 8-way bank conflict. By adding another column to the tile we shift the banks again but it is not sufficient to resolve all bank conflicts since two work-items of a warp access bank 0. To resolve all bank conflicts we have to add two columns which fully resolves the bank conflicts. Another solution would be to change the width of the tile from 16 to 32. This way, we have similar setup as before. A whole warp accesses one row in global memory and one column in shared memory when storing the transposed tile. By adding one column to this new tile we shift the banks in another way such that no bank is accessed twice.

The rewrite rule needed to introduce the *increase* primitive is defined as:

REWRITE RULE 9 (INCREASE IDENTITY):

$$f \rightarrow \textit{increase}\ l\ r \circ f \mid f \circ \textit{increase}\ l\ r$$

As we have seen in this section, some optimizations are highly depending on low-level hardware details. For example one has to carefully consider the choice of tile sizes and work-group configurations according to the warp scheduling and the amount of banks in a GPU. In the next section, we examine one final optimization to further improve the performance of stencil computations by reducing boundary checks.

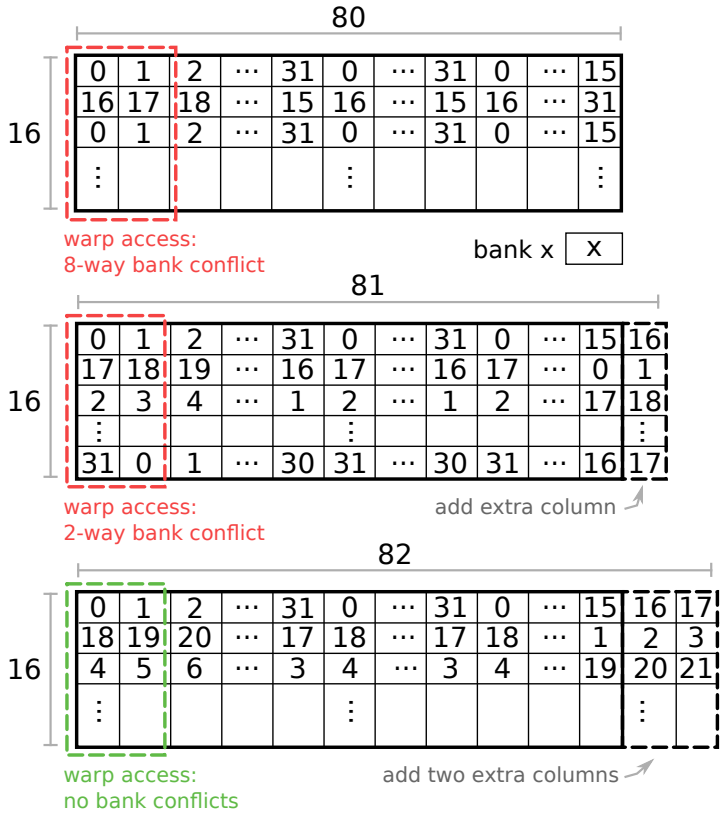


Figure 26: Column convolution: Arrangement of banks for transposed tile on GPUs with 32 banks

3.2.5 *Loop Unrolling and Reducing Boundary Checks*

Stencil kernels in low-level languages like OpenCL or CUDA typically contain small loops in their imperative kernel code. Similarly, our low-level primitives like *mapSeq* or *mapLocal* are used in the code generation which is described later to create for-loops. However if the loop is iterated only a few times, it is almost always beneficial to unroll the loop to avoid its overhead. A potentially beneficial place to unroll loops is when loading a tile from global to local memory. Taking the column convolution as an example, every work-group copies a 16×80 tile to local memory using work-groups of size 16×8. Thus, the tile is loaded to local memory iterating a loop 10 times as visualized in Figure 27. The critical parts of this process to load this tile from global to local memory are the halo regions. When copying a halo region to local memory we need to check for potential out-of-bounds accesses and apply the boundary handling in these cases. An intuitive implementation of this loop can be found in the kernels that implement the optimization discussed in the last section, for example in Appendix A.1 in Listing 34. The loop that copies the tile to local memory is shown in Listing 14 for convenience. Boundary checks for both, the upper halo (line 13) and the lower halo (line 14), are

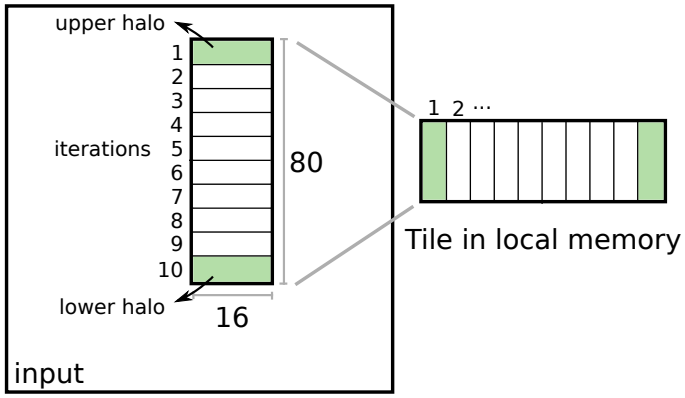


Figure 27: Visualizing iterations to load a tile from global to local memory

```
9   ...
10  for (int lid1 = get_local_id(1); lid1 < 80; lid1 = (8 + lid1)) {
11    int x = lid0 + (16 * wg0);
12    int y = (-8 + lid1 + (64 * wg1));
13    y = max(0, y);
14    y = min(4095, y);
15
16    L[lid1 + (81 * lid0)] = IN[x + (4096 * y)];
17  }
18  ...
```

Listing 14: Handwritten for-loop that copies a tile from global to local memory

executed in every iteration although each one is only needed in one specific iteration. By unrolling the loop into 10 single iterations executed subsequently, we are able to omit boundary checks in almost all iterations and only execute them when necessary.

PERFORMANCE The performance of a handwritten kernel that applies loop unrolling to reduce boundary checks compared to the previous column convolutions is illustrated in Figure 28. The kernel that implements the unrolled loop is shown in Appendix A.1 in Listing 36. Reducing boundary checks using loop unrolling achieved a speedup of 1.13 compared to the previous best column convolution.

REPRESENTATION IN LIFT Now, we examine how to functionally express this optimization using LIFT's IL. To express loop unrolling using our low-level primitives we introduce a new primitive called *mapSeqUnroll*. It is defined exactly as the *mapSeq* primitive with the slight difference that it indicates the code generation to always emit unrolled loops if possible. This is when the iteration count of a loop is statically known. Listing 15 shows the column convolution using tiling and local memory without unrolling the copying of the tile to local memory. Using this new primitive, we are able to express the

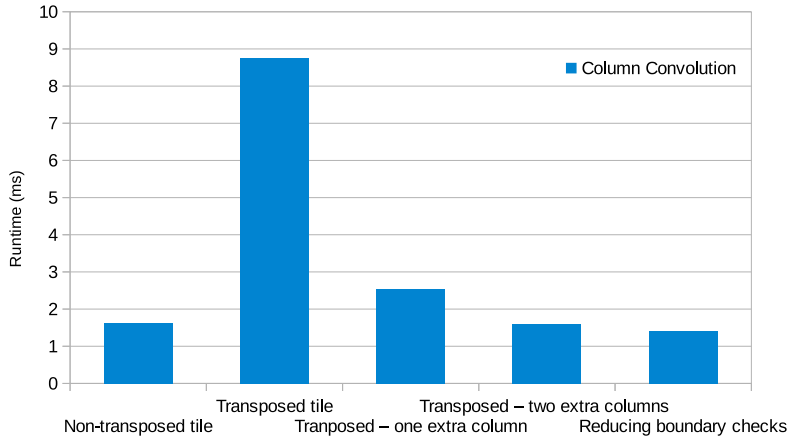


Figure 28: Performance of column convolution with reduced boundary checks compared to previous column convolutions

```
19  ...
20    transpose ∘
21      // using a configuration of 16 x 8 local work-items
22      toLocal(mapLocal_1(mapLocal_0 id)) ∘
23        transpose ∘ increase2d 0 0 0 1) tile:[[float]_16]_80))) ∘
24    slide2d 80 64 16 16 ∘ pad2d 8 8 0 0 b) input
```

Listing 15: Expression to copy a tile from global to local memory

optimization discussed above as show in Listing 16. We first explicitly divide the tile in 10 chunks using *split* 8 as shown in line 26 which results in an array of chunks. The tile is copied to local memory in 10 steps while the *mapSeqUnroll* states our intent to use an unrolled loop. This allows omit boundary checks for the first and last iteration which happens automatically during code generation which we discuss in the next chapter.

3.3 SUMMARY

In this chapter we started with a functional expression that described a 17×17 convolution stencil computation which is easy to write by a programmer familiar with functional programming. This expression is shown again in Listing 17. We incrementally formalized and applied multiple well-known optimizations for stencil computations like overlapped tiling and separating the convolution into a row and column convolution. In order to express some of these optimizations, we extended the functional low-level language of the LIFT framework by adding new low-level primitives like *increase* or *mapSeqUnroll* and reused the earlier introduced *slide* primitive. We systematically applied all optimizations one after each other and in a way an optimizing compiler could perform automatically. Since all applied optimiza-

```
19          ...
20          transpose ∘
21            // join the partitioned array in local memory again
22            map(join) ∘
23              // unroll the loop using the new mapSeqUnroll primitive
24              toLocal(mapLocal_1(mapSeqUnroll(mapLocal_0 id))) ∘
25            // split tile in 10 chunks of size 8
26            map(split 8) ∘
27          transpose ∘
28        increase2d 0 0 0 1) tile))) ∘
29      slide2d 80 64 16 16 ∘ pad 8 8 0 0 b) input
```

Listing 16: Low-level expression to copy a tile from global to local memory using loop unrolling

```
1 conv = λ b weights input .
2   (map(map( λ nbh .
3     (reduce (+) 0 ∘ map (∗) ∘ zip) (join nbh) weights)) ∘
4   slide2d 17 1 ∘ pad2d 8 8 b) input
```

Listing 17: High level expression for a 17×17 convolution

tions are provably semantics preserving we can guarantee that we did not change the semantics of the original high-level program. Finally we end with a fully optimized functional program of low-level primitives as shown in Listing 18.

Using the optimizations discussed in this chapter we are able to improve the performance of the 17×17 convolution achieving a speedup of 40.9 comparing the naive version with the most optimized version. This result emphasizes the importance of applying these optimizations in a structured way as there are many caveats during the optimization process where a wrong decision in optimization parameters may lead to a significant performance drawback.

At this point, one might ask two specific questions considering an arbitrary stencil expression: Which optimizations should be applied in which order and when is an application sufficiently optimized? This question is a well-known problem, also known as *phase ordering*, in optimizing state-of-the-art compilers like *clang* or *gcc*. To this day, there is no sufficient answer to this problem and optimizing compilers still use heuristics to decide which optimizations are applied when. However when talking about *optimizations* we are not considering *instruction combining* or *inlining* like the existing compilers. Furthermore, we talk about stencil-specific optimizations like *overlapped tiling* which existing compilers can not apply. Therefore, we defined means to systematically (and possibly automatically once heuristics or performance models are defined) apply application-specific optimizations using our functional approach.

```
1  // separating convolution - Section 3.2.3
2  conv = λ b weights_X weights_Y input .
3    (convColumn b weights_Y ∘ convRow b weights_X) input
4
5  convRow = λ b weights input .
6    // using low-level primitives - Section 3.1
7    (mapWorkgroup_1 (mapWorkgroup_0 (λ tile .
8      (mapLocal_1 (mapLocal_0 (λ nbh .
9        (toGlobal (mapSeq id) ∘
10         reduceSeq (+) 0 ∘ mapSeq (*) ∘ zip) (join nbh) weights)) ∘
11       slide2d 1 1 17 1 ∘
12     // utilizing local memory - Section 3.2.2.2
13     toLocal (mapLocal_1 (mapLocal_0 id))) tile)) ∘
14   // overlapped tiling - Section 3.2.2
15   slide2d 1 1 144 128 ∘ pad2d 0 0 8 8 b) input
16
17 convColumn = λ b weights input .
18   (mapWorkgroup_1 (mapWorkgroup_0 (λ tile .
19     (mapLocal_1 (mapLocal_0 (λ nbh .
20       (toGlobal (mapSeq id) ∘
21         reduceSeq (+) 0 ∘ mapSeq (*) ∘ zip) (join nbh) weights)) ∘
22       slide2d 17 1 1 1 ∘
23         transpose ∘
24           map(join) ∘
25             toLocal (mapLocal_0 (mapSeqUnroll (mapLocal_1 id))) ∘
26           // reduce boundary checks - Section 3.2.5
27           map(split 8) ∘
28         // transpose tile - Section 3.2.4
29         transpose ∘
30       // avoid bank conflicts - Section 3.2.4.1
31       increase2d 0 0 0 1) tile)) ∘
32   // global memory coalescing - Section 3.2.3.3
33   slide2d 80 64 16 16 ∘ pad2d 8 8 0 0 b) input
```

Listing 18: Low-level expression for a 17×17 convolution applying all optimizations discussed in this thesis

4

GENERATING HIGH PERFORMANCE OPENCL CODE

In this chapter we explain how we implemented the code generation producing high performance OpenCL kernels using our low-level expression written in our functional data parallel IL (Intermediate Language) introduced in the previous chapter.

Given an expression written in LIFT's IL, the compilation process is divided into several steps as visualized in Figure 29. These steps are discussed in more details below:

TYPE INFERENCE The first step in LIFT's compilation process is a static type check and type inference. The compiler implements a *dependent type systems* that considers array lengths, possible ranges of values which will be of more importance later, and OpenCL memory address spaces. The static type check validates that input and output types match at every step of the computation. This ensure that all primitives are composed and nested correctly preventing the user to write invalid code. Type checking and inference is one of the most important steps during the compilation process because the type information is key to achieve high performance OpenCL code. For example, this information is extensively used in the next two steps

MEMORY ALLOCATION During memory allocation, the type information received in the first step is used to determine the required buffer size and corresponding OpenCL address space for any significant intermediate result. An intermediate result is regarded as significant if data has been modified. A simple

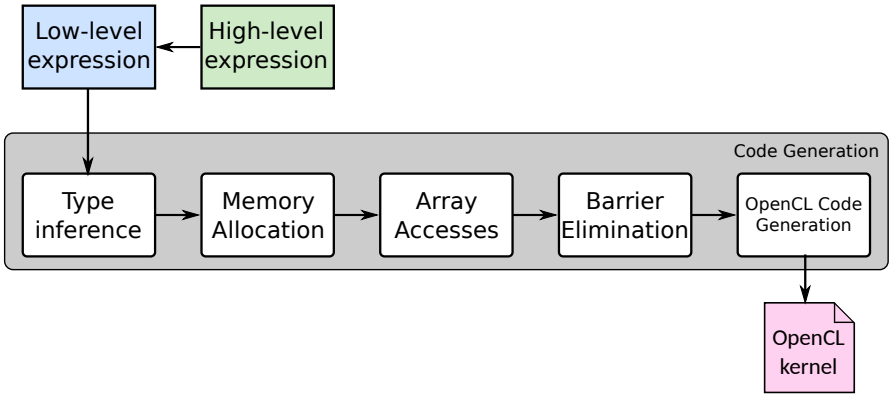


Figure 29: Compilation steps to compile a high-level expression to an OpenCL kernel

memory allocation would allocate a new buffer for ever intermediate result. This would be highly inefficient since primitives like *slide* or *pad* only shape the data before it is accessed instead of modifying it.

ARRAY ACCESSES All array accesses in the LIFT framework are implicit rather than explicit. Each primitive implicitly defines how its input should be accessed - *map* for example is defined to access each element once and apply a unary function to it. Since there are no means to explicitly access arrays, data races are avoided by construction. However, this introduces two main challenges: First, unnecessary intermediate results need to be avoided. Therefore, one has to store the information of how memory should be accessed in the compiler when a data-layout primitive is encountered. Second, accesses to multidimensional arrays which have a flat representation in memory need to be efficiently converted to one dimensional accesses.

BARRIER ELMINATION To ensure memory consistency, threads that access the same memory location must synchronize. The only primitives that potentially allow multiple threads to access the same memory location are the parallel *map* primitives. Therefore, all threads need to synchronize after every occurring parallel *map*. This is realized be emitting a `return` after *mapWorkgroup* and *mapGlobal* because OpenCL does not allow to synchronize global threads or work-groups. However, a barrier is emitted after every *mapLocal* primitive to synchronize all work-items of a work-group. In some cases these barriers can be eliminated, for example when there is no sharing because all work-items continue to use the same memory locations. If it can be statically proved that barriers are not necessary, these barriers are eliminated.

OPENCL CODE GENERATION As a final step, the lift compiler generates the low-level OpenCL kernel code. The compiler follows the data flow and emits matching code snippets for every primitive. At this point, every previous compilation step has been executed successfully and low-level optimizations are performed. As an example, using the type information received earlier, the compiler infers the amount of threads assigned to a *mapLocal* primitive. Therefore it can eliminate the loop which would normally be emitted if the number of threads exceeds or is equal to the amount of elements processed.

All these compilation steps emphasize the importance of preserving the information captured in the functional primitives and their types. In contrast to most existing code generation frameworks, LIFT generates low-level loop based code at the very last stage of the com-

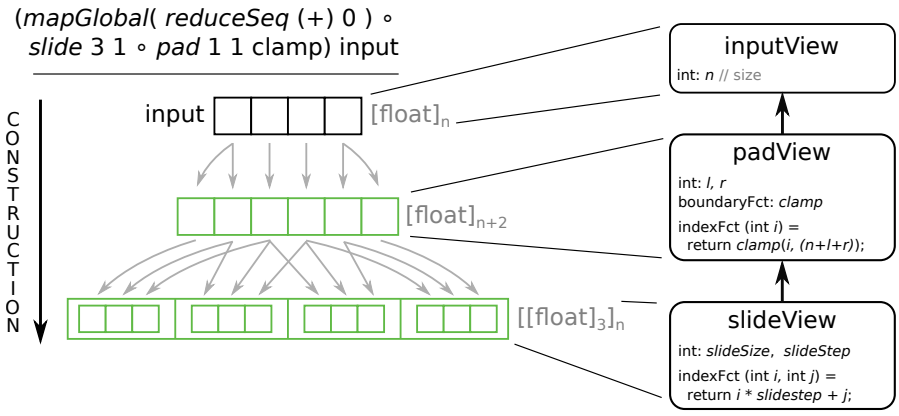


Figure 30: Construction of LIFT's views for a 3-point Jacobi stencil

pilation process and is therefore able to utilize unique opportunities for optimizations as much as possible.

In the following section we discuss LIFT's view system which is used to handle data-layout primitives during the *Array Access* stage. Since both new high-level primitives (*slide* and *pad*) introduced in this thesis shape the layout of the data rather than modifying it, this step is of the compilation process is particularly interesting.

4.1 LIFT VIEW SYSTEM

When a data-layout primitive is encountered during compilation, a compiler-internal data structure called *view* is created. Views store the information how the memory should be accessed by subsequent primitives. Once data is going to be modified, the view system is used to resolve memory accesses and the result is stored in a new buffer allocated in the memory allocation phase. This view system can be compared to views in SQL which provide a virtual result of a query that can also be used in subsequent queries.

The view system is divided into two parts: the view *construction* where information is gathered of how to access data, and the view *consumption* where this information is used to generate array accesses. Primitives that do not modify data but shape it construct a view data structure. These particularly include the two new primitives *slide* and *pad* which we focus on in this section.

We explain both construction and consumption using a simple 3-point Jacobi stencil example shown in Listing 19:

```
1 mapGlobal(reduceSeq (+) 0) ∘ slide 3 1 ∘ pad 1 1 clamp
```

Listing 19: Low-Level Expression for a simple 3-Point Jacobi stencil

VIEW CONSTRUCTION The View construction for this example is visualized in Figure 30. The left-hand side visualizes the intermedi-

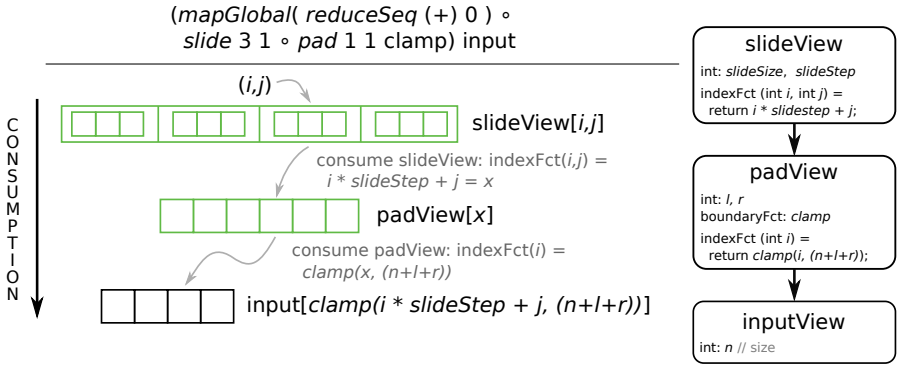


Figure 31: Consumption of LIFT's views for a 3-point Jacobi stencil

ate results when applying the data-layout primitives *pad* and *slide* successively. The right-hand side visualizes the corresponding view data structures representing these intermediate results in our compiler. Furthermore, the input is also represented by a corresponding *inputView* as shown on the top right corner. Views are stored in a linked list, such that every view is connected to the previously created view. The first primitive that is encountered is the *pad* primitive. Since it is data-layout primitive, a view data structure is created which is linked to the *inputView*. Every view is defined by the type of the primitive which caused its creation. We start with an input array of length n. Once the *pad* primitive is encountered, a new view is created storing information regarding the type of this primitive. Thus, the *pad*-view stores the amount of elements added on the left and right-hand side and the utilized boundary function. This information is necessary to later compute array indices using the function *indexFct* during view consumption. Intuitively the *pad*-view represents an array of size $n + 2$ as shown in the center of Figure 30,

The next primitive encountered is the *slide* primitive which again creates another view structure that is connected to its predecessor *pad*-view. The *slide*-view stores the step and size of the encountered *slide* primitive to define its *indexFct*. This view represents a two dimensional data structure as shown at the bottom. Therefore, the index function defined by the *slide*-view needs to transform a two dimensional access to a one dimensional access.

VIEW CONSUMPTION The first primitive after *pad* and *slide* which we already dealt with is a *mapGlobal* primitive with a nested *reduceSeq* primitive. These primitives are not data-layout primitives but describe computation which leads to the consumption of the views created so far. The consumption of the views is visualized in Figure 31. Again, the intermediate results of evaluating the *pad* and *slide* primitive are shown on the left-hand side while the view data structures are shown on the right-hand side. Views are always consumed in the opposite order of construction. The most recently created view is the

slide-view (on the top right) which represents a two dimensional array. This array is accessed by the *mapGlobal(reduceSeq* $(+)$ $0)$ function. The left-hand side of Figure 31 visualizes how accessing one particular element of the *slide*-view is resolved. This element is accessed using an outer index provided by the *mapGlobal* primitive (i in the figure) which selects a specific neighborhood created by *slide* and an inner index provided by the *reduceSeq* primitive that accesses elements inside a neighborhood (j in the figure). Since the *slide*-view only simulates a two dimensional data structure that has a flattened representation in memory, the two dimensional access needs to be converted to a one dimensional access again. This conversion happens during view-consumption is and is defined in the *slide*-view by the index function *indexFct*. The one dimensional access is computed using the formula $i * slideStep + j$ as illustrated in Figure 31.

This newly computed index is then used to access the predecessor *pad*-view which itself resolves another layer of indirection. The *pad*-view (in the middle on the right) is consumed by applying the boundary function to the given index. In our case, the $clamp$ function ensures that indices that would exceed the bounds of the input array are clamped to the outermost in-bound indices. The final index computation is then passed to the *inputView* which is used to emit the array access in the OpenCL kernel code.

SUMMARY The original input array is now accessed using the computed index $clamp(i * slideStep + j, (n + l + r))$ defined by the index function. This index was computed by consuming one view at a time while passing on each result until there is no more layer of indirection.

The code implementing array accesses for this particular example is show in Listing 20. The input array is accessed in lines 6 and 7 exactly as visualized in Figure 31 and as discussed above.

```
1 for (int gid = get_global_id(0);
2     gid<N;gid = (gid + get_global_size(0))){
3   acc = 0.0f;
4   for (int j = 0;j<3;j = (1 + j)){
5     acc = add(acc,
6     IN[( ((-1 + gid + j) >= 0) ?
7        ( ((-1 + gid + j) < N) ? (-1 + gid + j) : (-1 + N) ) : 0 )]);
8   }
9 }
```

Listing 20: OpenCL code generated for accessing the input array for 3-point Jacobi stencil

Similar to *slide* and *pad*, each data-layout primitive like *split*, *join* or *reorder* defines its own view data structure including instructions of how to consume incoming indices. Views of different data-layout primitives can be arbitrarily connected as long as the functional program is type safe. This may lead to an almost arbitrary amount of

```
1 (((((2 * i) + (i * N) + j) / (2 + N)) +
2  ((((2 * i) + (i * N) + j) % (2 + N)) * 2) +
3  ((((2 * i) + (i * N) + j) % (2 + N)) * M)) / (2 + M))
```

Listing 21: Unsimplified automatically generated array andex

layers of indirection that need to be resolved before accessing memory. Depending on the amount of connected views that need to be consumed, index computations for accessing arrays might grow significantly. This fact motivates an important phase during code generation which we discuss in the following section.

4.2 INDEX COMPUTATION SIMPLIFICATION

Expressing computations using functional primitives allows to omit explicit array accesses. Race conditions are avoided by construction since arbitrary accesses to memory are not allowed. Furthermore, automatically generating correct indices is easily done in small steps by constructing and consuming view data structures as explained in the previous section. However, in order to generate efficient array accesses, index computations need to be significantly simplified. Chaining multiple views can easily lead to large automatically generated indices containing multiple repetitive subexpressions as shown in Listing 21. This is a small but representative example as the automatically generated indices might span across hundred or more lines of code. This arithmetic expression can easily be simplified by a human as we see in the following. However if this expression is emitted unchanged into the OpenCL kernel, multiple costly and unnecessary operations are executed which results in a significant performance drawback as we observe in the following chapter. Therefore, arithmetic expression simplification is an inevitable and highly important phase during code generation in order to generate high performance OpenCL kernels.

Let us now examine how the expression shown in Listing 21 can be simplified. For convenience we repeat the index computation and use a mathematical notation:

$$\frac{\frac{2i+iN+j}{2+N} + 2((2i+iN+j)\%(2+N)) + M((2i+iN+j)\%(2+N))}{2+M}$$

Here % represents OpenCL's %-modulo operation and every division is integer division. LIFT's arithmetic expressions are always transformed to a certain canonical form which defines that terms are always written as sums of products. By factorizing some of these sums, we are able to reduce fractions as shown in the following equations:

$$\frac{\frac{(2+N)i+j}{2+N} + (2+M)((2+N)i+j)\%(2+N))}{2+M}$$

$$
\begin{aligned}
&= \underbrace{\frac{i + \frac{j}{2+N}}{2+M}}_{0} + \underbrace{((2+N)i + j)\%(2+N)}_{j} \\
&= j
\end{aligned}
\tag{15}
$$

If we can statically proof that $j < 2 + N$ we can simplify the arithmetic subexpression $\frac{i}{2+N}$ to zero since we are computing integer division which always rounds the result to the closest integer that is smaller or equals the result. Subsequently, if we can also statically proof that $i < 2 + M$ we can simplify the whole fraction of Equation 15 to 0. The right term of the sum can easily be simplified to j using modulo arithmetic. Using LIFT's symbolic arithmetic simplifier which relies on type information to extract lengths of arrays and possible ranges of all variables, we are able to automatically perform all simplification steps discussed automatically. Thus, in this particular example we were able to reduce the index computation shown in Listing 21 which contains 24 operations to a single variable.

In this work, we significantly improved LIFT's arithmetic expression simplifier to simplify indices generated from expressions containing new primitives like *slide* and *pad* introduced in this thesis.

4.3 OPENCL CODE GENERATION

The final step of the LIFT compilation process is the OpenCL code generation. In this stage, the compiler follows the data flow of the given expression and emits small code snippets for certain primitives as discussed above. The generated code for our 3-point Jacobi example we discussed in the previous section is show in Listing 22. For low-level *map* primitives including the *mapGlobal* primitive used in this example, for-loops are generated as shown in line 18. This particular for-loop iterates over the global-IDs of all work-items because we used the *mapGlobal* primitive. If we statically know during compilation that the number of work-items exceeds the number of elements to process, no for-loop is emitted. The *reduceSeq* primitive nested inside the *mapGlobal* primitive causes the second for-loop in line 23 which is nested in the for-loop emitted for the *mapGlobal* primitive. For primitives like *slide* or *pad*, no code is emitted because their information has been used in the views and during array index computations. This results in the array access shown in line 25 as discussed in the previous chapter. Since *reduceSeq* implicitly stores its result in private memory we applied *toGlobal(mapGlobal*(`id`)*)* afterwards to copy the result back to global memory. This can also be observed in the generated code. After the for-loop emitted for the *reduceSeq* primitive (lines 22-27) the result of the reduction is stored in the variable `v__11` (line 11) which resides in the private memory of each work-item. Ap-

```
// (toGlobal(mapGlobal id) o
// mapGlobal(reduceSeq (+) 0) o slide 3 1 o pad 1 1 clamp) input

float add(float x, float y){
  { return x+y; }
}
float id(float x){
  { return x; }
}
kernel void KERNEL(const global float* restrict v__9,
    global float* v__15, int v_N_0){

  /* Static local memory */
  /* Typed Value memory */
  float v__11;
  /* Private Memory */
  for (int v_gl_id_6 = get_global_id(0);v_gl_id_6<v_N_0;
       v_gl_id_6 = (v_gl_id_6 + get_global_size(0))){
    float v_tmp_20 = 0.0f;
    v__11 = v_tmp_20;
    /* reduce_seq */
    for (int v_i_7 = 0;v_i_7<3;v_i_7 = (1 + v_i_7)){
      v__11 = add(v__11,
                  v__9[( ((-1 + v_gl_id_6 + v_i_7) >= 0) ?
                       ( ((-1 + v_gl_id_6 + v_i_7) < v_N_0) ?
            (-1 + v_gl_id_6 + v_i_7) : (-1 + v_N_0) ) : 0 )]);
  }
  /* end reduce_seq */
  }
  for (int v_gl_id_8 = get_global_id(0);v_gl_id_8<v_N_0;
      v_gl_id_8 = (v_gl_id_8 + get_global_size(0))){
    v__15[v_gl_id_8] = id(v__11);
  }
}
```

Listing 22: OpenCL kernel generated for a 3-point Jacobi stencil

plying *toGlobal(mapGlobal(*id*))* results in the last for-loop emitted in line 30 which is used to copy the result to the output buffer. Note that *id* used in the second *mapGlobal* primitive and the operator + used in the *reduceSeq* primitive are the only customizing functions in this expressions. Both customizing functions have been emitted as simple C functions above the kernel (lines 4 and 7) and are called at the appropriate places (line 23 and line 32)

4.4 SUMMARY

In this chapter we briefly discussed the compilation stages of the LIFT framework. We specifically focused on the view system which is used to avoid unnecessary intermediate results. These views are used as layers of indirection before accessing data stored in memory. However, when many views are connected together, the automatically generated array indices become complicated. Therefore, we also particularly focused on how to optimize these generated indices and discussed the significant simplification potential by using simple rules of algebra. Finally we analyzed a generated OpenCL kernel for a simple 3-point Jacobi example and discussed which primitives of the given expression caused which parts of the kernel to be emitted.

5 EVALUATION

In this chapter, we evaluate the performance of our functional code generation approach for stencil applications. Specifically, we examine if our low-level expression are compiled to OpenCL kernels with competitive performance compared to the handwritten kernels discussed in Chapter 3. Furthermore, we evaluate the impact of unsimplified array accesses to examine the importance of arithmetic expression simplification discussed in the previous chapter. Finally, compare our generated kernel for the 17×17 convolution example to the Nvidia Toolkit example *convolutionSeparable* discussed in [51].

5.1 EXPERIMENTAL AND HARDWARE SETUP

To conduct our experiments, we used an Nvidia Kepler K20c Graphics Card (Compute Capability 3.5) and the driver version 361.42. It contains 13 SMX (Streaming Multiprocessors) each containing 192 processors cores (2496 cores in total) running with a clock speed of 706 MHz. Each SMX is able to schedule 4 warps concurrently using four warp schedulers and eight instruction dispatch units. Furthermore, each SMX contains 64 KB configurable local memory and L1 cache. In our configuration, the size of the local memory is 48 KB and the size of the L1 cache 16 KB. The Kepler K20c contains 5GB of global memory size with a bandwidth of 208 GB/sec. We use the Ubuntu 16.04.1 LTS Operating System with the kernel version 4.4.0-36-generic x86_64. The used OpenCL Platform is NVIDIA CUDA version 1.2 CUDA 8.0.20.

Every experiment was conducted 100 times and we always present the median of the measured results. We only consider kernel runtimes, hence, we ignore data transfer times from and to the OpenCL device.

5.2 PERFORMANCE OF HANDWRITTEN CONVOLUTION KERNELS

In the previous chapter, we reported the performance of handwritten OpenCL kernels to evaluate the effect of specific optimizations. We were able to speed up the naive version by a factor of 41. This emphasizes the importance of optimizing stencil computations for GPUs. In this section we analyze and compare the performance of generated OpenCL kernels using the low-level expressions written in LIFT's IL to the handwritten OpenCL kernels. Figure 32 visualizes the performance of the handwritten kernels implementing all optimizations

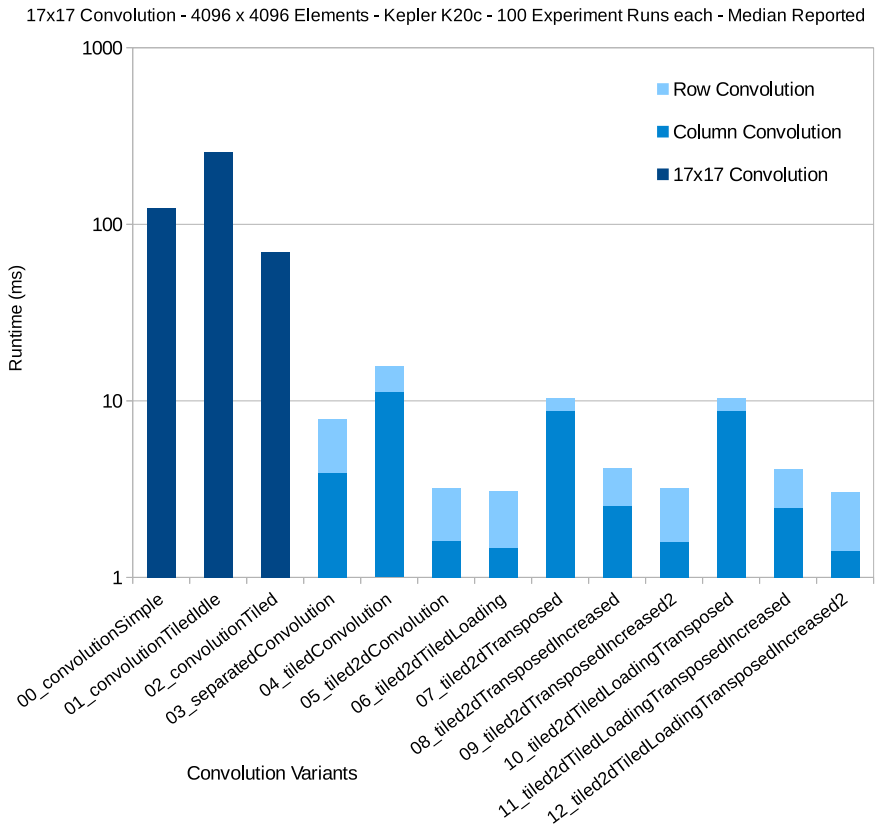


Figure 32: Performance of handwritten OpenCL kernels (incrementally) implementing specific optimizations

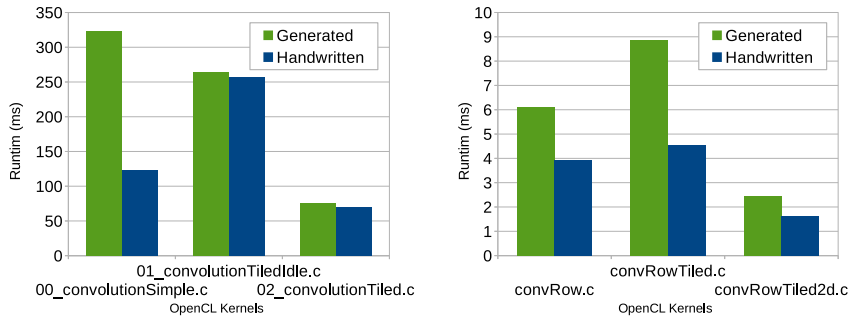
discussed in the previous chapter using a logarithmic scale. Table 1 shows a short description of which optimization is implemented by which kernel and where it is discussed in depth in the previous chapter. Since we split the convolution in a row and column convolution, thus in two different kernels, we are able to compare three different kinds of kernels: The kernels computing the complete convolution, the kernels that compute the row convolution and finally the kernels that compute the column convolution which showed the most optimization potential.

5.3 PERFORMANCE OF GENERATED CONVOLUTION KERNELS

We start by comparing the first three kernels which all compute the complete 17×17 convolution without separating it into two kernels. To generate the OpenCL kernels, we use the low-level expressions discussed in the previous chapter. Figure 33 (a) compares the performance of the generated kernels using the expressions discussed in Section 3.2.1 and Section 3.2.2 to the handwritten references. All generated kernels are slower compared to the handwritten versions while the first kernel being up to 2.61 times slower. This performance differ-

KERNEL(S)	DESCRIPTION	SECTION
00	No optimizations | Every global thread computes one output element	Section 3.2.1
01	Overlapped Tiling and local memory usage | Idle work-items in work-group during computation	Section 3.2.2
02	Overlapped Tiling and local memory usage | Improved work-group size	Section 3.2.3.3
03	Separate Convolution into row and column convolution	Section 3.2.3
04	Separation + Overlapped Tiling	Section 3.2.3.2
05	Separation + Overlapped Tiling | improved tile sizes	Section 3.2.3.3
06	Separation + Overlapped Tiling + reducing boundary checks	Section 3.2.5
07	Separation + Overlapped Tiling + transposing column convolution tile	Section 3.2.4
08	Separation + Overlapped Tiling + transposing column convolution tile + increase buffer to avoid bank conflicts	Section 3.2.4.1
09	Separation + Overlapped Tiling + transposing column convolution tile + resolve all bank conflicts	Section 3.2.4.1
10	Separation + Overlapped Tiling + transposing + reducing boundary checks	Section 3.2.5
11	Separation + Overlapped Tiling + transposing + reducing boundary checks + increase buffer to avoid bank conflicts	Section 3.2.4.1
12	Separation + Overlapped Tiling + transposing + reducing boundary checks + resolve all bank conflicts	Section 3.3

Table 1: Description of evaluated handwritten OpenCL kernels including references to sections where each optimization is discussed



(a) Comparing performance of the generated 17x17 convolution kernels to the handwritten references

(b) Comparing performance of the generated row convolution to the handwritten convolutions

Figure 33: Comparing generated kernels to handwritten references for complete and row convolution

ence is explainable by comparing the generated code to the handwritten kernel. Listing 23 compares the generated for-loop to compute a single output element to the handwritten for-loop. The generated kernel contains a large unsimplified arithmetic expression spanning more than 25 lines. Obviously, executing all operations in this expression causes the significant performance difference compared to the handwritten version. This shows that there is still potential improvement in arithmetic expression simplification during code generation. At the same time, it emphasizes the importance of simplifying these expression as they have a significant impact on performance. The other two generated kernels only show minor performance differences.

PERFORMANCE OF ROW AND COLUMN CONVOLUTION KERNELS

Next, we compare the performance of generated row convolution kernels to handwritten ones. These results are shown in Figure 33 (b). Comparing the performance of the row convolution kernels, we observe a significantly lower difference between generated and handwritten kernels compared to the full convolution examples. However, the generated kernels are still measurably slower than the handwritten references. Again, this is caused by arithmetic expressions in array indices that are not as simple as possible.

Finally, we compare the performance of the column convolution kernels for which we introduced several low-level optimizations in the previous chapter. For example, we introduced to transpose a tile before loading it into local memory while avoiding bank conflicts. Thus, the column convolution is the most optimized expression we examined in this thesis. Figure 34 shows the performance differences of the generated and handwritten column convolution kernels. Every generated column kernel matches the performance of the handwritten OpenCL kernels. Therefore, the more an expression is optimized,

```
// generated for-loop
for (int v_i_11 = 0;v_i_11<289;v_i_11 = (1 + v_i_11)){
  v__16 = multAndSumUp(v__16, v__13[((((4096 * ((v_gl_id_9 +
  (v_i_11 / 17) + (4112 * ( ((-8 + v_gl_id_10 +
  (v_i_11 % 17)) >= 0) ? ( ((-8 + v_gl_id_10 +
  (v_i_11 % 17)) < 4096) ? (-8 + v_gl_id_10 +
  (v_i_11 % 17)) : 4095 ) : 0 ))) % 4112)) +
  // ... skipping 25 lines
  ( ((-8 + v_gl_id_10 + (v_i_11 % 17)) >= 0) ?
  ( ((-8 + v_gl_id_10 + (v_i_11 % 17)) < 4096) ?
  (-8 + v_gl_id_10 + (v_i_11 % 17)) : 4095 ) : 0 )))
  % 4112)) : 4095 ) : 0 )))], v__14[v_i_11]);}

// handwritten for-loop
for (int i = 0; i < 289; i++) {
  int x = gid0 - 8 + (i % 17);
  int y = gid1 - 8 + (i / 17);
  x = max(0, x);
  x = min(x, 4095);
  y = max(0, y);
  y = min(y, 4095);
  acc = acc + (IN[x + 4095 * y] * W[i]);
}
```

Listing 23: Comparison of for-loops that copy a tile from global to local memory

the better are our performance results for our generated kernels. In case of the column convolution kernels, the arithmetic expressions are as simple as possible. Therefore, they do not cause a performance drawback as in the other kernels seen before.

To summarize, every optimization discussed in the previous chapter is successfully encoded using functional low-level expressions. The previous figures showed that our generated kernels have competitive performance compared the handwritten references. In the cases where the generated kernels are significantly slower, its just a matter of improving the arithmetic expression simplification instead of improving the functional expression.

5.4 MEASURING THE OVERHEAD OF UNSIMPLIFIED ARITHMETIC EXPRESSIONS

Unsimplified arithmetic expressions caused the performance drawbacks of the kernels evaluated in the previous section. In this section, we examine the impact of our arithmetic expression simplifier by measuring the performance of kernels with and without simplified expressions. The results are shown in Figure 35. This figure shows the speedup of kernels with simplified expressions compared to the versions without simplification. Thus, a higher bar indicates the importance of simplification for this particular example. To get representa-

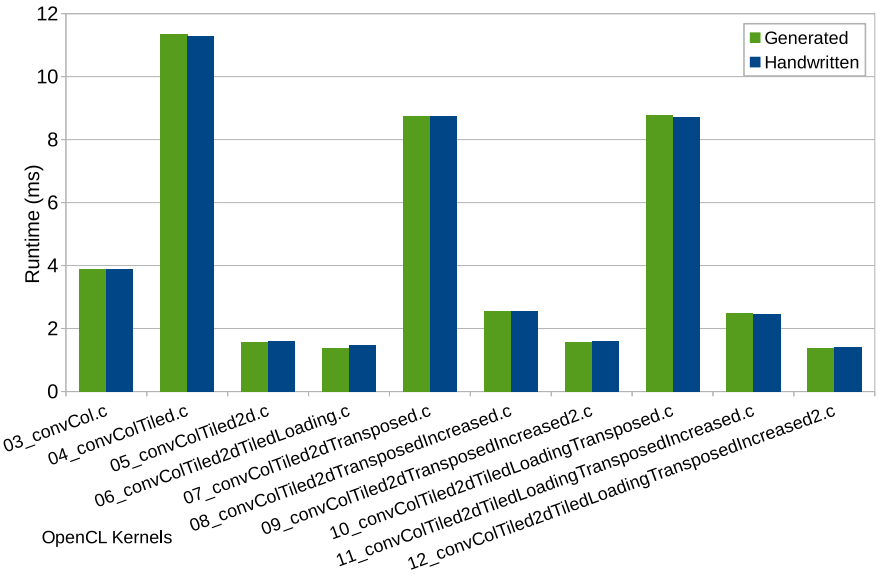


Figure 34: Comparing performance of the generated column convolution to the handwritten convolution

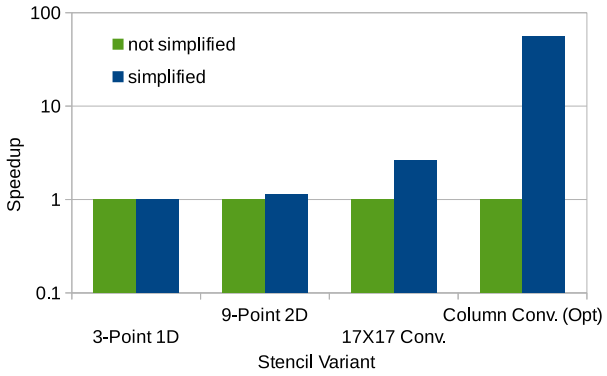


Figure 35: Speedup of kernels using simplified arithmetic expressions compared to kernels without arithmetic simplification

tive results, we examine four different stencil variants with increasing complexity. The first stencil we evaluate is a simple one dimensional 3-point stencil. In this case arithmetic expression simplification has no impact at all because the generated expressions are already as simple as they can be. The second stencil is a two dimensional 9-point stencil without applying any optimizations. However, this stencil has increasing complexity compared to the 3-point stencil because we use the functions *slide2d* and *pad2d* as introduced in the previous chapters. These functions apply the primitives *slide* and *pad* multiple times potentially leading to complex arithmetic expressions. However, as we observe in Figure 35, the impact of simplifying the expression is noticeable but not significant. The third stencil we evaluate is the 17×17 convolution applying overlapped tiling. Now that we applied some optimizations, the low-level expressions increase in complexity. This

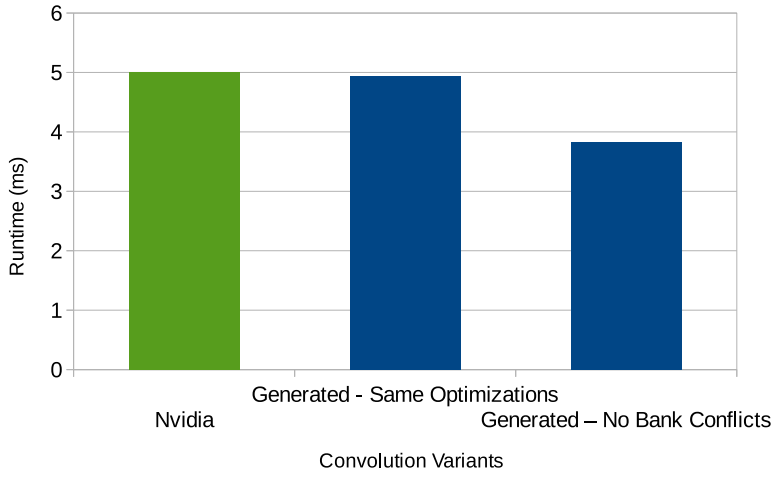


Figure 36: Performance of the generated 17×17 convolution and the ConvolutionSeparable Example (Nvidia Toolkit)

causes the generated arithmetic expressions to become complex as well because we use multiple *views* to flatten multidimensional array indices as described in Section 4.1. As a last example, we examine the fully optimized column convolution discussed in the previous chapter. Since this is the most optimized low-level expression we discussed in this thesis, the arithmetic expressions in the generated OpenCL are the most complex we have observed for stencil applications so far. Simplifying these expressions using basic rules of algebra achieves a speedup of 55.4 for this example. This emphasizes the importance of simplification during code generation.

To summarize, the more complex a particular low-level expression is, the more complex are the arithmetic expressions generated in the OpenCL Kernel code. Thus, simplifying them becomes non-negligible for complex computations expressed with low-level primitives.

5.5 PERFORMANCE COMPARED TO NVIDIA TOOLKIT EXAMPLE

Finally, we compare the performance of our generated kernels to the performance of the Nvidia Toolkit example *ConvolutionSeparable* described in [51]. This example implements a constant boundary handling. Hence, on out-of-bound accesses it returns a constant value. Since we currently do not support constant boundary handling, we rewrote that part of the CUDA example to implement a `clamp` boundary condition as defined in this thesis. We also confirmed that this does not change the performance of the original kernel by measuring a difference of 1% when executing the example with both boundary conditions. Figure 34 shows the performance differences of the generated and handwritten column convolution kernels. The first generated kernel applies the same optimizations as the CUDA kernel.

We executed both, the CUDA application and our generated convolution, using the exact same kernel launch configuration and tile sizes. As visualized, the performance of both kernels matches as expected. Thus, applying the same optimizations (using a functional expression) leads to the same performance. This shows, that we were able to successfully formalize the optimization described in [51] using our functional IL. Furthermore the compilation of such an expression using our code generator results in an imperative OpenCL kernel that has the same performance as the hand-tuned kernel by Nvidia. As described in the previous chapter, this example introduces bank-conflicts on modern GPUs like the Kepler K20c which we used to conduct our experiments. By resolving these bank-conflicts we are able to outperform Nvidia toolkit example achieving a speedup of 1.3. This emphasizes the flexibility of our approach to formalize optimizations using functional primitives. Resolving these bank conflict was just a matter of changing a numerical parameter for a single primitive instead of making changes in low-level imperative code.

6 CONCLUSION

In this thesis, we started by analyzing and decomposing the stencil pattern into its fundamental algorithmic parts. For two of these parts we introduced new high-level primitives *slide* and *pad*. Afterwards we provided multiple examples of how to use these primitives and the power of function composition to express several one- and multi-dimensional stencil computations. Therefore, we provided a powerful and flexible approach to express stencil computations by composing intuitive and simple functional primitives.

In the subsequent chapter we analyzed and formalized well-known optimizations for stencil computations. We began with lowering a simple high-level expression into a low-level expression written in LIFT's IL. We incrementally applied several optimizations like overlapped tiling or utilizing local memory using a set of formal rewrite rules. We proved that these rules are semantics preserving and applied them systematically to rewrite an expression into a more efficient one. Doing this, we introduced a new low-level primitive *increase* to avoid bank conflicts when using local memory, introduced the *mapSeqUnroll* primitive to unroll loops and reused the high-level primitive *slide* to apply overlapped tiling. We evaluated handwritten OpenCL kernels to confirm that the applied optimizations indeed improve performance. Comparing the naive version we started with, with the most optimized version, we are able to improve the performance by a factor of 41, emphasizing the necessity to optimize stencil computations on GPUs. Afterwards we described the LIFT's code generation process and briefly discussed every compilation stage. We introduced and discussed the extensions to LIFT's view system used to generate code for the newly introduced primitives.

Finally, we evaluated the performance of OpenCL kernels generated using the LIFT Framework. We incrementally rewrote expressions and generated kernels after applying every optimization. The performance of our generated kernels is competitive to the handwritten kernels introduced earlier. Some of these generated kernels show further potential for improvements in terms of simplifying array accesses. As a last step we compared the performance of our generated kernels to the performance of the Nvidia Toolkit *convolutionSeparable* example. When generating a kernel that applies the same optimization we achieve the same performance as the hand-tuned version from Nvidia. Since the example introduces bank-conflicts on modern GPUs we are able to outperform the Nvidia example by resolving all of these bank-conflicts.

To summarize, we achieve to generate high-performance OpenCL kernel for stencil computations using the LIFT Framework. We provide a flexible programming framework achieving the same level of abstraction as existing library approaches without relying on hard-coded specialized solutions for multidimensional stencils. Using the ideas of Backus, Bird, Cole and many more we are able to rewrite functional expressions into more efficient expressions that eventually are used to generate high-performance device-specific OpenCL kernels.

APPENDIX

A.1 OPENCL KERNELS IMPLEMENTING OPTIMIZATIONS FOR THE 17×17 CONVOLUTION

In the following we show all kernels used in Chapter 3 to obtain performance numbers for specific optimizations.

```
1  kernel void KERNEL(const global float *restrict IN,
2         const global float *restrict W, global float *OUT) {
3
4    int gid1 = get_global_id(1);
5    int gid0 = get_global_id(0);
6    float acc = 0.0f;
7
8    for (int i = 0; i < 289; i++) {
9      int x = gid0 - 8 + (i % 17);
10     int y = gid1 - 8 + (i / 17);
11     x = max(0, x);
12     x = min(x, 4095);
13     y = max(0, y);
14     y = min(y, 4095);
15     acc = acc + (IN[x + 4095 * y] * W[i]);
16   }
17
18   OUT[(gid0 + (4095 * gid1))] = acc;
19 }
```

Listing 24: OpenCL kernel implementing a naive 17×17 convolution

```
kernel void KERNEL(const global float *restrict IN,
       const global float *restrict W, global float *OUT) {

  local float L[1024];
  float acc;
  for (int wg1 = get_group_id(1); wg1 < 256; wg1 += 128) {
    for (int wg0 = get_group_id(0); wg0 < 256; wg0 += 128) {

      int lid1 = get_local_id(1);
      int lid0 = get_local_id(0);

      int x = wg0 * 32 + lid0 - 8;
      int y = wg1 * 32 + lid1 - 8;
      x = max(0, x);
      y = max(0, y);
      x = min(4095, x);
      y = min(4095, y);

      L[(lid0 + (32 * lid1))] = IN[y * 4096 + x];

      barrier(CLK_LOCAL_MEM_FENCE);

      if (get_local_id(1) < 16) {
        int lid1 = get_local_id(1);

        if (get_local_id(0) < 16) {
          int lid0 = get_local_id(0);

          acc = 0.0f;
          for (int i = 0; i < 289; i++) {
            int x = lid0 + (i % 17);
            int y = lid1 + (i / 17);

            acc = acc + (L[(x + 32 * y)] * W[i]);
          }

          int x = lid0 + (16 * wg0);
          int y = lid1 + (16 * wg1);
          OUT[x + 4096 * y] = acc;
        }
      }
      barrier(CLK_GLOBAL_MEM_FENCE);
    }
  }
}
```

Listing 25: OpenCL kernel implementing 17×17 convolution using local memory

```
kernel void KERNEL(const global float *restrict IN,
      const global float *restrict W, global float *OUT) {

  local float L[1024];
  float acc;
  int wg1 = get_group_id(1);
  int wg0 = get_group_id(0);

  for (int lid1 = get_local_id(1); lid1 < 32; lid1 = (16 + lid1)) {
    for (int lid0 = get_local_id(0); lid0 < 32; lid0 = (16 + lid0)) {

      int x = wg0 * 32 + lid0 - 8;
      int y = wg1 * 32 + lid1 - 8;
      x = max(0, x);
      y = max(0, y);
      x = min(4095, x);
      y = min(4095, y);

      L[(lid0 + (32 * lid1))] = IN[y * 4096 + x];
    }
  }

  barrier(CLK_LOCAL_MEM_FENCE);

  int lid1 = get_local_id(1);
  int lid0 = get_local_id(0);
  acc = 0.0f;

  for (int i = 0; i < 289; i++) {
    int x = lid0 + (i % 17);
    int y = lid1 + (i / 17);

    acc = acc + (L[(x + 32 * y)] * W[i]);
  }

  int x = lid0 + (16 * wg0);
  int y = lid1 + (16 * wg1);
  OUT[x + 4096 * y] = acc;
}
```

Listing 26: OpenCL kernel implementing 17 × 17 convolution avoiding idle threads

```
kernel void KERNEL(const global float *restrict IN,
       const global float *restrict W, global float *OUT) {
  float acc;
  int gid1 = get_global_id(1);
  int gid0 = get_global_id(0);
  acc = 0.0f;

  acc = acc + (IN[max(0, gid0 - 8) + 4096 * gid1] * W[0]);
  acc = acc + (IN[max(0, gid0 - 7) + 4096 * gid1] * W[1]);
  acc = acc + (IN[max(0, gid0 - 6) + 4096 * gid1] * W[2]);
  acc = acc + (IN[max(0, gid0 - 5) + 4096 * gid1] * W[3]);
  acc = acc + (IN[max(0, gid0 - 4) + 4096 * gid1] * W[4]);
  acc = acc + (IN[max(0, gid0 - 3) + 4096 * gid1] * W[5]);
  acc = acc + (IN[max(0, gid0 - 2) + 4096 * gid1] * W[6]);
  acc = acc + (IN[max(0, gid0 - 1) + 4096 * gid1] * W[7]);
  acc = acc + (IN[(gid0 + (4096 * gid1))] * W[8]);
  acc = acc + (IN[min(4095, gid0 + 1) + 4096 * gid1] * W[9]);
  acc = acc + (IN[min(4095, gid0 + 2) + 4096 * gid1] * W[10]);
  acc = acc + (IN[min(4095, gid0 + 3) + 4096 * gid1] * W[11]);
  acc = acc + (IN[min(4095, gid0 + 4) + 4096 * gid1] * W[12]);
  acc = acc + (IN[min(4095, gid0 + 5) + 4096 * gid1] * W[13]);
  acc = acc + (IN[min(4095, gid0 + 6) + 4096 * gid1] * W[14]);
  acc = acc + (IN[min(4095, gid0 + 7) + 4096 * gid1] * W[15]);
  acc = acc + (IN[min(4095, gid0 + 8) + 4096 * gid1] * W[16]);

  OUT[(gid0 + (4096 * gid1))] = acc;
}
```

Listing 27: OpenCL kernel implementing 17-point row convolution

```
kernel void KERNEL(const global float *restrict IN,
       const global float *restrict W, global float *OUT) {

 float a;
 int gid1 = get_global_id(1);
 int gid0 = get_global_id(0);
 a = 0.0f;

 a=a+(IN[(gid0 + (4096 * (((-8 + gid1) >= 0) ? (-8 + gid1) : 0 )))] * W[0]);
 a=a+(IN[(gid0 + (4096 * (((-7 + gid1) >= 0) ? (-7 + gid1) : 0 )))] * W[1]);
 a=a+(IN[(gid0 + (4096 * (((-6 + gid1) >= 0) ? (-6 + gid1) : 0 )))] * W[2]);
 a=a+(IN[(gid0 + (4096 * (((-5 + gid1) >= 0) ? (-5 + gid1) : 0 )))] * W[3]);
 a=a+(IN[(gid0 + (4096 * (((-4 + gid1) >= 0) ? (-4 + gid1) : 0 )))] * W[4]);
 a=a+(IN[(gid0 + (4096 * (((-3 + gid1) >= 0) ? (-3 + gid1) : 0 )))] * W[5]);
 a=a+(IN[(gid0 + (4096 * (((-2 + gid1) >= 0) ? (-2 + gid1) : 0 )))] * W[6]);
 a=a+(IN[(gid0 + (4096 * (((-1 + gid1) >= 0) ? (-1 + gid1) : 0 )))] * W[7]);
 a=a+(IN[(gid0 + (4096 * gid1))] * W[8]);
 a=a+(IN[(gid0 + (4096 * (((1 + gid1) < 4096) ? (1 + gid1) : 4095 )))] * W[9]);
 a=a+(IN[(gid0 + (4096 * (((2 + gid1) < 4096) ? (2 + gid1) : 4095 )))] * W[10]);
 a=a+(IN[(gid0 + (4096 * (((3 + gid1) < 4096) ? (3 + gid1) : 4095 )))] * W[11]);
 a=a+(IN[(gid0 + (4096 * (((4 + gid1) < 4096) ? (4 + gid1) : 4095 )))] * W[12]);
 a=a+(IN[(gid0 + (4096 * (((5 + gid1) < 4096) ? (5 + gid1) : 4095 )))] * W[13]);
 a=a+(IN[(gid0 + (4096 * (((6 + gid1) < 4096) ? (6 + gid1) : 4095 )))] * W[14]);
 a=a+(IN[(gid0 + (4096 * (((7 + gid1) < 4096) ? (7 + gid1) : 4095 )))] * W[15]);
 a=a+(IN[(gid0 + (4096 * (((8 + gid1) < 4096) ? (8 + gid1) : 4095 )))] * W[16]);

 OUT[(gid0 + (4096 * gid1))] = a;
}
```

Listing 28: OpenCL kernel implementing 17-point column convolution

```
kernel void KERNEL(const global float *restrict IN,
      const global float *restrict W, global float *OUT) {

  local float L[144];
  float acc;
  int wg1 = get_group_id(1);
  int wg0 = get_group_id(0);
  int lid1 = get_local_id(1);

  for (int lid0 = get_local_id(0); lid0 < 144; lid0 = (16 + lid0)) {

    int x = lid0 + (128 * wg0) - 8;
    x = max(0, x);
    x = min(4095, x);

    L[lid0] = IN[x + 4096 * wg1];
  }

  barrier(CLK_LOCAL_MEM_FENCE);

  for (int lid0 = get_local_id(0); lid0 < 128; lid0 = (16 + lid0)) {

    acc = 0.0f;
    acc = acc + (L[lid0] * W[0]);
    acc = acc + (L[1 + lid0] * W[1]);
    acc = acc + (L[2 + lid0] * W[2]);
    acc = acc + (L[3 + lid0] * W[3]);
    acc = acc + (L[4 + lid0] * W[4]);
    acc = acc + (L[5 + lid0] * W[5]);
    acc = acc + (L[6 + lid0] * W[6]);
    acc = acc + (L[7 + lid0] * W[7]);
    acc = acc + (L[8 + lid0] * W[8]);
    acc = acc + (L[9 + lid0] * W[9]);
    acc = acc + (L[10 + lid0] * W[10]);
    acc = acc + (L[11 + lid0] * W[11]);
    acc = acc + (L[12 + lid0] * W[12]);
    acc = acc + (L[13 + lid0] * W[13]);
    acc = acc + (L[14 + lid0] * W[14]);
    acc = acc + (L[15 + lid0] * W[15]);
    acc = acc + (L[16 + lid0] * W[16]);

    OUT[(lid0 + (128 * wg0) + (4096 * wg1))] = acc;
  }
}
```

Listing 29: OpenCL kernel implementing tiled 17-point row convolution

```
kernel void KERNEL(const global float *restrict IN,
       const global float *restrict W, global float *OUT) {

  local float L[80];
  float acc;
  int wg1 = get_group_id(1);
  int wg0 = get_group_id(0);

  for (int lid1 = get_local_id(1); lid1 < 80; lid1 = (8 + lid1)) {
    int lid0 = get_local_id(0);
    int y = (-8 + lid1 + (64 * wg1));
    y = max(0, y);
    y = min(4095, y);

    L[lid1] = IN[(wg0 + (4096 * y))];
  }

  barrier(CLK_LOCAL_MEM_FENCE);

  for (int lid1 = get_local_id(1); lid1 < 64; lid1 = (8 + lid1)) {

    int lid0 = get_local_id(0);
    acc = 0.0f;

    acc = acc + (L[lid1] * W[0]);
    acc = acc + (L[1 + lid1] * W[1]);
    acc = acc + (L[2 + lid1] * W[2]);
    acc = acc + (L[3 + lid1] * W[3]);
    acc = acc + (L[4 + lid1] * W[4]);
    acc = acc + (L[5 + lid1] * W[5]);
    acc = acc + (L[6 + lid1] * W[6]);
    acc = acc + (L[7 + lid1] * W[7]);
    acc = acc + (L[8 + lid1] * W[8]);
    acc = acc + (L[9 + lid1] * W[9]);
    acc = acc + (L[10 + lid1] * W[10]);
    acc = acc + (L[11 + lid1] * W[11]);
    acc = acc + (L[12 + lid1] * W[12]);
    acc = acc + (L[13 + lid1] * W[13]);
    acc = acc + (L[14 + lid1] * W[14]);
    acc = acc + (L[15 + lid1] * W[15]);
    acc = acc + (L[16 + lid1] * W[16]);

    OUT[(wg0 + 4096 * (lid1 + 64 * wg1))] = acc;
  }
}
```

Listing 30: OpenCL kernel implementing tiled 17-point column convolution

```
kernel void KERNEL(const global float *restrict IN,
      const global float *restrict W, global float *OUT) {

  local float L[576];
  float acc;
  int wg1 = get_group_id(1);
  int wg0 = get_group_id(0);
  int lid1 = get_local_id(1);

  for (int lid0 = get_local_id(0); lid0 < 144; lid0 = (16 + lid0)) {

    int x = wg0 * 128 + lid0 - 8;
    int y = wg1 * 4 + lid1;

    x = max(0, x);
    x = min(4095, x);

    L[lid0 + (144 * lid1)] = IN[y * 4096 + x];
  }

  barrier(CLK_LOCAL_MEM_FENCE);

  for (int lid0 = get_local_id(0); lid0 < 128; lid0 = (16 + lid0)) {
    acc = 0.0f;

    acc = acc + (L[(lid0 + (144 * lid1))] * W[0]);
    acc = acc + (L[(1 + lid0 + (144 * lid1))] * W[1]);
    acc = acc + (L[(2 + lid0 + (144 * lid1))] * W[2]);
    acc = acc + (L[(3 + lid0 + (144 * lid1))] * W[3]);
    acc = acc + (L[(4 + lid0 + (144 * lid1))] * W[4]);
    acc = acc + (L[(5 + lid0 + (144 * lid1))] * W[5]);
    acc = acc + (L[(6 + lid0 + (144 * lid1))] * W[6]);
    acc = acc + (L[(7 + lid0 + (144 * lid1))] * W[7]);
    acc = acc + (L[(8 + lid0 + (144 * lid1))] * W[8]);
    acc = acc + (L[(9 + lid0 + (144 * lid1))] * W[9]);
    acc = acc + (L[(10 + lid0 + (144 * lid1))] * W[10]);
    acc = acc + (L[(11 + lid0 + (144 * lid1))] * W[11]);
    acc = acc + (L[(12 + lid0 + (144 * lid1))] * W[12]);
    acc = acc + (L[(13 + lid0 + (144 * lid1))] * W[13]);
    acc = acc + (L[(14 + lid0 + (144 * lid1))] * W[14]);
    acc = acc + (L[(15 + lid0 + (144 * lid1))] * W[15]);
    acc = acc + (L[(16 + lid0 + (144 * lid1))] * W[16]);

    int x = lid0 + 128 * wg0;
    int y = lid1 + 4 * wg1;
    OUT[x + 4096 * y] = acc;
  }
}
```

Listing 31: OpenCL kernel implementing improved tiled 17-point row convolution

```
kernel void KERNEL(const global float *restrict IN,
       const global float *restrict W, global float *OUT) {

  local float L[1280];
  float acc;
  int wg1 = get_group_id(1);
  int wg0 = get_group_id(0);
  for (int lid1 = get_local_id(1); lid1 < 80; lid1 = (8 + lid1)) {

    int lid0 = get_local_id(0);
    int x = wg0 * 16 + lid0;
    int y = wg1 * 64 + lid1 - 8;

    y = max(0, y);
    y = min(4095, y);

    L[(lid0 + (16 * lid1))] = IN[y * 4096 + x];
  }

  barrier(CLK_LOCAL_MEM_FENCE);

  for (int lid1 = get_local_id(1); lid1 < 64; lid1 = (8 + lid1)) {
    int lid0 = get_local_id(0);
    acc = 0.0f;

    acc = acc + (L[(lid0 + (16 * lid1))] *  W[0]);
    acc = acc + (L[(16 + lid0 + (16 * lid1))] * W[1]);
    acc = acc + (L[(32 + lid0 + (16 * lid1))] * W[2]);
    acc = acc + (L[(48 + lid0 + (16 * lid1))] * W[3]);
    acc = acc + (L[(64 + lid0 + (16 * lid1))] * W[4]);
    acc = acc + (L[(80 + lid0 + (16 * lid1))] * W[5]);
    acc = acc + (L[(96 + lid0 + (16 * lid1))] * W[6]);
    acc = acc + (L[(112 + lid0 + (16 * lid1))] * W[7]);
    acc = acc + (L[(128 + lid0 + (16 * lid1))] * W[8]);
    acc = acc + (L[(144 + lid0 + (16 * lid1))] * W[9]);
    acc = acc + (L[(160 + lid0 + (16 * lid1))] * W[10]);
    acc = acc + (L[(176 + lid0 + (16 * lid1))] * W[11]);
    acc = acc + (L[(192 + lid0 + (16 * lid1))] * W[12]);
    acc = acc + (L[(208 + lid0 + (16 * lid1))] * W[13]);
    acc = acc + (L[(224 + lid0 + (16 * lid1))] * W[14]);
    acc = acc + (L[(240 + lid0 + (16 * lid1))] * W[15]);
    acc = acc + (L[(256 + lid0 + (16 * lid1))] * W[16]);

    int x = lid0 + 16 * wg0;
    int y = lid1 + 64 * wg1;
    OUT[x + 4096 * y] = acc;
  }
}
```

Listing 32: OpenCL kernel implementing improved tiled 17-point column convolution

```
kernel void KERNEL(const global float *restrict IN,
       const global float *restrict W, global float *OUT) {

  local float L[1280];
  float acc;
  int wg1 = get_group_id(1);
  int wg0 = get_group_id(0);
  int lid0 = get_local_id(0);

  for (int lid1 = get_local_id(1); lid1 < 80; lid1 = (8 + lid1)) {
    int x = lid0 + (16 * wg0);
    int y = (-8 + lid1 + (64 * wg1));
    y = max(0, y);
    y = min(4095, y);

    L[lid1 + (80 * lid0)] = IN[x + (4096 * y)];
  }

  barrier(CLK_LOCAL_MEM_FENCE);

  for (int lid1 = get_local_id(1); lid1 < 64; lid1 = (8 + lid1)) {
    int lid0 = get_local_id(0);
    acc = 0.0f;

    // clang-format off
    acc = acc + (L[(lid1 + (80 * lid0))] *  W[0]);
    acc = acc + (L[(1 + lid1 + (80 * lid0))] * W[1]);
    acc = acc + (L[(2 + lid1 + (80 * lid0))] * W[2]);
    acc = acc + (L[(3 + lid1 + (80 * lid0))] * W[3]);
    acc = acc + (L[(4 + lid1 + (80 * lid0))] * W[4]);
    acc = acc + (L[(5 + lid1 + (80 * lid0))] * W[5]);
    acc = acc + (L[(6 + lid1 + (80 * lid0))] * W[6]);
    acc = acc + (L[(7 + lid1 + (80 * lid0))] * W[7]);
    acc = acc + (L[(8 + lid1 + (80 * lid0))] * W[8]);
    acc = acc + (L[(9 + lid1 + (80 * lid0))] * W[9]);
    acc = acc + (L[(10 + lid1 + (80 * lid0))] * W[10]);
    acc = acc + (L[(11 + lid1 + (80 * lid0))] * W[11]);
    acc = acc + (L[(12 + lid1 + (80 * lid0))] * W[12]);
    acc = acc + (L[(13 + lid1 + (80 * lid0))] * W[13]);
    acc = acc + (L[(14 + lid1 + (80 * lid0))] * W[14]);
    acc = acc + (L[(15 + lid1 + (80 * lid0))] * W[15]);
    acc = acc + (L[(16 + lid1 + (80 * lid0))] * W[16]);
    // clang-format on

    int x = lid0 + 16 * wg0;
    int y = lid1 + 64 * wg1;
    OUT[x + 4096 * y] = acc;
  }
}
```

Listing 33: OpenCL kernel implementing transposed tile 17-point column convolution

```
kernel void KERNEL(const global float *restrict IN,
       const global float *restrict W, global float *OUT) {

  local float L[1296];
  float acc;
  int wg1 = get_group_id(1);
  int wg0 = get_group_id(0);
  int lid0 = get_local_id(0);

  for (int lid1 = get_local_id(1); lid1 < 80; lid1 = (8 + lid1)) {
    int x = lid0 + (16 * wg0);
    int y = (-8 + lid1 + (64 * wg1));
    y = max(0, y);
    y = min(4095, y);

    L[lid1 + (81 * lid0)] = IN[x + (4096 * y)];
  }

  barrier(CLK_LOCAL_MEM_FENCE);

  for (int lid1 = get_local_id(1); lid1 < 64; lid1 = (8 + lid1)) {
    int lid0 = get_local_id(0);
    acc = 0.0f;

    acc = acc + (L[(lid1 + (81 * lid0))] *  W[0]);
    acc = acc + (L[(1 + lid1 + (81 * lid0))] * W[1]);
    acc = acc + (L[(2 + lid1 + (81 * lid0))] * W[2]);
    acc = acc + (L[(3 + lid1 + (81 * lid0))] * W[3]);
    acc = acc + (L[(4 + lid1 + (81 * lid0))] * W[4]);
    acc = acc + (L[(5 + lid1 + (81 * lid0))] * W[5]);
    acc = acc + (L[(6 + lid1 + (81 * lid0))] * W[6]);
    acc = acc + (L[(7 + lid1 + (81 * lid0))] * W[7]);
    acc = acc + (L[(8 + lid1 + (81 * lid0))] * W[8]);
    acc = acc + (L[(9 + lid1 + (81 * lid0))] * W[9]);
    acc = acc + (L[(10 + lid1 + (81 * lid0))] * W[10]);
    acc = acc + (L[(11 + lid1 + (81 * lid0))] * W[11]);
    acc = acc + (L[(12 + lid1 + (81 * lid0))] * W[12]);
    acc = acc + (L[(13 + lid1 + (81 * lid0))] * W[13]);
    acc = acc + (L[(14 + lid1 + (81 * lid0))] * W[14]);
    acc = acc + (L[(15 + lid1 + (81 * lid0))] * W[15]);
    acc = acc + (L[(16 + lid1 + (81 * lid0))] * W[16]);

    int x = lid0 + 16 * wg0;
    int y = lid1 + 64 * wg1;
    OUT[x + 4096 * y] = acc;
  }
}
```

Listing 34: OpenCL kernel implementing inreased transposed tile 17-point column convolution

```
kernel void KERNEL(const global float *restrict IN,
       const global float *restrict W, global float *OUT) {

  local float L[1312];
  float acc;
  int wg1 = get_group_id(1);
  int wg0 = get_group_id(0);
  int lid0 = get_local_id(0);

  for (int lid1 = get_local_id(1); lid1 < 80; lid1 = (8 + lid1)) {
    int x = lid0 + (16 * wg0);
    int y = (-8 + lid1 + (64 * wg1));
    y = max(0, y);
    y = min(4095, y);

    L[lid1 + (82 * lid0)] = IN[x + (4096 * y)];
  }

  barrier(CLK_LOCAL_MEM_FENCE);

  for (int lid1 = get_local_id(1); lid1 < 64; lid1 = (8 + lid1)) {
    int lid0 = get_local_id(0);
    acc = 0.0f;

    acc = acc + (L[(lid1 + (82 * lid0))] *  W[0]);
    acc = acc + (L[(1 + lid1 + (82 * lid0))] * W[1]);
    acc = acc + (L[(2 + lid1 + (82 * lid0))] * W[2]);
    acc = acc + (L[(3 + lid1 + (82 * lid0))] * W[3]);
    acc = acc + (L[(4 + lid1 + (82 * lid0))] * W[4]);
    acc = acc + (L[(5 + lid1 + (82 * lid0))] * W[5]);
    acc = acc + (L[(6 + lid1 + (82 * lid0))] * W[6]);
    acc = acc + (L[(7 + lid1 + (82 * lid0))] * W[7]);
    acc = acc + (L[(8 + lid1 + (82 * lid0))] * W[8]);
    acc = acc + (L[(9 + lid1 + (82 * lid0))] * W[9]);
    acc = acc + (L[(10 + lid1 + (82 * lid0))] * W[10]);
    acc = acc + (L[(11 + lid1 + (82 * lid0))] * W[11]);
    acc = acc + (L[(12 + lid1 + (82 * lid0))] * W[12]);
    acc = acc + (L[(13 + lid1 + (82 * lid0))] * W[13]);
    acc = acc + (L[(14 + lid1 + (82 * lid0))] * W[14]);
    acc = acc + (L[(15 + lid1 + (82 * lid0))] * W[15]);
    acc = acc + (L[(16 + lid1 + (82 * lid0))] * W[16]);

    int x = lid0 + 16 * wg0;
    int y = lid1 + 64 * wg1;
    OUT[x + 4096 * y] = acc;
  }
}
```

Listing 35: OpenCL kernel implementing wider increased transposed tile 17-point column convolution

```
kernel void KERNEL(const global float *restrict IN,
       const global float *restrict W, global float *OUT) {

  local float L[1312];
  float acc;
  int wg1 = get_group_id(1);
  int wg0 = get_group_id(0);
  int lid0 = get_local_id(0);
  int lid1 = get_local_id(1);
  int x = lid0 + (16 * wg0);
  int y = lid1 + (64 * wg1);

  int upperHalo = max(0, (y -8));
  int lowerHalo = min(4095, (y+64));

  L[     (lid1 + (82 * lid0))] = IN[(4096 * upperHalo) + x];
  L[(8 +  lid1 + (82 * lid0))] = IN[(4096 * y) + x];
  L[(16 + lid1 + (82 * lid0))] = IN[(4096 * (y + 8)) + x];
  L[(24 + lid1 + (82 * lid0))] = IN[(4096 * (y + 16)) + x];
  L[(32 + lid1 + (82 * lid0))] = IN[(4096 * (y + 24)) + x];
  L[(40 + lid1 + (82 * lid0))] = IN[(4096 * (y + 32)) + x];
  L[(48 + lid1 + (82 * lid0))] = IN[(4096 * (y + 40)) + x];
  L[(56 + lid1 + (82 * lid0))] = IN[(4096 * (y + 48)) + x];
  L[(64 + lid1 + (82 * lid0))] = IN[(4096 * (y + 56)) + x];
  L[(72 + lid1 + (82 * lid0))] = IN[(4096 * lowerHalo) + x];

  barrier(CLK_LOCAL_MEM_FENCE);

  for (int lid1 = get_local_id(1); lid1 < 64; lid1 = (8 + lid1)) {

    int lid0 = get_local_id(0);
    acc = 0.0f;

    acc = acc + (L[(lid1 + (82 * lid0))] * W[0]);
    acc = acc + (L[(1 + lid1 + (82 * lid0))] * W[1]);
    acc = acc + (L[(2 + lid1 + (82 * lid0))] * W[2]);
    acc = acc + (L[(3 + lid1 + (82 * lid0))] * W[3]);
    acc = acc + (L[(4 + lid1 + (82 * lid0))] * W[4]);
    acc = acc + (L[(5 + lid1 + (82 * lid0))] * W[5]);
    acc = acc + (L[(6 + lid1 + (82 * lid0))] * W[6]);
    acc = acc + (L[(7 + lid1 + (82 * lid0))] * W[7]);
    acc = acc + (L[(8 + lid1 + (82 * lid0))] * W[8]);
    acc = acc + (L[(9 + lid1 + (82 * lid0))] * W[9]);
    acc = acc + (L[(10 + lid1 + (82 * lid0))] * W[10]);
    acc = acc + (L[(11 + lid1 + (82 * lid0))] * W[11]);
    acc = acc + (L[(12 + lid1 + (82 * lid0))] * W[12]);
    acc = acc + (L[(13 + lid1 + (82 * lid0))] * W[13]);
    acc = acc + (L[(14 + lid1 + (82 * lid0))] * W[14]);
    acc = acc + (L[(15 + lid1 + (82 * lid0))] * W[15]);
    acc = acc + (L[(16 + lid1 + (82 * lid0))] * W[16]);

    int x = lid0 + 16 * wg0;
    int y = lid1 + 64 * wg1;
    OUT[x + 4096 * y] = acc;
  }
}
```

Listing 36: OpenCL kernel implementing wider increased transposed tile 17-point column convolution

A.2 CORRECTNESS PROOFS FOR REWRITE RULES

In this section, we give proofs for several rewrite rules used in this thesis.

PROOF A.2.1 (OVERLAPPED TILING RULE): *Let* $xs = [x_1, \ldots, x_m]$.

$$(\mathit{join} \circ \mathit{map}(\mathit{slide}\ n\ s) \circ \mathit{slide}\ u\ v)\ xs$$

(definition of slide)

$$= (\mathit{join} \circ \mathit{map}(\mathit{slide}\ n\ s))\ [T_1, \ldots, T_p] \quad \mathit{where}\ p = \frac{m-u+v}{v} \quad \mathit{and}\ T_i = [t_{(i,1)}, \ldots, t_{(i,u)}],\ t_{(i,j)} = x_{(i-1)v+j} \tag{16}$$

(definition of map)

$$= \mathit{join}\ [\mathit{slide}\ n\ s\ T_1, \ldots, \mathit{slide}\ n\ s\ T_p]$$

(definition of slide)

$$\begin{aligned} &\mathit{slide}\ n\ s\ T_i \\ &= \mathit{slide}\ n\ s\ [t_{(i,1)}, \ldots, t_{(i,u)}] \\ &= [W_{(i,1)}, \ldots, W_{(i,q)}] \end{aligned}$$

$$\mathit{where}\ q = \frac{u-n+s}{s} \tag{17}$$

$$\mathit{and}\ W_{(i,k)} = [w_{(i,k,1)}, \ldots, w_{(i,k,n)}], \tag{18}$$

$$w_{(i,k,j)} = t_{(i,(k-1)s+j)} \tag{19}$$

$$= \mathit{join}\ [[W_{(1,1)}, \ldots, W_{(1,q)}], \ldots, [W_{(p,1)}, \ldots, W_{(p,q)}]]$$

(definition of join)

$$= [W_{(1,1)}, W_{(1,2)}, \ldots, W_{(p,q)}]$$

(using Equation 18)

$$\begin{aligned} =&[[w_{(1,1,1)}, \ldots, w_{(1,1,n)}], \\ &[w_{(1,2,1)}, \ldots, w_{(1,2,n)}], \\ &\ldots, \\ &[w_{(p,q,1)}, \ldots, w_{(p,q,n)}]] \end{aligned}$$

(using Equation 19 and Equation 17)

$$
\begin{aligned}
=&[[t_{(1,1)},\ldots,t_{(1,n)}],\\
&\ [t_{(1,s+1)},\ldots,t_{(1,s+n)}],\\
&\ \ldots,\\
&\ [t_{(p,u-n)},\ldots,t_{(p,u)}]]
\end{aligned}
$$

(using Equation 16)

$$= [[x_1,\ldots,x_n],[x_{(s+1)},\ldots,x_{s+n}],\ldots,[x_{m-n},\ldots,x_m]]$$

(definition of slide)

$$= \mathit{slide}\ n\ s\ [x_1,\ldots,x_m]$$

□

PROOF A.2.2 (MAP-JOIN REORDER RULE):
Let $xs = [[x_1,\ldots,x_k],[x_{k+1},\ldots,x_{2k}],\ldots,[x_{n-k},\ldots,x_n]]$

$$(\mathit{join} \circ \mathit{map}(\mathit{map}\ f))\ xs$$

(definition of map)

$$
\begin{aligned}
=\ \mathit{join}\ [&\mathit{map}\ f\ [x_1,\ldots,x_k],\\
&\mathit{map}\ f\ [x_{k+1},\ldots,x_{2k}],\\
&\ldots,\\
&\mathit{map}\ f\ [x_{n-k},\ldots,x_n]]
\end{aligned}
$$

(definition of map)

$$
\begin{aligned}
=\ \mathit{join}\ [&[f\ x_1,\ldots,f\ x_k],\\
&[f\ x_{k+1},\ldots,f\ x_{2k}],\\
&\ldots,\\
&[f\ x_{n-k},\ldots,f\ x_n]]
\end{aligned}
$$

(definition of join)

$$= [f\ x_1, f\ x_2,\ldots,f\ x_n]$$

(definition of map)

$$= \mathit{map}\ f\ xs$$

□

A.3 SYSTEMATICAL REWRITING OF FUNCTIONAL EXPRESSIONS

In the following we transform the expression shown in Listing 5 to the expression shown in Listing 6 using rewrite rules:

EXAMPLE A.1 (LOADING TILES TO LOCAL MEMORY):

$$
\begin{aligned}
&\lambda\ b\ weights\ input\ .\\
&(mapWorkgroup_1(mapWorkgroup_0(\lambda\ tile\ .\\
&\quad(mapLocal_1(mapLocal_0(\lambda\ nbh\ .\\
&\qquad(reduceSeq\ (+)\ 0 \circ mapSeq\ (*) \circ\ zip)\ (join\ nbh)\ weights)) \circ\\
&\quad slide2d\ 17\ 1)\ tile)) \circ\\
&slide2d\ 32\ 16 \circ pad2d\ 8\ 8\ b)\ input
\end{aligned}
$$

(using the identity three times - Rewrite Rule 6)

$$
\begin{aligned}
&\lambda\ b\ weights\ input\ .\\
&(mapWorkgroup_1(mapWorkgroup_0(\lambda\ tile\ .\\
&\quad(mapLocal_1(mapLocal_0(\lambda\ nbh\ .\\
&\qquad(map\ id \circ\\
&\qquad\quad reduceSeq\ (+)\ 0 \circ mapSeq\ (*) \circ\ zip)\ (join\ nbh)\ weights)) \circ\\
&\qquad slide2d\ 17\ 1)\ tile)) \circ\\
&\quad map(map\ id) \circ\\
&slide2d\ 32\ 16 \circ pad2d\ 8\ 8\ b)\ input
\end{aligned}
$$

(using the low-level map rule three times - Rewrite Rule 1)

$$
\begin{aligned}
&\lambda\ b\ weights\ input\ .\\
&(mapWorkgroup_1(mapWorkgroup_0(\lambda\ tile\ .\\
&\quad(mapLocal_1(mapLocal_0(\lambda\ nbh\ .\\
&\qquad(mapSeq\ id \circ\\
&\qquad\quad reduceSeq\ (+)\ 0 \circ mapSeq\ (*) \circ\ zip)\ (join\ nbh)\ weights)) \circ\\
&\qquad slide2d\ 17\ 1)\ tile)) \circ\\
&\quad mapLocal_1(mapLocal_0\ id) \circ\\
&slide2d\ 32\ 16 \circ pad2d\ 8\ 8\ b)\ input
\end{aligned}
$$

(using the global and local memory rule - Rewrite Rule 7)

$$
\begin{aligned}
&\lambda\ b\ weights\ input\ .\\
&(mapWorkgroup_1(mapWorkgroup_0(\lambda\ tile\ .\\
&\quad(mapLocal_1(mapLocal_0(\lambda\ nbh\ .\\
&\qquad(toGlobal(mapSeq\ id) \circ\\
&\qquad\quad reduceSeq\ (+)\ 0 \circ mapSeq\ (*) \circ\ zip)\ (join\ nbh)\ weights)) \circ\\
&\qquad slide2d\ 17\ 1)\ tile)) \circ\\
&\quad toLocal(mapLocal_1(mapLocal_0\ id)) \circ\\
&slide2d\ 32\ 16 \circ pad2d\ 8\ 8\ b)\ input
\end{aligned}
$$

BIBLIOGRAPHY

[1] 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*. Tech. rep. University of California, Berkeley, 2006.

[2] Olivier Aumage, Denis Barthou, and Alexandre Honorat. "A Stencil DSEL for Single Code Accelerated Computing with SYCL." In: *SYCL 2016 1st SYCL Programming Workshop during the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2016.

[3] John Backus. "Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs." In: *Communications of the ACM* 21.8 (1978), pp. 613–641.

[4] Peter Bastian, Markus Blatt, Christian Engwer, Andreas Dedner, Robert Klöfkorn, S Kuttanikkad, Mario Ohlberger, and Oliver Sander. "The distributed and unified numerics environment (DUNE)." In: *Proc. of the 19th Symposium on Simulation Technique in Hannover*. 2006.

[5] Anne Benoit, Murray Cole, Stephen Gilmore, and Jane Hillston. "Flexible skeletal programming with eSkel." In: *European Conference on Parallel Processing*. Springer. 2005, pp. 761–770.

[6] Mauro Bianco and Ugo Varetto. "A generic library for stencil computations." In: *arXiv preprint arXiv:1207.1746* (2012).

[7] Richard S Bird. "An introduction to the theory of lists." In: *Logic of programming and calculi of discrete design*. 1987, pp. 5–42.

[8] Richard S Bird. "Lectures on constructive functional programming." In: *Constructive Methods in Computing Science*. Springer, 1989, pp. 151–217.

[9] Rainer Bleck, Claes Rooth, Dingming Hu, and Linda T Smith. "Salinity-driven thermocline transients in a wind-and thermohaline-forced isopycnic coordinate model of the North Atlantic." In: *Journal of Physical Oceanography* 22.12 (1992), pp. 1486–1505.

[10] Tobias Brandvik and Graham Pullan. "SBLOCK: A framework for efficient stencil-based PDE solvers on multi-core platforms." In: *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*. IEEE. 2010, pp. 1181–1188.

[11] Stefan Breuer, Michel Steuwer, and Sergei Gorlatch. "Extending the SkelCL Skeleton Library for Stencil Computations on Multi-GPU Systems." In: *Proceedings of the 1st International Workshop on High-Performance Stencil Computations*. 2014, pp. 15–21.

[12] John Canny. "A computational approach to edge detection." In: *IEEE Transactions on pattern analysis and machine intelligence* 6 (1986), pp. 679–698.

[13] Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. "Accelerating Haskell array codes with multicore GPUs." In: *Proceedings of the sixth workshop on Declarative aspects of multicore programming*. ACM. 2011, pp. 3–14.

[14] Milosz Ciznicki, Michal Kulczewski, Piotr Kopta, and Krzysztof Kurowski. "Scaling the GCR Solver Using a High-Level Stencil Framework on Multi-and Many-Core Architectures." In: *Parallel Processing and Applied Mathematics*. Springer, 2016, pp. 594–606.

[15] Murray I Cole. "Algorithmic skeletons: A structured approach to the management of parallel computation." PhD thesis. University of Edinburgh, 1988.

[16] Murray Cole. "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming." In: *Parallel computing* 30.3 (2004), pp. 389–406.

[17] Usman Dastgeer and Christoph Kessler. "A performance-portable generic component for 2D convolution computations on GPU-based systems." In: *Proc. MULTIPROG-2012 Workshop at HiPEAC-2012, Paris*. 2012, pp. 1–12.

[18] Edsger W Dijkstra. "The humble programmer." In: *Communications of the ACM* 15.10 (1972), pp. 859–866.

[19] Fabian Dütsch, Karim Djelassi, Michael Haidl, and Sergei Gorlatch. "HLSF: A High-Level; C++-Based Framework for Stencil Computations on Accelerators." In: *Proceedings of the Second Workshop on Optimizing Stencil Computations*. ACM. 2014, pp. 41–4.

[20] Johan Enmyren and Christoph W. Kessler. "SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems." In: *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*. HLPP '10. ACM, 2010, pp. 5–14.

[21] Thomas L Falch and Anne C Elster. "ImageCL: An Image Processing Language for Performance Portability on Heterogeneous Systems." In: *arXiv preprint arXiv:1605.06399* (2016).

[22] Matteo Frigo and Volker Strumpen. "Cache oblivious stencil computations." In: *Proceedings of the 19th annual international conference on Supercomputing*. ACM. 2005, pp. 361–366.

[23] Joseph D Garvey. "Automatic Performance Tuning of Stencil Computations on Graphics Processing Units." PhD thesis. University of Toronto, 2015.

[24] Sergei Gorlatch. "Send-Recv considered harmful? Myths and truths about parallel programming." In: *International Conference on Parallel Computing Technologies*. Springer. 2001, pp. 243–257.

[25] Tobias Grosser, Albert Cohen, Paul HJ Kelly, J Ramanujam, P Sadayappan, and Sven Verdoolaege. "Split tiling for GPUs: automatic parallelization using trapezoidal tiles." In: *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. ACM. 2013, pp. 24–31.

[26] Tobias Grosser, Sven Verdoolaege, Albert Cohen, and P Sadayappan. "The relation between diamond tiling and hexagonal tiling." In: *Parallel Processing Letters* 24.03 (2014).

[27] Jia Guo, Ganesh Bikshandi, Basilio B Fraguela, and David Padua. "Writing productive stencil codes with overlapped tiling." In: *Concurrency and Computation: Practice and Experience* 21.1 (2009), pp. 25–39.

[28] Adam Harries, Michel Steuwer, Murray Cole, Alan Gray, and Christophe Dubach. "Compositional Compilation for Sparse, Irregular Data Parallelism." In: *Workshop on High-Level Programming for Heterogeneous and Hierarchical Parallel Systems (HLPGPU) 2016 @ HiPEAC, Prague, Czech Republic, January 19, 2016*. Jan. 2016.

[29] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. "A stencil compiler for short-vector SIMD architectures." In: *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM. 2013, pp. 13–24.

[30] Shoaib Kamil. "A generalized framework for auto-tuning stencil computations." In: *Lawrence Berkeley National Laboratory* (2009).

[31] Shoaib Kamil, Cy Chan, Leonid Oliker, John Shalf, and Samuel Williams. "An auto-tuning framework for parallel multicore stencil computations." In: *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE. 2010, pp. 1–12.

[32] Shoaib Kamil, Derrick Coetzee, Scott Beamer, Henry Cook, Ekaterina Gonina, Jonathan Harper, Jeffrey Morlan, and Armando Fox. "Portable parallel performance from sequential, productive, embedded domain-specific languages." In: *ACM SIGPLAN Notices*. Vol. 47. 8. ACM. 2012, pp. 303–304.

[33] John F Karpovich, Matthew Judd, W Timothy Strayer, and Andrew S Grimshaw. "A parallel object-oriented framework for stencil algorithms." In: *High Performance Distributed Computing, 1993., Proceedings the 2nd International Symposium on*. IEEE. 1993, pp. 34–41.

[34] John F Karpovich, Matthew Judd, W Timothy Strayer, and Andrew S Grimshaw. "A parallel object-oriented framework for stencil algorithms." In: *High Performance Distributed Computing, 1993., Proceedings the 2nd International Symposium on*. IEEE. 1993, pp. 34–41.

[35] DaeGon Kim, Lakshminarayanan Renganarayanan, Dave Rostron, Sanjay Rajopadhye, and Michelle Mills Strout. "Multi-level tiling: M for the price of one." In: *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. ACM. 2007, p. 51.

[36] Herbert Kuchen. "A skeleton library." In: *European Conference on Parallel Processing*. Springer. 2002, pp. 620–629.

[37] Michael Lesniak. "PASTHA: parallelizing stencil calculations in Haskell." In: *Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming*. ACM. 2010, pp. 5–14.

[38] Ben Lippmeier and Gabriele Keller. "Efficient parallel stencil convolution in Haskell." In: *ACM SIGPLAN Notices* 46.12 (2012), pp. 59–70.

[39] Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña-Marí. "Parallel functional programming in Eden." In: *Journal of Functional Programming* 15.03 (2005), pp. 431–475.

[40] Tareq Malas, Georg Hager, Hatem Ltaief, and David Keyes. "Multi-dimensional intra-tile parallelization for memory-starved stencil computations." In: *arXiv preprint arXiv:1510.04995* (2015).

[41] Azamat Mametjanov, Daniel Lowell, Ching-Chen Ma, and Boyana Norris. "Autotuning stencil-based computations on GPUs." In: *2012 IEEE international conference on cluster computing*. IEEE. 2012, pp. 266–274.

[42] Naoya Maruyama and Takayuki Aoki. "Optimizing stencil computations for NVIDIA Kepler GPUs." In: *Proceedings of the 1st International Workshop on High-Performance Stencil Computations, Vienna*. 2014, pp. 89–95.

[43] Naoya Maruyama, Kento Sato, Tatsuo Nomura, and Satoshi Matsuoka. "Physis: an implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers." In: *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE. 2011, pp. 1–12.

[44] Trevor L McDonell, Manuel MT Chakravarty, Gabriele Keller, and Ben Lippmeier. "Optimising purely functional GPU programs." In: *ACM SIGPLAN Notices* 48.9 (2013), pp. 49–60.

[45] AC McKellar and Edward G Coffman Jr. "Organizing matrices and matrix operations for paged memory systems." In: *Communications of the ACM* 12.3 (1969), pp. 153–165.

[46] Richard Membarth, Frank Hannig, Jürgen Teich, and Harald Köstler. "Towards domain-specific computing for stencil codes in HPC." In: *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:* IEEE. 2012, pp. 1133–1138.

[47] Cole Murray. *Algorithmic skeletons: structured management of parallel computation*. 1989.

[48] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. "3.5-D blocking optimization for stencil computations on modern CPUs and GPUs." In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society. 2010, pp. 1–13.

[49] Dominic Orchard and Alan Mycroft. "Efficient and Correct Stencil Computation via Pattern Matching and Static Typing." In: *arXiv preprint arXiv:1109.0777* (2011).

[50] Alyson D Pereira, Luiz Ramos, and Luís FW Góes. "PSkel: A stencil programming framework for CPU-GPU systems." In: *Concurrency and Computation: Practice and Experience* 27.17 (2015), pp. 4938–4953.

[51] Victor Podlozhnyuk. "Image convolution with CUDA." In: *NVIDIA Corporation white paper, June* 2097.3 (2007).

[52] 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), pp. 519–530.

[53] 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*. ACM. 2016, pp. 22–31.

[54] Lakshminarayanan Renganarayana, Manjukumar Harthikote-Matha, Rinku Dewri, and Sanjay Rajopadhye. "Towards optimal multi-level tiling for stencil computations." In: *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE. 2007, pp. 1–10.

[55] Takashi Shimokawabe, Takayuki Aoki, and Naoyuki Onodera. "High-productivity Framework for Large-scale GPU/CPU Stencil Applications." In: *Procedia Computer Science* 80 (2016), pp. 1646–1657.

[56] Irwin Sobel and Gary Feldman. "A 3x3 isotropic gradient operator for image processing." In: *a talk at the Stanford Artificial Project in* (1968), pp. 271–272.

[57] Michel Steuwer. "Improving Programmability and Performance Portability on Many-Core Processors." PhD thesis. 2015.

[58] Michel Steuwer, Christian Fensch, and Christophe Dubach. "Patterns and Rewrite Rules for Systematic Code Generation (From High-Level Functional Patterns to High-Performance OpenCL Code)." In: *arXiv preprint arXiv:1502.02389* (2015).

[59] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. "Skelcl-a portable skeleton library for high-level gpu programming." In: *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. IEEE. 2011, pp. 1176–1182.

[60] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. "Generating Performance Portable Code using Rewrite Rules." In: ICFP. 2015.

[61] Michelle Mills Strout. "Compilers for Regular and Irregular Stencils: Some Shared Problems and Solutions." In: *Proceedings of Workshop on Optimizing Stencil Computations (WOSC)*. 2013.

[62] Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and Hans-Peter Seidel. "Cache accurate time skewing in iterative stencil computations." In: *2011 International Conference on Parallel Processing*. IEEE. 2011, pp. 571–581.

[63] Markus Stuermer and U Ruede. "A framework that supports in writing performance-optimized stencil-based codes." In: *Universität Erlangen-Nürnberg, Tech. Rep* (2010), pp. 10–5.

[64] Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. "Delite: A compiler architecture for performance-oriented embedded domain-specific languages." In: *ACM Transactions on Embedded Computing Systems (TECS)* 13.4s (2014), p. 134.

[65] Yuan Tang, Rezaul Alam Chowdhury, Bradley C Kuszmaul, Chi-Keung Luk, and Charles E Leiserson. "The pochoir stencil compiler." In: *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*. ACM. 2011, pp. 117–128.

[66] Abhishek Udupa, R Govindarajan, and Matthew J Thazhuthaveetil. "Software pipelined execution of stream programs on GPUs." In: *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*. IEEE. 2009, pp. 200–209.

[67] Didem Unat, Xing Cai, and Scott B Baden. "Mint: realizing CUDA performance in 3D stencil methods with annotated C." In: *Proceedings of the international conference on Supercomputing*. ACM. 2011, pp. 214–224.

[68] Gerhard Wellein, Georg Hager, Thomas Zeiser, Markus Wittmann, and Holger Fehske. "Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization." In: *2009 33rd Annual IEEE International Computer Software and Applications Conference*. Vol. 1. IEEE. 2009, pp. 579–586.

[69] Rinse Wester and Jan Kuper. "Deriving stencil hardware accelerators from a single higher-order function." In: *Communicating Processes Architectures 2014*. Ed. by P.H. Welch. Open Channel publishing, 2014, pp. 205–218.

[70] Markus Wittmann, Georg Hager, and Gerhard Wellein. "Multicore-aware parallel temporal blocking of stencil codes for shared and distributed memory." In: *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*. IEEE. 2010, pp. 1–7.

[71] Michael E Wolf and Monica S Lam. "A data locality optimizing algorithm." In: *ACM Sigplan Notices*. Vol. 26. 6. ACM. 1991, pp. 30–44.

[72] Laurence T Yang and Minyi Guo. *High-performance computing: paradigm and infrastructure*. Vol. 44. John Wiley & Sons, 2005.

[73] Xing Zhou. "Tiling optimizations for stencil computations." PhD thesis. University of Illinois at Urbana-Champaign, 2013.

DECLARATION

I hereby confirm that this thesis on *An Extension of a Functional Intermediate Language for Parallelizing Stencil Computations and its Optimizing GPU Implementation using OpenCL* is solely my own work and that I have used no sources or aids other than the ones stated. All passages in my thesis for which other sources, including electronic media, have been used, be it direct quotes or content references, have been acknowledged as such and the sources cited.

I agree to have my thesis checked in order to rule out potential similarities with other works and to have my thesis stored in a database for this purpose.

Ibbenbüren, September 2016

Bastian Hagedorn

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

Final Version as of March 21, 2017 (`classicthesis` version 4.2).