A Functional Approach for the Automatic Parallelization of Non-Associative Reductions via Semiring Algebra

Federico Pizzuti

Master of Science by Research

Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh

2017

Abstract

Recent developments in computer architecture have yielded to a surge in the number of parallel devices, such as general purpose GPUs and application specific accelerators, making parallel programming more common than ever before.However, parallel programming poses many challenges to the programmer, as implementing correct solution requires expert knowledge of the target platform, in addition to advanced programming skills.

A popular solution to this problem is the adoption of a pattern-based programming model, wherein the code is written using high-level transformations. While automatic parallelisation for some of these transformations is easily performed, this is not the case for others. In particular, the automatic parallelisation of Reduction is commonly supported only for a restricted class of well known associative operators: outside of these cases, the programmer must either resort to handwritten ad-hoc implementations or settle for using a sequential implementation.

This document proposes a method for the automatic parallelisation of Reductions which, by taking advantage of properties of purely functional programs and the algebraic notion of a semiring, is capable of handling cases in which the reduction operator is not necessarily associative, but instead satisfies a much weaker set of properties. It then presents a series of additions to the Lift compiler implementing said methods, and finally proceeds to evaluate the performance of optimised Lift programs, demonstrating that the automatically parallelised code can outperform equivalent handwritten implementations, without requiring any explicit intervention from the programmer.

Acknowledgements

This work was supported in part by the EPSRC Centre for Doctoral Training in Pervasive Parallelism, funded by the UK Engineering and Physical Sciences Research Council and the University of Edinburgh.

I'd like to thank my supervisor, Dr Cristophe Dubach, for the support offered during throughout the course of this project and the writing of this document. I'd like to thanks Naums Mogers and Rodrigo Rocha for their help in finding suitable editing tools and judgements on this document's stylistic choices.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

1.1.1 Heterogeneous programming

In recent times, a shift in computer architecture has been recorded, with interest steadily moving away from the improvement of faster single-CPU machines towards the development of scalable, massively parallel processing units. From mass-produced devices backed by technological giants such as General Purpose GPUs, many-cores processing units such as Intel's Xeon Phi and Google's Tensor Processing Unit, to avant-guard application specific accelerators such as the machine-learning oriented PuDianNao(Liu et al., 2015), the market for heterogeneous platforms could not be more varied.

By comparison, programming languages and compilers have been lagging behind. Generating parallel programs targeting a potentially heterogeneous system has still very high barriers to entry: firstly, due to the inherent difficulties in writing parallel code by hand, and secondly due to technological friction introduced by having to code using different technologies in order to target the plethora of available system.

A commonly proposed and well established solution for reducing the difficulty of parallel programming is to encourage the programmer to think of their software in terms of a sequence of well-behaved parallel patterns, the most common of which as Map and Reduce (Cole, 1989). But using this approach in isolation is not sufficient for resolving the question of automatic parallelisation: while deriving a parallel implementation for Map is a trivial exercise, Reduction offer a much more challenging example, in which

the validity to derive a parallel implementation is subject to the satisfaction several, non-trivial conditions.

1.1.2 Reductions

A reduction is defined as the pairwise combination of elements belonging to some collection, resulting in a single element as a result. Such reductions make an appearance in all sorts of programming languages in various forms, ranging from the multiply-and-accumulate for loops of C to the higher-order "foldl/foldr" functions of Haskell. Many algorithms include a reduction as a component: computing the total sum of a series of numbers, finding the maximum or minimum value in an array, or arbitrary aggregation of complex entries in a big data context - such as in the very famous aptly named MapReduce framework(Dean and Ghemawat, 2008). All these algorithms are, in their essence, reductions.

As previously stated, one of the most interesting properties of reductions is that a parallel implementation is possible, in case the reduction operator needs to be an associative function. This observation has long been known, and strategies for efficient parallel implementations are widely available. The conditions seem therefore perfect for reductions to be the ideal application for automatic parallelisation techniques: they are a simple abstraction, they have widespread adoption and have a relatively straightforward efficient implementation. One could easily expect compilers to be able to mechanically handle most reductions, liberating the programmer from the burden of handwriting parallel implementations.

1.1.3 Automatic parallelisation of reductions

The reality of the situation is however much more disappointing that one might presuppose, as the problem of identifying and optimizing reductions is still far from being solved. The chief issue to be overcome - and the reason why automatic parallelisation of reductions is not common - lies in the process of identification and separation of the reduction from the provided source code.

The problem can be conceptualised as follows. Most automatic parallelisers ultimately works by identifying within a program a fragment of sequential code for which a semantically-equivalent but parallel implementation is possible. In some sense, we

can think of auto-parallelisation as divided into two steps: first, an "understanding" step, in which the compiler captures the semantics of a particular fragment of code, and then a "generating" step, in which parallel code is generated implementing the identified semantic.

We propose that, according to this understanding, the difficulties in automatic parallelisation lie in the "understanding" step, and that there are two main phenomena at the root of the problem.

- Source code might not include all the necessary information for the compiler to identify potential for parallelism even if present. For example, a parallel reduction might not be identified as such because of the compiler's inability to infer that the operator is an associative function - a fact that it is not known to be derivable directly from the source code.

- Source code might be overly specific, with implementation details obscuring the algorithmic nature of the program. This is often the case of source written in imperative programming languages, which describe programs as series of state transformations and side-effecting commands. Deriving an understanding for such source code is a very complicated task, as the same language features can be used to express both algorithmic features (which express the fundamental functionality of a program, and must be present independently of whether the implementation is sequential or parallel) and purely implementation features (which are tied to the sequential nature of the handwritten version). The compiler must be able to filter away the former of the latter.

The issues highlighted above are one of the reasons why most automatic parallelisers must limit themselves to some form of dependence analysis. However, this approach can now be considered sufficiently explored: current state of the art dependence analysis is almost optimal(Murphy et al., 2015), and yet automatic parallelisation is not in widespread use.

1.2 The proposed solution

The previous section has detailed the need for automatic parallelisation in an increasingly heterogeneous world, while at the same time underlying the fundamental shortcomings that prevent widespread successful usage of the available techniques. In par-

ticular, it has highlighted the issues of recovering domain-specific information from the source code and of the "noisiness" of imperative source code to be the chief roadblocks.

In this document, we present a different approach to automatic parallelisation by showing how these challenges can be tackled more easily by adopting a functional programming style.
In particular, we propose an algorithm capable of automatic identification and generation of parallel reductions from an high-level sequential description. The algorithm is applicable in even in some of the cases where the operator is not directly associative, by transforming the problem into an equivalent instance of a general form reduction, using the associative matrix multiplication operator.

In more detail, we use the algebraic properties of semirings as theorized in abstract algebra to enrich the compiler's understanding of source code, allowing it to reason about the associativity of operators. At the same time the adoption of a high-level functional programming language allows to minimize the effect of implementation-specific "noisy" code, which greatly simplifies the application of the previously proposed static analysis.

1.3 Contributions

The work presented here includes the following original contributions

- An algorithm for the transformation of sequential reductions into a parallel equivalent implementation in a functional setting, starting from the multiply-accumulate transformation into matrix multiplications presented in(Sato and Iwasaki, 2011).

- An extension of the basic transformations, using properties of semiring algebra to implement "asymmetrically typed" reductions through the use of higher-dimensional matrices.

- An algorithm for the identification of reductions which can be parallelised using the method proposed above, based on pattern-matching syntactic constraints over the source code, eliminating the need for more complex analyses presented in (Sato and Iwasaki, 2011)

- The sketch of a proof of correctness via rewrite rules for the proposed transformation.

- A concrete implementation of the transformation for the Lift compiler, resulting in the generation of an efficient implementation targeting OpenCL-compliant GPUs.

- The necessary compiler infrastructure integrating the transformation within the wider context of the Lift programming language.

- The design of a future expression language for Lift, allowing for a more natural and direct expression of Reduction operators.

- Several handwritten implementations of the efficient parallel matrix multiplication reduction, for both the CPU (OpenMP, pthreads) and the GPU(OpenCL), to be used as a baseline and a comparison with the Lift optimised implementation.

1.4 Thesis outline

The rest of the thesis is divided into the following chapters

- Chapter 2 presents the required knowledge to understand the rest of the document. It contains a quick tour of the technologies used by the project: GPUs, OpenCL and Lift, and of the mathematical theories used through the document.

- Chapter 3 offers an overview of the field of parallel programming, with particular interest to both automatic parallelisation and to emerging programming models for heterogeneous architectures.

- Chapter 4 first explains the optimisation in a general functional programming setting, explaining the simplest form possible. It then proceeds to detail possible generalisations to the basic form, and concludes with a proof correctness.

- Chapter 5 focuses on the concrete implementation of the work of Chapter 4 into the Lift compiler, discussing the required software engineering decisions and the necessary extensions to the current compiler infrastructure required to support the transformation.

- Chapter 6 presents a series of handwritten implementations for the optimised parallel matrix multiplication reduction, to be used both as comparison for the Lift implementation and as an informative tool for the exploration of different implementation strategies.
- Chapter 7 details the results of a series of experiments performed to evaluate the quality of the compiler extensions provided, with the associated discussion and analysis
- Chapter 8 concludes the document, summarising the work and the results shown. It also includes a critical analysis of the project, and a section including possible future work.

1.5 Summary

This chapter has motivated the need of automatic parallelisation in the ever more heterogeneous and parallel world of computing. It has highlighted the importance of the reduction algorithmic skeleton, and detailed the limitations of current approaches towards its transparent parallelisation. Finally, it has proposed a method for addressing the shortcomings through the usage of an analysis based on properties of semiring algebra, to be applied in a functional programming context.

Chapter 2

Background

2.1 General Purpose Graphics Processing Units

Graphical Processing Units are among the most widespread accelerators in use, being found in a large proportion of personal computers and workstations, and spanning a vast range of prices and performance characteristics.

There are many differences in the performance characteristics of CPUs and GPUs. While the former are particularly good at executing single-instruction single-data (SISD) programs, which have a limited amount of parallelism but potentially much irregularity, the latter shine at the execution of single-instruction, multiple data (SIMD) programs, which expose a significant amount of parallelism but must be highly regular. Moreover, CPUs are fundamentally latency-oriented devices, while GPUs are throughput-oriented.

In light of these differences, it's easy to see how GPUs make for very good additions to CPU-centric systems, with both devices compensating each other's shortcomings. This is particularly important in recent times, with Moore's Law's slowdown pushing developers more and more towards the writing of parallel programs.

2.1.1 From GPUs to General Purpose GPUs

As the name suggests, Graphical Processing Units where originally introduced into computer systems in order to speed up graphical computations. This is because many computer graphics algorithms execute small, identical and independent computations

on each pixel being displayed. This made CPUs particularly inefficient for displaying graphics, and specialized hardware implementing a selected pipeline of computation was developed to lessen the burden. These devices were the first GPUs

The second step in this evolution was the progressive generalization and programmability of the various steps within the GPU execution pipeline. This phase saw the introduction of shaders, user customizable fragments of code, used to specify the details of the various pixel computations, without however having the ability to influence the structure of the computation itself.

While shaders were originally intended solely to perform graphical computations, their introduction spurred the birth of "compute shaders", that is non-graphics related algorithms implemented within the shader language to take advantage of GPUs massively parallel architecture. This eventually led to the creation of even more general programming languages and tools for GPUs, and as of today one can use frameworks such as OpenCL and NVidia's CUDA to program a GPU in C.

This alternative use has pushed the development of GPUs into a new area, with more and more architectures aiming to improve the performance of general purpose parallel computations, particularly in domain of high contemporary interest, such as Machine Learning. In light of this shift in focus, the term Graphical Processing Unit is more and more commonly replaced by the new "General Purpose Graphical Processing Units", to highlight the widened scope and programmability of the platform.

2.1.2 GPGPU architecture

Having covered the history and the context behind GPGPUs, it is time to turn towards a more concrete definition of their characteristics from the point of view of a programmer wishing to employ them successfully.

As previously stated, GPUs are a throughput oriented architecture, meaning that the performance metric to be maximized is total amount of operation over a set time. GPUs have much higher throughput than CPUs, and this is achieved by adopting a massively parallel architecture. Let's now look at the details of a common GPU architecture, as found in many of NVidia's Graphics Processing Units.

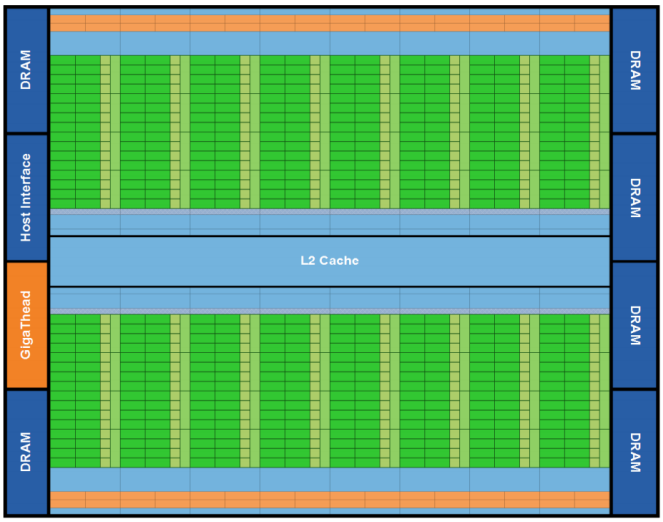

Figure 2.1: Overall structure of a GPU. From:(Glaskowsky, 2009)

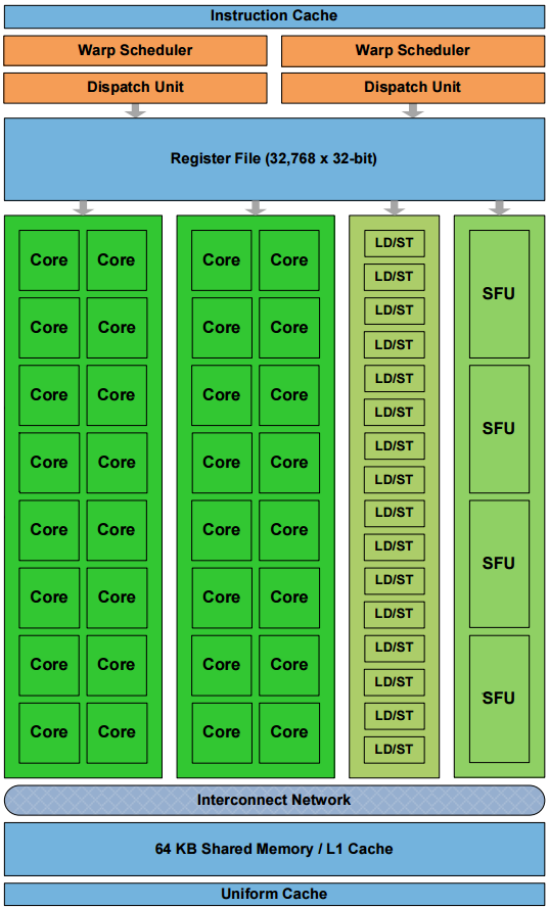

Figure 2.2: Internal structure of a Streaming Multiprocessor. From:(Glaskowsky, 2009)

At the top level, the GPU is composed of a few tens of small processors, usually called Streaming Multiprocessor(SM)(Glaskowsky, 2009). Each SM has many fea-

tures of a traditional CPU processor: an L1 data cache, an instruction cache, and a (very large) register file. Unlike a CPU processor however, an SM will be itself composed of many smaller processors, themselves known as Compute Cores (CC). Each CC fundamentally contains structures such as Floating Point computation units and Integer computation units.

A SM does not organize the computation of its CC directly. Instead, these are group together in structures known as Warps. CC's within a warp share a single program counter, and therefore will all execute the same instruction at the same time. The SM will then schedule the computation in terms of Warps.

As far as memory is concerned, GPUs tend to include on-chip blocks of fast RAM, partitioned among the different SM. This is meant to be used as fast local memory, and it's uses is preferred whenever possible over the distant system main memory - which is instead mostly used for input and output.

2.1.3 GPGPU programming model

The architecture presented above has several important consequences in the programming model. These are some of the considerations paramount to a good understanding of the functioning of GPUs:

- When thinking of parallel algorithms, it is usually convenient to think of all threads as existing in one unified space. This is not the case when programming GPUs: the process of scheduling logical threads onto Compute Cores implies that a division across Streaming Multiprocessors will be needed, and threads only shared a unified local address space within a single SM. Therefore, parallel programs optimized for a GPU might prefer structure their thread hierarchy in two steps: a "group", to be mapped onto the SM granularity, and then a "thread", to be mapped to a single CC. (This will become apparent when examining the "workgroup" and "workitem" concepts in OpenCL)

- The fact that threads grouped together execute in lockstep has potentially important performance implications. This means that divergent code (eg. the bodies of the two branches of after an if statement) is comparatively expensive to execute, as the two arms need to be executed serially, at each branch "stalling" those threads which have not taken the particular branch. This implies that instead of

if statements, one should try to prefer if-expressions, and try to remove as much code as possible from divergent branches.

- The memory access patterns of the program are to be taken into account when partitioning the threads across the various Streaming Multiprocessors groups. In particular, a program with many accesses to off-chip memory might benefit from having more, smaller groups - meaning that the GPU's will be able to consistently swap out Warps that are blocked performed memory accesses with other that are ready to execu. On the other hand, a program with fewer off-chip accesses might benefit more from fewer, larger groups, avoiding the performance penalties associated with context-switching between warps and, more importantly, potentially achieving better CC utilization.

2.2 OpenCL

2.2.1 Overview

Open Computing Language, often shortened as OpenCL, is a framework for the development of heterogeneous programs, supporting a variety of different platforms, including CPUs, GPUs and DSPs.

From the user's point of view, OpenCL consists of two main components: a guest device programming language and a host device library to organize and launch computations. The guest device language be considered a dialect of C, lacking the standard ANSI C library (notably without support for dynamic memory allocation), but enriched by language features key for heterogeneous programming, such as the ability to specify what kind of memory a given variable will be stored in, or function returning the id for a given thread.
The host library is a low-level C library used to identify a target device, compile a program for it and coordinate data transfers between host and guest.

2.2.2 Guest device programming

The guest programming language assumes a SIMD (single-program, multiple data) execution model, in which the same program is executed in parallel by all the threads, with the only possible source of divergence being the thread's coordinates (global_id

and local_id). This paradigm of execution is very conducive for writing programs targeting GPUs, which are capable of running a very large number of threads, while only having program control flow capabilities.

One of the fundamental ideas behind the SIMD execution model of OpenCL is the division of threads in groups, known as “workgroups”. Threads (also known as workitems) within a workgroup are able to synchronize their execution. This is the only form of synchronization supported as of OpenCL 1.2. AS such, no synchronization is possible across workgroups.

Another central feature of the OpenCL framework is the ability to control in which kind of memory a variable is stored. This is very important in an heterogeneous programming context, due to the fact that, unlike CPUs, alternative devices might have a very non-uniform memory hierarchy, with different kinds of memory types, both in terms of size and speed.
Since OpenCL is an abstraction over a large class of different device types, all the different kind of memory architectures can be expressed using only four different memory tags, with the following capabilities:

- Global - This tag is used for memory that is visible to all the threads in the system, independently from the workgroup they belong to. As this memory is allocated from the host memory pool, it tends to be both the most abundant and the slowest. It also has the function of communicating data between the host and guest, such as passing inputs and receiving the results of a guest program execution.

- Local - This tag is used for memory that is visible to all threads in a workgroup, but not across workgroups (that is to say, each workgroup is allocated one exclusive copy). It is much faster than Global memory, although much more scarce.

- Private - This tag is used for memory that is visible to a single thread, and as such is commonly used for local variables. This type of memory tends to have comparable speed with Local memory - although the exact performance characteristic depends on platform-specific implementation details (the standard only dictates the variable’s visibility).

- Constant - This tag can be only used for read-only variables, and has the same

visibility as Global memory. While the performance characteristics depend on the architecture of the guest device, on GPUs it tends to be a small but very fast. Writing to a constant memory address causes a compile-time error.

2.2.3 Host device programming

As previously stated, the OpenCL model structures an heterogeneous program according to a host-guest architecture. In this model, the host selects which device to run a given "kernel" (the name of a guest program) and then orchestrates input data transfers using platform library calls. Moreover, the host must also specify the number of workgroups/workitems (collectively known as worksizes) of the program, and also determine the size and types of buffers to be allocated for the kernel's execution.Only then the guest computation can start.

Although the OpenCL framework aims at providing a platform independent abstraction through its programming model, the specifics of the targeted platform often need to be taken into account if one wishes to implement highly efficient programs. One of the most complex decisions that need to be made is to determine the worksizes for a given program and dataset. Initially, common sense might be a useful guide, often informed by considerations like "a GPU prefers a large number of threads than a CPU" and "if the program needs to perform many accesses to global memory, then it should have many workgroups to mask the latency. However, this approach is unlikely to result in the individuation of the best possible parameter configuration.

There are several options to tackle this problem, such as the automated exploration (such as in Lift) or the use of autotuners. No such tool will be used in the work presented here, with manual exploration deemed sufficient, since the OpenCL parts of the project are there merely to serve as a litmus test for the usability of the proposed optimisation on GPUs, and the relative overall behaviour when compared with a CPU implementation. This also means that whatever performance numbers will be derived are very likely representing a conservative estimate of the maximum performance available to a real-world use case, where one would possibly enhance the methods proposed here by a more rigorous inference of worksizes.

2.3 Semirings, Matrices and Linear Functions

The optimisation presented in this project relies on algebraic properties of various mathematical structures, such as Semirings and Matrices. This section presents all the necessary concepts to understand the implementation mechanisms and its correctness.

2.3.1 Intuition behind Semirings

In mathematics, it is sometimes useful to abstract related entities and operations to derive general results. One such type of abstractions is algebraic structures: grouping of sets and operations over said sets that possess some particular properties. In some sense, these are the "generic programming" of mathematics.

One of these algebraic structures is of particular interest for this project: the semiring. Intuitively, a semiring is the abstraction of entities that behave similarly to numbers (this is valid for all but the Naturals without zero), when operated onto via addition and multiplication, with the relaxed condition that multiplication needs not be associative.
Examples of semirings include the numbers with + and * (as stated before) and the numbers with max and + (in this interesting example, + plays the roles of * in the previous semiring). Note that semirings are not restricted to numbers: sets make a semiring with union and intersection, and booleans with $\wedge$ and $\vee$.

The reason behind our interest in semirings lies in the fact that, if a property is proved considering an arbitrary semiring (and relying on none of its specific characteristics), then that property is valid for all the others Semirings as well. So since the algorithm for mechanic parallelisation presented in this document is defined solely in terms of a semiring, it will be automatically valid for all possible instances, even those that have yet to be found.

2.3.2 Formal definition

Formally speaking, a semiring in an algebraic data structure comprised of the following elements:

- S, a set

- $\oplus$, A commutative and associative binary operator over S
- $\otimes$, An associative binary operator over S, which distributes over $\oplus$
- 0, an item in S which is a left and right identity for $\oplus$, and is an annihilator for $\otimes$
- 1, an item in S which is a left and right identity for $\otimes$

2.3.3 Semirings and Matrices

The key property of semirings leveraged in the optimisation here proposed is that, if we can show that a set and some operations abide by the semiring laws, then it is possible to construct Matrices whose entries are elements of the set and for whom Matrix multiplication behaves in the commonly expected way.
To see why this is the case, let us first consider the conventionally accepted definition of a Matrix: a rectangular array of numbers. Then, we define Matrix multiplication of two matrices, one of size x-by-n and one of size n-by-y as:

$$A \times B = \sum_{n=1}^{N} A_{in} * B_{nj}$$

for all i and j.
Matrix multiplication has several important properties: for example, the presence of a known left and right identity, or the fact that matrix multiplication of square matrices is associative. These very properties le at the heart of the optimisation presented here, and we can use the definition of semiring above to allow the usage of Matrices and their multiplication outside the domain of the real numbers with + and *.

We can intuitively see why this is the case: the associative property of Matrix multiplication depends entirely on the associativity of its constituent operations: and that holds for $\oplus$ and $\otimes$ as well. Moreover, these operators must also have similar identity/annihilator behaviour with respect to their traditional counterparts, which means that Matrices over Semirings will also have a left/right identity.
It is good to mention for completeness that not all the properties of conventional matrices can be inherited using this abstraction: for example, matrix division is not a supported operation within this framework. Since these properties are not needed, however, this is of no real consequence.

2.4 Lift language and compiler

2.4.1 Language Overview

While the optimisation proposed in this document is language independent and should be applicable for functional languages in general, the specific implementation detailed here targets Lift, functional programming language focused on the expression of data-parallel computations.

The Lift programming language is particularly friendly to complex program analyses and transformation, in part due to the lack of side effects, which simplifies greatly the process of verifying whether a transformation is valid, and in part due to its tendency to structure the program as a composition of a small set of control primitives, such as Maps and Reduce. This in turns allows the compiler to work at a more coarse grained level and concentrate macroscopic structures within programs, without having to deal with the finer grain details of how these structures are implemented in practice.

2.4.2 High-level and low-level Lift

The primitives of the Lift language can be divided into roughly two groups: the high-level primitives and the low-level primitives. The first group is concerned solely with the algorithm description of a problem: this group include entries such as “Map” or “Transpose”. The second group instead contains platform and implementation specific primitives, such as “MapSeq” (the sequential map) and “MapGlb” (a parallel map using OpenCL global threads). Ideally, a programmer would write his software using the high-level set of primitives (resulting in a completely portable, implementation agnostic description of the computation). Then, when the code is generated for a given platform, the compiler first transforms the high-level set of rules into a suitable low-level expression, which would then specify the particulars of code generation.

The work proposed in this document mostly deals with the low-level kind of Lift primitives, as its goal is to transform a high-level Reduction into a particular parallel implementation for GPUs. What now follows is a quick overview of the primitives mentioned in the later chapters.

2.4.3 Notable primitives

What follows is a list of the Lift primitives used in the implementation

- MapSeq(f) - A sequential mapping function, given an array of inputs and a function f, generates and equally sized array of outputs, wherein each element e_n corresponds to $f(i_n)$. In OpenCL, it is implemented as a simple for loop.
- MapWrg(f) - Algorithmically identical to the above, with a parallel implementation, where input elements are assigned across OpenCL workgroups.
- MapLcl(f) - Algorithmically identical to the above, with a parallel implementation, where input elements are allocated across OpenCL workitems (threads). Only valid when nested within a MapWrg
- ReduceSeq(f, x) - A sequential reduction. Equivalent to a for loop combining input elements from an array into an accumulator, and then returning the accumulator as a result. It uses x as an initial accumulator.
- Split(N) - Splits an array into an array-of-arrays, with each element having length N
- Join() - Flattens an array of arrays.
- toLocal(f) - Applies f to its input, ensuring that the result is stored in a local memory buffer. Used solely for its performance implications.
- toGlobal(f) - Applies f to its input, ensuring that the result is stored in the GPU's global memory. The final result of any Lift program targeting OpenCL must be stored in global memory.

2.4.4 User Functions

So far we have only mentioned Lift's abilities concerning control primitives such as Map and Reduce, without mentioning how operations over scalar values are expressed. This is because Lift does not have a true expression language to speak of. Instead, Lift relies on programmer supplied "User Functions" for specifying such operations. These User Functions are written as snippets of C code, whose body is directly injected into the generated code at compile time, and are opaque to Lift. This fact will be analysed more in depth in Chapter 4, as the optimisation will have to generate some of these User Functions dynamically.

2.5 Summary

This Chapter has presented an overview of the required material for the full comprehension of the coming chapter. In particular, it has detailed the key technologies used (GPUs, OpenCL, the Lift programming language), as well as the theoretical knowledge pertaining to semirings that lay the foundation of the proposed optimisation algorithm.

Chapter 3

Related work

3.1 Overview

This chapter presents an overview of the state of the art in the field. It also aims to underline where shortcomings have been identified in these techniques, motivating the rest of the work presented in the document.

3.2 Automatic Parallelization

Automatic Parallelization refers to a host of techniques aimed at mechanically transforming a sequential program into an equivalent parallel implementation. Such techniques have been existing for many years, and usually employ some sort of static analysis to identify segments of the program which are independent, and hence parallelizable. The majority of these techniques have assumed an imperative context, tailoring their analysis around the properties of languages like C, or directly at lower level frameworks, such as LLVM.

3.2.1 Automatic Parallelization at the machine level: ICC

Possibly the simplest approach to parallelization, from a compiler's point of view, is to leverage low-level highly parallel features of the target hardware, offloading the problem of deriving an efficient implementation to the underlying metal, and therefore focusing exclusively on the identification of small subsets of programs that can take advantage of such instructions.
This is the approach used in the Intel's ICC compiler, which can be used to generate

highly optimised C++ and Fortran programs when targeting an Intel's architecture(Bik et al., 2002), by aggressively using techniques such as vectorisation, which is supported by most modern Intel CPUs via the utilisation of the SSE instructions and registers(Lomont, 2011). Following this approach, the ICC compiler focuses merely on recognising individual loops that can be converted into a SIMD form and then generates the appropriate vectorised implementation.
While ICC can perform more complex optimisations, such as automatically generating programs that take advantage of Intel's Hyperthreading technology when used in conjunction with OpenMP(Tian et al., 2002), all these techniques are mostly based about the leveraging of in-house knowledge about the implementing hardware, and are relatively straightforward and limited in scope from a program-to-program transformation point of view. Truly, one could even consider them "smart code generation technologies" rather than true "automatic parallelisation".

3.2.2 Automatic Parallelization via abstract representations

Another possible approach to automatic parallelisation is to derive some abstract representation of the program's source code, and then perform analyses and transformations on it. The intuition here is that through the selection of a proper abstract from one can move from the relatively "noisy" domain of source code to a cleaner environment, more amenable to automatic parallelisation efforts.

A very well known example of this approach is Polly, an automatic parallelisation infrastructure for LLVM, which relies on the transformation into a representation known as the "polyhedral model"(Grosser et al., 2012). Intuitively, the ideas behind the polyhedral model is a program representation focused on the expressions of loops in terms of their control and dependencies characteristics, which are modeled as "polytypes", n-dimensional geometrical objects, which each polytype representing a set of nested loops. Then, a large number of program transformations capable of extracting parallelism are reduced to a geometrical problems of partitioning the polytypes.

Another example of such an approach can be see in the work of Sato and Iwasaki, which utilize the algebraic properties of semirings to derive a matrix-multiplication form from C for loops implementing reductions.(Sato and Iwasaki, 2011). This is a perfect example of the power of utilizing abstract forms: once the original algorithm

is expressed in terms of a chain of matrix multiplications, parallelisation follows naturally due to the associative properties of the matrix multiplication operator (in fact, that work is fundamental inspiration behind the optimisations presented in this document).

3.2.3 Limitations

The main criticism of the methods discussed above is that the derivation of the abstract form, while greatly simplifying the process of parallelisation, might itself be an equally hard problem, greatly limiting the applicability of these techniques. By analyzing the works above, it is possible to see that a large effort is necessary in order to identify and separate the parts of the program which can benefit from the transformation. While this is fundamentally an unavoidable problem, much of the difficulty might be an artefact of the programming language optimised, rather than intrinsically associated with the abstract form selected. For example, Sato and Iwasaki's paper cover in depth methodologies to "untangle" idiomatically-written C for-loops, separating the parallelisable parts from the inherently sequential ones.

We can therefore conclude that while much research has gone in the field of automatic parallelisation, a central difficulty preventing widespread adoption is the fact that traditional imperative programs operate at a granularity that is too low for such techniques to be utilized to their fullest.

3.3 Conventional approaches to parallel programming

As underlined in the previous section, automatic parallelisation techniques suffer from structural applicability problems when applied to commonly used programming languages. A host of strategies have been developed to mitigate this issues. While each approach differs widely in implementation and scope, all these techniques are based on constraining the source program, as to superimpose a structure over the noisy nature of unconstrained source code. This is achieved either by providing extension to the source language (such as in the case of OpenMP and OpenCL) or by employing requiring the user to make calls to a particular library API to access the parallel features (such as pthreads and Thread Building Blocks). In this section, we will compare and contrast these common approaches to parallel programming.

3.3.1 Extension-based approaches

Extension-based approaches rely on the programmer writing in a slightly different dialect of the language when it comes to coding in parallel programs. This allows to repurpose otherwise sequential languages to express parallelism, while at the same time conserving much of the basic style, and therefore this methods tend to be very accessible.

3.3.1.1 OpenMP

Consider for example OpenMP, a framework for writing parallel programs in Fortran/C/C++. The OpenMP standard is based on the idea of decorating for-loops (the main parallelisable construct in these programming languages) with pragmas, that is with well-formatted preprocessor directives(Dagum and Menon, 1998). At compilation time, a compiler that supports the OpenMP standard will recognize the decorated loops and generate the necessary parallel implementation, using the information specified by the programmer in the pragma.

The pragma based approach has several benefits: firstly, it grants the OpenMP standard a significant amount of freedom in the definitions of what exactly each pragma should contain. Secondly, it maintains source compatibility, as compiler which does not support OpenMP can simply ignore the directives.
There are however also significant downsides to using OpenMP: the quality of the generated code is not always very high, and substantial complexities still arise when interacting with the source programming language, which can yield to inexplicably non-performant programs. To see an example of such complications, just consider indexing arrays within the body of a loop using C - this deceivingly easy operation might involve resolving aliases, or perhaps the index computation might be input-dependent in some way. In short, a framework such as OpenMP can be handy when applied to straightforward parallel loops but quickly becomes lacklustre when more complex code is encountered.

3.3.1.2 OpenCL & CUDA

Another potential avenue for extending a sequentially-oriented language like C is to derive a non-source compatible dialect and provide an ad-hoc compiler for it. This approach is particularly powerful in cases where implementing a different compiler is

necessary anyhow, such as when targeting a significantly different architecture from CPUs. As such, this is the approach adopted by the OpenCL standard(Khronos Group, 2016), which employs and implicitly SIMD dialect of C to express programs potentially targeting a whole host of different platforms. The OpenCL dialect of C closely resembles the C99 standard but has a few crucial differences, such as replacing the C standard library with a very constrained alternative - which for example lacks dynamic memory allocation.

In a framework like OpenCL, the entire burden of parallelisation is shifted onto the programmer. On the other hand, as the code is written in an environment which is implicitly parallel, this at least relieves the chore of having to decorate code explicitly. On the other hand, the OpenCL approach of writing the parallel parts of the program in their own incompatible dialect forces the programmer to split the source bases in parallel and non-parallel sections. Moreover, OpenCL requires the program to provide a significant amount of code interfacing the two.

From a usability standpoint, NVidia's CUDA offers and even more advanced model, wherein the program can be presented to the compiler as a single mixed-language source(Nvidia, 2010). It is the compiler's job to derive the two separate host-sequential guest-parallel programs, and removes the burden of generating the glue code to coordinate among the two.

3.3.2 Library-based approahces

An alternative to modifying the source programming language is to offer libraries to structure the parallel computation explicitly. This is the approach taken by Intel's Thread Building Blocks (TBB) library, which provides the programmer with a set of parallel algorithms, parametrised by user provided custom code(Reinders, 2007). The idea is that explicit parallelism only appears within each algorithm implementation, which is provided with the library. Therefore, all the user has to do is to derive an expression of their program in terms of such parallel API calls, by providing the suitable element-wise operations.

Taking this approach to the extreme, for some very common applications it is possible to simplify the life of the user even further, by providing full parallel implementation

of some particular algorithms with widespread usage. A salient example of this approach is CuBLAS(Nvidia, 2008): this famous linear algebra library exposes an API of fundamental functions, each of which has an high-performance GPU implementation provided by the NVidia engineers. Users of CuBLAS therefore do not need to concern themselves with notions of parallelism, as all the details are entirely encapsulated by the API.

This library based approaches are not without downsides, the most notable of all is that there applicability is limited to those platforms and systems for which such libraries are provided. Moreover, the libraries are black-boxes to the compilers, which limits the ability to optimize code further. This means that inter-pattern optimizations (from the simplest "map fusion" to more complex transformations such as the one presented in this document) are not easily supported.

3.4 Novel approaches to parallel programming

In recent times, new approaches to parallel programming have begun appearing. Many of these techniques take inspiration from the concepts previously presented - such as attempting to impose an algorithm structure over arbitrary source code - but at the same time try to completely alleviate the problems that arise out of writing the program in a traditional, sequentially oriented programming language, preferring a higher level, more declarative approach.

3.4.1 Application-specific DSLs

The Halide language for example is a DSL designed for the expression of efficient image processing pipelines(Ragan-Kelley et al., 2013). When using Halide, the programmer is expected first to specify an initial value function, from which the beginning of the computation can be derived, and a recursive reduction function, expressing the value of each point in terms of its neighbourhood. Both these functions are pure, in the sense that the Halide framework does not support arbitrary side-effecting statements. Once these functions are given to the framework, to together with some iteration bounds defining how long the computation must run, the framework takes over, executing the computation in a highly efficient manner: the user has provided an algorithmic description, but the framework is free to arrange the execution details as it sees

fit.

The Halide is a particularly relevant example of the work presented here, due to its interesting handling of Reduction operations: Halide also tries to automatically parallelise Reductions whenever possible, which means that it needs to device a system for determining whether an operator is associative. The approach to derive such inference however is radically different from the one proposed in this document: in Halide, an operator is classified as associative if it can be built up from some combination of associative operators, using a specified set of rules present in a database(Suriana et al., 2017). Using this technique, they are able to prove the associativity or complex operators, even spanning multiple dimensions.

This approach however has several shortcomings: firstly, while it can identify complex associative operators, it can not identify those operators for which a transformation into an associative operator is possible. This reduces the applicability of the algorithm greatly. Secondly, it requires a database to be pre-computed, and a compile time search, meaning that the solution might not scale arbitrarily as the data set and exploration space grows. Finally, this approach does not offer a sound, deterministic approach - being search based, it is not possible to provide a programming model that guarantees parallelisation - or, in case of parallelisation being impossible, that can communicate to the programmer the reason for this impossibility.

Other examples of the employment of DSLs to simplify parallel applications can be seen in the efforts of the Delite project(Chafi et al., 2011), which provides a framework for DSLs authors, in order to simplify the necessary work to embed a custom language within the Scala compiler. The authors of Delite suggest that at the current stage there is no practical solution to the difficulties of writing parallel software in an heterogeneous context, and therefore a good compromise is to facilitate DSL creation. And example of DSL currently implemented with Delite is OptiML, a DSL for machine learning(Sujeeth et al., 2011).

3.4.2 General purpose data parallel languages

The methods presented above attempted to simplify parallel heterogeneous programming by hiding the parallel implementation details within application-specific abstrac-

tions. Such approaches however rely on the existence of experts within each domain, capable both of choosing the right abstraction and possessing the skills required for the production of the DSLs. However, much research has taken place in recent times in the attempt of deriving a general purpose solution to the problem of parallel heterogeneous programming.

Such general approach is taken by Futhark, a project attempting to derive a platform-independent parallel programming language by relying on a functional programming paradigm(Henriksen et al., 2014). The intuition is that a parallel program can be seen as the composition of a number of primitive higher order functions, each performing pure transformations on their inputs. A Futhark program therefore can be seen as a totally platform independent construct, purely defining a series of algorithmic transformations.
Many of these primitive transformations however are known to have parallel implementations, and therefore the compiler can generate a parallel program for a target platform (CPU or GPU), resulting in an overall parallel program, without the user having to deal with explicit notions of parallelism per se.

Another project that operates in this scope is the Lift project. Lift is a functional programming language for expressing data-parallel computations(Steuwer et al., 2015), and as such shares many fundamental ideas with Futhark. In addition to the pattern-based approach of Futhark however, Lift provides a system of automated rewrite rules, which allow to derive many possible implementations from a single Lift program, and allow therefore to generate high-performance code for a host of various architectures(Remmelg et al., 2016). Lift is particularly relevant to the work presented in this document, as all the contributions presented here are ultimately included in the Lift compiler.

3.5 Summary

This Chapter offered a tour of the related research pertaining to automatic parallelisation and heterogeneous programming. It started by listing the traditional approaches and their shortcomings, and has then moved towards the more current actively research methods, contextualising the work proposed here within the broader area.

Chapter 4

Parallelising Reduction with Semiring Algebra

In this section, we show how the theoretical notions from abstract algebra can be used to create a framework for the parallelisation of programs. It starts by introducing the necessary concepts and the intuition behind a sequential-to-parallel transformation, heavily based on the work on automatic parallelisation of C programs as seen in (Sato and Iwasaki, 2011). The rest of the chapter is dedicated to adaptation of the methodology within a functional context, and presents a variety of potential extensions and generalization over the basic approach, a set of constraint for the verification of the applicability of the method based on syntactic rules, and a series of step-by-step transformations from the original form into the matrix based parallel version via rewriting rules.

4.1 Effects of associativity on the reductions

Before covering in depth the implementation of this project, let us analyse the optimisation in an abstract context. The general idea is as follows: consider a reduction having for operator a non-associative binary function. Such a reduction may not be executed in a tree-parallel fashion since the elements must be combined for the result to be correct. This fact can be seen if we consider reduction operations in terms of simple arithmetic formulas. Consider, for example, the subtraction of integers, a known non-associative operation.

$$3 - 1 - 1 - 1$$

The expression above can be seen as a reduction: the input is the sequence 3,1,1,1, and the reduction operator is subtraction. We can see that the result is determined by the order of execution of these steps (or, in mathematical notation, on the ordering of parenthesis). If execute the steps sequentially and in a tree-parallel order, we have:

$$(((3-1)-1)-1) \neq (3-1)-(1-1)$$

Contrast this with the addition operator, which is associative and therefore can be executed in a tree-parallel way:

$$(((3+1)+1)+1) = (3+1)+(1+1)$$

We have therefore derived a general rule: for a reduction to be tree-parallel, its operator needs to be associative.

At first, this conclusion seems to paint a bleak view with regards to the utility of parallel reductions. There are a significant number of algorithms which can be expressed as a reduction of non-associative operators, and following the logic presented so far, it seems that these algorithms are always condemned to be executed sequentially.

There is luckily a way out of this impasse: under certain circumstances, it is possible to derive a new reduction, operating on a transformed version of the original data, and using an equivalent, but this time associative, operator. The optimisation presented in this document does precisely the above: automatically deriving a parallel reduction in terms of matrix multiplication, from a non-associative reduction whose binary operator is constructed out of semiring functions. Before exploring the transformation more in depth, let's consider a concrete example.

4.2 Example: Polynomial Evaluation via Horner Scheme

The name Horner Scheme refers to a method for evaluating a polynomial at a given point. This algorithm has a straightforward implementation in terms of a non-associative reduction.

The algorithm accepts as input an ordered sequences of numbers, with the Nth entry

representing the coefficient of the monomial of the Nth factor in the polynomial. The evaluation of the polynomial at a point k is then computed by reducing left-to-right over this input sequence with the following operator:

$$\odot = \lambda(x,y). \rightarrow (k * x + y)$$

Now, $\odot$ is not an associative operator, and therefore the direct implementation of the reduction will be sequential, as the result of any given application of $\odot$ depends on all the previous ones. This dependency, however, is not arbitrary, but instead, has some notable properties.

First, notice that the reason for this dependency is the $k * x$ term of the expression, which can only be computed if x, the "prefix" of the reduction up to the current step, has alread been calculated. We can see this clearly by expanding a few steps of the reduction:

$$k * (k * (k * (x_0) + x_1) + x_2) + x_3$$

Unlike arbitrary operations, however, we know that multiplication distributes over addition. This allows us to transform this interleaving of products and sums into a sum-of-products form

$$k^3 * x_0 + k^2 * x_1 + k * x_2 + x_3$$

The expression in this form is now almost in a tree-parallel form: our “outermost” operations is just a reduction over a sum, which we know is tree parallel. However, what to do about the k^a terms?

An elegant solution to this question is to express the above chain of operations as a series of matrix multiplications. By taking a closer look at the expression, we can see that the above expression is equivalent to computing the following:

$$\begin{pmatrix} k & x_0 \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} k & x_1 \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} k & x_2 \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} k & x_3 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} k^3 & y \\ 0 & 1 \end{pmatrix}$$

and afterwards projecting out y from the result.

The expression above can be directly expressed as a reduction, with $\times$ as its operator.

Note that since we are multiplying square matrices, $\times$ is associative, and therefore the derived reduction is, in fact, tree parallel.

We have therefore achieved what we set out to do: transformed a non-associative reduction over one some data (in this case $\odot$ over integers) into an equivalent parallel reduction over another type of data ($\times$ over matrices), providing a pair of transformation to go from the original domain to the transformed one (the construction of matrices form ($x \rightarrow \begin{pmatrix} k & x \\ 0 & 1 \end{pmatrix}$) and one to return back to the original domain ($\begin{pmatrix} k & x \\ 0 & 1 \end{pmatrix} \rightarrow x$).

4.3 Generalizing over the data types with semirings

While the previous section covered a particular example, similar arguments to the one made above are valid for many other operators as well - although the exact specifics will vary from case to case. To reason more easily about the problem, and to allow the compiler to optimise whenever possible, it is necessary to derive a general framework to express and identify reductions that can benefit from this transformation.

As we have previously seen, the distributive property of * over + allowed us to "flatten" the expression, which in turn was fundamental for the expression of the operation in terms of a sum-of-products. Secondly, we used the fact that the only operators appearing in $\odot$ where + and * to express each term as a matrix. Thirdly, we needed to know that 0 and 1 are the neutral elements for + and * respectively. Moreover, finally, we relied on + being an associative operator - this might not have been clear from looking at the matrix form, but can be seen in the sum-of-products step.

It is not needed to introduce a new formalism to express all the conditions just mentioned: as what we have derived is the definition of the semiring, an algebraic abstraction over particular types of data and operations. (see background for a more in-depth explanation). Therefore, while the example seen so far was in terms of numerical operations, the same exact transformation can work on arbitrary types of data, as long as they respect the necessary properties. Examples of such other data types are the semiring of $(\{T,F\},\wedge,\vee,F,T)$,or the semiring of $(regexp,\|,\&,\varepsilon,\varepsilon)$ and many others.

4.4 Generalizing over the reduction form with higher-dimensional matrices

So far, we have only considered reductions of the type:

$$reduce : ((\alpha, \alpha) \to \alpha) \to [\alpha] \to \alpha$$

That is to say, reductions which use a binary operator and use it to combine pairwise elements of some collection.

There is, however, another, more general definition of reduction, with the following type

$$reduce : \alpha \to ((\alpha, \beta) \to \alpha) \to [\beta] \to \alpha$$

This operation, which is also known by the name of left fold, can be seen under two different interpretations: either as a reduction with a starting value, or has a different interpretation: given a starting state of type α and a collection of type $[\beta]$, compute a final state of the chain of state-transition specified by the given function. Notice how the latter does not expect α and β to be the same type - as one would expect the state of an algorithm to have a different type from its input.

It is possible, in some cases, to adapt the previously presented optimisation to work with left folds. The starting value extra parameter is not a difficult issue to tackle - all we need to do is to add one more operation to the reduction of matrix multiplications. The real difficulty lies with the fact of working with two different data types.
In general, we the transformation cannot support a reduction which has an arbitrarily typed state: the transformed version can still only operate over scalars belonging to the original semiring. It is in fact however possible to abstract over arity, by using higher-dimensional matrices.

For example, consider the Maximum Segment Size problem, which involves computing the maximum sum for any contiguous slice of a given input array of numbers.

This problem can be implemented in terms of a reduction with the following operator:

$$MssOp = \lambda((m,x),y). \rightarrow (max(m,x+y),x+y)$$

and then projecting out the first component of the resulting tuple.
This reduction operator has asymmetrical arity in its parameters: the accumulator and the result of each step are a tuple, while the input is a simple value.

Deriving a matrix-reduction form for the $MssOp$ operator is possible. First, notice that the scalar types and operators in the expression all belong to the $(Z+\{\infty\}, \max, +, \infty, 0)$ semiring. Now, for each input element y_i, we can define the following reduction step matrix for $MssOp$

$$\begin{pmatrix} 0 & y_i & -\infty \\ -\infty & y_i & -\infty \\ -\infty & -\infty & 0 \end{pmatrix}$$

To check that in fact, this matrix is correct, we can compute the application of a single step, by multiplying the matrix with a column vector, representing the accumulator of the reduction.

$$\begin{pmatrix} 0 & y_i & -\infty \\ -\infty & y_i & -\infty \\ -\infty & -\infty & 0 \end{pmatrix} \times \begin{pmatrix} m \\ x \\ 0 \end{pmatrix} = \begin{pmatrix} max(m+0, max(x+y_i, -\infty+0)), \\ max(m-\infty, max(x+y_i, -\infty+0)) \\ 0 \end{pmatrix}$$
$$= \begin{pmatrix} max(m, x+y_i), \\ x+y_i \\ 0 \end{pmatrix}$$

As we can see, the above multiplication implements the desired behaviour of the $MssOp$ and is, therefore, a valid implementation.

4.5 Constraints

We have previously stated that a necessary condition for the applicability of this transformation is that the initial reduction needs to operate over elements that belong to a semiring, and that it also must rely solely on the use of that semiring's operations. This is a necessary, but not sufficient condition.

The other, more stringent requirement, is that the reduction operator must be a linear function of the first parameter. This necessity arises from the fact that, during the

transformation from the original scalar domain into the matrix domain, we are in fact encoding partial applications of the operator with the parameter fixed, and as covered in the background section, for a function to be representable as a matrix, it needs to be linear.

Traditionally, verification of the linearity of a function is proved using mathematical tools such as equational reasoning. However, because this transformation is applied within the context of a compiler optimisation, such methods are inappropriate. Instead, it is more useful to describe syntactic rules matching over an expression's structure.

The approach adopted here is to define a single, general template for all formulas that satisfy the constraints. Here are the formulas for the 2-by-2 and 3-by-3 matrix representation case:

$$\lambda(x,y) \rightarrow a \otimes x \oplus b \otimes y \oplus c$$

$$\lambda((x_1,x_2),y) \rightarrow (a_1 \otimes x_1 \oplus b_1 \otimes x_2 \oplus c_1 \otimes y \oplus d_1),(a_2 \otimes x_1 \oplus b_2 \otimes x_2 \oplus c_2 \otimes y \oplus d_2)$$

With this scheme, constraint satisfaction is simply a matter of flattening expression by distributing all the $\otimes$ in the formula and then checking that every term is strictly included in the template.

4.6 Transformation steps

One of the strengths of the Lift programming language is the ability to express complex optimisations as chains of individually small rewrite rules. In this vein, this section will now present a step-by-step derivation of the parallel reduction in terms of such rewrite rules. For compactness and clarity, this section will continue using the maths-like notation employed so far, as opposed to utilising Lift, deferring the Lift-specific details to a later section. Furthermore, only the 2-by-2 case will be covered: similar arguments can be derived for any other arity.

The starting point for the transformation is any sequential reduction abiding by the constraints previously laid out.

$$reduce(\lambda(x,y) \rightarrow a \otimes x \oplus b \otimes y \oplus c, i_0)[i_1,...,i_N]$$

As a first step, let's factor together with the operations in the reduction, so that we obtain a clear structure.

$$reduce(\lambda(x,y) \rightarrow (a \otimes x) \oplus (b \otimes y \oplus c), i_0)[i_1,...,i_N]$$

Now, the term $b \otimes y \oplus c$ can be considered a function onto itself. Let's therefore define $t = \lambda(y) \to (b \otimes y \oplus c)$.

$$reduce(\lambda(x,y) \to (a \otimes x) \oplus t(y), i_0)[i_1, ..., i_N]$$

Since t is the first function applied to the y argument in the reduction operator, which corresponds to the input data, we can separate it as follows

$$reduce(\lambda(x,y) \to (a \otimes x) \oplus y, i_0) \circ map(t)[i_1, ..., i_N]$$

Let's now begin to move towards operating over matrices. By the arguments presented in previous sections, we see how $((a \otimes x) \oplus y)$ can be expressed as a single matrix-vector multiplication and a projection, by considering the expression

$$\begin{pmatrix} a & y \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ 1 \end{pmatrix} = \begin{pmatrix} a \otimes x \oplus y \\ 1 \end{pmatrix}$$

and then taking out the top element. We can therefore have

$$get_0 \circ reduce(\lambda(x,y) \to y \times x, \left(\begin{smallmatrix} i_0 \\ 1 \end{smallmatrix}\right)) \circ map(\lambda y \to \left(\begin{smallmatrix} a & y \\ 0 & 1 \end{smallmatrix}\right)) \circ map(t)[i_1, ..., i_N]$$

We can then fuse together the two maps, and by re-expanding t, we obtain

$$get_0 \circ reduce(\lambda(x,y) \to y \times x, \left(\begin{smallmatrix} i_0 \\ 1 \end{smallmatrix}\right)) \circ map(\lambda y \to \left(\begin{smallmatrix} a & (b \otimes y \oplus c \\ 0 & 1 \end{smallmatrix}\right))[i_1, ..., i_N]$$

The expression now has the desired form: a parallel map followed by a tree-parallel reduce.

4.7 Summary

This chapter has presented the optimisation algorithm in a language agnostic functional context. It has started by detailing the basic intuition as found in the literature, and has then provided original contributions, such as the generalisation of the algorithm for asymmetrically-typed reductions, a set of syntactic constraints to determine when the optimisation can be applied, and a sketch of correctness via rewrite rules.

Chapter 5

Lift implementation

5.1 Implementation of a parallel reduction

Just like in the hard-coded examples presented above, there are many possible implementations for the parallel Matrix Multiplication reduction in low-level Lift. For simplicity and ease of comparison with the handwritten versions, we will only consider the implementation for the 2-by-2 case.

5.2 Generated code

Here is the code as generated after the optimisation pass:

```
fun(
    ArrayType(TupleType(T,T),N),
    data => {
        Join() o MapWrg(
            MapLcl(toGlobal(id) o Get(1)) o ReduceSeq(
             fun(TupleType(T,T), ArrayType(TupleType(T,T)),
               (accum, data) =>
                toLocal(matMul).apply(accum,
                     fun(x => x.at(0)) o
                  ReduceSeq(matMulTuple, nautralPair)) o
                  o MapSeq(transform))
                  o Split(2) o
                  toLocal(MapSeq(id)) $ data
                 )
             ),
             neutralPair
           ) o Split(64)
        ) o Split(2048) $ data
```

```
19      }
20 )
```

It might be helpful to refactor the code above in it's main, coarser grained steps. Conceptually, the program above is equivalent to the following pseudo-Lift:

```
1 fun(
2     data => {
3            Join() o MapWrg(
4             MapLcl(matrixToT) o
5             outerReduceMatrices(
6                     innerReduceMatrices o
7                     buildMatrices o
8                     copyToLocalMemory,
9                     neutralPair) o Split(64)
10            )
11         ) o Split(2048) $ data
12     }
13 )
```

Listing 5.1: Simplified pseudo-Lift

As we can see, the optimised code, therefore, executes, for each workitem, the following steps (since the code is written in point-free style, the order of execution is bottom-to-top, right-to-left)

- copyToLocalMemory: copy the data from its split to local memory, to avoid storing intermediate results in slow, global memory.

- buildMatrices: maps a function deriving, for each input element, the equivalent matrix for one step of the reduction.

- innerReduceMatrices: a sequential reduction implementing matrix multiplication. We start with the identity matrix as a neutral element of the reduction.

- outerReduceMatrices: same as innerReduceMatrices, used to combine the results of each split. Equivalent to the "chunk computation" of the OpenCL version

- matrixToT: converts the results back from the matrix representation to the original datatype by projecting out the appropriate entries

5.3 Optimisation-language interface

In the previous section, the Lift optimised code contained several functions which are not existing Lift primitives. These are

- matrixToT: The function converting matrices into the resulting element
- matMulTuple: The semiring-specific matrix multiplication
- neutralPair: The semiring-specific identity matrix
- transform: A function transforming an input item into the corresponding iteration step matrix.

All these functions are specific to a particular semiring and therefore will be algorithm specific. Even more importantly, the transform function, tasked with generating the appropriate Matrices for each given iteration, and therefore depend also on the details of the reduction operation.

To have a truly automatic transformation - as opposed to just a conveniently expressed parallel pattern, it is necessary for the compiler to infer these four entities automatically. This can be done by augmenting the Lift language with the necessary capabilities to represent semirings and expressions.

5.3.1 The semiring representation

The first extension needed is to enrich the compiler with a concept of what a semiring is. To have that, first, the concept of an abstract operator needs to be modelled.

```
1 abstract class SemiringOp(neutralElem:Value) {
2     def cString(param1:String,param2:String)
3 }
```

The above definition might seem unusual, but it fits well within the current Lift implementation, wherein custom user functions are expressed as fragments of C code. In the above, the cString method of an operator is expected to generate the appropriate C implementation, given expression param1 and param2.
Armed with this structure, it is now possible to define the representation for a semiring

```
1 case class Semiring(plusLike:SemiringOp, timesLike:SemiringOp)
```

Listing 5.2: "The semiring data structure

With the above definition, it is now possible to derive a general implementation for both neutralPair and matMulTuple.

```
def neturalPair(semRng:Semiring) =
    (semRng.multLike.neutralElem, semRng.plusLike.neutralElem)

def matMulTuple(semRng:Semiring) = {
  val entry1 = semRng.timesLike.cString("a._0","b._0)
  val entry2 =
   semRng.timesLike.cString("a._0",semRng.plusLike.cString("b._1
      ↪", "a._1"))
  UserFun(/*Type parameters omitted*/s
      "{Tuple t = {$entry1, entry2}; return t;}"
   )
```

Here are a few example usages of this approach. First one must define the available operators and their C implementation.

```
val add = SemiringOp(0) {
    def cString(p1:String,p2:String) = s"$p1 + $p2"
}
val mult = SemiringOp(1) {
    def cString(p1:String,p2:String) = s"$p1 * $p2"
}
val max = SemiringOp(1) {
    def cString(p1:String,p2:String) = s"$p1 > $p2 ? $p1:$p2"
}
```

Then, it is possible to define specific instances for each algorithm. For example, to compute the Horner Polynomial evaluation, the following Semiring is to be used:

```
val floatNaturalSemRng = Semiring(plusLike = add, timesLike =
    ↪mult)
```

While to implement the Maximum Segment Size problem, it is possible to use:

```
val floatMaxSumSemRng = Semiring(plusLike = max, timesLike =
    ↪plus)
```

One limitation of the following approach is that all these definitions live at the compiler level, rather than at the language level. As of the time of writing this document, there are no specific plans for implementing a language level system for defining semiring instances. In future however it might be possible to support such user definitions, perhaps using a typeclass system similar to those found in other functional programming languages such as Haskell.

5.3.2 The LinearExpression representation

Supplying a definition of semiring is not sufficient. As noted the "transform" function, tasked with the transformation of the input from its original domain into its Matrix representation, depends on the specifics of the operator supplied to the reduction: a simple way to see this, is that each Matrix must represent a partially-applied version of the operator.

As noted in section "2.4.2 Linear Functions", we know from algebra that the functions that can be represented as Matrices coincide with the class of linear functions. Therefore, it is sensible to enrich the compiler with a notion of a Linear Expression, that is an expression whose body is a linear function. This representation is derived by reifying the expression as an ordered series of coefficients, representing the entries of the equivalent Matrix implementing that function.

5.3.3 Transform function generation

Assuming an instance of such representation is available, then it is straightforward to derive the C code implementing the "transform" function. For example, let's consider the operator seen before in the Horner Scheme example

$$\odot = \lambda(x,y). \rightarrow (k * x + y)$$

First, let all the coefficients be explicit:

$$\odot = \lambda(x,y). \rightarrow (k * x + 1 * y)$$

We can now simply express the above as a sequence of indices. Since the bottom row is always $[0,1]$, there is no need to keep track of it

$$[k,1]$$

Finally, from the above, the following C code is generated (remember that the 2-by-2 matrix generated is implemented as a simple pair, as the bottom row is always constant and therefore implied at runtime).

```
1 Tuple2 transform(float input) {
2     Tuple t = {k, 1*input};
3     return t;
4 }
```

At this point, the attentive reader might have a few questions:

- $\odot$ is a function that takes two parameters, and assuming the last row of the generated matrix must be $[0,0,1]$, why is there only a single row for both the variables x and y? Should it be represented as a 3-by-3 matrix?
- Why does the C code multiply the input by the coefficient of y, and not of x, or both?

The two questions above are deeply related. The answers to both lie in the fact that while $\odot$ is indeed a function of parameters, each matrix generated represent a single step of the reduction having y partially applied. After said partial application, $\odot$ becomes a single-parameter function of the reduction state and is therefore implemented as a 2-by-2 Matrix (by taking account the requirement of having the last row be $[0,1]$).

There is, however, a problem with the design as presented so far. What are the indices for the following operator?

$$\lambda(x,y) \rightarrow a*x+b*y+c$$

Clearly, $[a,b]$ would not suffice, as there is also the constant term c to be accounted for. However, $[a,b,c]$ would require us to move into 3-by-3 matrix territory, and while this would be a possible solution, it is inefficient.

A better solution to the problem above is to enhance our representation for a LinearExpression by including a new field representing the constant term. Then, the transform function generated can access this information. This allows to generate code of the form:

```
1 Tuple2 transform(float input) {
2     Tuple t = {a, b*(input+c)};
3     return t;
4 }
```

5.3.4 Inferring LinearExpressions in Lift

We have mentioned above a simple methodology for the inference of linear expressions given the formula of a given operator. However, at the time of the writing Lift does not possess an expression level language, and expects all custom user behaviour

to be expressed as snippets of C code.

While is in theory possible to construct a LinearExpression from a block of C, in practice this is a significantly involved task - especially if when arbitrary length blocks of code are supported. On the other hand, forcing users to derive the correct values for a LinearExpression manually is unnatural and potentially error prone.

A simple solution to the problem above is to supply a very compact expression language, resembling the simply typed lambda calculus plus arithmetic operators. It would then be easily possible to first, verify the constraints of applicability of the expression as previously seen, and then, once the constraints are checked, to generate the appropriate LinearExpression instance.
Since it would also be trivial to generate C code from the proposed expression language, this would give Lift the option to adopt a transparent interface to parallel reductions: whenever an expression satisfies the constraints, then the compiler would proceed down the LinearExpression path, eventually generating a parallel reduction. If on the other hand the constraints are not met, then the compiler can simply generate the direct C implementation for the operator, and execute the reduction sequentially with the already existing Lift primitives.

5.4 Summary

This chapter has presented the extensions to the Lift compiler used to implement the algorithm as proposed in the previous chapter, covering first the parallel matrix multiplication reduction generated by the optimisation, and then moved to detail the necessary analyses and steps to identify and generate that optimised code form Lift's sources. Finally, it concludes with the design of an expression language to be added to Lift, to make the optimisation completely transparent to the user.

Chapter 6

Handwritten implementations

This section documents the exploratory work used to both confirm the validity of the method as proposed in (Sato and Iwasaki, 2011), and also to explore the implications of applying the method to a GPU platform for the first time. This work resulted in the writing of a series of optimised parallel implementation for the 2-by-2 parallel matrix multiplication, which also served an informative purpose in the derivation of the optimised Lift form.

6.1 Motivation

In a previous Chapter, we have presented the general principles for the optimisation: a mathematically-sound approach for the transformation a class of reductions with non-associative operators into reductions with associative operators. In theory, this transformation of a program from not-parallel into parallel yields an arbitrarily large benefit. In practice, however, there might be engineering issues that limit the usability of the parallel implementation.
For example, the parallel versions of the algorithm perform a reduction over matrices, whereas the original directly work on scalars. This has the obvious implication that the optimised version uses much more memory than the naive version - and is, in fact, possible for this increased memory consumption might lead to worse performance.

Moreover, while the optimisation is in theory data and operation agnostic, the commonly envisioned use cases include many algorithms that revolve around arithmetic operations over numeric data-types. In practice, is possible that available sequential architectures are capable of optimising such instances much better than the general case,

optimisations that might be available in the parallel implementation: which would yield, in practice, to a smaller performance gain (if any) than predicted.

Furthermore, previous work on this subject has considered the behaviour of the optimised code in a CPU setting only: however, the work presented here is mostly geared towards implementation on GPU platforms. Before proceeding with a full implementation in Lift, a comparison between a CPU and a GPU implementation was needed, to confirm that GPUs are indeed the right platform for this sort of computation.

Finally, handwriting the OpenCL implementation served a secondary, didactic purpose for the author, which had no prior experience with OpenCL. This is of vital importance because the Lift's compiler primary target is indeed the OpenCL framework. Handwriting a high-performance implementation, including the various GPU specific optimisations that this entails, proved to be invaluable even for the successive phase of the project.

For all these reasons, a variety of programs have been written, implementing parallel reduction over 2-by-2 matrices (that is, manually implementing the code that the actual optimiser might generate), trying out different strategies and optimisations over a variety of platforms.

Note that, unlike the Lift implementation presented later, the following parallel version will expect their input to already appear in the form of matrices. The implications (or better, the lack thereof) of this choice will be tackled in depth in the Lift implementation section.

The rest of this section is concerned with the details of each of the implementations, and general comments on their relative performance. A more in depth presentation of the results can be found in the Evaluation chapter.

6.2 The baseline sequential implementation

The sequential baseline implementation is straightforward: a simple, multiply and add accumulating for loop, implemented in C.

```
1 float acc = 0;
```

```
for(int i = 0; i < N; i++) {
    acc = acc*k + a[i];
}
```

An important observation to be made is that, for all its naivety, this snippet of code is unusually hard to out-compete: such code is widespread, and there exist a plethora of established compiler techniques that can optimise loops of this form. Moreover, the algorithm incurs in no initial overhead - something that the parallel versions derived next will instead have to tackle with.

This version is also somewhat advantaged due to the nature of operation used: it is performing a simple, fast float multiplication and addition, raising even further the amount of data needed for a parallel version to overtake the sequential.

Finally, one must always bear in mind that the sequential implementation is doing significantly fewer operations, as it operates over scalars as opposed to matrices (although the parallel versions have been optimized to not fully materialize the matrices, and avoid doing computations for which the result is always constant).

6.3 OpenMP implementation

The first implementation attempt was to leverage OpenMP and its high-level interface. As such, no special efforts were taken towards the production of a sophisticated implementation but rather were limited to observe if the semi-automatic parallelisation facilities of OpenMP were sufficient for the task.

The implementation consists of a single for loop, reducing over tuples representing matrices. The for loop is tagged as parallel.

As expected, OpenMP is not able to generate competitively parallel code from the description above. This is not to say that it is impossible to express an efficient parallel implementation using OpenMP. Doing so, however, would require a significant amount of effort for the derivation of an implementation that would still have to endure the overheads associated with OpenMP's high-level nature.

6.4 Pthreads implementation

Unlike OpenMP, the pthreads library takes a much lower level approach to parallel programming - giving us the desired degree of control.

The pthread algorithm implements a simplified form of tree reduction: the input sequence of matrices is split in adjacent chunks, one per available thread. Then each thread proceeds to reduce sequentially in isolation. After the threads have all finished their reduction, the spawning thread will perform one more reduction of the remaining data.

This two step reduction fits perfectly with the characteristics of a CPU regarding parallelism, that is having threads with very high individual computational power, but very limited in number. By chunking the original input by the number of threads, we ensure maximum utilisation and balance. It is not necessary to implement a full tree reduction either, as the single-threaded second step will process merely "number of threads" elements - that is, very very few.

The pthread implementation performed well, beating the sequential baseline as expected even with a relatively small number of input matrices (a few thousand elements). It was therefore selected as the best implementation on the CPU platform, as far as comparison with GPU implementation is concerned.

6.5 OpenCL implementation

We now turn to the OpenCL implementation, which is the principal item of the preliminary exploration. The goals for this implementation were twofold: first, to measure the different impact of the optimisation on CPU vs GPU, therefore motivating the entire enterprise, and secondly to act as a tool of comparison for the Lift implementation.

Initially, the implemented algorithm followed a simple two-level reduction scheme: wherein the input data is divided into a series of contiguous chunks, each of which is a signed to one OpenCL workgroup. Then, within each chunk, a second contiguous subdivision takes place, with each sub-chunk being assigned to an individual thread. This division involves no actual runtime work besides a few index and ranges computations.

Then, each thread performs a sequential reduction and waits at a barrier for synchronisation. Finally, when all the threads in a workgroup have synchronised, thread number 0 performs one final sequential reduction. In a highly simplified pseudo-OpenCL, we have

```
1 matrix myAccumulator = identityMatrix;
2 for(int i = mySubchunkBegin; i < mySubchunkEnd; i++) {
3     myAccumulator = myAccumulator X input[myChunkAddress + i];
4 }
5 local_buffer[thread_id] = myAccumulator;
6 barrier();
7 if(thread_id == 0) {
8     myAccumulator = identityMatrix;
9     for(int = 0; i < num_threads; i++) {
10        myAccumulator = myAccumulator X local_buffer[thread_id];
11    }
12    output[workgroup_id] = myAccumulator;
13 }
```

There are several potential issues with this implementation strategy. Firstly, the result of the reduction is not a single matrix representing the result, like in the previously presented CPU implementations, but instead is an array of values, each representing the final result for one OpenCL workgroup. This, however, is a common feature of programs run on the GPU, as workgroup level synchronisation is not supported by default. A better solution is to execute yet one more reduction sequentially on the CPU host. This has virtually no impact on performance, as the amount of data to reduce is quite small compared to the initial input - just one entry per workgroup.

6.5.1 Memory Coalesced Accessed

More serious issues arise when dealing with memory access patterns. When performing computations on a non-integrated GPU, accesses through Global memory have a very significant latency compared with Local memory accesses. Therefore, GPU's can batch together multiple requests, increasing the maximum bandwidth. This is however only advantageous if all the access is to memory locations belonging to different cache lines.

The parallel matrix-multiplication kernel presented here is a memory bound kernel, that is to say, that memory operations are the performance bottleneck, rather than the calculations themselves. Therefore, speeding up memory accesses is essential. The

algorithm is therefore extended to include a first step wherein each thread copies the input from the slow Global memory into a local workgroup-shared buffer, to efficiently manipulate the data. For maximum performance, the accesses are "coalesced" together, by making sure that each thread copies the element at a fixed step stride, longer than the size of a cache line, thus ensuring that all the bandwidth available is used.

```
1 for(int i = 0; i < chunkSize/numThreads; i++) {
2     local_input[i*(thread_id % num_threads)] = input[i*(
        ↪thread_id % num_threads)];
3 }
4 barrier()
5 /** Same as before, but reading from local_input instead of
    ↪input**/
```

As a further optimisation, the kernel reuses the "local_input" buffer to store the result of the each thread_level sequential reductions.

6.5.2 Buffer Usage Optimisation

While the implementation of memory coalescing via a local buffer yields great performance boons, it also introduces a significant problem: local_buffer needs to be large enough to fit the and entire workgroup-level chunk. In theory, this could simply be achieved by increasing the number of workgroups. However, OpenCL devices are not guaranteed to support an arbitrarily large number of workgroups (notice that is not directly related to the number of compute cores on the physical device, as the OpenCL standard does not require workgroups to run in parallel, but merely concurrently).

A solution to the problem presented before is to restructure the algorithm by introducing one more level of iteration: each thread is no longer expected to process its entire slice of input in one go, but instead in a number of iterations dependent on the ratios of input size and buffer size. With this changes, it is now possible to decouple the size of the local memory buffers from the number of workgroups running.

```
1 for(int step = 0; step < localBufferSize/(inputSize/chunkSize);
    ↪step++) {
2     /*Same as before, but with the subChunk end computed based
        ↪on the localBufferSize*/
3 }
```

6.5.3 Multi-step Reduction

While all these adjustments are enough to make the OpenCL kernel much faster than the PThread CPU equivalent, there is still plenty of room for performance improvement. In particular, by timing the kernel at the current state, we would notice that a significant part of the runtime is spent in performing the single-threaded workgroup level reductions after each thread-level reduction step. Therefore, a new intermediate parallel reduction step is introduced.

```
1 /*Memory coalesced copy and thread-level reductions */
2 if(thread_id < 32) {
3   myAccumulator = identityMatrix;
4   for(int i = 0; i < num_threads/32; i++) {
5       myAccumulator = myAccumulator X local_input[thread_id*(
          ↪num_threads/32)+i];
6   }
7   local_input[thread_id] = myAccumulator;
8 }
9 /*Single-threaded workgroup reduction*/
```

Note very importantly that only 32 threads are used. There are two reasons behind this choice: firstly, the total number of threads running the kernel is always a power of two, and therefore 32 happens to be a good divisor, eliminating the need for input padding and alignment problems. Secondly, this kernel is intended to be tested on NVidia K20m GPU, which has a warp size of 32: which means that all the threads will execute in lockstep, eliminating the need for a barrier.

6.6 Summary

This Chapter has presented a series of handwritten implementations for the parallel matrix multiplication reduction, using different frameworks and platforms, to be used as both as a preliminary exploration tool for the generation of optimized code, and as a benchmark for its quality.

Chapter 7

Evaluation

In this chapter, we evaluate the performance of the optimised Lift code as presented in Chapter 5 by comparing it when the handwritten implementations presented in Chapter 6. Also, we explore the effects that various design decisions had on the algorithm performance. Moreover, as the previous comparisons are performed on both Integrated and Discrete graphics cards, we are also able to compare the performance of the algorithm on this two different device types.

7.1 Methodology

7.1.1 Experimental Setup

All the experiments presented here were conducted on two machines. The first was used to run all the experiments labelled with "CPU" and the "Integrated GPU". The machine has an Intel i5-6500 for CPU, and an Intel HD530 integrated GPU, and 16 GB of DDR4 ram - enough to make memory usage a non-issue for all the experiments.

The other machine was used to run all the experiments labelled as "NVidia GPU". The specific device used is a NVidia Tesla K20m. The machine also has an abundant 16 GB of RAM but used the older DDR3 technology.

All the figures presented refer to the average runtime, as each experiment has been run several for thirty-one repetitions, to minimise the effects of noise and random variations. The first iterations for each experiment have been discarded, as some devices - especially GPUs - are known to have a certain "warm up" time in which the perfor-

mance is not the same as at full regimen.

7.1.2 Goals and Strategy

The central aim of this evaluation is to ascertain whether the optimisation added to the Lift compiler can generate code that performs comparatively with a manually optimised equivalent implementation. This is mainly achieved by direct comparison of the runtimes of the best version for each technology-platform pair. Also, an untransformed CPU baseline implementation is provided for reference.

In addition to the above, we present a comparison between unoptimised and optimised Lift code, as to evaluate the performance gained by applying our technique within the context of Lift. Finally, two more experiments, exploring various minor aspects of the generated Lift implementation, and comparing their performance implications: the first one measuring the performance effects of matrix materialisation within the algorithm, and finally a quick comparison of the effects of varying the number of workgroups on the performance of the application.

Moreover, since the Lift exclusive experiments have been run on both the Integrated GPU and the NVidia GPU targets, they can be used as a tool to verify that the optimisation performs similarly well on both platforms.

7.1.3 Choice of input

All the experiments implement the 2-by-2 instance of Horner Polynomial evaluation. The input to the computation was produced by a handwritten generator algorithm. The correctness of the output was checked for all the experiments by comparing the execution's result with a sequential implementation running on the same exact data.

The results presented here operate on data ranging from 2^{16} to 2^{28} element matrices, which each element being either a floating point number or a pair of floating point numbers depending on the specific algorithm's internal representation of the matrices. The chosen sizes are large enough to allow for the effects of parallelism to be easily measured, while at the same time still fitting comfortably within the bounds of the available RAM on each system.

7.2 Results

7.2.1 Comparison of unoptimized Lift implementation

The following figures present a comparison between the following Lift program and its optimised counterpart.

```
MapWrg(MapSeq(toLocal(id)) o ReduceSeq(horner_step, 0)) o Split
    ↪(2048) $ data
```

Where horner_step is the custom C function, defined for some value of K

```
#DEFINE K /*some value here*/
float horner_step(float x, float y) {
    return K*x+y;
}
```

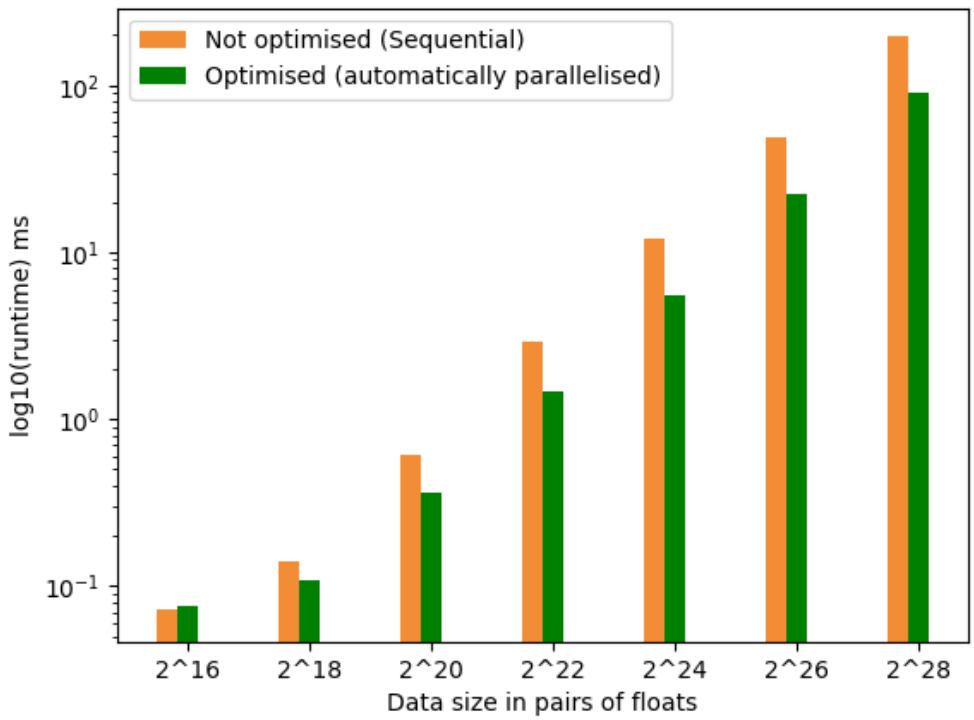

Figure 7.1: Median runtime comparison of unoptimized vs highest performing optimized Lift version on the Integrated GPU. Lower is better

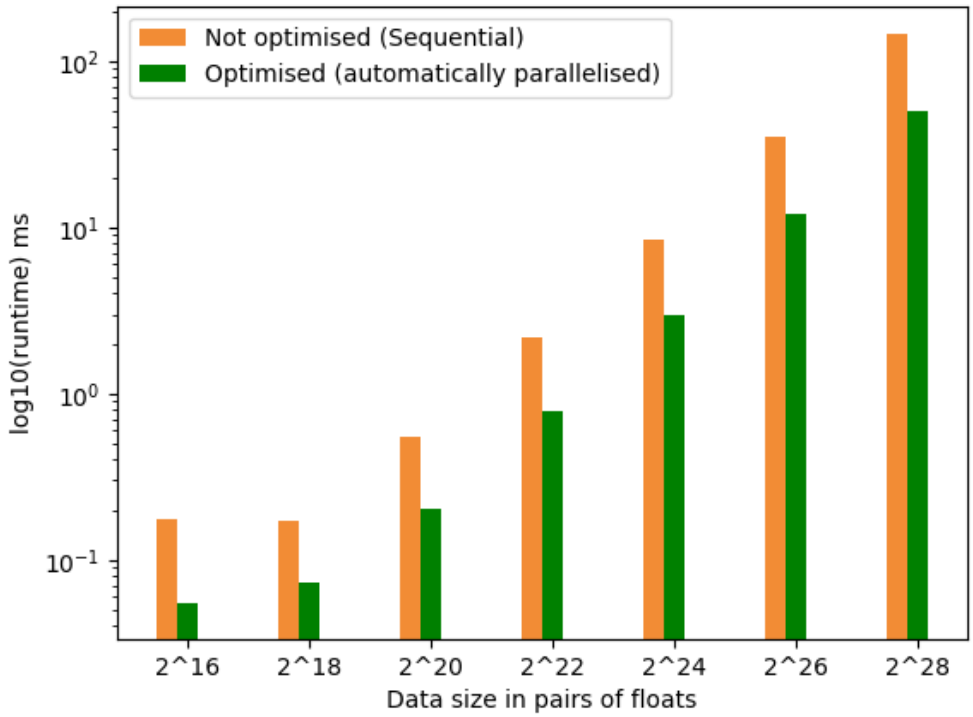

Figure 7.2: Median runtime comparison of unoptimized vs highest performing optimized Lift version on the NVidia GPU. Lower is better

As expected, the optimized version is consistently faster then the unoptimized baseline, across all data sizes and implementing platforms.

7.2.2 Effects of the variation of number of OpenCL workgroups

Ultimately, the compiled Lift code needs to be executed via OpenCL, which means that it is necessary to decide the local and global worksizes.
In general, this involves deciding both the number of workgroups and the number of threads per workgroup that the kernel needs to be run with. In the case of the optimised parallel matrix multiplication generated by the optimisation, however, the number of threads per workgroup has been fixed in the source code to be 64. This was motivated by memory usage considerations with regards to the allocation of static memory buffers and loop unrolling, which are crucial for the derivation of a high-performance implementation.

What remains to be decided therefore is the number of workgroups to be used. Since

each batch of input data was a power of two, it was also decided to have a power of two number of workgroups - to eliminate load-balancing problems from the investigation.

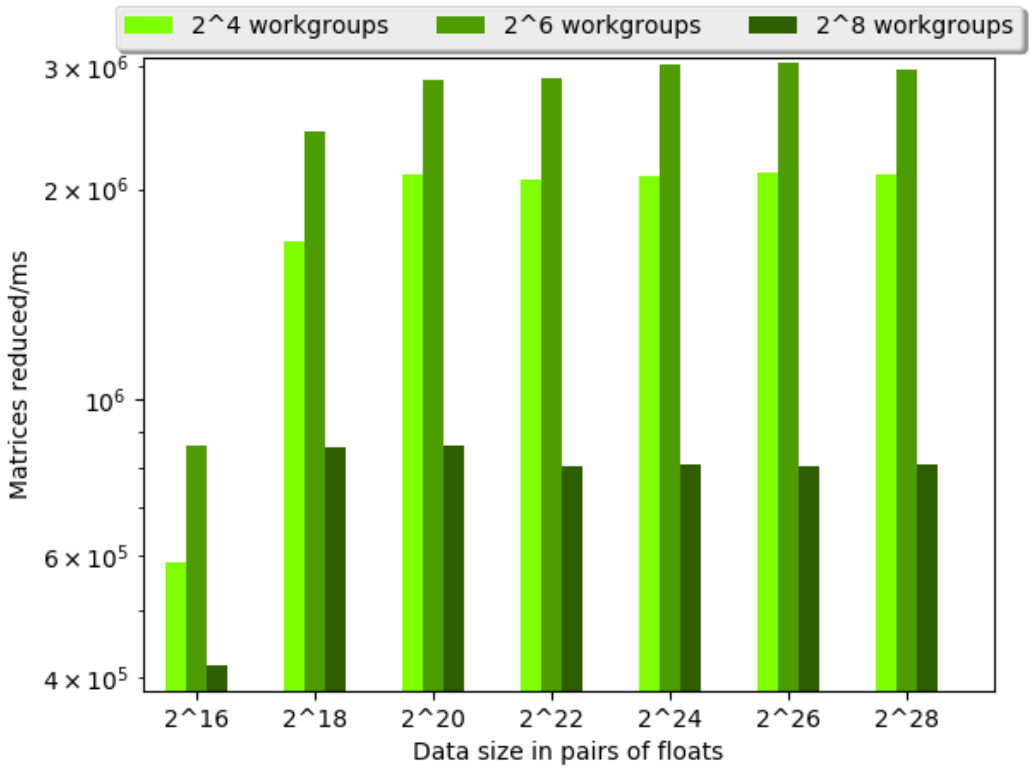

Figure 7.3: Effects of worksize variation on median runtime for Integrated GPU. All versions are non-materialised. Higher is better

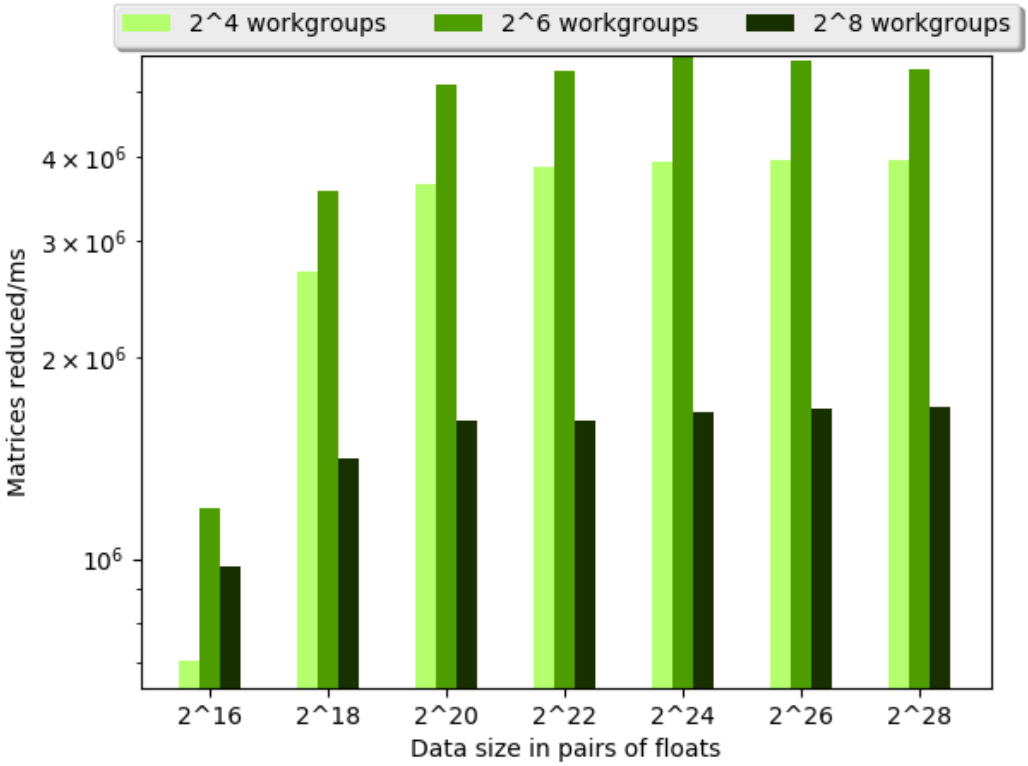

Figure 7.4: Effects of worksize variation on median runtime for Integrated GPU. All versions are non-materialised. Higher is better.

As we can see from the graphs above, the best performance is achieved with 2^6 workgroups. As such, this is the value employed for the Lift generated kernels in all the experiments shown.

7.2.3 Effects of matrix materialisation

We now analyse the effects of matrix materialisation, that is the performance impact of actually producing a concrete representation of the matrices. This is worth measuring because it is possible to avoid materialising the matrices, at the cost of increasing the complexity of the algorithm. In particular, the following versions have been explored:

- Full materialisation: the naive implementation, in which the matrices are represented verbatim in memory as an appropriate data structure (for example, in the 2-by-2 case, this means representing a matrix as four floats). This has the advantage of being the simplest method.

- Partial materialisation: Taking advantage of the fact that the bottom row of the

matrices is always constant, we can save space and computation time by avoiding to compute those entries. This, however, requires us to generate a custom matrix-multiplication operation that is aware of this fact and can correctly compute the values of the materialised entries, behaving as if a bottom row was present. In the 2-by-2 case, this reduces significantly both the space consumption and the number of operations performed by the algorithm.

- No materialisation: Exploiting the fact that each matrix is ultimately a function of some scalar input, we can avoid materialising the matrices at all, and just compute the entries lazily when they are required. Concerning partial materialisation, this reduces the space requirements but leaves the number of operations unaltered.

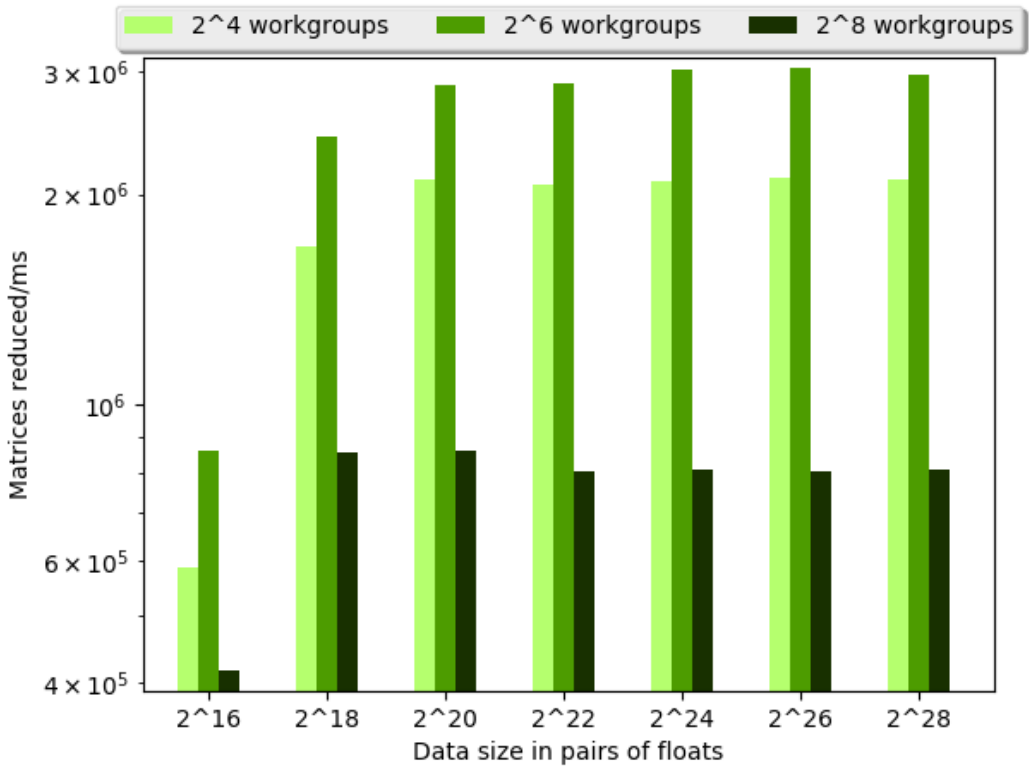

Figure 7.5: Effects of matrix materialisation on median runtime for the Integrated GPU. All implementation use the best worksize configuration. Higher is better

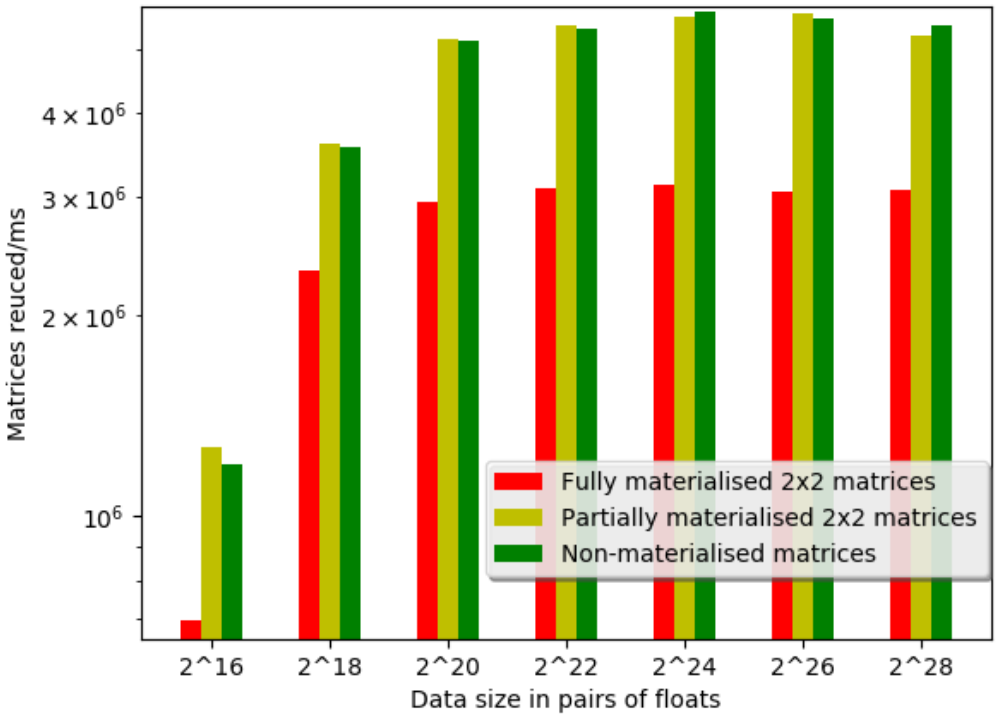

Figure 7.6: Effects of matrix materialisation on median runtime for the NVidia GPU. All implementation use the best worksize configuration. Higher is better

As we can see from the graphs above, full materialisation is significantly outperformed by the other two version. On the other hand, the difference between partial and no materialisation is much more narrow, particularly on the NVidia GPU. This is probably because on the Integrated GPU materialised matrices will be stored in main memory, while non-materialised matrices can easily fit into registers. On the NVidia GPU, this is not the case, as the number of matrices materialised per each compute unit is sufficiently small to fit in the high-speed private memory.

7.2.4 Comparison across all implementations

We conclude the evaluation by comparing the speedup over a sequential C baseline achieved by all the parallel matrix multiplication reduction implementations produced through the presented work, to evaluate the quality of the generated Lift code in a more general setting.

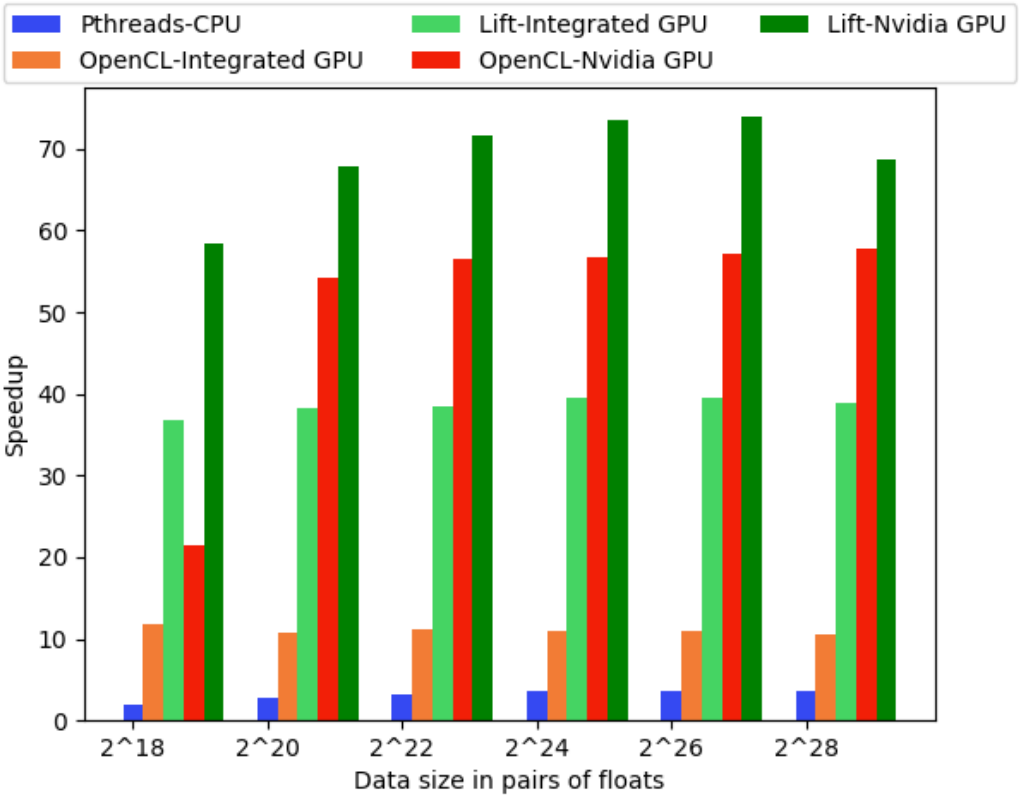

Figure 7.7: Speedup across all parallel implementations compared to the sequential C baseline. Higher is better

As we can see from the above, Lift implementations for both the Integrated and NVidia GPUs consistently outperform the manually written OpenCL GPU implementation, which is, as expected, much faster than the pthread parallel implementation on CPU.

However, it is not safe to conclude that the code generated by Lift is better than what can be achieved by OpenCL. From the graph above, we can see that for the 2^{18} entries experiment, the handwritten OpenCL on the NVidia GPU only reaches half the speedup of the Lift generated code for the integrated GPU. This suggests that the handwritten OpenCL code is not optimal.

Instead, one can merely conclude that the optimisation at applied for Lift results in kernels that can outperform handwritten average quality OpenCL kernels, while programming in a significantly simpler and safer environment.

7.2.5 Comparison between generated Lift and handwritten OpenCL versions

In the previous section, we have compared the generated Lift code with the handwritten OpenCL implementation and seen that the OpenCL kernel is outperformed by Lift, in some cases even when running on a more powerful device. What follows is a list of similarities and differences between the Lift and OpenCL implementations, which might be useful to understand the reason for this performance gap better.

- Both the OpenCL and Lift versions execute a two-step reduction, wherein each local thread is tasked to multiply together a contiguous slice of the input matrices, with a successive workgroup level reduction combining the individual results.

- Both the OpenCL and Lift versions represent matrices as fixed-size tuples, omitting the bottom-row, which is always a constant series of 0s followed by a single 1. Similarly, the matrix multiplication operator is not a full implementation of the generic operation, but instead is a specialised implementation of said tuples, and takes into account the lack of the last row, adjusting the result accordingly.

- The Lift version does not perform the reduction over each thread in place - this is because such optimisation cannot be performed in general, and the Lift compiler is not able to determine in which case it can be applied.

- The OpenCL version did not use private and static local memory, the Lift version does. This is enabled by the fact that the Lift compiler performs loop unrolling and can, therefore, know statically exactly how much memory will be needed for each reduction. The handwritten OpenCL version does not perform unrolling, and therefore all memory sizes are only known at runtime.

- The OpenCL kernel carries out a reduction over matrices represented as a pair of floats, expecting the host to generate the appropriate layout. The Lift version expects a single linear array of values, which are then converted to matrices on the fly. Originally, the Lift version did the same as the OpenCL - however, it has been found out that the conversion can happen locally without any loss of performance: this is because the longer computation time of the kernel is balanced by the smaller memory consumption, which allows for more parallelism given the fixed buffer sizes. (The motivating reason behind this change is that when

coding the handwritten exploratory tests, the concern was to measure how much faster a parallel matrix multiplication reduction is compared to the equivalent sequential multiply-and-accumulate loop. On the other hand, in the Lift case, the optimisation must operate within the context of a language - and therefore the transformations must happen in place).

- The Lift version was significantly easier to write and test, even after having taken into consideration the experience gained from manually implementing a similar task in OpenCL. In particular, Lift's type system and principled pattern-based programming model eliminated the need for writing explicit array indexing calculations, which have been perceived as being by far the most complex and error prone part in writing an efficient OpenCL kernel.

7.3 Summary

This chapter has evaluated the performance of the optimisation added to the Lift compiler. This has been achieved through a series of comparisons within optimized and unoptimized code. In addition, a variety of different optimisation versions have been compared, validating the effect of the design decisions used for each one. Finally, through a general comparison with handwritten parallel implementation on other devices, we have verified that the optimised Lift code can outperform reasonable-quality OpenCL code running on equivalent platforms.

Chapter 8

Conclusions

In this document, we have presented an algorithm for the derivation of a tree-parallel implementation to sequential reductions with non-associative operators, with an associated implementation for the Lift compiler. In addition to the transformation, we have designed a series of addition to Lift to integrate the transformation with the rest of the language better, and we have provided implementations for some of these additions. Moreover, through a series of comparisons with unoptimised Lift code and equivalent handwritten implementations for other platforms, we have established the usefulness of the approach.

8.1 Contributions

The work presented here includes the following original contributions

- An algorithm for the transformation of sequential reductions into a parallel equivalent implementation in a functional setting, starting from the multiply-accumulate transformation into matrix multiplications presented in(Sato and Iwasaki, 2011).

- An extension of the basic transformation, using properties of semiring algebra to implement "asymmetrically typed" reductions through the use of higher-dimensional matrices.

- An algorithm for the identification of reductions which can be parallelised using the method proposed above, based on pattern-matching syntactic constraints over the source code, eliminating the need for more complex analyses presented in (Sato and Iwasaki, 2011)

- The sketch of a proof of correctness via rewrite rules for the proposed transformation.
- A concrete implementation of the transformation for the Lift compiler, resulting in the generation of an efficient implementation targeting OpenCL-compliant GPUs.
- The necessary compiler infrastructure integrating the transformation within the wider context of the Lift programming language.
- The design of a future expression language for Lift, allowing for a more natural and direct expression of Reduction operators.
- Several handwritten implementations of the efficient parallel matrix multiplication reduction, for both the CPU (OpenMP, pthreads) and the GPU(OpenCL), to be used as a baseline and a comparison with the Lift optimised implementation.

8.2 Critical Analysis

Although the principal objectives set out for the work have been achieved, in hindsight, some of the work might have been done differently.

The principal shortcomings are those pertaining to the handwritten OpenCL implementation of the parallel reduction. Initially, it was thought as a didactic exercise to familiarise the author with OpenCL and GPU programming (a necessary step for the implementation of a Lift optimisation, since OpenCL, is the main target of the compiler). Ultimately, however, the very same handwritten implementation was used as the main benchmark to beat. A better choice would have been trying to either find some third-party implementation known to be highly performant or to supplement the OpenCL handwritten implementation with some other alternative high-level parallel programming framework's.

Another problem, also pertaining to the handwritten OpenCL implementation, was the fact that the worksize parameters for the execution have been inferred by manual exploration. Here the usage of an autotuner might have been a better choice, as it is possible that the shortcomings in the handwritten OpenCL version are not due to poor code quality, but rather by the lack of proper worksize tuning.

Another possible critique can be found in the fact that the Lift implementation was only tested for GPUs, but not for the CPU. Extending the evaluation to the CPU section would allow for a better comparison across device types. This crucial for any work done on the Lift project, which values "performance portability" as one of its central principles.

A final critique can be levied for the fact that ultimately full integration within the Lift language was not completed. While this can be mostly traced to Lift's lack of an expression language (and providing one would have meant providing very profound changes to the current compiler), this also means that the work done here cannot be truly called "automatic parallelisation": the human must still explicitly instruct the compiler to generate a parallel implementation, and has to provide both explicitly the semiring instance and write the reduction operation in a slightly constrained style.

8.3 Further work

As the work presented in this document presents a novel approach to automatic parallelisation in a functional programming context, there are plenty of avenues for development.

Regarding the Lift implementation of the optimisation, the next future should see the implementation of the expression language and constraint verification systems as previously presented in this document. This will allow the full automation of the optimisation, which would then become truly transparent at a source code level.

Another welcome addition would be the implementation of a parallel scan primitive to the Lift compiler. The principles of the transformation presented in this document can almost verbatim be reused to parallelise scan, and the constraint checking part is completely identical. This would extend the applicability of the transformation to an even wider class of algorithms, such as the *LU* decomposition algorithm for linear algebra.

In a more general context, a possible research opportunity lies in attempting to extend the algorithm even further. In this document, we have simply covered one possible

transformation (linear functions to matrix multiplication). However, it might be possible to derive further generalisations, perhaps allowing for nonlinear functions to be expressed as some combination of matrix multiplication and other operations, trying to keep the derived arrangement ultimately associative - and hence still parallel.

A fascinating new possibility is to take the framework and approach used in this algorithm and re-purpose it to solve radically different problems. For example, it might be possible to use the functional programming, syntactic constraint and rewrite rules combination to address the issue of transparent heterogeneous programming.

While the details of this possible application go outside the scope of this further work section, the intuition is as follows: instead of considering this algorithm a form of automatic parallelisation, interpret it as code generation for an abstract platform whose programming model only support matrix multiplication. Constraint checking would then be equivalent to ensuring that the platform supports the program, the abstract transformation into matrix reduction form would be equivalent to a high-level representation of the program, and finally the generation of the optimised OpenCL kernel corresponds to target code generation.
In this light, therefore, the act of generating the parallel reduction from the algorithm description becomes equivalent to code generation for this peculiar platform. A similar approach might be used to generate code for very constrained accelerators such as DSPs and Vision Processing Unit, all in a transparent way.

Bibliography

Bik, A. J., Girkar, M., Grey, P. M., and Tian, X. (2002). Automatic intra-register vectorization for the intel® architecture. *International Journal of Parallel Programming*, 30(2):65–98.

Chafi, H., Sujeeth, A. K., Brown, K. J., Lee, H., Atreya, A. R., and Olukotun, K. (2011). A domain-specific approach to heterogeneous parallelism. *ACM SIGPLAN Notices*, 46(8):35–46.

Cole, M. I. (1989). *Algorithmic skeletons: structured management of parallel computation*. Pitman London.

Dagum, L. and Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55.

Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

Glaskowsky, P. N. (2009). NvidiaâĂŹs fermi: the first complete gpu computing architecture. *White paper*, 18.

Grosser, T., Groesslinger, A., and Lengauer, C. (2012). PollyâĂŤperforming polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010.

Henriksen, T., Elsman, M., and Oancea, C. E. (2014). Size slicing: a hybrid approach to size inference in futhark. In *Proceedings of the 3rd ACM SIGPLAN workshop on Functional high-performance computing*, pages 31–42. ACM.

Khronos Group (2016). Opencl - the open standard for parallel programming of heterogeneous systems.

Liu, D., Chen, T., Liu, S., Zhou, J., Zhou, S., Teman, O., Feng, X., Zhou, X., and Chen, Y. (2015). Pudiannao: A polyvalent machine learning accelerator. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 369–381. ACM.

Lomont, C. (2011). Introduction to intel advanced vector extensions. *Intel White Paper*, pages 1–21.

Murphy, N., Jones, T., Campanoni, S., and Mullins, R. (2015). Limits of dependence analysis for automatic parallelization. In *Proceedings of the 18th International Workshop on Compilers for Parallel Computing (CPC 2015, London*.

Nvidia, C. (2008). Cublas library. *NVIDIA Corporation, Santa Clara, California*, 15(27):31.

Nvidia, C. (2010). Programming guide.

Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. (2013). Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530.

Reinders, J. (2007). *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.".

Remmelg, T., Lutz, T., Steuwer, M., and Dubach, C. (2016). Performance portable gpu code generation for matrix multiplication. In *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit*, pages 22–31. ACM.

Sato, S. and Iwasaki, H. (2011). Automatic parallelization via matrix multiplication. In *ACM SIGPLAN Notices*, volume 46, pages 470–479. ACM.

Steuwer, M., Fensch, C., and Dubach, C. (2015). Patterns and rewrite rules for systematic code generation (from high-level functional patterns to high-performance opencl code). *arXiv preprint arXiv:1502.02389*.

Sujeeth, A., Lee, H., Brown, K., Rompf, T., Chafi, H., Wu, M., Atreya, A., Odersky, M., and Olukotun, K. (2011). Optiml: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 609–616.

Suriana, P., Adams, A., and Kamil, S. (2017). Parallel associative reductions in halide. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, pages 281–291. IEEE Press.

Tian, X., Bik, A., Girkar, M., Grey, P., Saito, H., and Su, E. (2002). Intel® openmp c++/fortran compiler for hyper-threading technology: Implementation and performance. *Intel Technology Journal*, 6(1).