SEAN YOUNGSUNG LEE

ACCELERATING HASKELL ARRAY CODES
WITH ALGORITHMIC SKELETONS ON GPUs

A thesis submitted in fulfilment
of the requirements for the degree of
Doctor of Philosophy

THE UNIVERSITY OF NEW SOUTH WALES

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

2011
GPUs have been gaining popularity as general purpose parallel processors that deliver a performance to cost ratio superior to that of CPUs. However, programming on GPUs has remained a specialised area, as it often requires significant knowledge about the GPU architecture and platform-specific parallelisation of the algorithms that are implemented. Furthermore, the dominant programming models on GPUs limit functional decomposition of programs, as they require programmers to write separate functions to run on GPUs.

I present and quantitatively evaluate a GPU programming system that provides a high-level abstraction to facilitate the use of GPUs for general purpose array processing. The presented programming system liberates programmers of low-level details and enables functional decomposition of programs, whilst providing the facilities to sufficiently exploit the parallel processing capability of GPUs. Fundamentally, the presented programming system allows programmers to focus on what to program on GPUs instead of how to program GPUs.

The approach is based on algorithmic skeletons mapped to higher-order functions, which are commonly available in functional programming languages. The presented programming system (1) encapsulates the low-level control of GPUs and the GPU-specific parallelisation of the higher-order functions in algorithmic skeletons, (2) employs an embedded domain specific language as the interface that treats the parallel operations as expressions with parallel semantics, allowing functional decomposition of programs, and (3) manages the compilation of the embedded domain specific language into algorithmic skeleton instances and the execution of the algorithmic skeleton instances online. Programmers only need to write the operations to be executed on GPUs as expressions in the embedded domain specific language, and the presented programming system takes care of the rest.

Although the concrete implementation presented in this thesis involves an embedded domain specific language in Haskell, Accelerate, and a specific GPU platform, CUDA, the approach in this thesis can be applied to other similar configurations with appropriate adjustments.
ORIGINALITY STATEMENT

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.'

Signed

Date Sep 21, 2011
ACCELERATING HASKELL ARRAY CODES
WITH ALGORITHMIC SKELETONS ON GPUS

SEAN YOUNGSUNG LEE

THE UNIVERSITY OF NEW SOUTH WALES
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Sydney, Australia, 2011

Copyright © 2011 by Sean Youngsung Lee

Typeset by the author using the \LaTeX \textsuperscript{2e} typesetting system.
Printed in Australia.
Acknowledgments

I am deeply indebted to my supervisor, Gabriele Keller. She immensely helped me accomplish this work with her technical insight, constant guidance, kind encouragement, and warm support. She even spent a great deal of time proofreading this thesis. I also thank my co-supervisor, Manuel Chakravarty. He has made tremendous technical contributions to this work. Along with Gabriele Keller, he taught me various aspects of functional programming, programming language design, and parallel programming in depth.

The former and current members of the Programming Language and Systems group at the University of New South Wales have helped me tremendously, and I cannot thank them enough. Roman Leshchinskiy offered me brilliant ideas and feedback with a great sense of humour, whenever possible; Sean Seefried generously shared parts of his office cubicle while I was waiting to be assigned with my own in the early years of my PhD candidature, and spared much of his time to teach me $\lambda$-calculus; Donald Stewart and Simon Winwood were always willing to answer my questions related to functional programming without any hesitation, and they set a good example as PhD students with their enthusiasm and insight in a wide range of computer science topics; Trevor McDonell has been a good friend and colleague since he joined the group, and he has made invaluable contributions to this work; and there are many others —including but not limited to Gabriel Ditu, André Pang, Mark Wotton, Sarah Webster— I must thank for making the group such a pleasant place for a PhD student like me.

I would like to extend my gratitude to the Compute Compiler team at NVIDIA. They offered me internship opportunities, and eventually full-time employment, through which I was able to acquire better understanding into GPU programming and compiler engineering. Especially, Vinod Grover, my mentor and manager, never ceased to amaze me with wonderful ideas, and guided me with great wisdom and support.

Finally, I am forever grateful to my wife, Ami Woo, for her love, patience, and support, and to the rest of my family for their encouragement and prayer.

Portions of this thesis are based on the collaborative work published in [1, 2],
and acknowledgement is due to Manuel Chakravarty, Gabriele Keller, Trevor McDonell, and Vinod Grover for their contributions.

Seattle, WA
Sean Youngsung Lee
Abstract

GPUs have been gaining popularity as general purpose parallel processors that deliver a performance to cost ratio superior to that of CPUs. However, programming on GPUs has remained a specialised area, as it often requires significant knowledge about the GPU architecture and platform-specific parallelisation of the algorithms that are implemented. Furthermore, the dominant programming models on GPUs limit functional decomposition of programs, as they require programmers to write separate functions to run on GPUs.

I present and quantitatively evaluate a GPU programming system that provides a high-level abstraction to facilitate the use of GPUs for general purpose array processing. The presented programming system liberates programmers of low-level details and enables functional decomposition of programs, whilst providing the facilities to sufficiently exploit the parallel processing capability of GPUs. Fundamentally, the presented programming system allows programmers to focus on what to program on GPUs instead of how to program GPUs.

The approach is based on algorithmic skeletons mapped to higher-order functions, which are commonly available in functional programming languages. The presented programming system (1) encapsulates the low-level control of GPUs and the GPU-specific parallelisation of the higher-order functions in algorithmic skeletons, (2) employs an embedded domain specific language as the interface that treats the parallel operations as expressions with parallel semantics, allowing functional decomposition of programs, and (3) manages the compilation of the embedded domain specific language into algorithmic skeleton instances and the execution of the algorithmic skeleton instances online. Programmers only need to write the operations to be executed on GPUs as expressions in the embedded domain specific language, and the presented programming system takes care of the rest.

Although the concrete implementation presented in the thesis involves an embedded domain specific language in Haskell, Accelerate, and a specific GPU platform, CUDA, the approach in this thesis can be applied to other similar configurations with appropriate adjustments.
Contents

1 Introduction 1

2 Background 5

2.1 History of GPGPU .................. 5
2.2 CUDA .......................... 8
2.2.1 Hardware Overview .............. 8
2.2.2 Programming in CUDA ............ 9
2.2.3 Kernel Execution Control ....... 11
2.2.4 Memory Access .................. 12
2.3 Why Another Parallel Programming System for GPGPU? ...... 13
2.4 Skeleton-based Parallel Programming .............. 14

3 Accelerate 19

3.1 EDSL in Haskell .................. 20
3.2 Accelerate Language Constructs ... 21
3.2.1 Data Representation ............. 23
3.2.2 Type Reification and Type Dictionaries .......... 33
3.2.3 Collective Array Operations ...... 36
3.2.4 Scalar Operations ............... 42
3.3 Conversion to the De Bruijn Representation .......... 48
3.3.1 De Bruijn Indices and Environments ........ 48
3.3.2 Collective Array Operations ...... 50
3.3.3 Scalar Operations ............... 55
# Contents

## 4 CUDA Back End

4.1 Compilation Scheme ............................................. 63
4.2 Minimising the Runtime Overhead .............................. 67
  4.2.1 Data Transfers ........................................... 68
  4.2.2 Compilations of the Device Code ............................ 70
  4.2.3 Overlapping Data Transfers, Compilations, and Executions 73
  4.2.4 Caching of the Compiled Modules ............................ 76
4.3 Code Generation .................................................. 78
  4.3.1 Collective Array Operations ................................. 79
  4.3.2 Scalar Operations ......................................... 89

## 5 Quantitative Analysis and Evaluation

5.1 Environment ...................................................... 97
5.2 Applications .................................................... 98
  5.2.1 SAXPY, SSCAL, SASUM, SNRM2, SDOT ......................... 98
  5.2.2 Exclusive Prefix Sum ....................................... 99
  5.2.3 Outer Product ............................................. 100
  5.2.4 Sparse Matrix and Vector Multiplication ..................... 100
  5.2.5 Black-Scholes Options Pricing ............................. 101
5.3 Performance of Generated Device Code ......................... 102
  5.3.1 Applications with One Simple Array Operation ............... 102
  5.3.2 Applications with Multiple Simple Array Operations ........ 104
  5.3.3 Applications with One Complex Array Operation ............ 109
  5.3.4 Summary ................................................... 112
5.4 The Overall Performance and the Effect of Caching of Compiled  
  Modules .......................................................... 113
5.5 Summary .......................................................... 118

## 6 Related Work

6.1 Characteristics of Related GPU Programming Systems .......... 121
6.2 Haskell-based GPU Programming Systems ........................ 123
  6.2.1 Nikola .................................................... 123
7.2 Future Research ............................................................ 143
  7.2.1 Multi-GPU Support ..................................................... 143
  7.2.2 Optimisations .......................................................... 143
  7.2.3 Efficient Hashing of Computation ASTs ......................... 145
  7.2.4 Porting to Other Parallel Platforms ............................ 146
  7.2.5 Data Parallel Haskell and Nested Data Parallelism .......... 146

A Outer Product in CUDA .................................................... 147

B Manual Optimisations to Black-Scholes Options Pricing .......... 151

Bibliography .................................................................... 161
List of Figures

2.1 (a) fixed-function pipeline [3], (b) programmable pipeline with vertex shader programs and fragment shader programs [3, 4], and (c) programmable pipeline with unified shader programs ............... 6

2.2 Modern GPU architecture .............................................. 10

2.3 (a) SAXPY written in Accelerate and (b) the device code from the corresponding instance of `zipWith` skeleton ................. 18

3.1 An example of multi-dimension array of type `Array (Int, Int, Int) Float` .......................................................... 28

3.2 Slices of the cube in Figure 3.1 with various slice descriptors ........ 29

3.3 The relationship between the reified data types in Accelerate ........ 34

3.4 The Definition of Reified Dictionaries and Reified Types in Accelerate 35

3.5 Some of the smart constructors that Accelerate provides for the computation AST construction in the HOAS representation .... 37

3.6 The Accelerate collective array operations in the HOAS representa-
tion [5] .............................................................................. 38

3.6 The Accelerate collective array operations in the HOAS representa-
tion (continued) [5] ............................................................ 39

3.7 The Accelerate scalar expressions in the HOAS representation [5] 44

3.8 The Accelerate collective array operations in the De Bruijn repre-
sentation [5] ...................................................................... 51

3.8 The Accelerate collective array operations in the De Bruijn repre-
sentation (continued) [5] .................................................... 52
3.9 The Accelerate scalar expressions in the De Bruijn representation [5] 58

4.1 High-level overall structure of Accelerate ............................ 62
4.2 run in the CUDA back end ............................................. 64
4.3 (a) dot-product written in Accelerate, and (b) its AST representation 66
4.4 The data structure that keeps track of the device memory uses ... 68
4.5 The data structure that keeps track of the compiled modules ..... 71
4.6 The timeline of data transfers, code generation, compilation, and execution for the dot-product in Figure 4.3 ......................... 75
4.7 The zipWith skeleton in the CUDA back end [5] ...................... 81
4.8 Reading an array element and updating an array element, using the getn and set functions ........................................... 83
4.9 The code generation rules: the translation of the supported types 85
4.10 The selection and instantiation of the algorithmic skeletons ..... 88
4.11 The code generation rules: the translation of the scalar expressions 95

5.1 The runtime cost of SAXPY ........................................... 103
5.2 The runtime cost of SSCAL ............................................ 103
5.3 The runtime cost of exclusive prefix sum ............................ 104
5.4 The runtime cost of SASUM ........................................... 106
5.5 The runtime cost of SNRM2 .......................................... 106
5.6 The runtime cost of SDOT ............................................. 108
5.7 The runtime cost of outer product ................................... 109
5.8 The runtime cost of sparse matrix and vector multiplication ... 110
5.9 The runtime cost of Black-Scholes options pricing ................. 111
5.10 The overall performance of the benchmark applications and the reference implementations ........................................ 114
5.10 The overall performance of the benchmark applications and the reference implementations (continued) ...................... 115
5.11 The overall runtime cost increase of the Accelerate implementation of Black-Scholes options pricing by the number of options .... 117
List of Tables

3.1 Slice, CoSlice, and SliceDim computed with the slice descriptors from Figure 3.2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
3.2 The higher-order collective array operations in Accelerate . . . . . 41

5.1 The system configuration for the benchmarks . . . . . . . . . . . 98
5.2 The detailed runtime (µsec) of SASUM and SNRM2 . . . . . . . . 105

B.1 The comparison of the number of instructions and the number of divergent branches . . . . . . . . . . . . . . . . . . . . . . . . . . . . 152
Chapter 1

Introduction

In the foreseeable future, performance increase will only come through parallelism due to the paradigm shift in hardware development. This fundamentally changes the situation for all software development, as it imposes problems such as non-deterministic execution of parallel threads, race conditions, communication among multiple threads, synchronisation, and parallelisation of existing algorithms on programmers. Programmers are forced to spend more time and resources on figuring out how to program the hardware than what to program on the hardware.

Given that performance and correctness are two of the most important aspects in computing, hardware manufacturers and software engineers have put a great deal of effort into improving performance whilst preserving correctness of computation. As a part of the endeavour, Central Processing Unit (CPU) manufacturers realised the performance gain by increasing the clock speed on a single core until the early 2000s. The speed-ups brought by a higher clock frequency were almost free from the application programmers’ point of view. Then it became physically impractical to increase the clock frequency due to the heat and the power consumption, and CPU manufacturers turned their attention to multicore architectures. Consequently, the parallelism to exploit the multiple cores has become the key to performance gain.

Another noticeable trend to achieve massive performance gain on commodity
desktop computers has developed since the early 2000s: General Purpose Computation on Graphics Hardware (GPGPU). The architectural changes made to Graphics Processing Units (GPUs) in the early 2000s suggested possible uses of GPUs as co-processing units to CPUs on commodity desktop computers. GPUs, which were non-programmable and only usable for graphics rendering, have become able to process general purpose computation with programmability and flexibility. Furthermore, the performance to cost ratio of GPUs, which is exhibited by their massively parallel architecture, has also become greater than that of CPUs.

I focus on the GPU architecture and its use for general purpose computation in this thesis. However, the approach and the methodology I present are applicable to other architectures, such as multi-core CPUs and FPGAs, with little modification, and Svensson and Newton recently demonstrated the portability of this approach on multi-core CPUs [7].

The main problems I tackle in this thesis are as follows:

- The GPU architecture has an eccentric memory hierarchy, a specialised thread management, and a feature set that differs significantly from that of general purpose processing units such as CPUs. Existing programming models require programmers to possess significant knowledge about the GPU architecture and about parallelism at low-level to sufficiently exploit performance gain on GPUs.

- Existing programming models require programmers to write separate functions within a program, depending on whether they are to be executed on GPUs or CPUs. Consequently, they limit functional program decomposition.

I aim to alleviate the process to achieve performance gain close to maximum on GPUs by liberating programmers from low-level details and by allowing functional program decomposition. To this end, I use algorithmic skeletons [X] — or skeletons in short — that implement the semantics of the collective array operations in functional programming languages.
I also explore how the collective array operations written in Accelerate — a Haskell embedded domain specific language for describing the collective array operations— are compiled to algorithmic skeletons to show that we can write GPU programs succinctly with a high-level abstraction, yet they can be executed efficiently on GPUs with a well-designed compilation and execution scheme.

The main contributions of this thesis are:

• I show how skeleton-based parallel programming can be utilised in GPU programming to hide low-level details and how it allows programmers to focus on the arithmetic operations, instead of the hardware specific issues such as thread manipulation, memory management, synchronisations, etc.

• I present a set of skeletons that parallelise the collective array operations commonly found in functional programming languages such as Haskell and OCaml on GPUs. As each of these skeletons can be treated as an expression with parallel semantics at the application level, they provide the infrastructure to enable functional program decomposition.

• I present the online compilation and execution scheme for the collective array operations on GPUs. The compilation and execution scheme includes the skeleton-based code generation, the efficient execution configuration, and the techniques to minimise the runtime overhead introduced by the hardware constraints and the online compilation.

• I quantitatively analyse the use of skeleton-based parallelism in GPU programming. Although skeleton-based parallelism is well-known in parallel computing, and has been evaluated on many parallel architectures, it has not been evaluated on GPUs. I present a formal evaluation of skeleton-based parallelism on GPUs by comparing the GPU code generated with skeletons against the hand-optimised GPU code in terms of performance, and show that it is possible to achieve performance close to the optimal on GPUs with skeleton-based parallel programming.
Chapter 2 illustrates the history of GPGPU. The architectural characteristics of GPUs as of the year 2010 are introduced along with the existing GPU programming models and systems for them. This chapter also identifies the issues with the existing GPU programming systems and reasons about the need for another GPU programming system with a high-level abstraction more in detail.

Chapter 3 presents the details of the array language and its embedding in Haskell. I start this chapter with a general overview of embedded domain specific languages in Haskell, and I explain the Accelerate language constructs, the data structures that we use for the GPU subprograms written in Accelerate, and how we transform the GPU subprograms into a form which is better suited for the skeleton-based GPU code generation than the original form.

Chapter 4 elaborates on the overview of the skeleton-based approach in GPU programming, what is required to generate GPU code whose performance is close to the performance of hand-optimised GPU code, and how these properties are incorporated into skeletons. It also presents the execution scheme of the generated GPU code modules and the strategies that the back end employs to minimise the runtime overhead incurred by the online compilation.

Chapter 5 quantitatively evaluates the skeleton-based parallelism on GPUs, and validates its practicability and usability. It also identifies room for improvement on this approach.

Chapter 6 discusses related work in GPGPU programming. It compares Accelerate to other approaches based on Haskell, C++, and Python, as well as other approaches based on C#, F#, and Matlab. Some of them are, like Accelerate, implemented as EDSLs, and others as directive-based programming systems or as libraries.

Chapter 7 provides the conclusion with a discussion of future work.

It is to be noted that this thesis and its implementation exploit the language features in Haskell such as type classes [9][11], type families [12][13], etc. I suggest reading the cited references before the rest of the thesis to the readers who are not sufficiently familiar with Haskell.
Chapter 2

Background

In this chapter, I provide the background knowledge required to understand the approach of the GPGPU programming system that I present throughout this thesis. I give the overview of the history of GPGPU to explain how GPUs have become relevant to general purpose computation, in particular, parallel processing. Furthermore, I describe the modern GPU architecture to reason about the challenges in programming GPUs for general purpose computation that are primarily imposed by hardware constraints. Then I outline the approach that I implement to tackle the challenges, in which I employ a functional programming language as the means to express parallel operations on GPUs, and generate and execute the GPU code using a set of pre-defined parallelism patterns —or algorithmic skeletons. I defer the detailed review on related work to Chapter 6.

2.1 History of GPGPU

It was not until the early 2000s that the idea of using graphics hardware for general purpose computing took off. This was because graphics hardware at that time was meant for graphics applications only, and therefore non-programmable, equipped only with the fixed-function pipeline. Figure 2.1 (a) shows the most common configuration of the fixed-function pipeline. Using raw geometry data as input, the fixed-function pipeline applied a number of transformations to the
The evolution of graphics hardware between the early 2000s and the mid 2000s resulted in the dominance of programmable graphics hardware [4] [14]. The geometry transformation unit and the lighting unit of the fixed-function pipeline were replaced by programmable vertex processors. Similarly, the texture application unit and the pixel drawing unit were replaced by programmable fragment processors [3]. Figure 2.1 (b) shows the programmable pipeline from this period.

The programmability of the graphics pipeline during this period, however, was very limited. In fact, the enhancement of the programmability was to support custom vertex shaders and fragment shaders, thus, the programmability was centred around the graphics processing. Shader languages were used to program shaders, and they accepted input data in the form of textures, and produced
output data in the form of position values or fragment colour values, depending on the shader type. GLSL [15], Cg [16], and HLSL [17] are the notable shader languages which are still in use for graphics applications.

Even with the limited programmability, the idea of using graphics hardware for general purpose computing seemed attractive, and the implementations of non-graphics applications—database operations [18], Fast Fourier Transform [19], ray tracing [1], etc.—on graphics hardware started appearing. These applications proved that GPUs could demonstrate a computational performance superior to the performance of CPUs, especially with floating-point operations, when the parallelism among shader processors was efficiently exploited; NV40, NVIDIA’s GPU architecture in GeForce 6 products released in 2004 – 2005, was armed with 6 vertex processors and 16 fragment processors whilst single core architecture was still dominant in CPUs.

GPU programming, however, was inaccessible to programmers without experience in graphics processing, as the shader languages are graphics-centric and GPU programming required re-interpretation of general purpose operations in the context of graphics. For example, the execution of shader programs was controlled by drawing functions in the graphics framework associated with the shader language, which looked nothing like thread spawning in other general purpose programming environments.

The potential and challenges of GPGPU stimulated researches in general purpose programming model on GPUs, and Brook for GPU [20] and Sh [21, 22] were released to provide a general purpose programming layer on top of the shader languages. Chapter 6 discusses them in detail.

GPU manufacturers started paying attention to this trend, and NVIDIA was first to respond with a general purpose programming system on GPUs, Compute Unified Device Architecture (CUDA). They initially released CUDA in early 2007. The evolution of GPUs and CUDA technology accelerated, and GPUs sported the peak performance in Floating Point Operations per Second (FLOPS) about 10 times higher than the peak performance of CPUs as of June 2008, and the performance discrepancy is even broadening [23]. The graphics pipeline also changed.
Vertex processors and fragment processors were replaced by unified shader processors which are capable of both vertex processing and fragment processing. Figure 2.1 (c) shows this change.

In addition to CUDA, Apple proposed Open Computing Language (OpenCL) [24] as an open standard for GPGPU programming, in collaboration with AMD, IBM, Intel and NVIDIA, and Khronos Group, and OpenCL was approved as an open standard in 2008. OpenCL aims to provide a heterogeneous parallel programming system on CPUs and GPUs. AMD and NVIDIA made the initial release on their architectures in 2009.

GPGPU, as of 2011, is gaining more popularity as CUDA and OpenCL have liberated programmers from the burden to understand the graphics pipeline and re-interpret their computations in the context of graphics. GPGPU applications demonstrate up-to 2600-times speed up in comparison to CPUs [25].

2.2 CUDA

I have chosen NVIDIA CUDA as the target platform because it was the only general purpose GPU programming environment, which did not depend on shader programming, when this work was initiated in 2005. OpenCL, which was released in 2009, could also be used as the target platform of the approach and methods that I present to achieve similar outcomes, as it shares numerous traits with CUDA.

2.2.1 Hardware Overview

GPUs apply the same vertex shader or the same fragment shader to each vertex or each fragment respectively. Simply put, this is a single program, multiple data (SPMD) stream processing. Although vertex shader programs and fragment shader programs are programmed differently as they are executed at different stages in the graphics pipeline, they run on an array of unified shader processors with the same set of capabilities. The programmability of these unified shader
processors is general enough to accommodate arithmetic operations used in non-
graphics applications. In fact, graphics processing is one of the areas where
sophisticated arithmetic operations are required.

Modern GPUs are equipped with hundreds of unified shader processors, and
these processors are organised in a similar way that arithmetic clusters are organ-
ised on stream processors such as Imagine \cite{26,27} and Merrimac \cite{28}. Figure 2.2
shows the structure of recent NVIDIA GPUs. They are equipped with a number
of streaming multiprocessors (SMs), and each SM consists of a fixed number of
scalar processors (SPs), which are unified shader processors in graphics. For in-
stance, GeForce GTX 285 comes with 30 SMs and 240 SPs in total. Furnished
with a large number of programmable cores and the SPMD nature, GPUs are
recognised as many-core parallel processors. Even though the performance of a
GPU core is inferior to a CPU core, GPUs outperform CPUs when the parallelism
is exploited efficiently.

A number of registers are available on each SM as well as a fixed amount of
shared memory which is accessible from all SPs in the SM. Every SP on a GPU
has access to the global memory, and the global memory is where data exchanges
with the host memory are performed; parameters to GPU operations are mapped
to global memory and results from GPU operations are retrieved from global
memory. As the primary use of GPUs is graphics, GPUs also have read-only
texture memory and constant memory. The texture memory and the constant
memory are cached whereas the global memory is not.

2.2.2 Programming in CUDA

A CUDA application has two components: device code and host code. Functions
in device code are categorised into global functions and device functions. A global
function and its associated device functions form a kernel, which means a subpro-
gram that runs on each SP. Global functions can be invoked only from host code,
and device functions only from device code. Neither global functions nor device
functions can invoke functions in host code. Host code controls kernel launches
in the form of global function invocations, and it manages memory operations such as dynamic memory allocations and data transfers between host memory and GPU memory.

CUDA allows programming at two levels: *C for CUDA* and *CUDA Driver API*. *C for CUDA* provides a C-like interface with the support for many C++-style features to programmers. The CUDA compiler shipped by NVIDIA, *nvcc*, is used to compile host code and device code written in C for CUDA. The CUDA Driver API gives lower-level control to programmers, compared to *C for CUDA*. When programming in the CUDA Driver API, host code is written in C++ and compiled by the native C++ compiler.

Programming in the CUDA Driver API is generally considered to be more difficult than in *C for CUDA*. Housekeeping tasks, which are automatically taken care of by the runtime library in *C for CUDA*, are required to be conducted manually by programmers. Nonetheless, as the CUDA Driver API comes with features unavailable in *C for CUDA*, it is appealing to those who are already familiar
with CUDA; the CUDA Driver API supports *Just-In-Time compilation* (JIT) of
device code, dynamic loading/unloading of compiled device code modules, and
explicit control of CUDA context.

Regardless of whether programming in C for CUDA or in the CUDA Driver
API, programmers are required to understand the hardware architecture and the
mapping between the hardware and the software model in order to achieve the
maximum performance gain.

### 2.2.3 Kernel Execution Control

NVIDIA GPUs with compute capability 1.x and compute capability 2.x have 8
SPs per SM, and each SM is capable of running 32 threads in parallel at any
moment without time sharing. This implicit group of threads is called a *warp*.
When device code is executed, *thread blocks* are mapped onto SMs. As of compute
capability 2.x, up to 8 thread blocks can be resident and only up to one thread
block is active on each SM at any moment. The number of threads per block
is determined when the kernel is launched. When there are multiple warps in
a thread block, the scheduler swaps warps in and out to hide memory access
latency if possible. The group of the thread blocks launched for the same kernel
execution is called a *grid*.

With this hierarchy, there are a few points to keep in mind in order to max-
imise the throughput.

- Threads in a warp must not diverge, or the execution of branches within
  the warp is serialised and a good portion of parallelism may be lost.

- The SM *occupancy* (= the number of active warps per SM / the maxi-
mum number of active warps per SM) must be kept as high as possible to
  maximise overlap between warps that must wait and warps that can run.

- The use of registers must be kept as low as possible. Local memory with high
  latency is used when register spilling occurs, thus affecting the performance.
The SM occupancy does not always guarantee the optimal performance. However, it is considered as a heuristic that can be used to determine the optimal execution configuration. NVIDIA ships an occupancy calculator along with CUDA SDK.

2.2.4 Memory Access

Arithmetic operations are performed almost instantly (up to 8 operations per clock cycles per SM with compute capability 1.x and up to 32 operations per clock cycles per SM with compute capability 2.x), however memory access incurs different levels of latency depending on the memory type. Access to global memory, which is off-chip, incurs high latency up to 800 clock cycles, and it is not cached unlike texture memory access or constant memory access. Access to shared memory, which is on-chip, can be as fast as access to registers if the access pattern is configured optimally as specified in the CUDA Programming Guide [23].

As a result, global memory access creates a bottleneck in many applications, and shared memory is often used as software-managed cache to minimise global memory access [29]. This technique typically requires, in sequence, (1) fetching of input data from global memory into shared memory, (2) synchronisation of threads in each block, (3) operations performed on shared memory, (4) another synchronisation of threads, and (5) storing output data back to global memory. This may introduce additional instructions. However, the cost of these additional instructions is much less than than the latency of hundreds of clock cycles which may be incurred by redundant global memory access.

It is also important to satisfy the coalescing and the alignment requirements in global memory access, and to maintain shared memory access free from bank conflicts. Although the global memory coalescing rules and the shared memory bank conflict rules have been simplified with each new CUDA release, they are still important performance considerations. A detailed description can be found in CUDA Programming Guide [23]. Global memory access is serialised when it
is non-coalesced or misaligned, resulting in performance degradation caused by high latency. Shared memory access with bank conflicts is also serialised.

2.3 Why Another Parallel Programming System for GPGPU?

Although it is evident that GPGPU promises notable performance gain at a low hardware cost compared to other high-performance systems, this benefit comes with significant overhead in the software development due to the complexity imposed by the algorithm parallelisation and the characteristics of the underlying hardware.

Application programmers unfamiliar with the underlying hardware —GPUs in our case— are required to invest a great deal of effort into learning about its characteristics and the constraints derived from them. This process costs time and resources. Consequently, the development and maintenance cost rises, affecting the total production cost.

CUDA and OpenCL, two representative general purpose programming systems on modern GPUs, are C extensions that provide low-level access to the hardware. This approach may be preferable to other high-level approaches for the programmers who have already acquired necessary knowledge about the hardware manipulation and require as much control as possible on the hardware, but it introduces significant overheads for those who wish to focus on the computation rather than on the hardware control and manipulation, as the performance gain may vary depending on the programmers’ ability to manipulate the hardware features.

Control of side effects is another issue. Pointers and side effects play a central role in C, but side effects need to be tightly controlled in parallel programming environments. Defining a global state and maintaining it in parallel programming environments require a great deal of attention, or the state is altered unexpectedly.
and the correctness of the computation is lost. When programming in these C-like GPU programming systems, it is the programmers’ responsibility to control side effects tightly.

Furthermore, program decomposition in these programming systems is not based on functionalities but on architectural constraints.

To alleviate these problems, a GPGPU programming system is required, which provides a high-level abstraction without losing much performance gain, isolates side effects from the rest of the application, and ensures functional program decomposition.

Hence I choose to employ a purely functional language as the front end, which is furnished with collective array operations that have a well-known interpretation, liberating programmers from the architectural constraints and the side effects control. In addition, collective array operations exist as expressions in the front end, thus allowing the embedding of parallel operations anywhere in the application and ensuring functional program decomposition.

To execute the collective array operations from the front end on GPUs, I implement a compiler back end that generates and executes the CUDA kernels at runtime that are mapped to the operations from the front end. Architectural constraints and other restrictions are internally controlled in the back end, and not exposed to programmers.

2.4 Skeleton-based Parallel Programming

The CUDA back end generates the device code modules, following the approach of skeleton-based parallel programming, which is a technique well established in parallel computing. Computations are expressed as a collection of parallelism patterns, which are called algorithmic skeletons. It was first introduced by Cole in 1989 [8], and has been adopted by a number of parallel computing frameworks [30]. The examples include, but not limited to, Eden [31] and HDC [32], Haskell-based skeleton frameworks on the Parallel Virtual Machine (PVM) [33], and Message Passing Interface (MPI) [34], Lithium [35] [36] and Muskel [37] [38],
Java-based skeleton frameworks on Java Remote Method Invocation (RMI) [39], and Muesli [40, 41], a C++-based skeleton framework on MPI and OpenMP [42].

Algorithmic skeleton frameworks provide high-level abstractions to application programmers by encapsulating low-level details, such as data exchanges among processing elements (PEs) and synchronisations, into a set of parallelism patterns. They alleviate the development process by liberating application programmers from the low-level coordination management, which can be error-prone and often requires more attention than the implementation of the actual computation of interest.

The following example demonstrates the benefits of skeleton-based parallel programming effectively.

```
foldl f z [x₁, x₂, x₃, ..., xₙ]
```

*foldl* is a Haskell operation that reduces the given list to a value by applying a binary function, *f*, to the identity value, *z*, and the list elements, [*x₁*, *x₂*, *x₃*, ..., *xₙ*]. When sequentially executed, this operation evaluates to

```
f ( ... (f (f (f z x₁) x₂) x₃) ...) xₙ
```

When executed in parallel, this operation works as parallel reduction which requires one or more synchronisations depending on the size of the input list and the number of elements evaluated per PE.

If this operation were to be implemented with a low-level or no abstraction, the parallel reduction tree would have to be built explicitly, and the evaluation would have to be described with the correct co-ordination of synchronisations and the correct re-direction of the intermediate results from a phase to another. However, with the *foldl* operation pre-defined as an algorithmic skeleton, application programmers are liberated from dealing with these error-prone tasks and allowed to concentrate on the actual computations.

*filter* is another example, which can demonstrate the benefits of skeleton-based parallel programming.
The operation above returns a list of the elements which are from the input list, \( xs \), and satisfy the predicate, \( p \).

When implemented for a parallel environment, this operation is split into three parallel sub-operations: (1) a map operation that applies the predicate, \( p \), to each element of the list, \( xs \), and produces the list of flags (0 or 1), (2) a left scan operation that runs on the list of flags and produces the list of destination indices, and (3) a permute operation that moves the elements using the list of destination indices and the list of flags.

This operation requires pipelining of the intermediate results between the sub-operations and between phases within each sub-operation as well as the tight control of synchronisations. Using skeleton-based parallel programming, these requirements can be simplified, and this operation can be implemented as a composition of skeletons; many algorithmic skeleton frameworks provide facilities for pipelining.

Collective array (or list) operations — filter, foldl, map, permute, scanl, etc.— commonly found in functional programming fit into data-parallel algorithmic skeletons adequately. As the examples above demonstrate, these operations specify the evaluation patterns, in which the given operation or predicate is applied to the elements of the input lists.

I pre-define data-parallel algorithmic skeletons in CUDA for a set of collective array operations found in Haskell, and employ Accelerate, a Haskell EDSL, as the front end. The front end provides facilities to build abstract syntax trees (ASTs) that carry information such as the set of requested operations, the dependency among those operations, and the pipelining of the intermediate results. The requested operations are compiled to instances of CUDA skeletons and the necessary CPU-to-GPU data transfers are triggered while the ASTs are traversed.

Figure 2.3 shows (a) Scalar alpha X Plus Y (SAXPY) written in Accelerate and (b) the device code from the corresponding instance of the zipWith skeleton. Given 1.5 as the \( \alpha \) value, the scalar function, \( \lambda \, x \, y \rightarrow \text{constant} \, 1.5 \times x \).
+ y, passed to \texttt{zipWith} in Figure 2.3 (a) is translated to the semantically equivalent CUDA function, \texttt{apply}, and combined with the pre-defined skeleton. The definition of the \texttt{zipWith} skeleton is in \texttt{zipWith.inl} and it is instantiated with the \texttt{apply} function and other auxiliary functions generated during the compilation; the skeleton code remains untouched, and the contents of \texttt{zipWith.inl} is included in Figure 4.7 with a detailed explanation in Chapter 4. Chapter 5 contains more examples written in Accelerate.

The CPU-to-GPU data transfers and the device code execution are controlled by the CUDA back end through Haskell’s CUDA driver API binding. The technical details are explained in Chapter 4.
let xs' = use xs :: Acc (Vector Float)
ys' = use ys :: Acc (Vector Float)
f = \lambda x y \to constant 1.5 * x + y
in zipWith f xs' ys' :: Acc (Vector Float)

#include <accelerate_cuda_extras.h>
typedef float TyOut;
typedef float * ArrOut;
static inline __device__ void set(ArrOut d_out, const Ix idx, const TyOut val)
{
    d_out[idx] = val;
}
typedef float TyIn1;
typedef float * ArrIn1;
static inline __device__ TyIn1 get1(const ArrIn1 d_in1, const Ix idx)
{
    TyIn1 x = d_in1[idx];
    return x;
}
typedef float TyIn0;
typedef float * ArrIn0;
static inline __device__ TyIn0 get0(const ArrIn0 d_in0, const Ix idx)
{
    TyIn0 x = d_in0[idx];
    return x;
}
static inline __device__ TyOut apply(const TyIn1 x1, const TyIn0 x0)
{
    TyOut r = (float) 1.5 * x1 + x0;
    return r;
}
typedef DIM1 DimOut;
typedef DIM1 DimIn1;
typedef DIM1 DimIn0;
#include <zipWith.inl>

Figure 2.3: (a) SAXPY written in Accelerate and (b) the device code from the corresponding instance of zipWith skeleton
Chapter 3

Accelerate

Accelerate is the front end EDSL that I employ to express the collective array computations. The Accelerate language provides a means to describe collective array operations on parallel hardware platforms such as *Graphics Processing Units* (GPUs) [1] and multi-core *Central Processing Units* (CPUs) [7], by employing the front end to build computation *abstract syntax trees* (ASTs), which are fed into the back end of choice for further processing.

Accelerate has three important properties, which make it the ideal EDSL for algorithmic skeleton-based code generation and execution:

- The computation ASTs in Accelerate maintain full type information to ensure type correctness.

- The internal representation of Accelerate ASTs is designed to be compiler-friendly for code generation and optimisation, whilst the user level representation is designed to be human-friendly.

- The language constructs in Accelerate are centred around higher-order operations, which have well-known data-parallel interpretation [43].

We will explore algorithmic skeleton-based code generation and execution in Chapter 4.

Portions of this chapter are based on the collaborative work published in [1, 2].
In this chapter, I introduce and explain Accelerate —both the language and the front end— focusing on the subset that is supported by the CUDA back end, and I describe how Accelerate is utilised as the interface between the algorithmic skeletons in the CUDA back end and the host language, Haskell.

### 3.1 EDSL in Haskell

There are two ways to embed a domain specific language in Haskell: *deep embedding* and *shallow embedding*. Domain specific languages (DSLs) with deep embedding construct the computation ASTs, and DSLs with shallow embedding define the language constructs with functions of the particular types that represent the meaning of the computation in Haskell. The following example illustrates the difference between the two approaches.

```haskell
data Op where
    Const :: Int → Op
    Add :: (Op, Op) → Op
    Mul :: (Op, Op) → Op

run :: Op → Int
run = eval
    where
        eval :: Op → Int
        eval (Const c) = c
        eval (Add (o1, o2)) = eval o1 + eval o2
        eval (Mul (o1, o2)) = eval o1 * eval o2
```

The data type `Op` and the function `run` together define a simple domain specific language with deep embedding for additive and multiplicative operations on integers. The term `op = Add (Mul (Const 2, Const 4), Const 1)` represents an AST, `op`, for the arithmetic expression \(2 \times 4 + 1\), and `run op` produces 9 as the result.

The same language can be defined with shallow embedding as follows, and `add (mul (2, 4), 1)` evaluates to \(2 \times 4 + 1\) and produces 9 as the result:
Chapter 3. Accelerate

```haskell
  type Args = (Int, Int)
  type Op = Args -> Int

  add :: Op
  add (a1, a2) = a1 + a2

  mul :: Op
  mul (a1, a2) = a1 * a2
```

Shallow embedding is easier to implement, and facilitates expanding the language. However, with deep embedding it is easier to implement various program transformations, optimisations, and analyses than with shallow embedding, as the programs written in domain specific languages with deep embedding exist in the form of computation ASTs. Domain specific languages with deep embedding usually require a run function, which accepts the computation AST written in the language, and produces the result of the computation.

As we want to be able to produce code for various platforms, we chose deep embedding for Accelerate. The CUDA back end, which will be explained in Chapter 4, takes the computation AST, generates the CUDA code for it, executes the CUDA code on GPUs, and returns the result. The run function in the CUDA back end —or CUDA.run in short— plays the role of the interface between the Accelerate front end and the CUDA back end.

### 3.2 Accelerate Language Constructs

Accelerate is a two-level language, consisting of collective array operations and scalar operations. Collective array operations are based on the higher-order operations in Haskell, and are parameterised with one or more scalar operations. These operations include Replicate, Map, ZipWith, Fold, FoldSeg, Scan, Permute, and Backpermute. Collective array operations define the pattern in which the given scalar operations are applied, and scalar operations define the computation on each element of the array. Sections [3.2.3](#) and [3.2.4](#) provide the detailed explanation on the collective array operations and the scalar operations in Accelerate. As of Version 0.8.1.0, Accelerate supports only flat parallelism.
Accelerate maintains full type information of the computation ASTs to ensure the type safety, and to support polymorphism. Instead of imposing informal ad-hoc constraints, Accelerate uses types to validate the correctness of computation ASTs. Accelerate represents all type constraints as constraints of the host language, Haskell. That is, an ill-typed Accelerate computation AST results in an ill-typed expression in the host language and is rejected by the Haskell compiler. For example, scalar array operations alone cannot build computation ASTs, and collective array operations cannot parameterise other collective array operations. Accelerate ensures this property by using the types that computation ASTs carry; the type constructor of scalar operations is different from the type constructor of collective array operations, the Accelerate front end exports computation ASTs with the type constructor of collective array operations to the available back ends, and collective array operations are only parameterised with the objects with the type constructor of scalar operations. In addition, the type information associated with Accelerate computation ASTs allows runtime type inspection, which is convenient for online code generation and optimisation.

Accelerate uses Higher-Order Abstract Syntax (HOAS) [44] as the user-level representation of the language, and converts it to a representation with De Bruijn indices [45] —or the De Bruijn representation. Both the HOAS and the De Bruijn representations are convenient for manipulating computation ASTs as their variable representations do not depend on the variable names, freeing them of the need for explicit α-conversion and the variable name capturing problem during substitution by representing variables without names; variable uses are referred to by the binding sites in the HOAS representation, and by the variable binder indices in the De Bruijn representation. The HOAS representation does not require the explicit construction and binding of variable terms, and it lets the function abstraction support of the host language handle them, making it more human-friendly than the De Bruijn representation, whereas the De Bruijn representation treats variables binder indices as first-order terms, facilitating code generation and optimisation. The trade offs and the conversion between the two representations have been studied by programming language researchers to take advantage
of both \([46,48]\).

We will use the HOAS representation to discuss the language constructs, reflecting the two-level nature of the language, in the following subsection, and will discuss the details of the conversion from the HOAS representation to the De Bruijn representation in Section 3.3.

### 3.2.1 Data Representation

The internal data representation of Accelerate is designed to facilitate the data manipulation for the back ends whilst the data representation at the user level provides a higher abstraction and a more human-friendly interface. This section explores the internal representations used by arrays, array elements, shapes and slices, and tuples and tuple indices in Accelerate.

#### Arrays and Array Elements

Arrays in Accelerate are restricted to contain elements of primitive types, and tuples thereof. To reflect this, Accelerate defines a type class, \texttt{Elem}, to which the array element types at the user level belong, and the array type as follows:

```haskell
class (Show a, Typeable a,
       Typeable (ElemRepr a), Typeable (ElemRepr' a),
       ArrayElem (ElemRepr a), ArrayElem (ElemRepr' a)) ⇒ Elem a
...
fromElem :: a → ElemRepr a
toElem  :: ElemRepr a → a
...
fromElem' :: a → ElemRepr' a
toElem'  :: ElemRepr' a → a
...

data Array dim e where -- Multi-dimensional arrays
  Array :: (Ix dim, Elem e)
    ⇒ ElemRepr dim -- extent of dimensions = shape
    → ArrayData (ElemRepr e) -- data
    → Array dim e

newArray :: (Ix dim, Elem e) ⇒ dim → (dim → e) → Array dim e
```

The Accelerate array element types at the user level are instances of the class `Elem`, whereas the Accelerate array element types in the internal representation are instances of the class `ArrayElem`. According to the definition of the type class `Elem`, if `a` is an instance of the class `Elem`, then `ElemRepr a` and `ElemRepr' a` must be instances of the type class `ArrayElem`. `ElemRepr` and `ElemRepr'` are type families—or *type functions*—that map a user level element type to the corresponding type for the internal representation. Accelerate defines the type families `ElemRepr` and `ElemRepr'`, where `τ` is a primitive type supported by Accelerate, as follows:

```haskell
  type family ElemRepr a :: *
  type instance ElemRepr () = ()
  type instance ElemRepr τ = ((), τ)
  type instance ElemRepr (a, b) = (ElemRepr a, ElemRepr' b)
  type instance ElemRepr (a, b, c) = (ElemRepr (a, b), ElemRepr' c)
  ...
  type family ElemRepr' a :: *
  type instance ElemRepr' () = ()
  type instance ElemRepr' τ = τ
  type instance ElemRepr' (a, b) = (ElemRepr a, ElemRepr' b)
  type instance ElemRepr' (a, b, c) = (ElemRepr (a, b), ElemRepr' c)
  ...
```

Accelerate uses pairs to represent primitive types internally and nested pairs to represent *n*-tuples, giving a uniform interface to the value structure of various types; different versions of the same function would have to be defined, otherwise, to handle tuples with different sizes in many cases. Given a primitive type `τ` and an *n*-tuple type `(τ₁, τ₂, ..., τₙ)`, `ElemRepr` produces a pair `(((), τ)` and a nested pair `((((), τ₁), τ₂), ..., τₙ)` respectively. For example, `ElemRepr Int` produces `(((), Int)`, and `ElemRepr (Int, Float, Word)` produces `((((), Int), ((), Float)), (((), Word))`. The type family `ElemRepr'` exists to avoid producing overly nested pairs; if `ElemRepr'` did not exist and `ElemRepr` were used instead of `ElemRepr'` in the definition of `ElemRepr`, `ElemRepr (Int, Float, Word)` would produce an overly nested pair `((((), Int), (((), Float)), (((), Word))))`. The
type class `Elem` provides methods to convert values of type `a` to values of type `ElemRepr a` and of type `ElemRepr' a`, and vice versa: `fromElem`, `fromElem'`, `toElem`, and `toElem'`.

The Accelerate array type `Array dim e` wraps the dimension information, `ElemRepr dim` that I explain later in this section, and unboxed arrays of the Haskell standard unboxed array type, `UArray Int e`, that hold the actual data, `ArrayData (ElemRepr e)`. `ArrayData` is a data family that constructs nested pairs of Haskell unboxed arrays with the given element type, and the instances of `ArrayData` introduce data constructors, `AD_e`, where `e ∈ {Unit, Int, Intn, Word, Wordn, Float, Double, Pair}`, and `n ∈ {8, 16, 32, 64}`:

- If `τ` is unit type, `ArrayData (ElemRepr τ)` is `AD_Unit`, which contains no array.

- If `τ` is a primitive type, `ArrayData (ElemRepr τ)` is `AD_Pair AD_Unit (AD_τ (UArray Int τ))`.

- If `τ` is a tuple type, `(τ₀, ..., τₙ)`, `ArrayData (ElemRepr τ)` is `AD_Pair (... (AD_Pair AD_Unit (AD_τ₀ (UArray Int τ₀))) ...) (AD_τₙ (UArray Int τₙ))`.

This internal representation that maintains arrays of tuples as tuples of arrays facilitates marshalling arrays from the host memory to the target device memory; each array has elements of a primitive type, and the computation of the storage and alignment requirements is simpler with arrays that have primitive type elements than with arrays that have tuple elements.

The type class `ArrayElem` provides two methods for reading an element in an Accelerate array, `indexArrayData` and `readArrayData`. They pack the element from the internal arrays into a value of the Accelerate array element type. It also provides a method for updating an element in an Accelerate array, `writeArrayData`. It unpacks a value of the Accelerate array element type into the sub-values and updates the internal arrays with the sub-values. The
following code snippet shows the relevant part of the ArrayElem instance for pairs.

\[
\text{instance } \text{ArrayElem } \text{a, ArrayElem } \text{b) } \Rightarrow \text{ArrayElem } (\text{a, b) where } \ldots
\]

\[
\ldots
\]

\[
\text{indexArrayData } (\text{AD_Pair a b) i = (indexArrayData a i, indexArrayData b i)}
\]

\[
\text{readArrayData } (\text{AD_Pair a b) i = do}
\]

\[
x \leftarrow \text{readArrayData a i}
\]

\[
y \leftarrow \text{readArrayData b i}
\]

\[
\text{return } (x, y)
\]

\[
\text{writeArrayData } (\text{AD_Pair a b) i (x, y) = do}
\]

\[
\text{writeArrayData a i x}
\]

\[
\text{writeArrayData b i y}
\]

**Shapes and Slices**

In addition to the data representations introduced above, Accelerate also defines data representations related to shapes and slicing of multi-dimensional arrays. It defines the type class `Shape` at the user level, whose instances are the types of values that describe the shape, as follows:

\[
\text{class Elem sh } \Rightarrow \text{Shape sh}
\]

\[
\text{instance Shape ()}
\]

\[
\text{instance Shape Int}
\]

\[
\text{instance Shape All}
\]

\[
\text{instance (ShapeBase a, ShapeBase b) } \Rightarrow \text{Shape } (\text{a, b)}
\]

\[
\text{instance (ShapeBase a, ShapeBase b, ShapeBase c) } \Rightarrow \text{Shape } (\text{a, b, c)}
\]

\[
\ldots
\]

`ShapeBase` is another type class, which has only `Int` and `All`; the type `All` is used only for slicing and it specifies entire dimensions in the slice descriptors. That is, the code snippet above instantiates `Shape` with unit type, `Int`, `All`, and `n`-tuples with elements of type `Int`, type `All`, or a combination of these two.
-- Identifier for entire dimensions in slice descriptors
data All = All deriving (Typeable, Show)
type instance ElemRepr All = ()
type instance ElemRepr' All = ()

There exist two subclasses of \texttt{Shape} at the user level: \texttt{Ix} and \texttt{SliceIx}. The types of dimension descriptors and array indices, $n$-tuples with integer elements, are the instances of \texttt{Ix}; $n$ is 0 for singleton arrays, 1 for one-dimensional arrays, and $n$ for $n$-dimensional arrays. The types of slice descriptors are the instances of \texttt{SliceIx}; every instance of \texttt{Shape} can be used as a slice descriptor type. It is to be noted that, even though Accelerate supports multi-dimensional arrays, Haskell unboxed arrays which are internal to Accelerate arrays are always one-dimensional, and \textit{linear row-major indices} are computed from multi-dimensional indices.

The type classes \texttt{Ix} and \texttt{SliceIx} have corresponding type classes, \texttt{Repr.Ix} and \texttt{Repr.SliceIx}, whose instances are the internal representation of the instances of \texttt{Ix} and \texttt{SliceIx} —i.e., \texttt{ElemRepr} $\tau$ is an instance of \texttt{Repr.Ix} if $\tau$ is an instance of \texttt{Ix}, and \texttt{ElemRepr} $\tau$ is an instance of \texttt{Repr.SliceIx} if $\tau$ is an instance of \texttt{SliceIx}.

The example in Figure 3.1 demonstrates how dimension descriptors, array indices, and slice descriptors work with arrays. The type of the cube is \texttt{Array (Int, Int, Int) Float}, where \texttt{(Int, Int, Int)} is the type of the dimension descriptor, and an instance of \texttt{Ix}. The dimension of this array is \texttt{(4, 2, 3)}, and the array indices are also of type \texttt{(Int, Int, Int)}. For instance, the element at the index \texttt{(2, 1, 2)} is \texttt{22.0}. The dimension descriptor and the indices of this array can be converted to a nested pair of type \texttt{ElemRepr (Int, Int, Int)}, which expands to \texttt{(((((), Int), Int), Int)}), for the internal representation, and \texttt{ElemRepr (Int, Int, Int)} is an instance of \texttt{Repr.Ix}. The array elements of this cube are stored in the one-dimensional Haskell unboxed array, which is internal to this cube, in ascending order, from 0.0 to 23.0.

We can slice Accelerate arrays, using a value of a \texttt{SliceIx} instance: \texttt{unit}, a value of type \texttt{Int}, a value of type \texttt{All}, or an $n$-tuple with a combination of values
Figure 3.1: An example of multi-dimension array of type `Array (Int, Int, Int) Float`

of type `Int` and values of type `All`. Slicing the cube in Figure 3.1 with the slice descriptor `(All, All, All)` produces the identical cube. The result of slicing with various other slice descriptors is depicted in Figure 3.2.

As Figure 3.2 demonstrates, the dimension of the slice changes depending on the slice descriptor. Each type class `SliceIdx` and `Repr.SliceIx` has type families associated with them to compute the dimensions of the slice: `Slice`, `CoSlice` and `SliceDim`. Accelerate computes slice dimensions using `Slice`, `CoSlice`, and `SliceDim` associated with `Repr.SliceIx`, and converts the internal representations of slice dimensions to the user level representations, as the computation is simpler with the internal representations and slice dimensions ought to be accessible at the user level.

A slice descriptor of type `sl` can be used to slice an array, whose dimension is of type `SliceDim sl`, to produce a slice whose dimension is of type `Slice sl`. The type `CoSlice sl` is the complement of the slice dimension of type `Slice sl` in the array dimension of type `SliceDim sl`. The definition of these type families associated with `Repr.SliceIx` is as follows:
Figure 3.2: Slices of the cube in Figure 3.1 with various slice descriptors
Table 3.1 shows \texttt{Slice} \(sl\), \texttt{CoSlice} \(sl\), and \texttt{SliceDim} \(sl\) in the class \texttt{SliceIx}, and \texttt{Slice} (\texttt{ElemRepr} \(sl\)), \texttt{CoSlice} (\texttt{ElemRepr} \(sl\)), and \texttt{SliceDim} (\texttt{ElemRepr} \(sl\)) in the class \texttt{Repr.SliceIx}, where \(sl\) is the type of a slice descriptor from Figure 3.2.

The shape and slice representation, and the use of type families such as \texttt{Slice}, \texttt{CoSlice}, and \texttt{SliceDim} ensure the correctness of the slice operations; slicing of an array is allowed only with a slice descriptor that is valid in the given dimension.

**Tuples**

Accelerate represents expression tuples as heterogeneous list with the following structure:

\[
\textbf{data Tuple }c\text{ }t\text{ where }\\
\text{NilTup }::= \text{ Tuple }c\text{ }()\\
\text{SnocTup }::= \text{ Elem }t\Rightarrow \text{ Tuple }c\text{ }s\Rightarrow c\text{ }t\Rightarrow \text{ Tuple }c\text{ }(s\text{, }t)
\]

Given the expression representation \(c\)—which can be either the HOAS representation or the De Bruijn representation—Accelerate represents the appending of an expression \(e\) of type \(c\text{ }t\) to a tuple \(es\) of type \texttt{Tuple} \(c\text{ }s\) using \texttt{SnocTup}: \(es\text{ ‘SnocTup’ }e\). \texttt{NilTup} represents a tuple without any element.
Table 3.1: Slice, CoSlice, and SliceDim computed with the slice descriptors from Figure 3.2
The tuple type representation is similar to the array element representation. Accelerate defines a type class `IsTuple` and a type family associated with it—to disallow unit type and primitive types as tuple types—`TupleRepr` as follows:

```haskell
class IsTuple tup where
    type TupleRepr tup
    fromTuple :: tup \rightarrow TupleRepr tup
    toTuple   :: TupleRepr tup \rightarrow tup

instance IsTuple (a, b) where
    type TupleRepr (a, b) = (((), a), b)
    fromTuple (x, y)   = (((), x), y)
    toTuple (((), x), y) = (x, y)

instance IsTuple (a, b, c) where
    type TupleRepr (a, b, c) = (TupleRepr (a, b), c)
    fromTuple (x, y, z)  = ((((), x), y), z)
    toTuple ((((), x), y), z) = (x, y, z)

... 
```

`TupleRepr tup` is a nested pair essentially same as `ElemRepr tup` where `tup` is an `n`-tuple, and `fromTuple` and `toTuple` for `TupleRepr tup` are same as `fromElem` and `toElem` for `ElemRepr tup`. Accelerate ensures that unit type and primitive types are disallowed as tuple types and that the tuple types are represented in nested pairs rather than `n`-tuples by providing interfaces only to build tuples of type `Tuple c (TupleRepr t)` with the constraint `IsTuple t`.

Accelerate represents tuple indices with `ZeroTupIdx` and `SuccTupIdx`:

```haskell
data TupleIdx t e where
    ZeroTupIdx :: Elem s \Rightarrow TupleIdx (t, s) s
    SuccTupIdx :: TupleIdx t e \rightarrow TupleIdx (t, s) e
```

Tuple indices are of type `TupleIdx t e` where `t` is the type of the tuple that is indexed and `e` is the type of the element that is projected. Accelerate provides interfaces only to build tuple indices of type `TupleIdx (TupleRepr t) e` with the constraint `IsTuple t` to ensure that the tuple indices are only used on the tuples and the tuple types are represented in nested pairs.
ZeroTupIdx projects the last element, of type $s$, in the tuple whose element representation is of type $(t, s)$, and SuccTupIdx $ix$ projects what $ix$ of type TupleIdx $te$ projects in the first element of the nested pair tuple, which is another tuple of type $t$. That is, SuccTupIdx ZeroTupIdx projects the second last element, SuccTupIdx (SuccTupIdx ZeroTupIdx) the third last element, and so forth.

### 3.2.2 Type Reification and Type Dictionaries

Accelerate back ends, especially those that generate code in typed languages, often require an inspection of an expression type for correct code generation. For example, to generate code for Accelerate tuples in C, we need to generate structs that match the structure and the type of the Accelerate tuples. To facilitate runtime type inspection, Accelerate provides methods `elemType` and `elemType'` in the type class `Elem` and `TupleType` reifies instances of `Elem`:

```haskell
class ... ⇒ Elem a where
    elemType :: {-dummy-} a → TupleType (ElemRepr a)
    elemType' :: {-dummy-} a → TupleType (ElemRepr' a)
```

```haskell
data TupleType a where
    UnitTuple :: TupleType ()
    SingleTuple :: ScalarType a → TupleType a
    PairTuple :: TupleType a → TupleType b → TupleType (a, b)
```

Depending on what `elemType` returns, we know the data type of the given expression:

- If `elemType e` or `elemType' e` returns `UnitType`, $e$ is of unit type.

- If `elemType e` or `elemType' e` returns `SingleTuple t`, and $t$ is of type `ScalarType τ`, $e$ is of a primitive type $τ'$.

- If `elemType e` or `elemType' e` returns `PairTuple $t_0$ $t_1$`, and $t_0$ and $t_1$ are of type `TupleType τ_0` and of type `TupleType τ_1` respectively, $e$ is a pair of type $(τ_0, τ_1)$.
Figure 3.3: The relationship between the reified data types in Accelerate

**ScalarType a** reifies the scalar types that are supported by Accelerate, using two abstract data types, **NumType a** and **NonNumType a**, which respectively reify the numeric types and the non-numeric types supported by Accelerate. **NumType**, in turn, uses **IntegralType a** and **FloatingType a** to reify the numeric types. **IntegralType a**, **FloatingType a**, and **NonNumType a** reify the integral types, floating-point types, and non-numeric types supported by Accelerate and they carry dictionaries for the class constraints of type a. **BoundedType a** only reifies the types that are bounded, for which the maximum bounds and the minimum bounds can be defined.

Figure 3.3 illustrates the relationship between **ScalarType a**, **NumType a**, **NonNumType a**, **IntegralType a**, **FloatingType a**, and **BoundedType a**. Given the relation $\tau \rightarrow \overline{\tau}$, $\overline{\tau}$ is parameterised with $\tau$ and it reifies type $\tau$. Figure 3.4 shows the definition of reified dictionaries and reified types in Accelerate.

Accelerate uses the reified dictionaries and the reified types to ensure type correctness when building ASTs for some arithmetic operations. For example,
data IntegralDict a where
  IntegralDict :: (Bounded a, Enum a, Eq a, Ord a, Show a
                   , Bits a, Integral a, Num a, Real a, Storable a)
                   \Rightarrow IntegralDict a

data FloatingDict a where
  FloatingDict :: (Enum a, Eq a, Ord a, Show a, Floating a, Fractional a
                  , Num a, Real a, RealFrac a, RealFloat a, Storable a)
                  \Rightarrow FloatingDict a

data NonNumDict a where
  NonNumDict :: (Bounded a, Enum a, Eq a, Ord a, Show a, Storable a)
              \Rightarrow NonNumDict a

data IntegralType a where
  Type \tau_i :: IntegralDict \tau_i \Rightarrow IntegralType \tau_i

data FloatingType a where
  Type \tau_f :: FloatingDict \tau_f \Rightarrow FloatingType \tau_f

data NonNumType a where
  Type \tau_n :: NonNumDict \tau_n \Rightarrow NonNumType \tau_n

data NumType a where
  IntegralNumType :: IntegralType a \Rightarrow NumType a
  FloatingNumType :: FloatingType a \Rightarrow NumType a

data BoundedType a where
  IntegralBoundedType :: IntegralType a \Rightarrow BoundedType a
  NonNumBoundedType :: NonNumType a \Rightarrow BoundedType a

data ScalarType a where
  NumScalarType :: NumType a \Rightarrow ScalarType a
  NonNumScalarType :: NonNumType a \Rightarrow ScalarType a

where \tau_i \in \{\text{Int, Intn, Word, Wordn}\}, n \in \{8,16,32,64\},
\tau_f \in \{\text{Float, Double}\}, \text{ and } \tau_n \in \{\text{Bool, Char}\}

\textbf{Figure 3.4:} The Definition of Reified Dictionaries and Reified Types in Accelerate
the following data constructor in Accelerate builds an AST for a sine operation:

\[
\text{PrimSin :: FloatingType } a \rightarrow \text{PrimFun } (a \rightarrow a)
\]

The first parameter that \text{PrimSin} takes is a reified floating-point type to ensure that ASTs for sine operations are either of type \text{PrimFun} (\text{Float} \rightarrow \text{Float}) or of type \text{PrimFun} (\text{Double} \rightarrow \text{Double}). It is impossible to define a value of type \text{Floating} \tau, where \tau is an integral type or a non-numeric type as the construction of the reified floating-point type requires the reified floating-point dictionary as the first parameter.

### 3.2.3 Collective Array Operations

Arrays in Accelerate are of type \text{Array dim e}, which carries the dimension information of the array, \text{dim}, and the data type of the array elements, \text{e}. The ASTs for collective array operations in Accelerate are of type \text{Acc a}, and the evaluation of the collective array operation of type \text{Acc a} produces a value of type \text{a} as the output; \text{a} is either an Accelerate array type or a pair of Accelerate array types.

Figure 3.6 lists the type signature of the collective array operations in the HOAS representation. The Accelerate front end does not expose the operations in Figure 3.6 directly to its users. Instead, it provides smart constructors, which are functions that build the values of specific type with the given parameters, for collective array operations. Figure 3.5 provides some of the smart constructors.

The rest of this section explains the collective array operations listed in Figure 3.6 in more detail.

**Use**

To use Accelerate arrays in computation ASTs, we need to promote them to collective array operations of type \text{Acc a}, as each node in ASTs is required. \text{Use} is the collective array operation that takes an Accelerate array of type \text{Array dim e}, and promotes it to a collective array operation of type \text{Acc (Array dim e)}.

The back ends that compile computation ASTs for the device with dedicated memory must transfer the arrays promoted by the \text{Use} operations from the host
Chapter 3. Accelerate

zip :: (Ix dim, Elem a, Elem b) ⇒ Acc (Array dim a) → Acc (Array dim b) → Acc (Array dim (a, b))
zip = zipWith (λ x y → tuple (x, y))

zipWith :: (Ix dim, Elem a, Elem b, Elem c) ⇒ (Exp a → Exp b → Exp c) → Acc (Array dim a) → Acc (Array dim b) → Acc (Array dim c)
zipWith = ZipWith

scan :: Elem a ⇒ (Exp a → Exp a → Exp a) → Exp a → Acc (Vector a) → (Acc (Vector a), Acc (Scalar a))
scan f e arr = unpair (Scanl f e arr)

Figure 3.5: Some of the smart constructors that Accelerate provides for the computation AST construction in the HOAS representation

memory to the device memory to make the arrays available on the device for the computation. As collective array operations run on the device and produce the output on the device memory, the data transfer from the host memory to the device memory is not required for other collective array operations. The data transfer from the device memory to the host memory is always performed only once after executing the operation at the root of the given computation AST—the operations in a computation AST are executed in post-order.

Unit

Unit takes a scalar operation, evaluates it, and promotes it to a collective array operation with a singleton array. This operation, as well as Use, may trigger data transfers from the host memory to the device memory in the back ends for the device with dedicated memory.
data Acc a where

-- Extract the first component
-- from a pair of collective array operations
FstArray :: (Ix dim1, Elem e1, Elem e2) 
⇒ Acc (Array dim1 e1, Array dim2 e2) 
⇒ Acc (Array dim1 e1)

-- Extract the second component
-- from a pair of collective array operations
SndArray :: (Ix dim2, Elem e1, Elem e2) 
⇒ Acc (Array dim1 e1, Array dim2 e2) 
⇒ Acc (Array dim2 e2)

-- Array inlet
Use :: Array dim e 
⇒ Acc (Array dim e)

-- Capture a scalar (or a tuple of scalars) in a singleton array
Unit :: Elem e ⇒ Exp e ⇒ Acc (Array () e)

-- Change the shape of an array without altering its contents
Reshape :: Ix dim 
⇒ Exp dim 
⇒ Acc (Array dim’ e) 
⇒ Acc (Array dim e)

-- Replicate an array across one or more dimensions
Replicate :: (SliceIx slix, Elem e) 
⇒ Exp slix 
⇒ Acc (Array (Slice slix) e) 
⇒ Acc (Array (SliceDim slix) e)

-- Index a subarray out of an array;
-- i.e., the dimensions not indexed are returned whole
Index :: (SliceIx slix, Elem e) 
⇒ Acc (Array (SliceDim slix) e) 
⇒ Exp slix 
⇒ Acc (Array (Slice slix) e)

-- Apply the unary function to all elements of the array
Map :: (Elem e, Elem e’) 
⇒ (Exp e ⇒ Exp e’) 
⇒ Acc (Array dim e) 
⇒ Acc (Array dim e’)

Figure 3.6: The Accelerate collective array operations in the HOAS representation [5]
-- Apply the binary function pairwise to all elements of the arrays
ZipWith :: (Elem e1, Elem e2, Elem e3)  
⇒ (Exp e1 → Exp e2 → Exp e3)  
⇒ Acc (Array dim e1)  
⇒ Acc (Array dim e2)  
⇒ Acc (Array dim e3)  

-- Fold of an array
Fold :: Elem e  
⇒ (Exp e → Exp e → Exp e)  
⇒ Exp e  
⇒ Acc (Array dim e)  
⇒ Acc (Array () e)  

-- Segmented fold of an array
FoldSeg :: Elem e  
⇒ (Exp e → Exp e → Exp e)  
⇒ Exp e  
⇒ Acc (Array Int e)  
⇒ Acc (Array Int Int)  
⇒ Acc (Array Int e)  

-- Left-to-right pre-scan of a linear array
Scan :: Elem e  
⇒ (Exp e → Exp e → Exp e)  
⇒ Exp e  
⇒ Acc (Array Int e)  
⇒ Acc (Array Int e, Array () e)  

-- Generalised forward permutation
Permute :: (Ix dim, Ix dim', Elem e)  
⇒ (Exp e → Exp e → Exp e)  
⇒ Acc (Array dim' e)  
⇒ (Exp dim → Exp dim')  
⇒ Acc (Array dim e)  
⇒ Acc (Array dim' e)  

-- Generalised multi-dimensional backwards permutation
Backpermute :: (Ix dim, Ix dim', Elem e)  
⇒ Exp dim'  
⇒ (Exp dim' → Exp dim)  
⇒ Acc (Array dim e)  
⇒ Acc (Array dim' e)  

Figure 3.6: The Accelerate collective array operations in the HOAS representation (continued) [5]
Reshape and Index

Reshape is self-explanatory. It takes the new shape as a parameter to which the collective array operation is reshaped. If the size of the new shape is different from that of the original shape, Accelerate gives an error when evaluating the reshape operation. As Haskell arrays which are internal to Accelerate arrays are always one-dimensional, the array contents are not re-arranged; only the dimension information, based on which linear row-major indices are computed, is updated.

Index expressions slice collective array operations, using slice descriptors. The detailed explanation about how slicing works is in Section 3.2.1.

Higher-Order Collective Array Operations

Amongst all collective array operations available in Accelerate, the higher-order collective array operations —such as Replicate, Map, ZipWith, Fold, FoldSeg, Scan, Permute, and Backpermute— are the ones that actually describe computation patterns. Table 3.2 provides the description of each higher-order function.

The higher-order collective array operations are parameterised with scalar operations, and they apply the parameterising scalar operations to the input arrays according to the application pattern determined by higher-order collective array operation. It is worth noting that Accelerate restricts scalar operations to first-order, considering the parallel hardware platforms such as CUDA—CUDA only have limited support for function pointers as of Version 3.1 and function pointers are essential to emulate higher-order functions in C-like languages. That is, Accelerate scalar operations cannot produce another scalar operation as result, and Accelerate higher-order collective array operations can only produce arrays of values as the result.

For instance, the computation AST built by

\[
\text{zipWith } f (\text{map } g (\text{use } xs)) (\text{fst } (\text{scanl } h \text{ e (use } ys)))
\]

describes the computation that applies \( g \) to each element in \( xs \), performs a left-to-right pre-scan on \( ys \) using the combination function, \( h \), and the default value,
<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replicate</td>
<td>\texttt{replicate e slice} replicates the array \texttt{slice} in the direction \texttt{e} to construct another array.</td>
</tr>
<tr>
<td>Map</td>
<td>\texttt{map f xs} applies the scalar operation \texttt{f} to each element of \texttt{xs}.</td>
</tr>
<tr>
<td>ZipWith</td>
<td>\texttt{zipWith f xs ys} applies the scalar operation \texttt{f} to each element of \texttt{xs} and its corresponding element of \texttt{ys}.</td>
</tr>
<tr>
<td>Fold</td>
<td>\texttt{fold f x xs} applies \texttt{f} to \texttt{x} and the first element of \texttt{xs}, and applies \texttt{f} to the intermediate result and the second element of \texttt{xs}, and so forth.</td>
</tr>
<tr>
<td>FoldSeg</td>
<td>Given \texttt{foldSeg f x xs segd}, \texttt{xs} consists of a number of segments and \texttt{segd} is the segment descriptor. \texttt{foldSeg f x xs segd} evaluates to an array of the segment fold values; the length of the output array is the number of segments in \texttt{xs}.</td>
</tr>
<tr>
<td>Scan</td>
<td>\texttt{scan f x xs} is fundamentally the same as \texttt{fold f x xs}, but with the intermediate results in the output array. \texttt{scan f x xs} evaluates to a pair that consists of an array of the intermediate results and the fold value.</td>
</tr>
<tr>
<td>Permute</td>
<td>\texttt{permute f def p xs} creates the output array by copying \texttt{def}, then, for every element in \texttt{xs}, it applies \texttt{f} to the element at the position \texttt{i} in \texttt{xs} and the element at the position \texttt{p i} in the output array, and updates the element at the position \texttt{p i} in the output array with the result.</td>
</tr>
<tr>
<td>Backpermute</td>
<td>\texttt{backpermute sh p xs} produces an array whose element at the position \texttt{i} is a copy of the element at the position \texttt{p i} in \texttt{xs}. The shape of the output array is determined by \texttt{sh}.</td>
</tr>
</tbody>
</table>

Table 3.2: The higher-order collective array operations in Accelerate

\texttt{e}, and finally applies \texttt{f} to each corresponding element in the arrays produced by \texttt{map g (use xs)} and \texttt{fst $ scanl h e (use ys)}.

These higher-order collective array operations have a well-known data parallel interpretation \cite{43}, and there exist a number of frameworks that define the algorithmic skeletons or the templates of some of these higher-order collective array operations for various platforms \cite{30,32,35,38,40,41}. Thus, it is a reasonable recommendation that the back ends define the algorithmic skeletons and use them to generate the device code, and the CUDA back end does so.

**Extracting Pair Components**

Whilst all other collective operations are of type \texttt{Acc (Array dim e)}, \texttt{Scan} is of type \texttt{Acc (Array Int e, Array () e)}; the first component is the array containing the pre-scan results, and the second component is the singleton array with the
fold value.

In Haskell, we use \texttt{fst} and \texttt{snd} to extract pair components. However, we cannot apply \texttt{fst} and \texttt{snd} directly to a value of type \texttt{Acc (a, b)} as they are of the following types:

\[
\begin{align*}
\text{fst} & : (a, b) \rightarrow a \\
\text{snd} & : (a, b) \rightarrow b
\end{align*}
\]

To deal with this situation, Accelerate provides two collective array operations \texttt{FstArray} and \texttt{SndArray}, and a convenience function \texttt{unpair}:

\[
\begin{align*}
\text{unpair} & : (Ix \ dim1, Ix \ dim2, \text{Elem e1}, \text{Elem e2}) \\
& \Rightarrow \text{Acc (Array dim1 e1, Array dim2 e2)} \\
& \rightarrow (\text{Acc (Array dim1 e1), Acc (Array dim2 e2)})
\end{align*}
\]

\[
\text{unpair acc} = (\text{FstArray acc, SndArray acc})
\]

The convenience function \texttt{unpair} converts \texttt{Acc (a, b)} to \texttt{(Acc a, Acc b)}, and enables the use of \texttt{fst} and \texttt{snd}.

The smart constructor for \texttt{Scan} operation demonstrates the use of this convenience function: \texttt{scan f e arr = unpair (Scan f e arr)}. The first component of \texttt{Scan f e arr} is an array of the scan values and the second component is another array with only one element, the fold value.

### 3.2.4 Scalar Operations

Scalar operations in Accelerate consist of a number of scalar expressions, and Figure 3.7 lists the scalar expressions in the HOAS representations. Accelerate scalar expressions are of type \texttt{Exp t}.

Similarly to collective array operations, Accelerate provides smart constructors to build scalar expressions in the HOAS representation.

#### Constants

The \texttt{Const} expression comes with a smart constructor, \texttt{constant}.

\[
\begin{align*}
\text{constant} & : \text{Elem t} \Rightarrow t \rightarrow \text{Exp t} \\
\text{constant} & = \text{Const}
\end{align*}
\]
constant promotes a Haskell value of type \( t \) to an Accelerate constant in the HOAS representation of type \( \text{Exp} \ t \).

Tuples and Tuple Projections

The Tuple expressions of type \( \text{Exp} \ t \) take a heterogeneous list of type Tuple \( \text{Exp} \ (\text{TupleRepr} \ t) \), whose elements are scalar expressions. The structure of the heterogeneous list is explained in Section 3.2.1.

Instead of forcing users to explicitly build nested pairs as the heterogeneous lists, Accelerate provides a smart constructor, tuple, which builds nested pairs, given \( n \)-tuples, and the Tuple expressions in the HOAS representation, using the nested pairs.

```haskell
class Tuple tup where
  type TupleT tup
  tuple :: tup \rightarrow TupleT tup
  ...
```

As the Haskell type system considers tuples with different numbers of elements distinct, it is impossible to define a function that works on tuples with any number of elements without type families. Accelerate, thus, defines tuple using a type family, TupleT, which maps a \( n \)-tuple of scalar expression types, \((\text{Exp} \ a, \text{Exp} \ b, \ldots)\), to a scalar expression type of a tuple, \(\text{Exp} \ (a, b, \ldots)\). For instance, when \((\text{Exp} \ a, \text{Exp} \ b, \ldots)\) is given as \( \text{tup} \), the type of tuple becomes \((\text{Exp} \ a, \text{Exp} \ b, \ldots) \rightarrow \text{Exp} \ (a, b, \ldots)\), and tuple \((x_1, \ldots, x_n)\) runs Tuple (NilTup 'SnocTup' x1 'SnocTup' ... 'SnocTup' xn) to build the tuple.

Similarly to the Tuple expressions, Accelerate provides a smart constructor, untuple, for the Prj expressions:

```haskell
class Tuple tup where
  type TupleT tup
  ...
  untuple :: TupleT tup \rightarrow tup
```

which, given expressions, \( e_0 \) of type \( \text{Exp} \ (a, b) \) and \( e_1 \) of type \( \text{Exp} \ (a, b, c) \), returns:
data Exp t where
  -- Needed for conversion to de Bruijn form
Tag :: Elem t
    ⇒ Int -- environment size at defining occurrence
    ⇒ Exp t

  -- Constant values
Const :: Elem t ⇒ t ⇒ Exp t

  -- Tuples
Tuple :: (Elem t, IsTuple t)
    ⇒ Tuple.Tuple Exp (TupleRepr t)
    ⇒ Exp t
Prj :: (Elem t, IsTuple t)
    ⇒ TupleIdx (TupleRepr t) e
    ⇒ Exp t
    ⇒ Exp e

  -- Conditional expression (non-strict in 2nd and 3rd argument)
Cond :: Exp Bool
        ⇒ Exp t
        ⇒ Exp t
        ⇒ Exp t

  -- Primitive constants
PrimConst :: Elem t
    ⇒ PrimConst t
    ⇒ Exp t

  -- Primitive scalar operations
PrimApp :: (Elem a, Elem r)
    ⇒ PrimFun (a → r)
    ⇒ Exp a
    ⇒ Exp r

  -- Project a single scalar from an array
IndexScalar :: Acc (Array dim t)
    ⇒ Exp dim
    ⇒ Exp t

  -- Array shape
Shape :: Elem dim
    ⇒ Acc (Array dim e)
    ⇒ Exp dim

Figure 3.7: The Accelerate scalar expressions in the HOAS representation [5]
(SuccTupIdx ZeroTupIdx ‘Prj’ e₀, ZeroTupIdx ‘Prj’ e₀).

and

(SuccTupIdx (SuccTupIdx ZeroTupIdx) ‘Prj’ e₁,
 (SuccTupIdx ZeroTupIdx) ‘Prj’ e₁,
 ZeroTupIdx ‘Prj’ e₁).

**Conditional Expressions**

Each Cond expression consists of three other scalar expressions: one for the predicate, one for the consequent, and the other for the alternative.

**Primitive Constants**

There are three primitive constants available in Accelerate:

```haskell
data PrimConst ty where

  -- constants from Bounded
  PrimMinBound :: BoundedType a → PrimConst a
  PrimMaxBound :: BoundedType a → PrimConst a

  -- constant from Floating
  PrimPi :: FloatingType a → PrimConst a
```

and each comes with a smart constructor: `minBound`, `maxBound`, and `pi`. When evaluated, `minBound` and `maxBound` are meant to produce constants with the minimum bound and the maximum bound of the expression type, and `pi` a constant with the π(π) value.

The first two smart constructors, `minBound` and `maxBound`, build:

```haskell
  PrimConst (PrimMinBound boundedType)
  PrimConst (PrimMaxBound boundedType).
```

where `boundedType` is a function defined in the type class `IsBounded`, and it is passed to `PrimMinBound` and `PrimMaxBound` to ensure that these primitive constants are only available when the expression type is bounded. As the Accelerate
front end instantiates the type class \texttt{IsBounded} only with the bounded types, \texttt{minBound} and \texttt{maxBound} fail if they are called to form expressions of unbounded types.

The other smart constructor, \texttt{pi}, builds:

\[
\text{PrimConst (PrimPi floatingType)}
\]

and it works in the same way as \texttt{minBound} and \texttt{maxBound}. The only difference is that \texttt{floatingType} is a function defined in the type class \texttt{IsFloating} and the Accelerate front end instantiates this type class only with \texttt{Float} and \texttt{Double}.

**Primitive Scalar Operations**

\texttt{PrimApp} takes two parameters: a primitive scalar operator and the argument to the primitive scalar operator. The argument has to be a single value if the primitive scalar operator given to the \texttt{PrimApp} expression is unary, and a pair if binary. For example, the addition of \(x\) and \(y\) is represented as:

\[
\text{PrimAdd numType 'PrimApp' tup2 (x, y)}
\]

The Accelerate front end overloads arithmetic operators and mathematical functions such as \((+), (*)\), (-), \texttt{negate}, \texttt{abs}, \texttt{signum}, \texttt{fromInteger}, \texttt{sin}, \texttt{cos}, etc. to provide a human-friendly interface to build the primitive scalar operations in the above representation; \(x + y\) looks more natural than the representation above.

These overloaded operators and functions build the primitive scalar operators typically by passing a function to ensure the type correctness to the data constructor; \texttt{numType} in the example above, which is defined in the type class \texttt{IsNum}. The Accelerate front end defines that \texttt{IsNum} is a subclass of \texttt{Num} and of \texttt{IsScalar}, and \texttt{Exp t} is an instance of \texttt{Num} if \(t\) is an instance of \texttt{IsNum} and of \texttt{Elem}.

\[
\begin{aligned}
\text{class } (\texttt{Num} a, \texttt{IsScalar} a) & \Rightarrow \texttt{IsNum} a \text{ where} \\
\texttt{numType} & :: \texttt{NumType} a \\
\text{instance } (\texttt{Elem} t, \texttt{IsNum} t) & \Rightarrow \texttt{Num} (\texttt{Exp} t) \text{ where} \\
(+) \times x & = \text{PrimAdd} \texttt{numType 'PrimApp' tup2} (x, y) \\
\end{aligned}
\]
Chapter 3. Accelerate

The Accelerate front end defines similar constraints and contexts for all primitive scalar operators to ensure that the operators are only applied to the values of the correct type. For instance, the Accelerate front end rejects

\[(\text{constant 'a' :: Exp Char}) + (\text{constant 'b' :: Exp Char})\]

as \text{Char} is not an instance of \text{Num} and \text{IsNum}, and \text{Exp Char} is not an instance of \text{Num}.

For the operators which are not or only partially overloadable, Accelerate defines new operators with the standard operators names suffixed by an ‘*’. For instance, the equality operator has the following type:

\[(\==) :: (\text{Eq a}) \Rightarrow a \rightarrow a \rightarrow \text{Bool}\]

As \text{Bool} is a concrete type, we cannot overload it with \text{Exp Bool}. That is, we cannot use \((==)\) to build an AST for an equality expression. Instead, we define an operator \((==*)\) for equality expressions:

\[(==*) :: (\text{Elem t, IsScalar t}) \Rightarrow \text{Exp t} \rightarrow \text{Exp t} \rightarrow \text{Exp Bool}\]

**Array Indexing**

The scalar expression \text{IndexScalar} takes a collective array operation that is to be indexed and a scalar expression that is used as the index. The array indexing expression limits the data type of the index expression to be \text{dim}, which is an \(n\)-tuple of integers for \(n\)-dimensional collective array operation.

The Accelerate front end overloads the Haskell \((\!)\) operator to provide an interface to build \text{IndexScalar} expressions; \((\!)\) is the operator commonly used to index other Haskell arrays. For example,

\[(xs :: \text{Acc (Array Int Word32)}) ! (\text{constant 0})\]

the above array indexing expression builds the following:

\[
\text{IndexScalar (xs :: \text{Acc (Array Int Word32)}) (Const 0 :: \text{Exp Int}) :: Exp Word32}
\]

and the evaluation of this expression produces the \(0^{th}\) element of \(xs\).
Shapes

The scalar expression \texttt{Shape} extracts the dimension information of a collective array operation, and promotes the dimension information of type \(\tau\) to a scalar expression of type \(\text{Exp}\ \tau\).

3.3 Conversion to the De Bruijn Representation

Accelerate language constructs introduced in Section 3.2.3 and Section 3.2.4 build computation ASTs in the HOAS representation, which provides users with a syntactic structure similar to the syntactic structure of the host language, Haskell. To assist back end implementors with code generation and optimisation, Accelerate also provides a means to convert ASTs in the HOAS representation to the De Bruijn representation.

This section introduces the De Bruijn representation in Accelerate and the conversion from the HOAS to the De Bruijn representation.

3.3.1 De Bruijn Indices and Environments

As the De Bruijn representation explicitly carries variable terms and their bindings, we encode De Bruijn indices and environment layouts that keep track of variable bindings and their types with the following representation:

\begin{verbatim}
-- De Bruijn indices
data Idx env t where
  ZeroIdx :: Idx (env, s) s
  SuccIdx :: Idx env t -> Idx (env, s) t

-- Environment layouts
data Layout env env' where
  EmptyLayout :: Layout env ()
  PushLayout :: Typeable t
              => Layout env env' -> Idx env t -> Layout env (env', t)
\end{verbatim}

The De Bruijn index representation is very similar to the tuple index representation, \texttt{TupleIdx}, in Section 3.2.1 De Bruijn indices are of type \texttt{Idx env t}.
where \(\text{env}\) is the type of the environment that is indexed and \(t\) is the type of the variable that is projected. The environment layout types are represented as nested pairs; when a new variable is bound, the old environment becomes the first component in the pair and the new variable the second component.

\(\text{ZeroIdx}\) projects the variable bound by the \(0^{th}\) innermost binder, which is the last element, of type \(s\), in the environment, whose representation is of type \((\text{env}, s)\). And \(\text{SuccIdx}\ ix\) projects what \(ix\) of type \(\text{Idx env}\ t\) projects in the first element, of type \(\text{env}\), in the environment layout, which is of type \((\text{env}, s)\). That is, \(\text{SuccIdx}\ ZeroIdx\) projects the variable bound by the \(1^{st}\) innermost binder, and \(\text{SuccIdx}\ (\text{SuccIdx}\ ZeroIdx)\) the variable bound by the \(2^{nd}\) innermost binder, and so forth.

The data constructor \(\text{EmptyLayout}\)constructs an empty environment layout, and the data constructor \(\text{PushLayout}\) appends a De Bruijn index to an existing environment layout to create a new environment layout. Environment layouts are of type \(\text{Layout env}\ env'\). The type variable \(env'\) is used to compute the environment layout in terms of a nested pair of types as we push a De Bruijn index into the environment, and the type variable \(\text{env}\) is used to resolve the type of the De Bruijn indices in the environment to a concrete type.

Given only an index of type \(\text{Idx env}\ t\), the number of the environment layout, which the index is pushed to, is unknown; even \(\text{ZeroIdx}\) has a type variable \(\text{env}\), which is to be resolved later, as the first component of the environment type; the type variable \(t\) is resolved when the index is created.

```plaintext
let ix0 = ZeroIdx
ix1 = SuccIdx ZeroIdx
in EmptyLayout 'PushLayout' ix1 'PushLayout' ix0
```

The expression above creates an environment layout, which contains two De Bruijin indices: \(ix0\) and \(ix1\). The indices \(ix0\) and \(ix1\) can actually be pushed to an arbitrary environment layout, thus, their types are partially resolved to \(\text{Idx}\ ((\text{env0}, t), t)\) and to \(\text{Idx}\ ((\text{env1}, t), s)\ t\). Upon building the environment layout with these indices, the type of \(ix0\) and the type of \(ix1\) are further resolved to \(\text{Idx}\ ((\text{env1}, t), s)\ s\) and \(\text{Idx}\ ((\text{env1}, t), s)\ t\) respectively, and the type of the
environment layout to Layout \(((env_1, t), s) (((), t), s)\). Finally, we resolve the type of the environment layout and the indices in the environment layout to concrete types by passing the environment layout as a function parameter of type Layout env env.

### 3.3.2 Collective Array Operations

Collective array operations in the De Bruijn representation are of type \(\text{OpenAcc aenv a}\), where \(aenv\) is the type of the environment layout represented as a nested pair, which keeps track of array variable bindings — e.g., the empty environment is of unit type, and the environment with two variables of type \(\tau_0\) and \(\tau_1\) is of type \((((), \tau_0), \tau_1)\) — and \(a\) is the type of the output, which is either an array or a pair of arrays, depending on the operation.

Figure 3.8 lists the type signature of the collective array operations in the De Bruijn representation. The De Bruijn representation is not available at the user level. Instead, users build computation ASTs in the HOAS representation, and convert them into the De Bruijn representation using the conversion functions provided by the front end.

The rest of this section explains the conversion of computation ASTs in the HOAS representation to the De Bruijn representation. To prevent confusion, I prefix the \(\text{OpenAcc}\) data constructors with \(\text{AST}\). For example, \texttt{Use} is the data constructor for the HOAS representation whereas \texttt{AST.Use} is the data constructor for the De Bruijn representation.

The Accelerate front end defines the following conversion functions for collective array operations:

\[
\text{convertOpenAcc} :: \text{Layout aenv aenv} \\
\quad \rightarrow \text{Acc a} \\
\quad \rightarrow \text{OpenAcc aenv a}
\]

\[
\text{convertAcc} :: \text{Acc a} \rightarrow \text{AST.Acc a}
\]

The function \texttt{convertAcc} converts a computation AST in the HOAS representation to the De Bruijn representation by calling \texttt{convertOpenAcc} with an empty
data OpenAcc aenv a where

  -- Local binding to represent sharing and demand explicitly
  Let :: OpenAcc aenv (Array dim e) -- bound expression
       → OpenAcc (aenv, Array dim e)
       (Array dim' e') -- the bound expr’s scope
       → OpenAcc aenv (Array dim' e')

  -- Variant of 'Let' binding (and decomposing) a pair
  Let2 :: OpenAcc aenv (Array dim1 e1, Array dim2 e2) -- bound expressions
       → OpenAcc ((aenv, Array dim1 e1), Array dim2 e2)
       (Array dim' e') -- the bound expr’s scope
       → OpenAcc aenv (Array dim' e')

  -- Variable bound by a 'Let', represented by a de Bruijn index
  Avar :: Elem e
       ⇒ Idx aenv (Array dim e)
       → OpenAcc aenv (Array dim e)

  -- Array inlet
  Use :: Array dim e → OpenAcc aenv (Array dim e)

  -- Capture a scalar (or a tuple of scalars) in a singleton array
  Unit :: Elem e ⇒ Exp aenv e → OpenAcc aenv (Array () e)

  -- Change the shape of an array without altering its contents
  Reshape :: Ix dim
          ⇒ Exp aenv dim -- new shape
          → OpenAcc aenv (Array dim' e) -- array to be reshaped
          → OpenAcc aenv (Array dim e)

  -- Replicate an array across one or more dimensions
  Replicate :: (Ix dim, Elem slinx)
            ⇒ SliceIndex
            (ElemRepr slinx) (ElemRepr sl) co' (ElemRepr dim)
            → Exp aenv slinx -- slice value specification
            → OpenAcc aenv (Array sl e) -- data to be replicated
            → OpenAcc aenv (Array dim e)

  -- Index a subarray out of an array
  Index :: (Ix sl, Elem slinx)
         ⇒ SliceIndex
         (ElemRepr slinx) (ElemRepr sl) co' (ElemRepr dim)
         → OpenAcc aenv (Array dim e) -- array to be indexed
         → Exp aenv slinx -- slice value specification
         → OpenAcc aenv (Array sl e)

Figure 3.8: The Accelerate collective array operations in the De Bruijn representation [5]
-- Apply the unary function to all elements of the array
Map :: Elem e'  
\Rightarrow Fun aenv (e \rightarrow e')  
\rightarrow OpenAcc aenv (Array dim e)  
\rightarrow OpenAcc aenv (Array dim e')

-- Apply the binary function pairwise to all elements of the arrays
ZipWith :: Elem e3  
\Rightarrow Fun aenv (e_1 \rightarrow e_2 \rightarrow e_3)  
\rightarrow OpenAcc aenv (Array dim e_1)  
\rightarrow OpenAcc aenv (Array dim e_2)  
\rightarrow OpenAcc aenv (Array dim e_3)

-- Fold of an array
Fold :: Fun aenv (e \rightarrow e \rightarrow e) -- combination function  
\Rightarrow Exp aenv e -- default value  
\rightarrow OpenAcc aenv (Array dim e) -- folded array  
\rightarrow OpenAcc aenv (Array () e)

-- Segmented fold of an array
FoldSeg :: Fun aenv (e \rightarrow e \rightarrow e) -- combination function  
\Rightarrow Exp aenv e -- default value  
\rightarrow OpenAcc aenv (Array Int e) -- folded array  
\rightarrow OpenAcc aenv (Array Int Int) -- segment descriptor  
\rightarrow OpenAcc aenv (Array Int e)

-- Left-to-right pre-scan of a linear array
Scan :: Fun aenv (e \rightarrow e \rightarrow e) -- combination function  
\Rightarrow Exp aenv e -- default value  
\rightarrow OpenAcc aenv (Array Int e) -- linear array  
\rightarrow OpenAcc aenv (Array Int e, Array () e)

-- Generalised forward permutation
Permute :: Fun aenv (e \rightarrow e \rightarrow e) -- combination function  
\Rightarrow OpenAcc aenv (Array dim' e) -- default values  
\rightarrow Fun aenv (dim' \rightarrow dim') -- permutation function  
\rightarrow OpenAcc aenv (Array dim e) -- source array  
\rightarrow OpenAcc aenv (Array dim' e)

-- Generalised multi-dimensional backwards permutation
Backpermute :: Ix dim'  
\Rightarrow Exp aenv dim' -- dimensions of the result  
\rightarrow Fun aenv (dim' \rightarrow dim) -- permutation function  
\rightarrow OpenAcc aenv (Array dim e) -- source array  
\rightarrow OpenAcc aenv (Array dim' e)

Figure 3.8: The Accelerate collective array operations in the De Bruijn representation (continued)
Chapter 3. Accelerate

layout, EmptyLayout. The function convertOpenAcc recursively applies itself to each node in the computation AST whilst traversing the AST in post-order, and, at each node, it constructs the collective operation in the De Bruijn representation after the conversion of the sub-operations. The environment layout is passed around until the conversion is complete.

There are also other conversion functions that convert various forms of scalar operations or scalar expressions that parameterise collective array operations:

- The function convertFun1 converts an unary scalar operation in the HOAS representation to the De Bruijn representation.

- The function convertFun2 converts a binary scalar operation in the HOAS representation to the De Bruijn representation.

- The function convertExp converts a scalar expression in the HOAS representation to the De Bruijn representation.

These conversion functions are explained in more detail in Section 3.3.3.

\[
\text{convertOpenAcc } \text{alyt} \ (\text{ZipWith } f \ \text{acc1} \ \text{acc2}) \\
= \text{AST.ZipWith} \ (\text{convertFun2 } \text{alyt} \ f) \\
\quad \ (\text{convertOpenAcc } \text{alyt} \ \text{acc1}) \\
\quad \ (\text{convertOpenAcc } \text{alyt} \ \text{acc2})
\]

The code snippet above demonstrates how convertOpenAcc converts a computation AST in the HOAS representation, where ZipWith f acc1 acc2, acc1, and acc2 are collective array operations, f is a binary scalar operation that is applied pairwise to all elements of acc1 and acc2, and alyt is the environment layout. We convert f by calling convertFun2 with the environment layout, and acc1 and acc2 by recursively calling convertOpenAcc with the environment layout, then construct the collective array operation AST.ZipWith with the results from the conversion of f, acc1 and acc2.

We convert most collective array operations as above with a few exceptions: FstArray, SndArray, and Use.
Use

As both \texttt{Use} and \texttt{AST.Use} take an Accelerate array and promote it to a collective array operation, \texttt{convertOpenAcc} pattern-matches to extract the Accelerate array and use it to construct a collective array operation in the De Bruijn representation:

\[
\text{convertOpenAcc} _ {\text{Use array}} = \text{AST.Use array}
\]

Let-Bindings and Variable Indices

\texttt{FstArray} and \texttt{SndArray} take a collective array operation of type \texttt{Acc (Array dim1 e1, Array dim2 e2)}, and extract the collective array operation of type \texttt{Acc (Array dim1 e1)} and \texttt{Acc (Array dim2 e2)} respectively. They are converted to \texttt{AST.Let2} operations. The code snippet below provides the conversion from \texttt{FstArray} and \texttt{SndArray} to \texttt{Acc.Let2}:

\[
\begin{align*}
\text{convertOpenAcc alyt (FstArray acc)} &= \text{AST.Let2} (\text{convertOpenAcc alyt acc}) (\text{AST.Avar (SuccIdx ZeroIdx)}) \\
\text{convertOpenAcc alyt (SndArray acc)} &= \text{AST.Let2} (\text{convertOpenAcc alyt acc}) (\text{AST.Avar ZeroIdx})
\end{align*}
\]

where \texttt{acc} is of type \texttt{Acc (Array dim1 e1, Array dim2 e2)}, and the conversion of \texttt{acc} constructs an operation of type:

\[
\text{convertOpenAcc alyt acc :: OpenAcc aenv (Array dim1 e1, Array dim2, e2)}
\]

which is passed to \texttt{AST.Let2} as the first parameter. As the second parameter to \texttt{AST.Let2} is of type \texttt{OpenAcc ((aenv, Array dim1 e1), Array dim2 e2) (Array dim' e')}, the type of \texttt{AST.AVar ZeroIdx} and the type of \texttt{AST.AVar (SuccIdx ZeroIdx)} are resolved to:

\[
\begin{align*}
\text{AST.AVar ZeroIdx} &: \text{OpenAcc ((aenv, Array dim1 e1), Array dim2 e2) (Array dim2 e2)} \\
\text{AST.AVar (SuccIdx ZeroIdx)} &: \text{OpenAcc ((aenv, Array dim1 e1), Array dim2 e2) (Array dim1 e1)}
\end{align*}
\]
After all, the type of the AST.Let2 operation is either OpenAcc aenv (Array dim1 e1) or OpenAcc aenv (Array dim2 e2), which correspond to Acc (Array dim1 e1) and Acc (Array dim2 e2).

### 3.3.3 Scalar Operations

As Figure 3.9 shows, scalar expressions in the De Bruijn representation are of type OpenExp env aenv t, where env and aenv are the types of the environment layouts represented as nested pairs that keep track of scalar variable bindings and array variable bindings. As no scalar expression introduces variable bindings, the only bindings found in env are the parameters to the scalar operations. Scalar expressions inherit aenv from the collective array operation that they parameterise, and aenv is used if collective array operations appear as parameters to scalar expressions.

The conversion of scalar expressions is centred around two conversion functions: convertExp and convertOpenExp. The function convertExp converts a closed scalar operation by calling convertOpenExp with an empty environment layout EmptyLayout, and the function convertOpenExp converts a scalar operation in the given environment. Other conversion functions such as convertFun1 and convertFun2 also utilise these functions to convert the body of scalar operations to the De Bruijn representation.

\[
\text{convertOpenExp :: forall t env aenv.} \\
\text{Layout env env} \quad \text{-- scalar environment} \\
\rightarrow \text{Layout aenv aenv} \quad \text{-- array environment} \\
\rightarrow \text{Exp t} \quad \text{-- expression to be converted} \\
\rightarrow \text{AST.OpenExp env aenv t}
\]

\[
\text{convertExp :: Layout aenv aenv} \quad \text{-- array environment} \\
\rightarrow \text{Exp t} \quad \text{-- expression to be converted} \\
\rightarrow \text{AST.Exp aenv t}
\]
Unary Operations and Binary Operations

Unary scalar operations — such as the combination functions of Map operations, and the permutation functions of Permute and Backpermute operations — introduce one bound variable, and binary scalar operations — such as the combination functions of ZipWith, Fold, FoldSeg, Scan, and Permute operations — introduce two bound variables. The functions convertFun1 and convertFun2 convert these unary and binary operations into a function, taking care of the bound variables with λ-abstraction:

```haskell
data OpenFun env aenv t where
    Body :: OpenExp env aenv t → OpenFun env aenv t
    Lam :: Elem a ⇒ OpenFun (env, ElemRepr a) aenv t → OpenFun env aenv (a → t)

convertFun1 :: forall a b aenv. Elem a ⇒ Layout aenv aenv → (Exp a → Exp b) → AST.Fun aenv (a → b)

convertFun2 :: forall a b c aenv. (Elem a, Elem b) ⇒ Layout aenv aenv → (Exp a → Exp b → Exp c) → AST.Fun aenv (a → b → c)
```

The function convertFun1 defines a variable tag, Tag 0, for the operation parameter to the unary scalar operation, f, and applies f to the variable tag. It also defines the environment layout, lyt, for scalar variable bindings: EmptyLayout ‘PushLayout‘ ZeroIdx. Then it calls convertOpenExp to convert f (Tag 0) to the De Bruijn representation in the scalar environment lyt and the array environment alyt inherited from the collective array operation that f parameterises. The converted scalar operation is passed to Body to build a function body, which is, in turn, passed to Lam to complete the λ-abstraction.

The function convertFun2 converts binary operations in the same way that convertFun1 converts unary operations, except that it defines two variable tags,
Tag 1 and Tag 0, and it defines one more variable index in the scalar environment. The integer values associated with the tags are the numerical representation of the De Bruijn indices.

Variable Indices

The variable tags injected to the scalar operation AST are converted to variable expressions with De Bruijn indices. The function `convertOpenExp` pattern-matches on a variable tag to extract the numerical variable index, \( n \), and constructs a variable expression with the \( n^{th} \) De Bruijn index from the scalar environment layout that it received from `convertFun1` or `convertFun2`.

For example, Tag 1 and Tag 0 are converted to `AST.Var (SuccIdx ZeroIdx)` and `AST.Var ZeroIdx` respectively in the environment layout:

```haskell
... 'PushLayout' (SuccIdx ZeroIdx) 'PushLayout' ZeroIdx
```

Constants

Similarly to the conversion of variable tags, `convertOpenExp` pattern-matches on a `Const` expression of type `Exp t` to extract the constant of type `t` in the plain Haskell representation, and constructs the corresponding `AST.Const` expression of type `OpenExp env aenv t`. We cannot, however, use the extracted Haskell constant directly as the parameter to `AST.Const` because `AST.Const` takes a value represented in the internal data representation explained in Section 3.2.1. To address the representation mismatch, `convertOpenExp` converts the Haskell constant to the internal data representation, using `fromElem`, then constructs the constant expression of type `OpenExp env aenv t`.

Tuples and Tuple Projections

The function `convertOpenExp` extracts the tuple of type `Tuple Exp (TupleRepr t)` from a `Tuple` expression, and converts the tuple to the corresponding tuple of type `Tuple (OpenExp env aenv) (TupleRepr t)`, using another conversion function `convertTuple`:
data OpenExp env aenv t where

  -- Variable index, ranging only over tuples or scalars
  Var :: Elem t
  ⇒ Idx env (ElemRepr t)
  → OpenExp env aenv t

  -- Constant values
  Const :: Elem t ⇒ ElemRepr t → OpenExp env aenv t

  -- Tuples
  Tuple :: (Elem t, IsTuple t)
  ⇒ Tuple (OpenExp env aenv) (TupleRepr t)
  → OpenExp env aenv t
  Prj :: (Elem t, IsTuple t)
  ⇒ TupleIdx (TupleRepr t) e
  → OpenExp env aenv t
  → OpenExp env aenv e

  -- Conditional expression (non-strict in 2nd and 3rd argument)
  Cond :: OpenExp env aenv Bool
  → OpenExp env aenv t
  → OpenExp env aenv t
  → OpenExp env aenv t

  -- Primitive constants
  PrimConst :: Elem t
  ⇒ PrimConst t → OpenExp env aenv t

  -- Primitive scalar operations
  PrimApp :: (Elem a, Elem r)
  ⇒ PrimFun (a → r)
  → OpenExp env aenv a
  → OpenExp env aenv r

  -- Project a single scalar from an array
  -- the array expression cannot contain any free scalar variables
  IndexScalar :: OpenAcc aenv (Array dim t)
  ⇒ OpenExp env aenv dim
  → OpenExp env aenv t

  -- Array shape
  -- the array expression cannot contain any free scalar variables
  Shape :: Elem dim
  ⇒ OpenAcc aenv (Array dim e)
  → OpenExp env aenv dim

Figure 3.9: The Accelerate scalar expressions in the De Bruijn representation [5]
convertTuple :: Layout env env
  -> Layout aenv aenv
  -> Tuple Exp t
  -> Tuple (AST.OpenExp env aenv) t

convertTuple _lyt _alyt NilTup = NilTup
convertTuple lyt alyt (es 'SnocTup' e)
  = convertTuple lyt alyt es 'SnocTup' convertOpenExp lyt alyt e

which converts each expression of type Exp τ in the tuple to the corresponding expression of type OpenExp env aenv τ, using convertOpenExp.

A Prj expression carries a tuple index and a Tuple expression. The function convertOpenExp reuses the tuple index from the given Prj expression to build the corresponding AST.Prj expressions, and it converts the Tuple expression to the AST.Tuple expression by calling itself recursively with the Tuple expression, and passes the resulting expression to the data constructor AST.Tuple.

Conditional Expressions

The function convertOpenExp simply extracts the predicate scalar expression of type Exp Bool, the consequent scalar expression of type Exp t, and the alternative scalar expression of type Exp t, using pattern-matching. It runs itself recursively with all three scalar expressions, and constructs the AST.Cond expression with the resulting expressions.

Primitive Constants

As Accelerate primitive constants are of type PrimConst t regardless of the representation used by scalar expressions, the function convertOpenExp pattern-matches the given PrimConst expression of type Exp t to extract the primitive constant of type PrimConst t, and reuses the extracted primitive constant to construct the AST.PrimConst expression.
Primitive Scalar Operations

PrimApp expressions carry two parameters: a primitive scalar operator and an argument, to which the primitive scalar operator is applied. As primitive scalar operators are independent of the representation of the scalar expression, the function convertOpenExp converts only the argument to the De Bruijn representation by applying convertOpenExp recursively, before constructing the AST.PrimApp expression that corresponds to the given PrimApp expression. Then convertOpenExp constructs the AST.PrimApp expression with the primitive scalar operator directly from the PrimApp expression and the argument converted to the De Bruijn representation.

Array Indexing

The function convertOpenExp converts the collective array operation or the array that is indexed and the index expression from the given IndexScalar expression to the De Bruijn representation using convertOpenAcc and itself respectively. Then it constructs the AST.IndexScalar expression with the collective array operation and the scalar operation from the conversions.

Shape

Given a Shape expression, convertOpenExp converts the collective array operation or the array that the expression carries to the De Bruijn representation using convertOpenAcc, and constructs the AST.Shape expression with it.
Chapter 4

CUDA Back End

In the commodity CPU market, application portability is usually not that big of an issue anymore, because independent of the manufacturers, we have a dominant architecture, x86. This is not at all the case for GPUs. This is not likely to change soon, as it is still a very fluid market. Obviously, we cannot ask the user to recompile an application for every concrete architecture they want to execute it on, nor does it make sense to compile it for the ‘lowest denominator’.

It is needless to say that the binaries compiled for GPUs from different manufacturers are incompatible, and there are cases where we see binary incompatibility between GPU generations even within the same manufacturer. This is because GPU architectures are evolving rapidly and the manufacturers keep adding new features to their GPU architectures. The case of the CUDA-compatible NVIDIA GPUs illustrates the architecture incompatibility issue. Although the NVIDIA GPUs based on G80 architecture or later are CUDA-compatible, they have different compute capabilities and they support different feature sets. CUDA binaries (CUBINs) are not backward-compatible, and have to be recompiled to run on previous generation GPUs. It requires intervention from end-users who may or may not possess adequate knowledge for this task. Compiling CUDA code for the lowest compute capability cannot be the optimal solution to this issue either, as it may hinder end-users from utilising new features on the latest GPUs. For example, double-precision operations are demoted to single-precision operations.
if compiled for compute capability 1.2 or below, even when they are executed on GPUs with compute capability 1.3 or above, which support double-precision.

The fact that we cannot assume the presence of a GPU increases the development and deployment complexity in many heterogeneous environments. If the collective array operations were to be extracted from the source code by a pre-processor, compiled by the compiler tool-chain for a particular target device, and linked statically against the host code, the executable would be runnable only on the system with the target device, lacking the ability to adapt itself to other systems. For instance, the collective array operations compiled statically for compute-capable GPUs cannot run on the systems without such GPUs, and vice versa. That is, running the same application in a heterogeneous environment requires multiple executables to cover the various system configurations present in the environment.

The online compilation scheme outlined in this chapter, which generates code and compiles it at application-runtime, is implemented in the CUDA back end of Accelerate. The CUDA back end queries the compute capabilities of the available GPUs before launching the online compiler, and it solves the portability issue.
among different GPUs. Furthermore, as the online compiler is implemented in one of the Accelerate back ends, it is possible to enable the dynamic switching to other back ends in the absence of compute-capable GPUs.

This compilation scheme only requires passing of the Accelerate ASTs to an appropriate back end, as Figure 4.1 illustrates, thus reducing the maintenance cost for programmers. It also liberates the end-user from compiler manipulations and from the burden of organising multiple executables.

The remainder of this chapter outlines the online compilation scheme and explains the strategies we use to minimise the overhead incurred by the online compilation. After that, we discuss and explore the skeleton-based code generation more in detail.

### 4.1 Compilation Scheme

As domain specific languages with deep embedding in Haskell are usually supplied with a function that performs the actual computation with the given computation AST, the CUDA back end provides such a function, called `run` —or `CUDA.run` to distinguish it from the `run` function in other back ends— and presented in Figure 4.2. It is also the only entry point to the CUDA back end and the only function that is exported from the CUDA back end. The rest of the back end does not directly interact with the front end. Collective array operations are passed to `CUDA.run` as the computation AST in the HOAS representation \textsuperscript{44}, and `CUDA.run` returns the result of the operations. The interactions with CUDA —e.g., querying the compute-capability, data transfers between the host memory and the device memory, \texttt{nvcc} execution, synchronisations, etc.— are encapsulated behind the scene.

When a computation AST in the HOAS representation is given to `CUDA.run` as the input, the CUDA back end first converts the computation AST to the corresponding computation AST in the De Bruijn representation, using `convertAcc`, as shown in Figure 4.2 the Accelerate front end defines the function `convertAcc`,
run :: Arrays a ⇒ Acc a → a
run acc
  = unsafePerformIO
    $ evalCUDA (execute (convertAcc acc) >>= collect)
    'catch'
    λ e → INTERNAL_ERROR(error)
    "unhandled" (show (e :: CUDAException))
execute :: Arrays a ⇒ OpenAcc () a → CIO a
execute acc = compileAcc acc >>= executeAcc acc

Figure 4.2: run in the CUDA back end

and is accessible from any Accelerate back end. Then the CUDA back end traverses the computation AST in post-order twice: first with compileAcc for the data transfer and compilation phase, and again with executeAcc for the execution phase. During the first traversal, the CUDA back end issues an asynchronous data transfer of the associated input array from the host memory to the device memory, when it encounters a use operation. For data transfers, the CUDA back end assumes that the data fits in the device memory, requiring programmers to stage the computation if needed. As for other nodes, the CUDA back end generates the device code module for the array operation, and spawns an nvcc process to compile it to the CUBIN module. Upon the completion of the first traversal, the CUDA back end traverses the computation AST in post-order again to load and execute the CUBIN modules, observing the data dependency among the operations. Note that both traversals ignore the leaves of the AST, which are either scalar operations processed as part of the parent array operation, or host arrays handled by the parent use operation.

For instance, Figure 4.3 shows (a) dotp, an implementation of the dot-product, written in Accelerate, and (b) how Accelerate internally represents dotp as an AST. The CUDA back end traverses the AST twice. During the first traversal, the CUDA back end (1) transfers xs to the device memory asynchronously, (2) transfers ys to the device memory asynchronously, (3) generates the device code
module for \texttt{zipWith} and spawns an \texttt{nvcc} process to compile it to a CUBIN module, and (4) generates the device code module for \texttt{fold} and spawns another \texttt{nvcc} process to compile it to a CUBIN module. The AST is traversed again. During the second traversal, the CUDA back end does nothing at \texttt{use}_1 and \texttt{use}_2.

At \texttt{zipWith}, the CUDA back end waits for the completion of the \texttt{nvcc} process that has been compiling the \texttt{zipWith} device code module, then, dynamically loads and executes the \texttt{zipWith} CUBIN module. When the execution of the \texttt{zipWith} operation is completed and the other \texttt{nvcc} process finishes compiling the \texttt{fold} device code module, the CUDA back end loads and executes the \texttt{fold} CUBIN module, passing the intermediate result from the \texttt{zipWith} operation as an input. Finally, the function \texttt{collect} in the CUDA back end, invoked in Figure 4.2, marshals the result from \texttt{fold} operation back to the host memory. The CUDA scheduler guarantees that the asynchronous data transfers triggered earlier with \texttt{x}_s and \texttt{y}_s are completed before the execution of the \texttt{zipWith} operation.

The function \texttt{evalCUDA}, used in Figure 4.2, is responsible for initialising and finalising the internal state of the CUDA back end, using a state monad \texttt{CIO} and the state data structure \texttt{CUDAState}. The state monad \texttt{CIO} is defined as

\begin{verbatim}
    type CIO = StateT CUDAState IO
\end{verbatim}

and we will explore the important uses of \texttt{CUDAState} in Section 4.2.

**Dynamic Linking**

The CUDA back end loads the CUBIN modules dynamically using the Haskell binding to the CUDA driver API \cite{cuda_api}, which has the capability of loading and unloading CUBIN modules without restarting the application \cite{cubin_api}. The CUDA back end carries the host code written in the Haskell binding to the CUDA driver API, instead of generating the host code in plain CUDA and the foreign import declarations in Haskell. As a result, the CUDA back end generates, compiles, and loads the device code only, keeping the runtime overhead as small as possible.
dotp :: Vector Float → Vector Float → Acc (Scalar Float)
dotp xs ys =
    let xs' = use xs
        ys' = use ys
    in fold (+) (constant 0.0) (zipWith (*) xs' ys')

(a)

(b)

Figure 4.3: (a) dot-product written in Accelerate, and (b) its AST representation
Execution Configuration

For the execution of the CUBIN modules, it is important to figure out the optimal grid size and the optimal block size for each CUBIN module. Even with the same CUBIN module, the performance varies depending on these configurations, and the optimal configurations vary depending on the GPU architectures. When programming in plain CUDA, programmers are given the possibility to inspect the available resources of the underlying hardware — e.g., the number of registers, the maximum amount of shared memory, the maximum number of threads per block, etc. — and the resource usage of each module, and they can figure out the optimal configuration for the execution of a particular CUBIN module on a specific GPU architecture with the obtained information, using NVIDIA’s CUDA occupancy calculator. As the CUDA back end operates the compilation scheme behind the scene without any human intervention, the CUDA back end profiles the underlying GPU architecture and the resource usage of each CUBIN module in the execution phase, thus figuring out the optimal execution configuration by passing the relevant information to the Haskell implementation of the CUDA occupancy calculator.

The Haskell implementation of the CUDA occupancy calculator is a part of the Haskell cuda package \[49\], and it has been implemented upon my request. The Haskell implementation of the CUDA occupancy calculator tries out a number of configurations, compares the occupancies defined by those configurations, and returns the configuration that is likely to be optimal, and the CUDA back end adopts that configuration.

4.2 Minimising the Runtime Overhead

The data transfers between the host memory and the device memory are one of the most expensive operations in the applications which use GPUs as co-processors, due to the latency of the PCIe bus on which the data transfers take place. Even when programming GPUs in CUDA without any high-level abstraction, the data
type MemTable = HashTable WordPtr MemoryEntry

data CUDAState = CUDAState
{
  ...
  _memoryTable :: MemTable
  ...
}

data MemoryEntry = MemoryEntry
{
  _refcount :: Int,
  _memsize :: Int64,
  _arena :: WordPtr
}

Figure 4.4: The data structure that keeps track of the device memory uses transfers often place significant runtime overhead especially in data-intensive applications. A naïve implementation of the compilation scheme may exacerbate the situation by introducing redundant data transfers. Avoiding this is one of the critical issues in achieving the optimal performance.

In addition to the runtime overhead caused by the hardware limitations, the online compilation scheme presented in this chapter introduces other sources of runtime overhead such as generation and compilation of device code, which offline compilers do not perform at application-runtime.

The rest of this section explains how these issues are addressed in the CUDA back end to minimise the runtime overhead.

4.2.1 Data Transfers

As in all purely functional languages, variables in Accelerate are immutable. Consequently, the Accelerate ASTs given by the front end do not depend on the mutability of the input arrays, ensuring that it is acceptable to transfer an array to the device memory once and use the data on the device memory multiple times.

A naïve implementation of the compilation scheme presented in Section 4.1
would trigger multiple data transfers with the same array whenever it appears more than once in the AST, as a data transfer is triggered for each node with a use operation.

This is wasteful as the CUDA back end and the CUDA memory architecture guarantee that the device code does not make changes to the input arrays and the global memory on the device is persistent within the same application. Reflecting the nature of the functional programming, the device code generated by the CUDA back end never makes changes to the input arrays—the details on the code generation are presented in Section 4.3.

Removing redundant data transfers reduces the runtime overhead especially for the computations which reuse the input arrays many times. To implement this, we need to keep track of the data transfers from the host memory to the device memory, and for which we use the hash table _memoryTable in Figure 4.4.

We use the address of Haskell unboxed arrays of type UArray Int a, which are internal to Accelerate arrays, as the key. Each value in the hash table contains the reference counter, _refcount, the size of the array, _memsize, and the associated device memory pointer, _arena. The CUDA back end manages the device memory as follows with this hash table:

- Before allocating the device memory for a Haskell unboxed array during the first traversal of the computation AST, the CUDA back end looks up the entry for the array in _memoryTable.
  - If the entry is not found, the CUDA back end allocates the device memory, creates the entry in _memoryTable, and sets the reference counter to 0.
  - Otherwise, the CUDA back end does nothing.

- Before triggering the data transfer of a Haskell unboxed array during the first traversal of the computation AST, the CUDA back end inspects the reference counter in the _memoryTable entry for the array.
– If the reference counter of the array is 0, the CUDA back end triggers the data transfer of the array, and increments the reference counter.
– Otherwise, the CUDA back end only increments the reference counter.

• After executing an array operation during the second traversal of the computation AST, the CUDA back end decrements the reference counter of the Haskell unboxed arrays that are internal to the input Accelerate arrays.

– If the reference counter of a Haskell unboxed array reaches 0, the CUDA back end frees the device memory, and removes the relevant entry from \texttt{\_memoryTable}, ensuring that the array is no longer needed.

The following example illustrates a situation where an input array is used more than once:

\begin{verbatim}
let xs' = use xs
...
in zipWith f xs' (map g xs')
\end{verbatim}

It makes use of \texttt{xs} twice, and naïve implementation of the compilation scheme would transfer \texttt{xs} twice. However, the approach described above ensures that \texttt{xs} is transferred only once, thus reducing avoidable runtime overhead.

The CUDA back end triggers the data transfer of \texttt{xs} only at the node \texttt{xs'} associated with the \texttt{map} operation during the data transfer and compilation phase, and sets the reference counter of \texttt{xs} to 2 at the end of the data transfer and compilation phase. During the execution phase, the CUDA back end decrements the reference counter of \texttt{xs} after the execution of \texttt{map g xs'} and once again after the execution of \texttt{zipWith f xs'} (\ldots). The CUDA back end frees the device memory associated with \texttt{xs} at the end of the execution phase as the reference counter reaches 0.

### 4.2.2 Compilations of the Device Code

The cost to compile the device code is another kind of overhead incurred at application-runtime. However, this is different from the runtime overhead associated with data transfers in that it is introduced by the online compilation scheme,
Figure 4.5: The data structure that keeps track of the compiled modules

As with data transfers, a naïve implementation of the compilation scheme presented in Section 4.1 would generate the device code of the same operation, and compile it to a CUBIN module as many times as the operation appears in the AST. The approach to minimise the compilation cost of the device code is very similar to the approach to minimise the data transfer cost described in Section 4.2.1: compile each kernel only once and reuse the CUBIN module as much as possible.

To address this issue, we define the data structure in Figure 4.5 to keep track of what modules the CUDA back end has compiled already. To this end, we maintain a hash table. Each key consists of the string representation of the array operation kind, the scalar operation, the type information, and the array operation, etc. Each value consists of the module name, the ID of the nvcc process if the module is being compiled or the CUDA module object if the module has been compiled already.

The CUDA back end manages the device code generation and the compilation as follows during the first traversal of the input AST:
• Whenever the CUDA back end visits a node with a collective array operation, it creates the key with the information given by the operation, and looks up the entry with the key in _kernelTable.

  – If the key already exists in the hash table, we assume that we have already compiled the same operation, and no further action is required for the current node.

  – Otherwise, the CUDA back end generates the device code is generated for the operation with a unique module name, spawns an nvcc process to compile the generated device code to the CUBIN module, and the key and the module information are added to the hash table.

For instance, the following computation makes use of the same scalar operation, f, three times:

```plaintext
let xs' = use xs
   ys' = use ys
   zs' = use zs
...
in fold f x (zipWith f xs' (zipWith f ys' zs'))
```

A naïve implementation of the compilation scheme would generate three device code modules: one for fold f and two for zipWith f. Then it would compile all of them to the CUBIN modules. As the two device code modules generated for zipWith f must be identical, it is meaningless to generate and compile the same device code module more than once.

The approach described above reduces the runtime overhead by minimising the device code generation and the compilation. The CUDA back end generates and compiles the device code module for zipWith f once while visiting the node with zipWith f ys' zs', and adds the information about the device code module to the hash table. It does not generate the device code module for zipWith f again at the node with zipWith f xs' (...) as it finds the information about the identical device code module in the hash table.
4.2.3 Overlapping Data Transfers, Compilations, and Executions

In Section 4.2.1 and in Section 4.2.2, we explored ways to minimise the number of the data transfers, and the number of the device code modules that the CUDA back end generates and compiles. Another way to reduce the runtime overhead is to overlap the data transfers, the code generation and compilation, and the code execution. The idea is depicted in Figure 4.6. Given the dotp in Figure 4.3, if no overlap is implemented at all, the runtime cost of the dotp would be as follows:

\[
R_{\text{fold}}(+) \circ (\text{zipWith} \ (\cdot) \ x s' \ y s') = T_{xs'} + T_{ys'} + C_{\text{zipWith}}(\cdot) + N_{\text{zipWith}}(\cdot) + C_{\text{fold}}(+) + N_{\text{fold}}(+) + E_{\text{zipWith}}(\cdot) + E_{\text{fold}}(+) + T'
\]

- \(R_{\text{comp}}\): the cost of running \texttt{comp} on the device
- \(C_{\text{op}}\): the device code generation cost for \texttt{op}
- \(N_{\text{op}}\): the \texttt{nvcc} cost to compile the device code generated for \texttt{op}
- \(E_{\text{op}}\): the execution cost of the device code generated for \texttt{op}
- \(T_{\text{use}}\): the cost of transferring the array data associated with \texttt{use}
  to the device memory
- \(T'\): the cost of transferring the final result back to the host memory

Even when writing the same computation in plain CUDA, the runtime overhead associated with the data transfers and the device code executions is unavoidable, whereas the runtime overhead associated with the device code generation and compilation does not exist. That is, the runtime overhead from the code generation and compilation is what we need to minimise. Moreover, the \texttt{nvcc} execution often takes longer than the data transfers and/or the device code executions, and creates a bottleneck.

The compilation scheme presented in Section 4.1 takes advantage of asynchronous data transfers and the concurrent execution of the \texttt{nvcc} processes to cancel out the runtime overhead associated with the code generation and compilation as much as possible.
CUDA manages asynchronous operations such as asynchronous data transfers and the device code execution, using *streams*. Simply put, a stream is a queue for the asynchronous operations, and there always exists at least one stream in any CUDA application. When multiple streams exist in an application, the order in which they are executed is undefined. However, it is guaranteed that the operations in the same stream are always executed in order.

The CUDA back end triggers the data transfers from the host memory to the device memory are triggered, using non-blocking asynchronous function calls, and the control returns to the host side as soon as the requests for the data transfers are placed in the default stream. It allows the CUDA back end to generate and compile the device code modules on the host side, whilst the actual data transfers are in progress. This overlap cancels out some of the runtime overhead associated with the code generation and compilation.

The CUDA back end further minimises the runtime overhead by executing *nvcc* processes concurrently. The CUDA back end does not wait for the *nvcc* processes it spawned during the data transfer and compilation phase. Instead, it only puts the process IDs in the state and it defers waiting for the completion of the processes until the CUBIN modules are needed by the execution phase, thus allowing multiple *nvcc* processes run concurrently. On multi-core CPUs, this strategy reduces the cost of the compilation phase.

Before executing a particular operation in the execution phase, the CUDA back end waits only for the *nvcc* process which has been compiling the device code for the operation. Once the *nvcc* process finishes compiling the device code for the operation to a CUBIN module, a CUDA module object is created by loading the CUBIN module into the runtime system, and the process ID of the *nvcc* process in the state is replaced with the CUDA module object so that the module object may be reused if the same operation appears again in the AST. The CUDA back end leaves the other *nvcc* processes untouched. Figure 4.6 shows that the CUDA back end ensures the completion of the *nvcc* process that has been compiling the device code for the *zipWith (*)* operation before executing the *zipWith (*)* operation on the GPU, but leaves the other *nvcc* process that
Figure 4.6: The timeline of data transfers, code generation, compilation, and execution for the dot-product in Figure 4.3

has been compiling the device code for the fold (+) operation running. That is, the device code compilation and the device code execution overlap partially, thus cancelling out some of the runtime overhead introduced by the code generation and compilation.

In the best case scenario, the runtime overhead incurred by the compilation phase would be completely overlapped by the data transfers and the code execution, and the runtime cost of \texttt{dotp} would become:

\[
R_{\text{fold (+) 0 (zipWith (+) xs' ys')}} = T_{\text{xs'}} + T_{\text{ys'}} + E_{\text{zipWith (+)}} + E_{\text{fold (+)}} + T'
\]

The following section of this chapter presents another technique that can be used in conjunction with the technique presented in Section 4.2.2 to help increase the chance for the best case scenario to happen.
4.2.4 Caching of the Compiled Modules

Although it is possible to reuse the compiled CUBIN modules within a computation AST with the technique discussed in Section 4.2.2, the technique does not expand across multiple computation ASTs. That is, it cannot accommodate the situation where an application has more than one computation AST and these trees have common operations in them.

For example, the following function, \texttt{sum}, builds a computation AST with a single operation, \texttt{fold (+)}, which is also found in the computation AST of \texttt{dotp} in Figure 4.3.

\begin{verbatim}
sum :: Vector Float \rightarrow Acc (Scalar Float)
sum xs =
    let xs' = use xs
    in    fold (+) (constant 0.0) xs'
\end{verbatim}

Ideally, the device code for \texttt{fold (+)} would be generated and compiled once, and used by both \texttt{dotp} and \texttt{sum}. However, when these two computation ASTs are passed to \texttt{CUDA.run} separately as below, the device code for \texttt{fold (+)} is generated and compiled twice: once for \texttt{dotp} and once for \texttt{sum}.

\begin{verbatim}
main :: IO ()
main = do
    let xs = ... :: Vector Float
        ys = ... :: Vector Float
        zs = ... :: Vector Float
    r1 = CUDA.run (dotp xs ys)
    r2 = CUDA.run (sum zs)
    ...
\end{verbatim}

The control enters the CUDA back end twice, first for \texttt{r1} and again for \texttt{r2}. The CUDA back end is given an empty state with each \texttt{CUDA.run}. That is, the CUDA back end does not remember what happened previously with \texttt{CUDA.run}.
(dotp xs ys) when the control enters the CUDA back end for the second time with CUDA.run (sum zs).

It is possible to change CUDA.run with the type signature below to relay the final state from one CUDA.run to the next as the initial state:

\[
\text{CUDA.run :: Arrays } a \\
\Rightarrow \text{Maybe CUDAState } \rightarrow \text{Acc } a \rightarrow (a, \text{CUDAState})
\]

This approach, however, introduces another issue. It does not only expose CUDAState, which is internal to the CUDA back end, to programmers, but also breaks the convention all other back ends adhere to, thus requiring programmers to remember that CUDA.run works differently than run in other back ends.

It is also impractical to define a universal state managed by the front end as the front end has no information about the back ends which would come in the future and all back end maintainers would be required to limit the features in the back end or to expand the universal state for their needs. It would result in an unacceptable growth in the maintenance complexity.

Hence, we cache the compiled modules to share them across multiple invocations of CUDA.runs. Basically, the final state is encoded to a file using Data.Binary at the end of each CUDA.run, and the final state from the previous CUDA.run is reconstructed from the file at the beginning of each CUDA.run. The reconstructed state is used as the initial state instead of an empty state unless the file does not exist and the reconstructed state contains nothing.

The final state of CUDA.run (dotp xs ys) in the example above contains two CUBIN modules—one for zipWith (*) and the other for fold (+). The CUDA back end reconstructs this state and uses as the initial state of CUDA.run (sum zs) in the example above. It makes the CUDA back end aware of the existence of the CUBIN module for fold (+).

Furthermore, the file that keeps the final state, the generated device code, and the CUBIN modules are kept in the file system in a way that they can be reused when the application is launched again. That is, the device code generation and the compilation are only performed during the first execution of the application. From the second execution, the CUDA back end only needs to
look up the necessary CUBIN modules from the cache, thus reducing the runtime overhead.

With this approach, the device code for each operation is generated and compiled only once, but used as many times as needed in an application. It does not introduce any changes to the front end, and it does not break the convention that all back ends are required to conform to.

Although the CUDA back end implements an online compilation scheme, which may incur extra runtime overhead compared to offline compilation schemes, the compilation scheme and the techniques outlined in this chapter encapsulate the extra runtime overhead to increase the chance for the best case scenario to happen, which is presented in Section 4.2.3. In the best case scenario, the efficiency of the generated device code determines the performance discrepancy between the hand-crafted and the Accelerate CUDA applications. The following section explains the code generator in the CUDA back end.

4.3 Code Generation

Accelerate is a two-level language, and the structure of the computation ASTs given by the front end reflects the two-level nature of the language. The ASTs built by the language of scalar operations and the ASTs built by the language of collective array operations are of different types. Scalar operation ASTs are of \texttt{Exp t} type and collective array operations are of \texttt{Acc a} type when represented in HOAS. And scalar operation ASTs are of \texttt{OpenExp env aenv t} type and collective array operations are of \texttt{OpenAcc aenv t} type when the De Bruijn representation is used.

Scalar operation ASTs alone cannot form the computation ASTs passed to the back ends, as \texttt{run} in the back ends only accepts an AST of \texttt{Acc a} type, according to the convention which all back ends must conform to. Collective array operations require parameterisation with scalar operations of \texttt{Exp t} type, and the Haskell type system ensures that collective array operations cannot be parameterised with collective array operations.
Chapter 4. CUDA Back End

The CUDA back end converts the input AST in the HOAS representation to the equivalent AST in the De Bruijn representation before processing the input any further. That is, the type of the collective array operations in the input AST changes from \texttt{Acc c} to \texttt{OpenAcc aenv t}, and the type of the scalar array operations, which parameterise the collective operations in the input AST, changes from \texttt{Exp t} to \texttt{OpenExp env aenv t}.

The code generator of the CUDA back end also reflects the two-level nature of the language, and the code generation is centred around these two functions, which take computation ASTs in the De Bruijn representation as the input:

\begin{verbatim}
  type CG a = State [CExtDecl] a
  codeGenAcc' :: OpenAcc aenv a → CG CUTranslSkel
  codeGenFun :: OpenFun env aenv t → CG [CExpr]
  codeGenExp :: forall env aenv t. OpenExp env aenv t → CG [CExpr]
\end{verbatim}

\texttt{codeGenAcc'} translates each collective array operation of \texttt{OpenAcc aenv t} type to a CUDA device code module AST of type \texttt{CUTranslSkel}, embedding the CUDA expression list of type \texttt{[CExpr]} that \texttt{codeGenFun} or \texttt{codeGenExp} generated from the scalar operation of the \texttt{OpenFun env aenv t} type or the \texttt{OpenExp env aenv t} type, which is passed to the collective array operation.

4.3.1 Collective Array Operations

The code generation of collective array operations is based on algorithmic skeletons. Each collective array operation has a matching algorithmic skeleton predefined, and the device code for a collective array operation is generated by the instantiation of the algorithmic skeleton for the operation.

Each algorithmic skeleton comprises one or more \texttt{global} functions, depending on the collective operation. Each \texttt{global} function defines an internal operation for the collective operation. For example, \texttt{map} has only one \texttt{global} function that applies the scalar operation to each element in the algorithmic skeleton whereas \texttt{scan} has two: one for \texttt{up-sweep} and the other for \texttt{down-sweep} \cite{43, 50, 51}. Each \texttt{global} function is responsible for three tasks:
• reading the input data from the device memory,

• applying the scalar operation to the input data according to a certain parallelism pattern, and

• writing the result to the device memory.

As there exist a number of attempts to implement collective array operations on GPUs efficiently, instead of re-inventing the wheel, the algorithmic skeletons have been created based on the algorithms from some of those existing work, including *CUDA Data Parallel Primitives Library* (CUDPP) \[52\] and *Thrust* \[53\].

The algorithmic skeletons used in the CUDA back end are written in such a way that all data transactions, both reading from and writing to the device memory, satisfy the memory coalescing requirements and the memory alignment and size requirements as far as the nature of the collective array operation allows; the data transactions in permute and backpermute cannot be coalesced as the data access patterns are not uniform. Also, the implementation of the algorithmic skeletons ensures that bank conflicts are avoided if shared memory access is present in the algorithmic skeleton.

Initially, all the algorithmic skeletons were defined as CUDA ASTs with some place holders for the data types, the scalar operations, etc., but the co-authors of the Accelerate package \[5\] improved it and the algorithmic skeletons are now defined as CUDA source files which can be included in the generated output file. The code generator builds CUDA ASTs using *language-c* library \[54\] only for the definitions of the type synonyms and the scalar operations, and pretty-print them in the generated output file along with \#include "skeleton_name.inl". As a consequence, the algorithmic skeleton code has become clearer and more readable, and the code generation is taking less time, as the number of AST traversals has been reduced.

Figure \[4.7\] shows one of the algorithmic skeletons from the CUDA back end, zipWith. The green terms are the type synonyms for the actual data types. ArrOut is the output array type, and ArrIn1 and ArrIn0 are the input array
extern "C"
__global__ void zipWith
{
  ArrOut d_out,
  const ArrIn1 d_in1,
  const ArrIn0 d_in0,
  const DimOut shOut,
  const DimIn1 shIn1,
  const DimIn0 shIn0
}
{
  const Ix shapeSize = size(shOut);
  const Ix gridSize = __umul24(blockDim.x, gridDim.x);
  for (Ix ix = __umul24(blockDim.x, blockIdx.x) + threadIdx.x;
       ix < shapeSize; ix += gridSize)
  {
    Ix i1 = toIndex(shIn1, fromIndex(shOut, ix));
    Ix i0 = toIndex(shIn0, fromIndex(shOut, ix));
    set(d_out, ix, apply(get1(d_in1, i1), get0(d_in0, i0)));
  }
}

\textbf{Figure 4.7:} The \texttt{zipWith} skeleton in the CUDA back end \cite{5}
types. DimOut, DimIn1, and DimIn0 are the type synonyms that describe the
dimension of the output array and the input arrays. They are simply one int32_t
for one-dimensional arrays, or a struct of n int32_ts for n-dimensional arrays,
where n > 1.

Section 3.2.1 notes that, even though Accelerate supports multi-dimensional
arrays, Haskell unboxed arrays that are internal to Accelerate arrays are always
one-dimensional. Ix in Figure 4.7 is the type for the internal one-dimensional
array indexing variable, and is a type synonym for int32_t. The indexing of
the array elements is controlled by the variables ix, i1, and i0 of type Ix, mak-
ing sure that the access pattern conforms to the memory coalescing, alignment
and size requirements. The function fromIndex converts an internal array in-
dex to the multi-dimensional array index with the given shape, and the function
toIndex converts a multi-dimensional array index with the given shape to the
internal array index. The CUDA back end does not generate fromIndex and
toIndex, but pre-defines them for up to 9-dimensional arrays in a header file,
accelerate_cuda_shape.h, that is included in the skeleton instances.

The getn function takes one of the input arrays of type d_inn, and the
index, and returns the selected value of the input array element type, TyInn,
from the input array. The set function takes the output array of type ArrOut,
the index, and the value of the chosen output array element type, TyOut, and
places the value at the designated position in the array. As the Accelerate front
end maintains an array of tuples as a tuple of arrays internally, getting an element
from an array of tuples requires reading an element from each internal array and
packing the elements into a tuple. That’s what the getn function is responsible
for. Likewise, setting an element in an array of tuples requires unpacking the new
tuple value into primitive values and assigning them to each internal array, and
set is responsible for this. Figure 4.8 illustrates it. For the arrays with primitive
type elements, the getn and set functions are simpler as there is no need for
packing and unpacking.

With these concepts and the algorithmic skeletons pre-defined, the code gen-
eration for collective array operations boils down to:
An array of $n$-tuples, represented as an $n$-tuple of arrays.

Figure 4.8: Reading an array element and updating an array element, using the $\text{get}_{n}$ and $\text{set}$ functions.
• the definition of the type synonyms required by the algorithmic skeleton,

• the generation of the CUDA __device__ functions, apply, identity, and project, from the scalar operation which parameterises the collective array operation,
  
  – The identity function is required for fold, scanl, and scanr operations.
  – The project function is required for backpermute and permute.

• the generation of other auxiliary __device__ functions such as getn and set, and

• the instantiation of the algorithmic skeleton with the type synonyms and the CUDA __device__ functions.

The rest of this section explains the definition of the type synonyms and the instantiation of the algorithmic skeleton further, and the next section is dedicated to the generation of the CUDA __device__ functions from the scalar operations.

The Definition of the Type Synonyms

The instantiation of an algorithmic skeleton requires the definition of several type synonyms in the generated CUDA source file, as explained earlier with Figure 4.7. These include ArrOut, ArrIn, DimOut, DimIn, TyOut, and TyIn.

The CUDA back end must translate the Accelerate types to the CUDA types correctly in order to define the necessary type synonyms. Figure 4.9 defines the type translation rules. The mapping between the Accelerate primitive types and the CUDA primitive types, denoted by the relation $\mapsto$ in Figure 4.9, is straightforward as there is a matching CUDA primitive type for each primitive type available in Accelerate. The Accelerate primitive types are denoted by $\text{eTy}(\tau)$ in Figure 4.9 and the CUDA primitive types by $\text{cTy}(\tau)$. The mapping between the Accelerate array types and the CUDA array types is also straightforward as
\[\mathcal{P} = \{\text{Int, Int}_n, \text{Word, Word}_n, \text{Float, Double}\} \quad \text{where} \quad n \in \{8, 16, 32, 64\}\]

**primitive types**

\[
\begin{aligned}
\tau \in \mathcal{P} & \quad \tau \in \mathcal{P}' & \quad n \in \{32, 64\} \quad n = \text{sizeof(Int)} & \quad \text{eTy(Word)} \rightarrow \text{cTy(uint)} \\
\text{eTy(\tau)} & \quad \text{cTy(\tau)} & \quad \text{eTy}\{\text{int, } \text{uint}, \text{uint}_n, \text{float, double, void}\} & \quad \text{eTy(\text{int}_n)} \rightarrow \text{cTy(\text{int}_n)}
\end{aligned}
\]

\[
\begin{aligned}
\tau \in \mathcal{P}' & \quad \tau \in \mathcal{P}' & \quad n \in \{32, 64\} \quad n = \text{sizeof(Int)} & \quad \text{eTy(\text{word})} \rightarrow \text{cTy(\text{uint})} \\
\text{eTy(\text{Float})} & \quad \text{cTy(\text{Float})} & \quad \text{eTy(\text{Double})} \rightarrow \text{cTy(\text{Double})}
\end{aligned}
\]

\[
\tau \in \mathcal{P}' \quad \text{cTy(\tau)} \leftrightarrow \tau
\]

**tuples**

\[
\begin{aligned}
\text{eTy}(\tau_0) & \quad \text{eTy}(\tau_{n-1}) & \quad \ldots & \quad \text{eTy}(\tau_1) & \quad \text{eTy}(\tau_0) \\
\text{eTy}(\tau_0, \tau_{n-1}, \ldots, \tau_1, \tau_0) & \quad \text{cTy}(\tau_0) & \quad \text{cTy}(\tau_{n-1}) & \quad \ldots & \quad \text{cTy}(\tau_1) & \quad \text{cTy}(\tau_0)
\end{aligned}
\]

\[
\text{eTy}(\ldots) \text{ snoc eTy}(\tau) \equiv \text{eTy}(\ldots, \tau) \\
\text{cTy}(\ldots) \text{ snoc cTy}(\tau) \equiv \text{cTy}(\ldots, \tau)
\]

\[
\begin{aligned}
\text{eTy}(\ldots) & \quad \text{eTy}(\ldots) \quad \tau_0 \in \mathcal{P} \quad \text{eTy}(\tau) \rightarrow \text{cTy}(\tau') \\
\text{eTy}(\ldots) & \quad \text{eTy}(\ldots) \quad \text{snoc eTy}(\tau) \rightarrow \text{cTy}(\ldots) \quad \text{snoc cTy}(\tau')
\end{aligned}
\]

\[
\begin{aligned}
\text{cTy}(\tau_0) & \quad \text{cTy}(\tau_{n-1}) & \quad \ldots & \quad \text{cTy}(\tau_1) & \quad \text{cTy}(\tau_0) \\
\text{cTy}(\tau_0, \tau_{n-1}, \ldots, \tau_1, \tau_0) & \quad \text{struct } \{ \tau_0 \ast \tau_0; \tau_0 \ast \tau_0 \ast \tau_0 \ldots; \tau_1 \ast \tau_1; \tau_1 \ast \tau_1 \ast \tau_1 \ ldots \}
\end{aligned}
\]

**arrays**

\[
\begin{aligned}
\text{eTy}(\tau) & \quad \text{cTy}(\tau) \\
\text{aTy}(\tau) \quad \text{cTy}[\tau] & \quad \text{aTy}(\ldots) \text{ snoc aTy}(\tau) \equiv \text{aTy}(\ldots, \tau) \\
\text{aTy}(\ldots) \rightarrow \{ \text{cTy}(\tau) \} & \quad \tau_0 \in \mathcal{P} \\
\text{aTy}(\ldots) \rightarrow \{ \text{cTy}(\tau') \}
\end{aligned}
\]

\[
\begin{aligned}
\text{aTy}(\ldots) & \quad \text{aTy}(\ldots) \quad \text{snoc aTy}(\tau) \rightarrow \{ \text{cTy}(\ldots) \text{ snoc cTy}(\tau') \}
\end{aligned}
\]

\[
\begin{aligned}
\text{aTy}(\ldots) & \quad \text{aTy}(\ldots) \quad \text{snoc aTy}(\tau) \rightarrow \{ \text{cTy}(\ldots) \text{ snoc cTy}(\tau') \}
\end{aligned}
\]

\[
\begin{aligned}
\tau \in \mathcal{P}' & \quad \text{cTy}(\tau_0) \quad \text{cTy}(\tau_{n-1}) \quad \ldots \quad \text{cTy}(\tau_1) \quad \text{cTy}(\tau_0) \\
\text{cTy}(\tau) \leftrightarrow \tau & \quad \text{cTy}(\tau_0, \tau_{n-1}, \ldots, \tau_1, \tau_0) \leftrightarrow \text{struct } \{ \tau_0 \ast \tau_0; \tau_0 \ast \tau_0 \ast \tau_0 \ldots; \tau_1 \ast \tau_1; \tau_1 \ast \tau_1 \ast \tau_1 \ ldots \}
\end{aligned}
\]

**Figure 4.9:** The code generation rules: the translation of the supported types
long as the element types remain primitive. The translation between the abstract syntax and the concrete syntax in CUDA is denoted by the relation $\leftrightarrow$.

For example, given an Accelerate collective array operation that maps a Float array to an Int32 array, the CUDA back end translates the element type of the input array to float, according to $eTy(\text{Float}) \mapsto cTy(\text{float})$ and $cTy(\text{float}) \leftrightarrow \text{float}$, and the input array type to float *, according to $aTy(\text{Float}) \mapsto [cTy(\text{float})]$ and $[cTy(\text{float})] \leftrightarrow \text{float *}$. The CUDA back end translates the output array element type and the output array type to int32_t and int32_t * in the same way. The type synonyms for these types are generated as follows:

```c
typedef int32_t TyOut; /* the output array element type */
typedef int32_t *ArrOut; /* the output array type */
typedef float TyIn0; /* the input array element type */
typedef float *ArrIn0; /* the input array type */
```

The Accelerate tuple types in the rules are denoted by $eTy(\tau_0, \ldots, \tau_{n-1}, \tau_n)$ where each tuple element type is an Accelerate primitive type or another Accelerate tuple type, and they are mapped to the corresponding CUDA struct types. The CUDA back end translates each primitive element type in the Accelerate tuple type to a field type in the CUDA struct type. The CUDA back end flattens nested tuples. That is, if a element type $\tau$ is another tuple type $\tau'$, the CUDA back end inserts the element types of $\tau$, instead of $\tau$, into $\tau'$.

The CUDA back end translates an Accelerate array of tuples to a CUDA struct of arrays. This is to match the internal structure of the Accelerate arrays; the Accelerate front end internally maintains an array of tuples as a tuple of arrays.

If a collective array operation is defined to produce an array of (Double, (Int32, Float)), the CUDA back end generates the type synonyms and the set function for the output array element type and the output array type in the CUDA source file as follows:
typedef struct {
    double a2; int32_t a1; float a0;
} TyOut; /* the output array element type */

typedef struct {
    double *a2; int32_t *a1; float *a0;
} ArrOut; /* the output array type */

static inline __device__ void set(ArrOut d_out, 
    const Ix idx, 
    const TyOut val)
{
    d_out.a2[idx] = val.a2;
    d_out.a1[idx] = val.a1;
    d_out.a0[idx] = val.a0;
}

The Selection and the Instantiation of the Algorithmic Skeletons

The selection of the algorithmic skeletons is based on simple pattern-matching. Figure 4.10 shows the pattern matching and the algorithmic skeleton instantiation implemented in codeGenAcc'. It inspects the given collective array operation from the computation AST, and invokes one of the skeleton instantiation functions in the CUDA.CodeGen.Skeleton module for the required algorithmic skeletons such as mkReplicate, mkIndex, mkMap, mkZipWith, mkFold, mkFoldSeg, mkScan, mkPermute, mkBackpermute, etc—the functions codeGenReplicate and codeGenIndex are the wrapper functions for mkReplicate and mkIndex respectively that also generate the CUDA scalar expression to take a slice of the input array.

The functions codeGenExpType and codeGenAccType translate the Accelerate scalar expression types and array element types to a list of the CUDA primitive types, according to the relation $\mapsto$ in Figure 4.9. The skeleton instantiation functions generate the struct type or a primitive type from the list of the CUDA primitive types, according to the relation $\leftrightarrow$ in Figure 4.9, then they generate the typedef such as TyOut, ArrOut, TyIn, or ArrIn with the generated type. The
codegenAcc' :: OpenAcc aenv a → CGCUTranslSkel

codegenAcc' op@(Replicate sl e1 a1)
  = codeGenReplicate sl e1 a1 op <* codeGenExp e1

codegenAcc' op@(Index sl a1 e1)
  = codeGenIndex sl a1 op e1 <* codeGenExp e1

codegenAcc' (Fold f1 e1 _)
  = mkFold (codegenExpType e1) <$> codeGenExp e1 <*> codeGenFun f1

codegenAcc' (FoldSeg f1 e1 _ s)
  = mkFoldSeg (codegenExpType e1) (codegenAccType s) <$> codeGenExp e1 <*> codeGenFun f1

codegenAcc' (Scan f1 e1 _)
  = mkScan (codegenExpType e1) <$> codeGenExp e1 <*> codeGenFun f1

codegenAcc' op@(Map f1 a1)
  = mkMap (codegenAccType op) (codegenAccType a1) <$> codeGenFun f1

codegenAcc' op@(ZipWith f1 a1 a0)
  = mkZipWith (codegenAccType op) (accDim op) (codegenAccType a1) (accDim a1)
      (codegenAccType a0) (accDim a0) <$> codeGenFun f1

codegenAcc' op@(Permute f1 _ f2 a1)
  = mkPermute (codegenAccType a1) (accDim op) (accDim a1) <$> codeGenFun f1 <*> codeGenFun f2

codegenAcc' op@(Backpermute _ f1 a1)
  = mkBackpermute (codegenAccType a1) (accDim op) (accDim a1) <$> codeGenFun f1

Figure 4.10: The selection and instantiation of the algorithmic skeletons
function `accDim` returns a value of the type `Int` that indicates the dimensions of the given Accelerate array, with which the instantiation function generates the typedef such as `DimOut` or `DimInn`. In addition to the typedefs, the skeleton instantiation functions also define the `getn` functions for `TyInn` and `ArrInn`, and the `set` function for `TyOut` and `ArrOutn`.

\[
\text{codegenAccType} :: \text{OpenAcc aenv (Array dim e)} \rightarrow \text{[CType]}
\]
\[
\text{codegenExpType} :: \text{OpenExp aenv env t} \rightarrow \text{[CType]}
\]
\[
\text{accDim :: forall aenv dim e. OpenAcc aenv (Array dim e) \rightarrow Int}
\]

The functions `codegenFun` and `codegenExp` translate the Accelerate scalar operation to the CUDA scalar expression list of type `[CType]`, as previously mentioned in Section 4.3. The skeleton instantiation functions, then, inject the CUDA expression list translated from the scalar operations into the required CUDA `__device__` functions, and include the relevant skeleton code in the CUDA source file.

For example, Figures 2.3 and 4.7 show that the `zipWith` skeleton requires `set` to set values in the output array, `get1` to get the values from the first input array, `get0` to get the values from the second input array, and `apply` to compute the values. The `zipWith` skeleton code is instantiated and becomes usable only when these functions are defined, as the invocations to these functions are made in the skeleton code.

### 4.3.2 Scalar Operations

Each scalar operation that parameterises collective array operations in Accelerate consists of a number of scalar expressions. The available scalar expressions are listed in Figure 3.9. This section explains how these expressions are translated to a list of the CUDA expressions. The translation of the scalar operation is centred around the function `codegenExp`, mentioned in Sections 4.3 and 4.3.1 and shown in Figure 4.11.
Variable Indices

In Section 4.1, I mentioned that the CUDA back end converts the computation AST built in the HOAS representation by the front end to the corresponding De Bruijn representation. After the conversion, each variable bound to the given scalar operation is represented by an index; the variable bound by the $i^{th}$ innermost binder is represented by the index, $i$, and the representation of the indices is explained in Section 3.3.1. For example, the scalar operation, $f$, in Figure 2.3 is represented with De Bruijn indices as:

$$\lambda (\lambda \text{ constant } 1.5 \ast (\text{SuccIdx ZeroIdx}) \ast (\text{ZeroIdx}))$$

When generating the CUDA code, the code generator simply translates the variable index of each scalar variable to a variable identifier, $x$, suffixed with the numerical representation of the index, and creates a singleton list with the variable identifier. That is, ZeroIdx and (SuccIdx ZeroIdx) are translated to $[x0]$ and $[x1]$ respectively. If the variable represented by the index is a tuple, instead of the singleton list with the variable identifier, the code generator creates a list of the CUDA expressions to access the tuple elements, placing these CUDA expressions in reverse order. For example, if ZeroIdx is a pair, $[x0.a1, x0.a0]$ is created. From this list, the relevant CUDA expressions are selected by the projection expressions, and are used by other scalar expressions.

As each scalar operation is a closed operation and there is no scalar expression that introduces a new variable binding, the De Bruijn representation can be further simplified and generalised. The parameters of the binary scalar operations are always represented with (SuccIdx ZeroIdx) and ZeroIdx, and the parameters of the unary scalar operations with ZeroIdx. Consequently, the parameters of the binary scalar operations are always generated as $x1$ and $x0$ in the CUDA code, and the parameter of the unary scalar operations as $x0$.

Constants

The code generator handles constants and scalar variable indices similarly. It creates a list with $n$ elements, where $n$ is 0 if the constant is a unit, 1 if the
constant is a single value, or the number of the elements if the constant is a tuple.

Boolean constants are treated as integer constants; True as 1 and False as 0. Float constants in the CUDA code is always explicitly typed — e.g., 1.5 in Accelerate is translated to float 1.5 — as they are recognised as double constants by the nvcc compiler otherwise. The code generator places other constants from Accelerate in the CUDA code without any change, ensuring the type correctness internally.

Tuples and Tuple Projections

Given a tuple, the code generator generates the CUDA expressions for the tuple elements, and places them in a list, and from which the code generator selects the CUDA expressions to be used by other scalar expressions according to the projection expression. For the nested tuples, the code generator works on the flattened representation described in Figure 4.9.

Each projection expression is parameterised with a projection index and a tuple. The projection indices are represented with SuccTupIdx and ZeroTupIdx; the projection with ZeroTupIdx yields the list of the expressions that belong to the last element in the tuple, (SuccTupIdx ZeroTupIdx) the list of the expressions that belong to the second last element by

- generating a list with every CUDA expression of the given tuple,
- converting the projection index to an integer, \( i \), that indicates the position of the last expression that belongs to the projected tuple element, and
- selecting the expressions at the positions \( (i - n, i) \), where \( n \) is the number of the expressions that belong to the projected tuple element.

Conditional Expressions

Each conditional expression is parameterised with three other scalar expressions: Cond \( p, e_0, e_1 \). The code generator translates \( p, e_0 \), and \( e_1 \) to the CUDA expressions,
\( p', e'_0, \) and \( e'_1, \) and generates the conditional expression, \( p' ? e'_0 : e'_1 \) in the CUDA code.

**Primitive Constants**

The code generator takes the values from `minBound`, `maxBound`, and `pi` on the Haskell side, builds the CUDA constant expression with the value, and creates a singleton list with the CUDA constant expression.

**Primitive Scalar Operations**

The `PrimApp` scalar expression is parameterised with a primitive scalar operator and the tuple of the primitive scalar operation arguments. The primitive scalar operators include arithmetic operators, relational and equality operators, logical operators, and trigonometric operators.

The code generator translates the primitive scalar operator to the CUDA operator or to the CUDA mathematical function, and the argument tuple to a list of the CUDA expressions, and builds the CUDA AST with them for the given primitive scalar operation. For example, when `PrimApp (PrimAdd numType) (e_1, e_0)` and `PrimApp (PrimCos floatingType) e` are given, the code generator translates \((e_1, e_0)\) and \(e\) to \([e'_1, e'_0]\) and \([e']\). Then it builds \(e'_1 + e'_0\), and \(\cos(e')\) or \(\cosf(e')\) depending on the type.

**Array Indexing**

The array indexing expression is different from other scalar expressions in that it takes a collective array operation as one of the parameters. As the algorithmic skeletons in the CUDA back end require the CUDA `__device__` functions, such as `apply` and `identity`, to accept only scalars as parameters, we need to expose the arrays that are indexed globally in the algorithmic skeleton instance.

To address this issue, the initial implementation of the CUDA back end generated the plain device memory pointer declarations with the file scope for the collective array operations that are indexed, and the co-authors of Accelerate[5]
later improved it with texture references to take advantage of the texture caches.

For each collective array operation that is indexed, the CUDA back end generates a declaration of a texture reference, and it binds the device memory allocated for the result of the collective array operation to the texture reference during the execution phase. For an array of tuples, which the Accelerate front end maintains as a tuple of arrays, the CUDA back end declares a texture reference for each internal array.

The CUDA back end generates the calls to the texture fetching function provided by the CUDA driver API, `tex1DFetch`, for the actual indexing. The function `tex1DFetch` takes a texture reference and a numerical index, $i$, as the parameters, and returns the $i^{th}$ element from the device memory region bound to the texture reference.

For example, the `identity` function in the algorithmic skeleton instance for:

```haskell
let xs' = use (xs :: Vector Word32)
in  fold (+) (xs' ! 0) xs'
```

is generated as below:

```c
TexWord32 tex0;
static inline __device__ TyOut identity()
{
TyOut r = indexArray(tex0, 0);
return r;
}
```

The CUDA back end pre-defines `TexWord32` and `indexArray` for `Word32` as below in the header file `accelerate_cuda_texture.h`, which is included in the generated output:

```c
typedef texture<uint32_t, 1> TexWord32;
static __inline__ __device__ uint32_t indexArray(TexWord32 t, const int x)
{
return tex1Dfetch(t, x);
}
```

The device memory allocated for $xs'$ in the above Accelerate code is bound to the texture reference, `tex0`, and `tex1Dfetch(tex0, 0)` fetches the $0^{th}$ element in $xs'$ through `tex0`. 
Shape

The support for shape expressions is incomplete in the CUDA back end as of Version 0.8.1.0, however, the co-authors of Accelerate have completed it in the development branch and it will be a part of the next release. The main idea behind it is similar to the array indexing. For each array that is passed to the shape expression, the CUDA back end generates a global variable that holds the dimension information of the array, and it replaces the shape expression by the reference to the global variable with the dimension information in the generated CUDA code.
codeGenExp :: forall env aenv t. OpenExp env aenv t → CG [CExpr]

```haskell
codegenExp (Shape _) = return . unit -- incomplete
          $ CVar (internalIdent "shape") internalNode

codeGenExp (PrimConst c) = return . unit $ codeGenPrimConst c

codeGenExp (PrimApp f arg) = unit . codeGenPrim f <$> codeGenExp arg

codeGenExp (Const c) = return
          $ codeGenConst (elemType (undefine d::t)) c

codeGenExp (Tuple t) = codeGenTup t

codeGenExp prj@(Prj idx e) = reverse . take (length $ codeGenTupleType (expType prj))
          . drop (prjToInt idx (expType e)) . reverse <$> codeGenExp e

codeGenExp (Var i) =
          let var = CVar (internalIdent ('x':show (idxToInt i))) internalNode
          in case codeGenTupleType (elemType (undefined::t)) of
            [ ] → return [var]
            cps → return . reverse . take (length cps)
            . flip map (enumFrom 0 :: [Int])
            $ \c \rightarrow
              CMember var (internalIdent ('a':show c)) False internalNode

codeGenExp (Cond p e1 e2) = do
  [a] ← codeGenExp p
  zipWith (\b c → CCond a (Just b) c internalNode) <$> codeGenExp e1 <*> codeGenExp e2

codeGenExp (IndexScalar a1 e1) = do
  n ← length <$> get
  [i] ← codeGenExp e1
  let ty = codeGenTupleTex (accType a1)
      fv = map (\x → "tex" ++ show x) [n..]
  modify (++) zipWith globalDecl ty fv
  return (zipWith (indexArray i) fv ty)
```

**Figure 4.11:** The code generation rules: the translation of the scalar expressions
Chapter 5

Quantitative Analysis and Evaluation

I discussed the Accelerate language and the Accelerate front end which provides a means to describe collective array operations on parallel hardware platforms, and the CUDA back end which compiles collective array operations written in the Accelerate language to CUDA device modules and executes them. In this chapter, I provide a quantitative analysis and evaluation of the performance of Accelerate applications in comparison to hand-tuned CUDA applications with the same effect.

The results demonstrate the potential of the skeleton-based code generation and the functional programming constructs in GPGPU, and suggests several future improvements by identifying the bottlenecks. These suggested improvements, as a result of this analysis and user feedback, are being implemented actively in the development branch.

5.1 Environment

I employ the system configuration in Table 5.1 for the benchmarks. Tesla S1070 is a solution specifically designed for general purpose high-performance computation, and it bundles four GPUs in each unit. Each GPU in S1070 has 240 scalar
processors and 4GB GDDR3 memory. Of the four GPUs available in Tesla S1070, I use only one of them, as the CUDA back end does not support multi-GPU execution.

5.2 Applications

5.2.1 SAXPY, SSCAL, SASUM, SNRM2, SDOT

SAXPY, SSCAL, SASUM, SNRM2, and SDOT are vector operations defined in Basic Linear Algebra Subprograms (BLAS), which is a well-known application interface for vector and matrix operations.

I measure the performance of these operations written in Accelerate and run by the CUDA back end against the corresponding operations in CUBLAS library, which is a BLAS library written in CUDA by NVIDIA.

**SAXPY**

SAXPY takes a scalar value, $\alpha$, and two vectors, $X$ and $Y$, as the input. Then it produces $\alpha X + Y$ as the output. Figure 2.3(a) shows an Accelerate implementation of SAXPY with 1.5 as the $\alpha$ value.

**SSCAL**

SSCAL takes a scalar value, $\alpha$, and a vector $X$ as the input. Then it produces $\alpha X$ as the output. The following code snippet provides an implementation of SSCAL in Accelerate, given \texttt{alpha} as the $\alpha$ value and \texttt{xs} as the vector:
let xs' = use xs
in  map (constant alpha *) xs'

**SASUM**

SASUM takes a vector and produces the sum of the absolute values of the elements. In Accelerate, given a vector \(xs\), we write SASUM as below:

\[
\text{let } xs' = \text{use } xs
\text{in } \text{fold (+)} (\text{constant 0}) (\text{map abs xs'})
\]

**SNRM2**

SNRM2 produces the \(l^2\)-norm of the given vector. Mathematically, it is defined as \(|X| = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}\), where \(X = [x_1, x_2, \ldots, x_n]\), and an Accelerate implementation of SNRM2 is as follows, where \(xs\) is the input vector:

\[
\text{let } xs' = \text{use } xs
\text{in } \text{map sqrt} (\text{fold (+)} (\text{constant 0}) (\text{map (} \lambda x \rightarrow x \times x \text{) xs'}))
\]

**SDOT**

SDOT produces the dot product of two input vectors. The mathematical notation of dot product, \(X \cdot Y = \sum_{i=1}^{n} x_i y_i\), where \(X = [x_1, x_2, \ldots, x_n]\) and \(Y = [y_1, y_2, \ldots, y_n]\) are the input vectors, is mapped to the following code snippet written in Accelerate; \(X\) and \(Y\) are represented as \(xs\) and \(ys\) in the code snippet:

\[
\text{let } xs' = \text{use } xs
\text{ys' = use } ys
\text{in } \text{fold (+) (constant 0) (zipWith (*) xs' ys')}
\]

### 5.2.2 Exclusive Prefix Sum

The exclusive prefix sum is a prefix scan whose combination function is the addition operator. Given a vector \([x_1, x_2, \ldots, x_n]\) and the identity value 0, the exclusive prefix sum of the vector is \([0, 0 + x_1, 0 + x_1 + x_2, \ldots, 0 + x_1 + x_2 + \ldots + x_{n-1}]\).

I compare the following Accelerate implementation of the exclusive prefix sum against the CUDA implementation found in the CUDPP library [52]:

---
5.2.3 Outer Product

Given two input vectors, $X$ and $Y$, $X \otimes Y$ is the outer product of these vectors:

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \quad Y = [y_1, y_2, \ldots, y_n] \quad X \otimes Y = \begin{bmatrix} x_1 y_1, & x_1 y_2, & \ldots, & x_1 y_n \\ x_2 y_1, & x_2 y_2, & \ldots, & x_2 y_n \\ \vdots & \vdots & \ddots & \vdots \\ x_m y_1, & x_m y_2, & \ldots, & x_m y_n \end{bmatrix}$$

I compare the following Accelerate implementation of outer product against a hand-written CUDA implementation found in Appendix A:

```haskell
let xs' = use xs 
    ys' = use ys 
  in scan (+) (constant 0) xs'
```

5.2.4 Sparse Matrix and Vector Multiplication

I compare an Accelerate implementation of sparse-matrix and vector multiplication, which is based on the Data Parallel Haskell implementation that Chakravarty et al. wrote [55], following the algorithm that Blelloch et al. used in their Nesl implementation [56], against the implementation found in the CUDPP library [52].

The sparse matrix above is represented with a vector of the non-zero values, $V$, a vector of the column indices of the non-zero values, $I$, and the length of each row, $S$:

$$V = [1, 2, 2, 4, 2, 3] \quad I = [0, 2, 0, 1, 1, 2] \quad S = [2, 2, 0, 2]$$
When this matrix is multiplied by a vector, \( Y = [4, 2, 5] \), we get the result \([14, 16, 0, 19]\) by

- backpermuting the vector using the column indices to produce another vector \( Y' = [4, 5, 4, 2, 2, 5]\),

- multiplying \( V \) and \( Y' \) element-wise to get an intermediate values, \( Z = [4, 10, 8, 8, 4, 15]\), and

- running the segmented sum on \( Z \) using the segment descriptor \( S \).

The following is the Accelerate implementation of sparse matrix and vector multiplication, where \( \text{segd}, \text{inds}, \) and \( \text{vals} \) are the segment descriptor, the column indices, and the values in the sparse matrix.

```plaintext
let segd' = use (segd :: Vector Int)
    inds' = use (inds :: Vector Int)
    vals' = use (vals :: Vector Float)
    vec' = use (vec :: Vector Float)

    vecVals = backpermute (shape inds') (\i -> inds' ! i) vec'
    products = zipWith (*) vecVals vals'

in foldSeg (+) 0 products segd'
```

### 5.2.5 Black-Scholes Options Pricing

Black-Scholes options pricing calculates call and put prices for the given options according to the formula that Black and Scholes defined \([57]\). I compare an Accelerate implementation of Black-Scholes options pricing against the one in the CUDA SDK.

Its scalar operation is the most complex among the applications in the benchmark suite, although it has just one collective array operation \( \text{map} \) that takes an array of tuples —the current option price, the strike price, and the time to expiration— as the input and produces an array of pairs —the price of a call option and the price of a put option. The mathematical details can be found in the article written by Black and Scholes \([57]\) and in the book written by Hull \([58]\).
5.3 Performance of Generated Device Code

I group the applications explained in the previous section into three:

- applications with one simple array operation, i.e., SAXPY, SSCAL, and exclusive prefix sum,

- applications with multiple simple array operations, i.e., SASUM, SNRM2, SDOT, outer product and sparse matrix and vector multiplication, and

- applications with one complex array operation, i.e., Black-Scholes options pricing.

and I analyse and evaluate the applications each group by comparing the performance of the generated device code modules against the hand-tuned device code modules, the CUBLAS operations, or the CUDPP operations. The execution cost of each module is measured by the NVIDIA CUDA profiler, which executes the application multiple times, averaging the timings.

5.3.1 Applications with One Simple Array Operation

SAXPY written in Accelerate consists of a `zipWith` operation, which runs a simple scalar operation on the input array elements. That is, the CUDA back end generates only one device code module, `zipWith`, compiles it, and executes it on the given GPU for SAXPY. Figure 5.1 displays the execution cost of the `zipWith` device code module and the execution cost of the `saxpy_gld_main` device code module that CUBLAS SAXPY runs internally. As seen in the figure, the Accelerate SAXPY achieves similar performance to the CUBLAS SAXPY.

Similar to SAXPY, SSCAL written in Accelerate consists of a simple `map` operation. Figure 5.2 displays the execution cost of the `map` device code module in comparison to the `sscal_gld_main` device code module that CUBLAS SSCAL runs internally, and Accelerate SSCAL and CUBLAS SSCAL have similar performance.
Chapter 5. Quantitative Analysis and Evaluation

Figure 5.1: The runtime cost of SAXPY

Figure 5.2: The runtime cost of SSCAL
Table 5.1: The runtime cost of exclusive prefix sum

<table>
<thead>
<tr>
<th>Number of Elements (million)</th>
<th>Tesla S1070 Accelerate</th>
<th>Tesla S1070 CUDPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1,750</td>
<td>1,750</td>
</tr>
<tr>
<td>4</td>
<td>3,500</td>
<td>3,500</td>
</tr>
<tr>
<td>6</td>
<td>5,250</td>
<td>5,250</td>
</tr>
<tr>
<td>8</td>
<td>7,000</td>
<td>7,000</td>
</tr>
<tr>
<td>10</td>
<td>8,881</td>
<td>8,881</td>
</tr>
<tr>
<td>12</td>
<td>10,752</td>
<td>10,752</td>
</tr>
<tr>
<td>14</td>
<td>12,624</td>
<td>12,624</td>
</tr>
<tr>
<td>16</td>
<td>14,496</td>
<td>14,496</td>
</tr>
<tr>
<td>18</td>
<td>16,368</td>
<td>16,368</td>
</tr>
</tbody>
</table>

Figure 5.3: The runtime cost of exclusive prefix sum

Exclusive prefix sum consists of one simple array operation, `scan`, however, unlike the cases with `map` and `zipWith`, the Accelerate CUDA back end generates two kernels for a `scan` device code module: one for up-sweep and the other for down-sweep [50]. The CUDPP `cudppScan` device code module is also organised similarly with two kernels. The execution cost in Figure 5.3 is the sum of the execution cost of both kernels. The figure shows that the performance of exclusive prefix sum written in Accelerate is about a half of the performance of the CUDPP counterpart. This performance discrepancy is expected to be addressed by replacing the existing algorithmic skeleton for `scan` with a more efficient algorithmic skeleton.

5.3.2 Applications with Multiple Simple Array Operations

SASUM, SNRM2, SDOT, and outer product written in Accelerate consist of multiple collective array operations. Each collective array operation is parameterised with a single scalar operation.
Figures 5.4 to 5.7 show that the Accelerate implementation of these programs is outperformed by their CUBLAS counterparts. Accelerate SASUM is about 3 times slower than CUBLAS SASUM, Accelerate SNRM2 about 3 times slower than CUBLAS SNRM2, and Accelerate SDOT about 2 times slower than CUBLAS SDOT.

The reason for this performance discrepancy is that each of these operations is implemented as a single kernel in CUBLAS, whereas they are implemented as multiple kernels in Accelerate. That is, the Accelerate implementation of these operations processes more arrays than the CUBLAS implementation. For example, CUBLAS SASUM processes one input array to generate the output value, whereas Accelerate SASUM maps one input array to one intermediate array, which is then reduced to the output value.

We can close the performance discrepancy between Accelerate SASUM and CUBLAS SASUM, and between Accelerate SNRM2 and CUBLAS SNRM2 with array fusion (also called deforestation) [59–61]. In plain Haskell, a map operation and a fold operation are fused into another fold operation; fold (+) 0 (map abs xs) is fused into fold (λx y → x + abs y) 0 xs. Accelerate, however, does not fuse SASUM into a fold operation, and manual fusion is also not trivial as the Accelerate fold operation requires the first argument to be an associative binary scalar operation and the second argument to be its neutral element. Although the binary scalar operation λ x y → x + abs y is associative, the second...
### Chapter 5. Quantitative Analysis and Evaluation

<table>
<thead>
<tr>
<th>Number of Elements (millioin)</th>
<th>Tesla S1070 Accelerate</th>
<th>Tesla S1070 CUBLAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>301.696</td>
<td>588.352</td>
</tr>
<tr>
<td>4</td>
<td>285.832</td>
<td>588.352</td>
</tr>
<tr>
<td>6</td>
<td>285.832</td>
<td>588.352</td>
</tr>
<tr>
<td>8</td>
<td>285.832</td>
<td>588.352</td>
</tr>
<tr>
<td>10</td>
<td>285.832</td>
<td>588.352</td>
</tr>
<tr>
<td>12</td>
<td>285.832</td>
<td>588.352</td>
</tr>
<tr>
<td>14</td>
<td>285.832</td>
<td>588.352</td>
</tr>
<tr>
<td>16</td>
<td>285.832</td>
<td>588.352</td>
</tr>
<tr>
<td>18</td>
<td>285.832</td>
<td>588.352</td>
</tr>
</tbody>
</table>

#### Figure 5.4: The runtime cost of SASUM

<table>
<thead>
<tr>
<th>Number of Elements (millioin)</th>
<th>Tesla S1070 Accelerate</th>
<th>Tesla S1070 CUBLAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>303.648</td>
<td>605.088</td>
</tr>
<tr>
<td>4</td>
<td>285.832</td>
<td>588.352</td>
</tr>
<tr>
<td>6</td>
<td>285.832</td>
<td>588.352</td>
</tr>
<tr>
<td>8</td>
<td>285.832</td>
<td>588.352</td>
</tr>
<tr>
<td>10</td>
<td>285.832</td>
<td>588.352</td>
</tr>
<tr>
<td>12</td>
<td>285.832</td>
<td>588.352</td>
</tr>
<tr>
<td>14</td>
<td>285.832</td>
<td>588.352</td>
</tr>
<tr>
<td>16</td>
<td>285.832</td>
<td>588.352</td>
</tr>
<tr>
<td>18</td>
<td>285.832</td>
<td>588.352</td>
</tr>
</tbody>
</table>

#### Figure 5.5: The runtime cost of SNRM2
argument 0 is not its neutral element — $0 + \text{abs}(-x)$ is not $-x$. We need to devise an implementation of array fusion that accommodates the characteristics of Accelerate collective array operations.

Table 5.2 lets us estimate the effect of array fusion. The runtime cost of CUBLAS SASUM is similar to the runtime cost of the fold operation in Accelerate SASUM, and the runtime cost of CUBLAS SNRM2 is slightly higher than the fold operation in Accelerate SNRM2. If the map and fold operations in Accelerate SASUM were fused, the runtime cost of Accelerate SASUM would be slightly higher than the runtime cost of the fold operation, and the performance discrepancy between Accelerate SASUM and CUBLAS SASUM be reduced significantly. If the first map and fold in Accelerate SNRM2 were fused, we would see the similar effect as with Accelerate SASUM.

Fusing zipWith and fold in Accelerate SDOT is not as obvious as fusing collective array operations in Accelerate SASUM and Accelerate SNRM2, because zipWith and fold take different numbers of input arrays. One simple solution is to define an algorithmic skeleton, e.g., mapFold, that generalises map and fold over the number of input arrays. With mapFold and array fusion, we would be able to close the performance discrepancy between Accelerate SDOT and CUBLAS SDOT.

Outer product is another application with multiple simple collective array operations. The Accelerate implementation of outer product replicates two input arrays to produce matrices, and multiplies them element-wise to produce the output.

The Accelerate implementation of outer product accesses about six times as many data elements as the CUDA implementation listed in Appendix A. Given two input arrays of size $m$ and of size $n$, the replicate operation on the first array reads $m$ elements $n$ times and writes $m \times n$ elements in the first intermediate matrix, and the replicate operation on the second array reads $n$ elements $m$ times and writes $m \times n$ elements in the second intermediate matrix. The zipWith reads $m \times n$ elements from each intermediate matrix and writes $m \times n$ elements in the output. In total, the Accelerate implementation of outer product reads
$4 \times m \times n$ elements and writes $3 \times m \times n$ elements. On the other hand, the CUDA implementation listed in Appendix A reads the $m$ elements in the first input array $n/16$ times and $n$ elements in the second input array $m/16$ times, and writes $m \times n$ elements in the output. In total, the CUDA implementation of outer product in Appendix A reads $(2\times m \times n)/16$ elements and writes $m \times n$ elements. Therefore, the Accelerate implementation of outer product accesses elements about 6 times more than the CUDA implementation — $(4 \times m \times n + 3 \times m \times n)/((2 \times m \times n/16) + (m \times n))$.

The difference in the global memory access of the Accelerate implementation of outer product and of the CUDA implementation of outer product results in the performance discrepancy depicted in Figure 5.7 as global memory access is one of the most expensive operations on GPUs. To reduce the global memory access in the Accelerate implementation, we need an index transformation based array fusion similar to what Keller et al. use in their Repa library [62]; instead of producing intermediate matrices, we can make the zipWith operation fetch the input array elements by computing the indices based on the transformation using their index transformation based array fusion or similar [63].

**Figure 5.6:** The runtime cost of SDOT
Figure 5.7: The runtime cost of outer product

Figure 5.8 shows that the performance of the Accelerate implementation of sparse matrix and vector multiplication is similar to the performance of the CUDPP counterpart. The CUDPP implementation also runs multiple device code modules internally, and their semantics are roughly the same as the collective array operation used as part of the Accelerate implementation of sparse matrix and vector multiplication.

The labels on X axis in Figure 5.8 are the matrix sizes, and each label has three values: the number of rows in the matrix, the number of columns, and the number of non-zero elements. To control the number of the non-zero elements and the matrix dimensions, I use the dense matrix represented with the representation that we discussed in Section 5.2.4 and use the algorithm for sparse matrices.

5.3.3 Applications with One Complex Array Operation

Among the benchmark application that we have, Black-Scholes options pricing is the only one with a complex array operation. Black-Scholes options pricing takes an array of three-tuples (the current option price, the strike price, the time
time to expiration), and it produces an array of pairs (the price for a call option, the price for a put option).

Since I provided a benchmark with Black-Scholes options pricing in 2009 [2], the design of the Accelerate front end and the Accelerate CUDA back end changed significantly. Furthermore, the benchmark in the past involved manual code translation for missing compilation functionality, whereas this benchmark involves no manual intervention, except where indicated otherwise.

The result that I present in this section is less favourable than other benchmark results that I presented in the previous sections and than the Black-Scholes options pricing benchmark in the past [2]. Figure 5.9 shows that the Accelerate implementation of Black-Scholes options pricing is about 23 times slower than the implementation in the CUDA SDK.

The reason for this poor performance lies in the lack of common sub-expression elimination (CSE); CSE has not been implemented in Accelerate as of Accelerate-0.8.0.0. A large portion of the scalar computation is identical in the call option computation and the put option computation. When computing the price for a
call option and the price for a put option, the CUDA SDK implementation binds the values of the common sub-expressions to variables and reuses them as much as possible, whereas the Accelerate implementation builds a pair of computation ASTs and nothing is shared between these two ASTs. That is, a significant amount of runtime cost is wasted, computing the same values redundantly. In addition, when the scalar operation contains a conditional. In the CUDA SDK implementation, the conditional is written:

\[
\text{if } (d > 0) \quad \text{cnd} = 1.0f - \text{cnd};
\]

As the value \(d\) depends on the randomly generated input values, this conditional may introduce a divergence in the execution. However, the divergence is relatively small compared to what we see in the Accelerate implementation. In the Accelerate implementation, \(d\) and \(\text{cnd}\) are not values, but they are sub-trees in the scalar computation AST. That is, this conditional builds a scalar computation AST with the sub-tree \(d\) as the predicate, the sub-tree \(\text{cnd}\) as the alternative, and another scalar computation AST that contains a copy of \(\text{cnd}\) as the consequent.
Both $d$ and $cnd$ are used for call options and for put options.

After all, the Accelerate implementation of Black-Scholes options pricing computes a number of values more than once unnecessarily, and both the scalar computation AST of the price for a call option and the scalar computation AST of the price for a put option are divergent. Redundant computations increase the instruction count excessively and the divergence serialises a substantial portion of the parallel execution. The Accelerate implementation of Black-Scholes options pricing executes more than 23 million instructions in total to process 9 million options and introduces divergent branches whereas the CUDA SDK implementation executed less than a million instructions without divergent branches; the \texttt{nvcc} compiler optimises the conditional in the CUDA SDK implementation to a predicated execution instead of creating a divergent branch.

When producing the Black-Schole options pricing benchmark in the past, the manual code translation assumed the scalar-level CSE, thus exhibiting performance closer to the performance of the CUDA SDK implementation. This benchmark result strongly suggests the necessity of the scalar-level CSE, and the yellow line and the red line in Figure 5.9 validate the suggestion. The yellow line and the red line show the performance of the code generated by the Accelerate CUDA back end and optimised manually with the scalar-level CSE. Appendix \ref{appendix-b} discusses the details of the manual optimisations.

### 5.3.4 Summary

In this section, we have evaluated the performance of the device code module generated with algorithmic skeletons that map higher-order operations written in Accelerate. Through the evaluation of SAXPY, SSCAL, exclusive prefix sums, and sparse matrix and vector multiplications written in Accelerate, we have found that it is possible to produce device code modules with algorithmic skeletons that achieve similar performance as the hand-tuned device code modules. As the implementation of the CUDA back end and its algorithmic skeletons become maturer and stabilised, any existing performance discrepancies are expected to
become narrower.

Furthermore, through the evaluation of SASUM, SNRM2, outer product, and Black-Scholes options pricing written in Accelerate, we have also found that we need optimisations such as array fusion, especially index transformation based, and CSE to reduce the number of intermediate arrays and to avoid redundant computations. And the evaluation of SDOT operations written in Accelerate has suggested a need for a general mapFold algorithmic skeleton. Some of the optimisations are already being resolved in the development branch: one of the co-authors is making progress on fusion, another co-author has already implemented the array-level CSE. The scalar-level CSE, which is much needed to improve the performance of the Accelerate implementation of Black-Scholes, is yet to be implemented.

Although every benchmark result has not been favourable, we have been able to identify the cause of the performance discrepancies and propose possible approaches to close the performance discrepancies.

5.4 The Overall Performance and the Effect of Caching of Compiled Modules

To evaluate the overall performance, I measured the runtime cost between the CUDA context initialisation and the CUDA context finalisation — i.e., the data transfer cost and the device code execution cost. In addition, I also measured the device code generation cost and device code compilation cost if the device code modules are dynamically generated and compiled.

Figure 5.10 shows the overall performance of our Accelerate benchmark applications and the reference implementations. It is to be noted that the Y axis is in a logarithmic scale. The performance of our Acceleate benchmark applications is presented in two configurations: one with the persistent caching of compiled device code modules and the other without it. The performance of the Accelerate implementation of the benchmark applications is much poorer than the
Figure 5.10: The overall performance of the benchmark applications and the reference implementations
Figure 5.10: The overall performance of the benchmark applications and the reference implementations (continued)
performance of the reference implementation written in plain CUDA, CUDPP, or CUBLAS, when the persistent caching of compiled device code modules is disabled, as the cost of spawning and executing the \texttt{nvcc} processes is dominant in the overall runtime cost, and it is much larger than the actual computation cost or the data transfer cost. Furthermore, in some cases, the performance of the Accelerate implementation is not linear when the persistent caching is disabled, and this is caused by the external factors inherited from the spawning of the \texttt{nvcc} processes; we cannot control from the applications the way that the operating systems schedule the main application process and the \texttt{nvcc} process.

Figure 5.10 suggests that the persistent caching of compiled device code modules reduces the runtime cost significantly and that the performance of the Accelerate implementation of the benchmark applications is linear when the persistent caching is enabled.

Let us compare the overall performance of the Accelerate implementations with the persistent caching against the reference implementations. For the applications such as SAXPY, SSCAL, and sparse matrix and vector multiplication, the performance of the Accelerate implementation is similar to the performance of the reference implementation, as the arrays of the same size are transferred and the performance of the generated device code modules is similar to the performance of the device code modules in the reference implementation. For the rest of the applications, the performance discrepancies that we discussed in the previous section are reflected in the overall performance, however, the overall performance of the Accelerate implementation with the persistent caching is within an order of magnitude.

Another advantage of the persistent caching is that, when the underlying GPU supports asynchronous data transfers, we can overlap the input array transfers and the overhead incurred by the Accelerate pipeline such as the device code execution preparation, even when the input size is relatively small. Without code generation and code compilation, the execution preparation only requires the dynamic loading of the module and the relevant parameter settings. Figure 5.11 provides an example. The runtime cost of the Accelerate implementation of
**Figure 5.11:** The overall runtime cost increase of the Accelerate implementation of Black-Scholes options pricing by the number of options
Black-Schole options pricing with the persistent caching stays at the similar level until the input size is increased over 10,000. That is, the constant cost is dominant in the overall cost, and the execution preparation, which is not dependent on the input size, is the source of the constant cost; the data transfer cost depends on the data sizes. As the runtime cost for data transfers becomes more expensive than the execution preparation cost, the gradient of the line in Figure 5.11 steepens, and the runtime cost of data transfers conceals the overhead incurred by the Accelerate pipeline. As code generation cost and code compilation cost are much greater than the execution preparation cost, we would need much bigger input arrays to conceal code generation cost and code compilation cost with the data transfer cost, if the persistent caching were not implemented.

5.5 Summary

Throughout this chapter, we have discussed and evaluated the performance of several applications that are written in the Accelerate language, and compiled and executed by the Accelerate CUDA back end, and also compared it against the performance of the hand-tuned CUDA implementation of the same applications.

The findings of the analysis and the evaluation are summarised to the following:

- The Accelerate CUDA back end is able to produce device code modules whose performance is comparable to the device code modules written in CUDA for applications consisting of one simple array operation.

- We must implement array fusion, especially index transformation based array fusion, in the Accelerate pipeline to reduce the creation of intermediate arrays.

- We must generalise map and fold to fuse map and fold regardless of the number of input arrays.
• We need common sub-expression elimination in the Accelerate pipeline to avoid redundant computations and reduce divergent branches in the generated device code, which cause performance degradation.

• The persistent caching of the compiled device code modules is, in fact, effective in reducing the overall runtime cost of the applications written in Accelerate.

• We must keep improving the algorithmic skeletons to obtain better performance.

Some of the performance results presented in this chapter are not favourable. Nonetheless, considering that both the Accelerate front end and the Accelerate CUDA back end are in their early stage of development and that they provide a high-level abstraction to alleviate GPU programming, the performance presented in this chapter demonstrated the potential of the approaches in this thesis—mainly, the use of algorithmic skeletons that map higher-order functions to generate GPU device code modules.
Chapter 6

Related Work

There are numerous projects which share Accelerate’s aim of facilitating access to the processing power of modern GPUs at the programming system level. This chapter introduces some of those attempts that are noteworthy and also related to Accelerate. The systems introduced here are designed for general purpose computations with the exception of a few that are graphics-specific. I aim to identify the advantages and the disadvantages of the Accelerate design and implementation by exploring the characteristics of each GPU programming system, and comparing them against Accelerate and each other.

6.1 Characteristics of Related GPU Programming Systems

This section discusses and compares the important characteristics of the related GPU programming systems such as programming mode and code generation approach. More detailed discussion on each GPU programming system related to Accelerate follows in the following sections.

Embedded Domain Specific Language

Similar to Accelerate, many of the GPU programming system discussed in this chapter are implemented as embedded domain specific languages (EDSL). The
EDSL-based programming systems generally integrate GPU operations into the host language seamlessly, allowing programmers, who are familiar with the host language, to adopt them without much endeavour.

The EDSL-based programming systems that generate GPU code and execute them at application runtime liberate programmers from the manipulation of GPU platform-specific compiler tools and environment configuration; those programming systems internally manipulate GPU platform-specific tools. One disadvantage of these programming systems is that code generation and compilation incur runtime overhead.

There are some EDSL-based programming systems that only generate GPU code such as Barracuda [64]. Even though the integration with the host language is somewhat limited in these programming systems, they still provide host language programmers with a familiar programming environment.

The implementation of the compiler for the EDSL-based programming systems is simpler than that of the programming systems that change the host language constructs.

**Directive and Annotation-Based Parallelisation**

A number of C++-based GPU programming systems, influenced by OpenMP [42], parallelise the annotated regions or functions of the program for the target GPU platform. They provide a fully-fledged host language compiler, modified to recognise directives, generate GPU code, and manipulate the target platform compiler tools such as the `nvcc` compiler of CUDA. One of the design principles of these systems is that programs written for these systems must be compiled to sequential programs when the compiler discards the directives.

Most of these programming systems allow functional decomposition of programs as the annotated regions can be embedded in any part of a function, and they do not distinguish GPU code from host code at the function level. The abstraction level that they provide is typically lower than the EDSL-based programming systems, and they allow finer-grained control on GPU operations. However, it also indicates that they require programmers to have significant knowledge...
about the target GPU platform.

**C++ Template Library**

GPU programming systems based on C++ template library approach implement a set of GPU operations and the data structures tailored for those GPU operations in a template library. It has an advantage that, unlike EDSL-based programming systems, the whole program can be compiled by the target platform compiler tools. The generation of GPU code is taken care of by the template instantiation. It is not possible to implement this approach in non-C++ host languages, as it is dependent on the C++-specific feature.

**Algorithmic Skeleton-Based Programming**

The GPU programming systems, which are based on algorithmic skeletons, do not provide fine-control on GPU operations. However, they provide higher level abstraction by encapsulating low-level details within algorithmic skeletons.

The algorithmic skeleton-based GPU programming systems that are introduced in this chapter only implement data parallelism and the composition of the algorithmic skeletons. Nonetheless, it still allows us to sufficiently exploit GPUs for parallel array processing, as the GPU architecture is better suited for single program, multiple data (SPMD) processing.

### 6.2 Haskell-based GPU Programming Systems

#### 6.2.1 Nikola

Nikola and Accelerate are similar in that they provide a high-level GPU programming system by embedding a domain specific language in Haskell. Nikola’s language constructs, implementation strategy, code generation techniques, kernel execution model are somewhat different from Accelerate’s. Nikola explicitly generates and schedules loops, whereas we use algorithmic skeletons. The Nikola language supports a more limited range of operations than Accelerate. It only
supports operations which can be mapped in a single GPU kernel, and whose memory requirements can be pre-determined statically, i.e., no support for operations such as \texttt{replicate}. For the operations that require a series of multiple kernels, application programmers must schedule the compilation and the execution of each kernel explicitly, and the glue code that Nikola generates triggers data transfers between host and device for each kernel execution, often incurring unnecessary overhead.

The advantages of Nikola over Accelerate is that it supports offline compilation as well as online compilation, and provides facilities for direct embedding of CUDA functions with a small amount of glue code, if needed. In addition, it implements a form of sharing observation at the scalar operation level.

### 6.2.2 Barracuda

Barracuda \cite{64} is another embedded domain specific language in Haskell for GPU programming. Similarly to Accelerate, Barracuda is an array language, which describes data parallelism with collective array operations. But the supported collective array operations are limited to \texttt{map}, \texttt{zipWith}, \texttt{reduce}, and \texttt{slice}, and the supported element types to floating-point numbers, integers, and Boolean values. Contrary to Accelerate’s type system which disallows collective array operations in the places where scalar operations are expected, Barracuda’s type system allows some of the array operations in such places. Then its back end hoists out such array operations when possible, otherwise it generates a sequential loop, which can be inefficient on GPUs.

The back end of Barracuda generates CUDA code with C++ wrapper code so that it can be integrated into C++ applications, but it does not compile or execute the generated CUDA code. Although it is still at an early stage of development, it already supports some important optimisations, namely shared memory optimisation, and array fusion. As mentioned in Section \ref{shared_memory_optimisation} the on-chip shared memory has a much higher bandwidth than the off-chip global memory, and it is
often used as software-managed cache to minimise the off-chip global memory access. Barracuda generates code to utilise the shared memory as software-managed cache when it is applicable and beneficial, aiming to achieve performance close to optimal. It also fuses collective array operations to avoid creating temporary arrays that hold intermediate results and to minimise kernel launches, when a collective array operation is given to another as an array argument.

### 6.2.3 Obsidian

Obsidian [66] is another EDSL in Haskell for GPU programming. Compared to Accelerate, Obsidian provides a lower-level language constructs; Obsidian is designed as a means to describe and fine-tune the algorithms that implement the collective array operations, which Accelerate provides as built-in language constructs using skeletons. Obsidian follows the online compilation, targeting CUDA as Accelerate, Nikola, and Barracuda do. The current implementation of Obsidian is focused on 1-dimension array operations.

Similarly to Nikola, Obsidian does not support the operations whose output size is dependent on the value of the input. Another limitation that Obsidian shares with Nikola is that the coordination of multiple kernels is not supported, thus Obsidian can only generate a single kernel with the expressions given to execute, which is equivalent to CUDA.run in Accelerate, and executes it. That is, it shares the limitation with Nikola that application programmers must schedule each kernel for compilation and execution separately and explicitly.

### 6.2.4 Haskgraph

Haskgraph [67] is also a Haskell EDSL that targets CUDA, and it follows the online compilation approach. Haskgraph provides parallel operation primitives such as mapS, zipWithS, foldS, etc. on Haskell streams.

Haskgraph uses deep embedding for scalar expressions, but shallow embedding for parallel stream operations. It builds an AST for a set of scalar operations
and passes it to one of the parallel operations. Then the parallel stream operation analyses the scalar operation AST, generates the CUDA code, executes the generated code, and returns the result. Overall, by not requiring an additional function that controls the compilation and the execution of the parallel operations —e.g., \texttt{CUDA.run} in Accelerate, \texttt{execute} in Obsidian, etc.— Haskgraph’s language interface may seem more familiar to Haskell application programmers. The downside of this approach is that, when a sequence of parallel stream operations are run, the intermediate results have to be transferred from the device memory to the host memory, then back to the device memory; data transfers are one of the most expensive operations in CUDA.

When generating CUDA code, Haskgraph hardwires some constants such as the number of the elements in the input stream. This may help save some processing cycles, but it makes the code harder to re-use unless the same constants can be re-used —e.g., the number of elements in the input stream stays the same.

### 6.2.5 Vertigo

Vertigo \cite{68} was the first Haskell EDSL for GPU programming. Its target platform, however, is different from the target platform of the aforementioned Haskell EDSLs; it targets DirectX 9 shader, whose architecture is graphics-oriented, and provides a much more limited instruction set for general purpose computations than CUDA platform. Reflecting the target architecture, the examples and the language constructs presented in the Vertigo paper demonstrate the use of the language for graphics shaders. However, the techniques can be used in more general settings.
6.3 C++-based GPU Programming Systems

6.3.1 Brook for GPUs

Brook for GPUs [20] is one of the earliest GPU programming system for general purpose computations. Its contribution was significant as it provided the programming system designed for general purpose computation, even though general purpose computation APIs were unavailable on GPUs when it was released in 2004, and it motivated a number of researches in GPGPU, including Accelerate.

Brook language extends the standard C for stream (or GPU array) declarations, and kernel definitions. The stream declaration is similar to the C-style array declaration except that it uses angle brackets instead of square brackets, e.g., \texttt{float2 stream<4,2>} declares two-dimensional stream in which each element is of type \texttt{float2}. The elements of the streams are accessible only in kernels — functions annotated with the keyword \texttt{kernel}. It supports \texttt{float}, \texttt{floatn}, \texttt{fixed}, \texttt{fixedn}, \texttt{shortfixed}, \texttt{shortfixedn}, and user-defined \texttt{struct}s, which consists of the aforementioned types, as the stream element type, where \( n \in \{2, 3, 4\} \). Generally, kernel bodies only supports the subset of C/C++ that shader languages such as NVIDIA’s Cg and Microsoft’s HLSL support.

The Brook system consists of two components: the Brook runtime (BRT) and the Brook compiler (\texttt{brcc}). BRT provides built-in operations that perform reduction, scatter, gather, and data transfers between the host memory and GPU memory. The \texttt{brcc} compiler maps kernels to graphics shaders and generates C++ code. The C++ code generated by \texttt{brcc} is compiled and linked with BRT, using a regular C++ compiler.

6.3.2 Sh

Sh [21, 22] is a meta-programming API, implemented as a C++ library. Shader programs written in Sh are represented as a series of function calls, which build a parse tree, and the parse tree is compiled online to low-level GPU assembly code. The embedding of shader program constructs as function calls in C++ allows the
type checking and the syntactic analysis of the shader programs by the host C++
compiler when the host application is built. Sh also allows programmers to utilise
the standard C++ features and libraries, and overloads many C++ operators for
seamless integration of Sh into C++.

In addition to the language constructs for shader program writing, Sh provides
a means to connect or combine multiple shader programs without re-writing them,
through the overloaded \( \& \) and \( \& \) operators, and other manipulation APIs. Sh also
provides facilities for stream processing using shader programs.

As one of the earliest GPU programming systems with Brook for GPUs, its
embedding of the domain specific language, and online compilation inspired many
GPU programming systems introduced in this chapter, including Accelerate.

The authors of Sh turned it into a product RapidMind \[69, 70\] that targets
multi-core CPUs, GPU (OpenCL), etc. RapidMind was later acquired by Intel
in 2009, and integrated into Intel Ct, then eventually into Array Building Blocks
(ArBB) \[71, 72\].

### 6.3.3 GPU++

GPU++ \[73\] is a C++-based GPU programming system similar to Sh in that
it provides a means to build a directed acyclic graph (DAG) for GPU kernels,
and compiles the DAG to low-level GPU assembly code. GPU++, however, is
designed for general purpose computations rather than graphics-centric opera-
tions. Besides, GPU++ provides a more abstract interface than Sh by enabling
object-oriented programming in kernels, providing a unified language features for
vertex shaders and fragment shaders. After building a DAG, GPU++ automatically
performs numerous optimisation such as algebraic re-association, common
sub-expression elimination, dead code elimination, constant folding, substitution
of the operations that are not natively supported with the ones that are natively
supported, etc. that many of the programming systems introduced in this chapter
do not support.
6.3.4 Thrust

Thrust \cite{53} is a C++ library written in CUDA. It provides vector containers and vector algorithms such as searching, copying, reductions, prefix sums, sorting, etc. that can be used on CPUs and GPUs with an interface similar to C++ Standard Template Library. These vector algorithms takes scalar operators defined as function objects. Application programmers are allowed to write custom scalar operators, and Thrust comes with a number of built-in scalar functions for arithmetic operations, comparison operations, logical and bitwise operations, and identity operations.

6.3.5 Skeletal Parallel Framework of Sato et al.

Sato et al. proposed and implemented a skeletal parallel framework in C for GPU programming \cite{74}. They introduce a new array kind, \textit{wrapped array}, which can be used as input or output arrays to skeletons. When an wrapped array is used for the first time by a skeleton, a data transfer from the host memory to the device memory is triggered. They define \texttt{map} and its variations, \texttt{zipWith}, \texttt{reduce}, and \texttt{generate}, which is similar to \texttt{replicate} in Accelerate, as CUDA kernel skeletons. They also provide a fusion optimiser to reduce the number of kernel launches and avoid unnecessary intermediate results in the device memory.

The compilation scheme of this framework takes advantage of the C front end, \texttt{cfront} and the C back end, \texttt{hir2c} of an existing compiler framework, COINS\footnote{http://coins-project.is.titech.ac.jp/international/index.html}. The compilation scheme (1) constructs \textit{high-level internal representation} (HIR) of the C source code that uses the skeletons, using \texttt{cfront}, (2) performs fusion on the HIR, (3) generates the corresponding C source code from the HIR, using \texttt{hir2c}, (4) imports the skeletons that are used into the output C source code, and (5) links the output of (4) against the relevant runtime libraries.

This framework resembles Accelerate in the use of skeletons, but there exist several noticeable differences between them. The compilation scheme in this
framework is offline, whereas it is online in Accelerate; this framework supports a smaller number of GPU operations than Accelerate, in particular, it does not support operations whose input array size and output array size may differ; this framework implements fusion, which Accelerate does not support yet; the destruction of wrapped arrays is explicit in this framework, whereas the destruction of arrays on GPU is implicit in Accelerate; this framework only supports one-dimensional arrays, whereas Accelerate supports various array shapes.

6.3.6 SkePU

SkePU [75] is a C++ library for skeleton-based GPU programming. Its language constructs are similar to the framework of Sato et al. from Section 6.3.5 in the range of the supported collective operations, in its use of skeletons, and in that it only supports one-dimensional vectors of a special type, `skepu::Vector`, for the collective operations—the vectors of the `skepu::Vector` type are comparable to wrapped arrays in the framework of Sato et al. On the other hand, SkePU does not require a specialised compiler and SkePU takes advantage of C++ features such as operator overloading and iterators to provide a high-level abstraction.

Scalar operators passed to SkePU collective operations are implemented as class objects with special methods, which are invoked by the collective array skeletons. SkePU defines a set of macros for convenience that are expanded to scalar operator definitions. Collective operation skeletons are defined as function object templates, i.e., their usage is similar to function templates from the application programmers’ perspective. When the CUDA back end is used, the `nvcc` compiler must be used to compile the application as the skeletons are instantiated to CUDA code.

The noticeable advantages of SkePU over Accelerate are: (1) SkePU is able to utilise multiple GPUs on a system and (2) SkePU supports multiple back ends. SkePU, however, has a smaller number of GPU operations than Accelerate, lacking the operations whose input array size and output array size may differ, and the supported vectors are one-dimensional whereas Accelerate supports various
array shapes.

6.3.7 MTPS

MTPS [76] is another skeleton-based C++ programming system that targets GPUs as well as CPUs. MTPS provides lower level abstraction than Accelerate, the skeletal parallel framework of Sato et al., and SkePU. MTPS defines a skeleton parallel_for that takes a function object as an input and executes it in parallel. The function objects passed to parallel_for must be written according to a particular convention that (1) they have to overload operator() with the operation that is to be executed on the data collection, (2) they have to implement the method getNbElement() that returns the number of elements in the data collection, (3) they have to define the type Device using typedef, and (4) they must carry a view of the data collection so that parallel_for knows on which data collection it has to map the operation. MTPS also requires data collection and their views to be organised in a certain way.

The authors of MTPS state that MTPS has been designed to be a low-level EDSL that will be a building block for a high-level EDSL [76]. That is, MTPS has different usability goals than Accelerate, which has been designed to provide a high-level abstraction.

6.3.8 PGI Accelerator

PGI Accelerator [77, 78] is a directive-based C and Fortran programming system that targets CUDA. There are three types of directives: the compute region directive, the compute region directive, and the loop directive. The compute region directive marks the beginning and the end of a region where the loops are to be parallelised. The PGI Accelerator compiler identifies the data that is required on GPUs for the region, and inserts the code that transfers such data to the device memory at the beginning of the compute region. The compiler also identifies the data that needs to be transferred back to the host memory, inserts the code that handles the data transfer at the end of the region. The data region
directive gives the control on the data transfers to application programmers, as it allows them to specify which data needs to be in the device memory for which part of the application. The loop directive provides a means to fine-tune the mapping of a loop on the CUDA architecture in terms of the thread and block organisation.

In contrast to Accelerate, which conceals the low-level details from application programmers, PGI Accelerator exposes much of the low-level details through the data region directive and the loop directive, if application programmers are willing to make an effort to fine-tune their applications; otherwise, they simply have to mark the compute region. This could be beneficial for the applications that are performance critical.

6.3.9 hiCUDA

hiCUDA \cite{79,80} is similar to PGI Accelerator in that it is also directive-based. In contrast to PGI Accelerator with which application programmers only have to mark the region that they wish to accelerate and leave the rest to the compiler, unless they want to fine-tune their applications, hiCUDA requires application programmers to manage low-level details. For example, application programmers are supposed to put \texttt{#pragma hicuda barrier}, which is comparable to \texttt{__syncthreads();} in CUDA, where a synchronisation is needed. If a loop does not have a partitioning directive, then the loop is executed on every thread, and it is not automatically partitioned.

Although the level of abstraction that hiCUDA provides is lower than PGI Accelerator’s, it still provides high-level abstraction over CUDA. It allows embedding of kernels in a C function, using directives —\texttt{#pragma hicuda kernel} marks the beginning of a kernel and \texttt{#pragma hicuda kernel_end} the end of a kernel— and it hides the manipulation of block IDs and thread IDs, using the loop partition directive.
6.3.10 HMPP

HMPP [81] is another directive-based GPU programming system. It has a multiple back ends that target CUDA, OpenCL, and multi-core CPUs. Unless specified, it uses the first available compatible platform. It has three kinds of directives: the codelet directive, the execution directive, the data transfer directive. The codelet directive is used on pure functions that are to be accelerated. The execution directive is used on the codelet invocations. The data transfer directive provides a means to specify when the codelet arguments are to be transferred between the host memory and the device memory. Without the data transfer directive, the codelet arguments are transferred to the device memory before the execution and the results are transferred back to the host memory after the execution.

Overall, HMPP provides a similar level of abstraction as PGI Accelerator with the advantage of supporting the acceleration at the function level.

6.3.11 OpenMP to GPGPU

Lee et al. implemented a framework that compiles OpenMP programs to CUDA programs [82]. The OpenMP to GPGPU framework consists of two parts: the OpenMP stream optimiser and the OpenMP-to-GPGPU (O₂G) translator with CUDA optimiser. The OpenMP stream optimiser transforms OpenMP programs, which are typically optimised for shared memory multiprocessing, into the form that is semantically equivalent but optimised for SPMD stream processing. The O₂G translator generates CUDA programs from the given OpenMP programs, optimising the programs with the CUDA specific features.

Fundamentally, they map parallel constructs to CUDA kernels, parallelise work-sharing constructs, partition a kernel further into two at synchronisation constructs to enforce global synchronisations, and define a set of rules to map the OpenMP data access model to the CUDA data access model.

This framework is similar to PGI Accelerator, hiCUDA, and HMPP in that it is also directive-based. However, this framework does not only provide an
abstraction on CUDA, but also allows to re-target existing OpenMP applications on GPUs.

6.3.12 CUDA-Lite

CUDA-Lite \cite{83} defines a set of annotations for application programmers to mark functions to run on GPUs, GPU arrays for which access must be optimised, and loops that are to be transformed appropriately for the CUDA platform. Based on the annotations given in the C source code, CUDA-lite generates optimised CUDA code.

The optimisation strategies of CUDA-Lite are specialised for coalescing global memory access. It assumes that every memory access in the user code is global memory access, and it transforms the global memory access to various patterns as needed.

6.4 Python-based GPU Programming Systems

6.4.1 PyCUDA

PyCUDA \cite{84, 85} is a GPU meta-programming system in Python, which comes with a multiple abstraction levels and performs runtime code generation. PyCUDA’s code generation strategy provides a greater degree of flexibility, compared to Accelerate’s, whose code generation is skeleton-based.

At the lowest level, it provides a binding to every feature of the CUDA runtime system that manages memory manipulation, kernel launches, etc. as well as a means to compile and load the strings that contain GPU kernels.

PyCUDA provides a high-level abstraction through the \texttt{GPUArray} class, which implements algebraic operations in the \texttt{NumPy} package \cite{86}, a Python array package. PyCUDA allows customisations of scalar operations applied to element-wise operations, or reduction operation, so that the code generator can generate user-defined kernels.
The runtime code generator in PyCUDA can be further customised in three ways:

- Given a kernel code string, PyCUDA allows simple keyword replacements. Using this feature, it is possible to replace a number of variables with a simple constant, and increase performance gain.

- Given a kernel code template string, users can define the value of the template arguments when generating the concrete kernel. It sounds similar to C++-style template programming, but it is different from C++-style in that the template instantiation and the code generation is done at runtime.

- PyCUDA provides facilities to build CUDA ASTs and manipulate them through the CodePy package, and it generates code from the ASTs.

Using the same approach, the authors of PyCUDA also implemented PyOpenCL, which targets OpenCL, instead of CUDA. Copperhead in Section 6.4.3 internally uses PyCUDA to generate code for a similar set of parallel operations as Accelerate.

### 6.4.2 CLyther

CLyther is another Python GPU programming system that targets OpenCL. The language constructs of CLyther embed OpenCL device language in Python, and allow users to write OpenCL kernels as Python functions, which are written as strings in C-style OpenCL programming environment. Then it generates the C-style OpenCL kernels, and executes them on GPUs by using the bindings to OpenCL runtime APIs.

CLyther increases the extensibility of the kernels, and it has a potential to enable even object-oriented OpenCL programming, whilst keeping the level of abstraction relatively low.
```python
def saxpy(a, x, y):
    """Add a vector y to a scaled vector a*x""
    return map(lambda xi, yi: a * xi + yi, x, y)
```

**Figure 6.1:** SAXPY written in Copperhead 4

### 6.4.3 Copperhead

Copperhead [89] is a Python EDSL for GPU programming. The building blocks in Copperhead language are collective operations such as replicate, map, zip, gather, permute, reduce, and scan like in Accelerate, Nikola, Barracuda, and Haskgraph. It lets users define functions with these collective operations and decorate such functions with `@cu` as shown in Figure 6.1 whereas Accelerate uses the type system to distinguish the Accelerate programs from the host program.

When a Copperhead program—i.e., Python function with the `@cu` decorator—is defined, the Copperhead front end builds an AST out of the source code using Python `ast` module. Then, when a Copperhead program is called for the first time, the Copperhead back end generates the CUDA kernels for the AST that the front end created from the program as well as the high-level C++ host code that launches the CUDA kernels, and it invokes the host code.

Although the host language, Python, is neither a purely functional programming language nor a statically typed language, Copperhead is a statically typed functional programming language. The Copperhead front end compiler infers type statically and rejects the programs that are not type-safe, and it disallows any operation with side-effects. Besides, the front end performs program transformations that are commonly found in functional programming languages such as closure conversion.

Compared to Accelerate compiler, Copperhead compiler gives more control of the code generation to the user: users can pass options to the compiler to generate a particular variant of the CUDA kernel from the same Copperhead program. Copperhead also performs some optimisations that Accelerate currently does not perform: It fuses collective operations, and sequentialises a program if necessary.
6.4.4 Theano

Theano [90] is a Python library that provides a means to build mathematical expression graphs using a symbolic representation. Theano supports the algebraic operations that NumPy [86] provides, and it allows the construction of the mathematical expressions graphs using those operations. Besides, Theano supports advanced mathematical operations such as mathematical expression simplification, symbolic differentiation of complex mathematical expressions, etc., making Theano more powerful and expressive in mathematical domains than other GPU programming systems introduced in this chapter.

The GPU back end of Theano can compile a given expression graph into CUDA kernels, and executes them on GPUs. Theano also provides a means to compile different parts of the expression graph into different kernels and execute them separately. Memory transfers between the host memory and the device memory can be controlled implicitly or explicitly in Theano, depending on the user preference.

6.4.5 PyGPU

PyGPU [91–93], is one of the early GPU programming EDSLs in Python. The problem domain that it aims to address is image processing. The language constructs in PyGPU are centred around image processing operations, and graphics primitives. Its back end targets NVIDIA’s Cg 1.4 shader, whose architecture is graphics-oriented similarly to DirectX 9 shader that Vertigo targets.

The functions written in PyGPU are decorated with @gpu, which is comparable to @cu in Copperhead. They are compiled into Cg shaders and executed on GPUs using PyCg, which is a binding to Cg 1.4 runtime. The compilation techniques that PyGPU uses can be adapted to more general settings.

6.4.6 PyStream

PyStream [94] is a GPU programming system that compiles a subset of Python specifically designed for the development of shader programs. PyStream comes
with an offline compiler that generates GLSL shaders along with a Python class that manages shaders, texture bindings, etc. The supported subset was chosen based on the restrictions imposed by the GLSL architecture, the compilation difficulties—as PyStream compiler deals with a language that is originally designed as an interpreted language—and other engineering difficulties.

PyStream shader programs are written with a convention that each shader program is defined as a Python class that implements special methods such as `shadeVertex`, and `shadeFragment` that define a vertex shader and a fragment shader. As long as the conventions are followed, users are allowed to leverage the significant portion of the abstraction and features that Python provides.

PyStream is different from Accelerate not only in the problem domain that it tackles, but also in that PyStream restricts the host language, Python, for the domain, and it is compiled statically, whereas Accelerate embeds a domain specific language in the host language, Haskell, and it is compiled dynamically.

### 6.4.7 UnPython and Jit4GPU

Garg and Amaral designed and implemented a GPU programming system in Python that compiles for loops to GPU kernels and executes them on GPUs [95]. This GPU programming system has two components: unPython, the offline compiler, and jit4GPU, the online compiler. UnPython compiles a module written in a subset of Python with annotations into C code. The C code generated by unPython is, then, linked with the rest of the application and jit4GPU, instead of the original Python module. When the application is executed, jit4GPU analyses the C code module from unPython, compiles it to GPU kernels, and executed them on GPUs.

The functions to be compiled to GPU kernels, are annotated with the decorator `@unpython.gpu`. A special iterator `prange` is used, instead of `xrange` or `range`, to denote the for loops that are to be parallelised on GPUs. During the compilation, jit4GPU runs various optimisations such as loop unrolling, loop fusion, memory access coalescing, and redundant memory access elimination. If
the range of a given loop is too large to fit in the device memory, it splits the loop into multiple tiles.

This programming system distinguishes itself from other Python GPU programming systems introduced in this chapter and Accelerate by parallelising loop constructs rather than higher order functions, and by targeting AMD GPUs. In addition to the GPU back end, it also has a CPU back end that leverages OpenMP.

6.5 Other GPU Programming Systems

6.5.1 Accelerator

Accelerator [96, 97] is a library, that is accessible from C#, F#, and C++ on the Windows platform. It targets DirectX 9 shaders, which does not provide access to several modern GPU features.

It defines a data-parallel array type and a set of array operations that the back ends can efficiently compile and execute on the target architectures, such as the data-parallel array construction, the conversion between data-parallel arrays and host array, element-wise operations, reductions, array transformations, and linear algebra operations. Accelerator builds a DAG for a set of collective array operations that are applied to a set of input, whereas Accelerate builds an AST. The online compilation and execution of a GPU program are triggered when the conversion of the result to host arrays.

The language constructs in Accelerator are different from Accelerate’s in that Accelerator’s are based on scalar operators lifted for array operations whereas Accelerate’s element-wise operations are based on higher order functions such as map, and zipWith. For example, the addition of two arrays —xs and ys— is expressed as Add(xs, ys) in Accelerator, but as zipWith (λ x y → x + y) xs ys in Accelerate. Accelerator also overloads some of the built-in operators for convenience. Accelerator generally provides less syntactic sugar than Accelerate.

Accelerator has an advantage over Accelerate that it is GPU vendor-neutral
as long as DirectX 9 is supported, and it supports multiple host languages, but it has an disadvantage that it is OS-specific. In addition to GPUs, Accelerator also targets multi-core CPUs (online compilation), and FPGAs (offline compilation).

6.5.2 Jacket

The Jacket [98] extension to Matlab defines new data types for matrices allocated in the device memory. Data transfers are triggered by casting between a host matrix and a device matrix, i.e., casting a host matrix to a device matrix triggers a data transfer from the host memory to the device memory, and vice versa. Jacket overloads built-in matrix operations for device matrices, and when those operations are executed on device matrices, GPU kernels are launched, taking those matrices as the input.

In general, Jacket provides a uniform interface regardless of the execution space —CPUs or GPUs. Jacket is also available as a C/C++ library that allows C/C++ programmers to execute the Matlab-style matrix operations using GPUs.
Chapter 7

Conclusion

As Moore’s law predicted, the performance of microprocessors has increased exponentially, and the costs per GFLOPS have decreased significantly over the past few decades. On the other hand, software development costs have not necessarily decreased, and in many cases, they have rather increased substantially due to the growing complexity of hardware platforms. GPU programming for general purpose computation is one of the examples that demonstrates this trend. The performance per cost that GPUs promise has become greater than what CPUs promise, however, it generally takes more time and effort to achieve the maximum performance on GPUs than on CPUs.

Throughout this thesis, we have explored a GPU programming language and system that provides facilities to alleviate the process to achieve performance gain close to maximum on GPUs. This chapter summarises the main contributions of this thesis, and lists some of the future research tasks to improve this GPU programming system even further.

7.1 Main Contributions

In this thesis, I introduced the Accelerate language—a Haskell EDSL specifically designed for array processing on parallel hardware platforms— which describes collective array operations in terms of higher-order functions commonly found in
functional programming languages. We decided to use an EDSL, instead of a new standalone language, so that we could leverage the implementation related features of the host language such as type system, foreign function interface, and I/O control. Moreover, learning a new language is an overhead for application programmers, and implementing full scale compilers like GHC and GCC for a new language from the ground up requires a great deal of effort and is out of the problem domain that I aimed to address. I also presented the Accelerate CUDA back end that generates device code modules using pre-defined algorithmic skeletons—which are mapped to the collective array operations in the language—and compiles and executes the generated device code modules online. As the algorithmic skeletons in the CUDA back end and the execution pipeline encapsulate the low-level details such as parallelisation, thread manipulation, memory management, and synchronisation, this approach allows application programmers to write parallel array programs without dealing with the characteristics and the restrictions of the underlying hardware platform. Furthermore, as collective array operations are expressions in Accelerate, which can be placed anywhere in applications, application programmers are able to decompose their applications based on the functionalities, and do not have to focus on the hardware constraints.

Although application programmers are given less control on the hardware—thus less room for fine-tuning—compared to other programming systems such as CUDA, I have demonstrated that they can still write applications in the Accelerate language and generate the device code modules, using the CUDA back end, whose performance is comparable to the device code modules directly written in CUDA in many cases. In the cases where the generated device code modules are outperformed by the device code modules directly written in CUDA, I analysed and evaluated the cause of the performance degradation and proposed potential solutions to close the performance discrepancies.

In summary, I have presented and evaluated a skeleton-based GPU programming system with a high-level abstraction that allows application programmers to focus more on what to program on the hardware than on how to program the hardware without unreasonable performance degradation in many cases.
7.2 Future Research

The benchmark results in Chapter 5 show that there is room for optimisations to improve the performance of the Accelerate pipeline, both the front end and the back end. There are also other features that we can implement in addition to the existing Accelerate features. I outline some of them in this section.

7.2.1 Multi-GPU Support

As of Accelerate-0.8.1.0, the CUDA back end only supports a single GPU. That is, even when there are more than one GPUs available in the system, the CUDA back end launches the device code modules on the first GPU among the available GPUs. We can take one of two approaches, or a combination of both to utilise multiple GPUs: data parallelism and task parallelism. The former is that we run each Accelerate operation using all of the available GPUs, and the latter is that we run each independent Accelerate operation on each available GPU.

We need to analyse which approach is more suitable for the Accelerate design. Then, we need to study how we want to deal with inter-GPU communication — inter-GPU communication is unavailable unless through the host memory— if we choose to implement data parallelism with multi-GPUs, or how we want to partition computation ASTs and schedule the sub-tasks if we choose to implement task parallelism with multi-GPUs.

7.2.2 Optimisations

Array Fusion

In Chapter 5, we discussed the necessity of array fusion, which is a well-known technique in the Haskell community to optimise a series of higher-order function invocations. For example, we can fuse the following combination of \texttt{scanl} and \texttt{map} into to one \texttt{scanl}:

\[
\text{scanl } f \ x \ \$\ \text{map } g \ \text{xs} \xrightarrow{\text{fusion}} \text{scanl } (\lambda x y \rightarrow f x (g y)) \ x \ \text{xs}
\]
Keller et al. presented an array fusion implementation strategy for array computations [63], and implemented an array library Repa [62], which can be adopted in Accelerate to reduce the number of the device code modules and the intermediate arrays. For the example above, if the array fusion from Repa was implemented in Accelerate, a back end would only have to generate and launch the device code, which reads each element from the input array \( xs \) — according to the index computation — and applies \( g \) to the element, then apply \( f \) to the left value and the result from the application of \( g \) to the input element. Index computation depends on whether the intermediate operations are structure-preserving or not: if the intermediate operations are structure-preserving, the index computation is simple and the index of the input element is the same as the index of the output element; if one or more intermediate operations are index space transformations such as Replicate and Index, the index computation becomes more complex, however, as 3.2.1 outlined, there is a set of rules for shapes and indices, and with which we can compute indices. If we succeeded with this approach, computation ASTs fed into back ends will only consist of the reduction operations such as scan and fold, and the operation at the root of the computation AST. As this optimisation can benefit other back ends, which may come in the future, as well as the CUDA back end, we plan to implement a middle-end with this optimisation.

In addition, to facilitate array fusion, we may need to generalise some of higher-order operations and their algorithmic skeletons. For example, we need to generalise fold and scan over the number of input arrays so that we can fuse these with both map and zipWith.

**Sharing and Common Sub-expression Elimination**

The benchmark with Black-Scholes options pricing suggests that we need to implement sharing and common sub-expression elimination (CSE). The array-level CSE has been already implemented in the Accelerate development branch, and will be included in the next release. However, Accelerate still lacks the scalar-level CSE. In fact, it would be desirable if the nvcc compiler itself would handle
it, as the performance degradation with redundant scalar computations is a general issue that occurs not only in those programs generated by the CUDA back end of Accelerate, but also in other CUDA programs similarly structured with redundant computations. Nonetheless, we could implement the scalar-level CSE in Accelerate and generate device code without redundant computations.

### 7.2.3 Efficient Hashing of Computation ASTs

The performance of the hash function and the hash table is not ideal as of Accelerate-0.8.1.0. The hash function takes the string representation of the collective array operations and their parameterising scalar operations, and hashes them to 32-bit integers, using a sample hash function supplied by the Haskell hash table library. Although the length of the string representation of a collective array operation is constant, the length of the string representation of a scalar operation can grow unexpectedly as users have the freedom to build scalar operation ASTs of any size as long as Accelerate supports each scalar expression of the scalar operation. When a complex and lengthy scalar operation is given, the hash function is likely to cause a bottleneck in the pipeline.

To define a hash table that works directly with collective array operations and scalar operations, we need to instantiate the type class `Eq` with `OpenAcc aenv t` and `OpenExp env aenv t` by implementing the `==` operator for each. It is possible to implement the `==` operator by comparing each node in computation ASTs, however, given the fact that the traversal of computation ASTs is slow, it does not necessarily speed up the hash performance. Moreover, we cannot rely on pointer equality as we support persistent caching of the CUDA modules, the hash function must generate the same hash key from the same operation across the application executions. In fact, even within a single application execution, the Haskell garbage collector may move objects around. The quality of the hash keys is another issue; it is required to generate hash keys that give an uniform distribution without much overhead.
In summary, to reduce the runtime overhead, we need an efficient hash function that produces the hash keys of computation ASTs with an uniform distribution in constant time, and ensures the referential transparency.

### 7.2.4 Porting to Other Parallel Platforms

There are a number of parallel platforms that have emerged in the last decades, including OpenCL [24], OpenMP [42], ArBB [71,72], FPGAs, etc. As they are all designed to process arrays efficiently, they are suitable platforms for Accelerate to target.

The implementation of Accelerate back ends for these platforms will alleviate the portability issues in heterogeneous environments as Accelerate provides a high-level abstraction focused on the computation rather than the hardware details, and each back end is meant to deal with the hardware details with efficient algorithmic skeletons. It will allow us to write applications once, focusing on the computations, whilst gaining speed-ups from various parallel platforms.

### 7.2.5 Data Parallel Haskell and Nested Data Parallelism

Data Parallel Haskell [101] (DPH) is a GHC extension to support nested data parallelism on parallel hardware platforms. As of GHC 6.12.3, its primary focus is multi-core CPUs and it is still work in progress. The authors and maintainers of the DPH project believe that they can target nested data parallelism and their approach to other parallel hardware platforms such as GPUs [55,102,103].

One approach to realise DPH on GPUs is to use Accelerate and its CUDA back end as a DPH back end. However, it requires a significant amount of research as Accelerate and GPU architectures only support flat data parallelism as of Version 0.8.1.0. It may require some new hardware features or flattening technique specifically designed to fit nested data parallelism in GPU architectures.
Appendix A

Outer Product in CUDA

The implementation of outer product I use in Chapter 5 is based on the tiling technique from the matrix transpose example in the CUDA SDK with 16-by-16 tiles, and each tile is processed by each thread block. That is, the size of the input arrays are limited to multiples of 16, and the device code `cudaOuterProduct` is launched with the following configuration, where `d_x` and `d_y` are the input arrays of size `m` and `n`, and `d_z` is the output array of size `m * n` with the row-major format.

```c
... outerProduct
  <<<dim3 (n / TILE_DIM, m / TILE_DIM), dim3 (TILE_DIM, TILE_DIM)>>
  (m, n, d_x, d_y, d_z);
...```

The device code basically performs outer product with the relevant input array segments. For example, the block (0, 0) takes the first 16 elements from `d_x` and the first 16 elements from `d_y`, produces the outer product of these two sub-arrays, and stores the result in `d_z`.

The device code, first of all, defines two arrays in the shared memory and the global indices to fetch the input array segments relevant to the block:

```c
__shared__ float _d_x[TILE_DIM];
__shared__ float _d_y[TILE_DIM];
```
Appendix A. Outer Product in CUDA

int d_xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
int d_yIndex = blockIdx.x * TILE_DIM + threadIdx.x;
int d_zIndex = ((blockIdx.y * TILE_DIM + threadIdx.y) * n) + d_yIndex;

It is to be noted that \( d_y \)Index uses \( threadIdx.x \) instead of \( threadIdx.y \) to ensure the coalescing of the global memory access; it is safe to use \( threadIdx.x \) instead of \( threadIdx.y \) as the tiles are square. The output index \( d_z \)Index is computed to store the output matrix in the row-major format, ensuring global memory coalescing.

The first half-warp and the second half-warp in the block fetch the first input array segment and the second input array segment respectively, and we synchronise all threads in the block to ensure the availability of the data before we process them.

if (threadIdx.y == 0) {
    d_x[threadIdx.x] = d_x[d_xIndex];
} else if (threadIdx.y == 1) {
    d_y[threadIdx.x] = d_y[d_yIndex];
}

__syncthreads();

Finally, the following code snippet computes the outer product of the input array segments. As all threads in the half-warp access the same element in \( d_x \)—the element is broadcasted— and different elements in \( d_y \), this computation does not cause bank conflicts.

\[
    z[d_zIndex] = d_x[threadIdx.y] \times d_y[threadIdx.x];
\]

Figure A.1 provides the complete implementation of the outer product device code.
Appendix A. Outer Product in CUDA

Figure A.1: The device code of outer product

```c
__global__ void outerProduct(int m, int n, const float *d_x, const float *d_y, float *d_z)
{
    __shared__ float _d_x[TILE_DIM];
    __shared__ float _d_y[TILE_DIM];

    int d_xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
    int d_yIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int index = ((blockIdx.y * TILE_DIM + threadIdx.y) * n) + d_yIndex;

    if (threadIdx.y == 0) {
        _d_x[threadIdx.x] = d_x[d_xIndex];
    } else if (threadIdx.y == 1) {
        _d_y[threadIdx.x] = d_y[d_yIndex];
    }

    __syncthreads();

    z[d_zIndex] = _d_x[threadIdx.y] * _d_y[threadIdx.x];
}
```
Appendix B

Manual Optimisations to Black-Scholes Options Pricing

This appendix provides the detailed information about the optimisations manually applied to the device code module that the Accelerate CUDA back end generates for Black-Scholes options pricing in Figure 5.9. The blue line in Figure 5.9 presents the performance of the generated device code module when no manual optimisation is applied, and the yellow line and the red line in the same figure present the performance of the generated device code module when manual optimisations are applied.

To evaluate the effect of the manual optimisations, we need to look at the generated device code module shown in Figure B.1. The scalar function `apply` has one large statement that defines the return value, which spans over four pages. Black-Scholes Options Pricing on 9 million input options with this generated code module executes more than 23 million instructions on Tesla S1070, whereas only about 0.74 million instructions are executed with the CUDA SDK implementation. In addition, the generated code module introduces divergent branches, which is not found in the CUDA SDK implementation; the conditional statement in the CUDA SDK implementation is easily optimised to a predicated execution as the consequent expression only subtracts a value from a constant.

The first manual optimisation applied to the generated device code module
Table B.1: The comparison of the number of instructions and the number of divergent branches

<table>
<thead>
<tr>
<th>9,000,000 Options</th>
<th>CUDA SDK</th>
<th>Without Sharing</th>
<th>Sharing CND Values</th>
<th>Sharing CND Values and D Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions</td>
<td>741,888</td>
<td>23,554,761</td>
<td>7,295,091</td>
<td>1,350,552</td>
</tr>
<tr>
<td>Divergent Branches</td>
<td>0</td>
<td>37,376</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Registers Per Thread</td>
<td>17</td>
<td>22</td>
<td>22</td>
<td>19</td>
</tr>
</tbody>
</table>

is the sharing of the yellow-coloured expression and the green-coloured expression. These expressions are equivalent to \( \text{CNDD1} \) and \( \text{CNDD2} \) respectively in the CUDA SDK implementation, and each of them appear four times in the generated code. Figure B.2 shows the code after applying the first optimisation. The yellow-coloured expression is bound to the variable \( \text{cndd1} \) and the green-coloured expression to the variable \( \text{cndd2} \), and these variable are used in the places of these expressions.

As Table B.1 presents, this optimisation does not only reduce the number of the instructions, but it also removes the divergent branches. This optimisation improves the performance by more than 300%, and the yellow line in Figure 5.9 depicts the performance of the generated device code module with the first optimisation.

The second manual optimisation applied to the generated device code on top of the first manual optimisation is the sharing of the red-coloured expression and the blue-coloured expression in Figure B.2 and Figure B.3 shows the resulting code. In fact, the red-coloured expression also appears in the underlined part of blue-coloured expression, and the sharing is applied there as well. The red expression represents the value of the variable \( d1 \) in the CUDA SDK implementation and the blue expression the value of the variable \( d2 \). The second optimisation binds these two expression to the variables \( d1 \) and \( d2 \), and use these variables instead of the lengthy expressions.

As the result of the second optimisation, the number of the instructions executed on Tesla S1070 decreases to approximately 1/17, and the performance
improves approximately 18 times in comparison to the case without sharing. The red line in Figure 5.9 and Table B.1 illustrates the benefits of the second optimisation.

After all, these two manual optimisations improve the performance of the generated device code significantly to the range comparable to the performance of the CUDA SDK implementation. It proves the importance of the recovery of sharing, and the co-authors of the Accelerate are completing the implementation of a variant of Gill’s 104 observable sharing for the future release.
static inline __device__ TyOut apply(const TyIn0 x0)
{
    TyOut r =
    { x0.a2 * ((logf(x0.a2 / x0.a1) + ((float) 2.0e-2 + (float) 0.5 * (float) 0.3 * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0)) >
        (float) 0.0) ? (float) 1.0 - (float) 0.3989423 * expf((float) -0.5 *
        ((logf(x0.a2 / x0.a1) + ((float) 2.0e-2 + (float) 0.5 * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0))) *
        ((logf(x0.a2 / x0.a1) + ((float) 2.0e-2 + (float) 0.5 * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0))) * (float) 1.0 / ((float) 1.0 + (float) 0.2316419 * fabsf((logf(x0.a2 / x0.a1) + ((float) 2.0e-2 + (float) 0.5 * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0)))) *
        (float) 1.0 / ((float) 1.0 + (float) 0.2316419 * fabsf((logf(x0.a2 / x0.a1) + ((float) 2.0e-2 + (float) 0.5 * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0)))) *
        (float) 0.3 * sqrtf(x0.a0)) * ((float) 0.31938154 + (float) 1.0 / ((float) 1.0 + (float) 0.2316419 * fabsf((logf(x0.a2 / x0.a1) + ((float) 2.0e-2 + (float) 0.5 * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0)))) * (float) 0.3989423 * expf((float) -0.5 *
        ((logf(x0.a2 / x0.a1) + ((float) 2.0e-2 + (float) 0.5 * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0))) *
        (float) 1.0 / ((float) 1.0 + (float) 0.2316419 * fabsf((logf(x0.a2 / x0.a1) + ((float) 2.0e-2 + (float) 0.5 * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0)))) *
        (float) 1.0 / ((float) 1.0 + (float) 0.2316419 * fabsf((logf(x0.a2 / x0.a1) + ((float) 2.0e-2 + (float) 0.5 * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0)))) *
        (float) 0.3 * sqrtf(x0.a0)) * (float) 1.3302745)))):
        (float) 0.3989423 * expf((float) -0.5 * ((logf(x0.a2 / x0.a1) + ((float) 2.0e-2 + (float) 0.5 * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0)))) *
        (float) 1.0 / ((float) 1.0 + (float) 0.2316419 * fabsf((logf(x0.a2 / x0.a1) + ((float) 2.0e-2 + (float) 0.5 * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0)))) *
        (float) 1.0 / ((float) 1.0 + (float) 0.2316419 * fabsf((logf(x0.a2 / x0.a1) + ((float) 2.0e-2 + (float) 0.5 * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0)))) *
        (float) 0.3989423 * expf((float) -0.5 * ((logf(x0.a2 / x0.a1) + ((float) 2.0e-2 + (float) 0.5 * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0)))) *
        (float) 1.0 / ((float) 1.0 + (float) 0.2316419 * fabsf((logf(x0.a2 / x0.a1) + ((float) 2.0e-2 + (float) 0.5 * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0)))) *
        (float) 1.0 / ((float) 1.0 + (float) 0.2316419 * fabsf((logf(x0.a2 / x0.a1) + ((float) 2.0e-2 + (float) 0.5 * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0)))) *
        (float) 1.0 / ((float) 1.0 + (float) 0.2316419 * fabsf((logf(x0.a2 / x0.a1) + ((float) 2.0e-2 + (float) 0.5 * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0)))) *
        (float) 0.3 * sqrtf(x0.a0)) * (float) 1.3302745)))) - x0.a1 * expf(-(float)
\begin{verbatim}
2.0e-2 * x0.a0)) * ((logf(x0.a2 / x0.a1) + (float) 2.0e-2 + (float)
0.5 * (float) 0.3 * (float) 0.3) * x0.a0) / ((float) 0.3 * 
sqrtf(x0.a0)) - (float) 0.3 * sqrtf(x0.a0) > (float) 0.0 ? (float)
1.0 - (float) 0.3989423 * expf((float) -0.5 * ((logf(x0.a2 / x0.a1)
+ (float) 2.0e-2 + (float) 0.5 * (float) 0.3) * (float) 0.3) *
x0.a0) / ((float) 0.3 * sqrtf(x0.a0)) - (float) 0.3 * sqrtf(x0.a0))
* ((logf(x0.a2 / x0.a1) + (float) 2.0e-2 + (float) 0.5 * (float)
0.3 * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0)) - (float)
0.3 * sqrtf(x0.a0)) * ((float) 1.0 / ((float) 1.0 + (float)
0.2316419 * fabsf((logf(x0.a2 / x0.a1) + (float) 2.0e-2 + (float)
0.5 * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0))) * ((float) 0.
3 * sqrtf(x0.a0)) - (float) 0.3 * sqrtf(x0.a0)) - (float) 0.3 * 
sqrtf(x0.a0))) * ((float) -0.35656378 + (float) 1.0 / ((float) 1.0 +
(float) 0.2316419 * fabsf((logf(x0.a2 / x0.a1) + (float) 2.0e-2 +
(float) 0.5 * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0))
- (float) 0.3 * sqrtf(x0.a0)) * ((float) 1.7814779 + (float)
1.0 / ((float) 1.0 + (float) 0.2316419 * fabsf((logf(x0.a2 / x0.
a1) + (float) 2.0e-2 + (float) 0.5) * (float) 0.3) * (float) 0.3)
* x0.a0) / ((float) 0.3 * sqrtf(x0.a0)) - (float) 0.3 *
sqrtf(x0.a0))) * ((float) -1.8212559 + (float) 1.0 / ((float) 1.0 +
(float) 0.2316419 * fabsf((logf(x0.a2 / x0.a1) + (float) 2.0e-2 +
(float) 0.5 * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0))
- (float) 0.3 * sqrtf(x0.a0))) * ((float) 0.31938154 + (float)
1.0 / ((float) 1.0 + (float) 0.2316419 * fabsf((logf(x0.a2 / x0.
a1) + (float) 2.0e-2 + (float) 0.5 * (float) 0.3) * (float) 0.3)
* x0.a0) / ((float) 0.3 * sqrtf(x0.a0)) - (float) 0.3 * 
sqrtf(x0.a0))) * ((float) 0.3989423 * expf((float) -0.5 * ((logf(x0.
a2 / x0.a1) + (float) 2.0e-2 + (float) 0.5 * (float) 0.3 * (float)
0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0)) - (float) 0.3 * sqrtf(x0.
a0)) - (float) 0.5 * (float) 0.3 * (float) 0.3) * x0.a0) / ((float)
0.3 * sqrtf(x0.a0))) * ((float) 1.3302745))))) : (float) 0.3998423
* expf((float) -0.5 * ((logf(x0.a2 / x0.a1) + (float) 2.0e-2 + (float)
0.5 * (float) 0.3) * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.
a0)) - (float) 0.3 * sqrtf(x0.a0)) * ((logf(x0.a2 / x0.a1) + (float)
2.0e-2 + (float) 0.5) * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.
a0)) - (float) 0.3 * sqrtf(x0.a0)) * ((float) 0.31938154 + (float)
1.0 / ((float) 1.0 + (float) 0.2316419 * fabsf((logf(x0.a2 / x0.
a1) + (float) 2.0e-2 + (float) 0.5) * (float) 0.3) * (float) 0.3)
* x0.a0) / ((float) 0.3 * sqrtf(x0.a0)) - (float) 0.3 *
sqrtf(x0.a0))) * ((float) 0.3989423 * expf((float) -0.5 * ((logf(x0.
a2 / x0.a1) + (float) 2.0e-2 + (float) 0.5 * (float) 0.3 * (float)
0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0)) - (float) 0.3 * sqrtf(x0.
a0)) - (float) 0.5 * (float) 0.3 * (float) 0.3) * x0.a0) / ((float)
0.3 * sqrtf(x0.a0))) * ((float) 0.31938154 + (float) 1.0 / ((float)
1.0 + (float) 0.2316419 * fabsf((logf(x0.a2 / x0.a1) + (float)
2.0e-2 + (float) 0.5) * (float) 0.3) * (float) 0.3) * x0.a0) / ((float)
0.3 * sqrtf(x0.a0)) - (float) 0.3 * sqrtf(x0.a0))) * ((float) 0.3998423
* expf((float) -0.5 * ((logf(x0.a2 / x0.a1) + (float) 2.0e-2 + (float)
0.5 * (float) 0.3) * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.
a0)) - (float) 0.3 * sqrtf(x0.a0)) - (float) 0.5 * (float) 0.3 * 

Figure B.1: The Black-Scholes Options Pricing device code generated by Accelerate without sharing (Continued)
\end{verbatim}
Figure B.1: The Black-Scholes Options Pricing device code generated by Accelerate without sharing (Continued)
Appendix B. Manual Optimisations to Black-Scholes Options Pricing

Figure B.1: The Black-Scholes Options Pricing device code generated by Accelerate without sharing (Continued)
static inline __device__ TyOut apply(const TyIn0 x0)
{
    float cndd1 =
        (float) 0.3989423 * expf((float) -0.5 * ((logf(x0.a2 / x0.a1) + ((float) 2.0e-2 + (float) 0.5 * (float) 0.3 * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0))) +
            (float) 2.0e-2 + (float) 0.5 * (float) 0.3 * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0))) * ((logf(x0.a2 / x0.a1) + ((float) 2.0e-2 + (float) 0.5 * (float) 0.3 * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0))) +
            (float) 0.31938154 + (float) 1.0 / ((float) 1.0 + (float) 0.2316419 * fabsf((logf(x0.a2 / x0.a1) + ((float) 2.0e-2 + (float) 0.5 * (float) 0.3 * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0))) +
            (float) 1.0))));

    return r;
}
float cndd2 =
(float) 0.3989423 * expf((float) -0.5 * (logf(x0.a2 / x0.a1) +
((float) 2.0e-2 + (float) 0.5 * (float) 0.3 * (float) 0.3) * x0.a0) /
((float) 0.3 * sqrtf(x0.a0)) - (float) 0.3 * sqrtf(x0.a0)) *
((logf(x0.a2 / x0.a1) + ((float) 2.0e-2 + (float) 0.5 * (float) 0.3 *
(float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0)) - (float) 0.3 *
sqrtf(x0.a0))) * (float) 1.0 / ((float) 1.0 + (float) 0.2316419 *
fabsf((logf(x0.a2 / x0.a1) + ((float) 2.0e-2 + (float) 0.5 * (float)
0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0)) - (float)
0.3 * sqrtf(x0.a0)) - (float) 0.3 * sqrtf(x0.a0)) - (float)
1.0 + (float) 0.2316419 * fabsf((logf(x0.a2 / x0.a1) + ((float)
2.0e-2 + (float) 0.5 * (float) 0.3 * (float) 0.3) * x0.a0) / ((float)
0.3 * sqrtf(x0.a0)) - (float) 0.3 * sqrtf(x0.a0)) - (float)
-0.35656378 + (float) 1.0 / ((float) 1.0 + (float) 0.2316419 *
fabsf((logf(x0.a2 / x0.a1) + ((float) 2.0e-2 + (float) 0.5 * (float)
0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0)) - (float) 0.3 * sqrtf(x0.a0)) - (float) 0.3 * sqrtf(x0.a0)) - (float)
0.3 * sqrtf(x0.a0)) - (float) 1.0 + (float) 0.2316419 * fabsf((logf(x0.a2 /
x0.a1) + ((float) 2.0e-2 + (float) 0.5 * (float) 0.3) * x0.a0) / ((float)
0.3 * sqrtf(x0.a0)) - (float) 0.3 * sqrtf(x0.a0)) - (float)
1.0 + (float) 0.3 * sqrtf(x0.a0)) - (float) 1.3302745));

TyOut r =
{ x0.a2 * ((logf(x0.a2 / x0.a1) + ((float) 2.0e-2 + (float) 0.5 *
(float) 0.3 * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0)) >
(float) 0.0 ? (float) 1.0 - cndd1 : cndd1) - x0.a1 * expf(-((float)
2.0e-2 * x0.a0)) * ((logf(x0.a2 / x0.a1) + ((float) 2.0e-2 + (float)
0.5 * (float) 0.3 * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0)) -
(float) 0.3 * sqrtf(x0.a0)) - (float) 0.3 * sqrtf(x0.a0) > (float) 0.0 ? (float)
1.0 - cndd2 : cndd2)
, x0.a1 * expf(-((float) 2.0e-2 * x0.a0)) * ((float) 1.0 -
((logf(x0.a2 / x0.a1) + ((float) 2.0e-2 + (float) 0.5) * (float) 0.3 *
(float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0)) - (float) 0.3 *
sqrtf(x0.a0) > (float) 0.0 ? (float) 1.0 - cndd2 : cndd2)) - x0.a2
* ((float) 1.0 - ((logf(x0.a2 / x0.a1) + ((float) 2.0e-2 + (float)
0.5 * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0)) - (float) 0.3 *
sqrtf(x0.a0)) > (float) 0.0 ? (float) 1.0 - cndd1 : cndd1) }

return r;
static inline __device__ TyOut apply(const TyIn0 x0)
{
    float d1 =
        (logf(x0.a2 / x0.a1) + ((float) 2.0e-2 + (float) 0.5 * (float) 0.3 * (float) 0.3) * x0.a0) / ((float) 0.3 * sqrtf(x0.a0));
    float d2 =
        d1 - (float) 0.3 * sqrtf(x0.a0);
    float cndd1 =
        (float) 0.3989423 * expf((float) -0.5 * d1 * d1) * ((float) 1.0 / ((float) 1.0 + (float) 0.31938154 + (float) 0.2316419 * fabsf(d1)) * ((float) 0.31938154 + (float) 1.0 / ((float) 1.0 + (float) 0.2316419 * fabsf(d1)) * ((float) -0.35656378 + (float) 1.0 / ((float) 1.0 + (float) 0.2316419 * fabsf(d1)) * ((float) 1.7814779 + (float) 1.0 / ((float) 1.0 + (float) 0.2316419 * fabsf(d1)) * ((float) 1.3302745)))));
    float cndd2 =
        (float) 0.3989423 * expf((float) -0.5 * d2 * d2) * ((float) 1.0 / ((float) 1.0 + (float) 0.31938154 + (float) 0.2316419 * fabsf(d2)) * ((float) 0.31938154 + (float) 1.0 / ((float) 1.0 + (float) 0.2316419 * fabsf(d2)) * ((float) -0.35656378 + (float) 1.0 / ((float) 1.0 + (float) 0.2316419 * fabsf(d2)) * ((float) 1.7814779 + (float) 1.0 / ((float) 1.0 + (float) 0.2316419 * fabsf(d2)) * ((float) -1.8212559 + (float) 1.0 / ((float) 1.0 + (float) 0.2316419 * fabsf(d2)) * (float) 1.3302745))));
    TyOut r =
        { x0.a2 * (d1 > (float) 0.0 ? (float) 1.0 - cndd1 : cndd1) - x0.a1 * expf(-((float) 2.0e-2 * x0.a0)) * (d2 > (float) 0.0 ? (float) 1.0 - cndd2 : cndd2),
          x0.a1 * expf(-((float) 2.0e-2 * x0.a0)) * ((float) 1.0 - (d2 > (float) 0.0 ? (float) 1.0 - cndd2 : cndd2)) - x0.a2 * ((float) 1.0 - (d1 > (float) 0.0 ? (float) 1.0 - cndd1 : cndd1))};
    return r;
}

Figure B.3: The Black-Scholes Options Pricing device code generated by Accelerate sharing CND values and D values
Bibliography


