

---

---

# Optimising Purely Functional GPU Programs

---

---

TREVOR L. McDONELL

A THESIS IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF  
DOCTOR OF PHILOSOPHY



**UNSW**  
AUSTRALIA

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING  
UNIVERSITY OF NEW SOUTH WALES

2015



This thesis is submitted in fulfilment  
of the requirements for the degree of

**DOCTOR OF PHILOSOPHY**

School of Computer Science and Engineering  
The University of New South Wales  
Australia



---

---

## Contents

---

---

<b>Optimising Purely Functional GPU Programs</b>	<b>i</b>
<b>Contents</b>	<b>v</b>
<b>Figures</b>	<b>xi</b>
<b>Listings</b>	<b>xiii</b>
<b>Tables</b>	<b>xv</b>
<b>Abstract</b>	<b>xvii</b>
<b>Acknowledgements</b>	<b>xix</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Basics</b>	<b>5</b>
2.1 Data parallelism . . . . .	5
2.2 GPU computing . . . . .	6
2.3 CUDA . . . . .	7
2.4 Data parallel algorithms in CUDA . . . . .	8
2.4.1 Reduction . . . . .	8
2.4.2 Scan . . . . .	10
2.4.3 Backpermute . . . . .	12
2.4.4 Permute . . . . .	13
2.4.5 Stencil . . . . .	16
2.4.6 Segmented operations . . . . .	19
2.5 Embedded domain-specific languages . . . . .	20
2.5.1 Shallow embedding . . . . .	20
2.5.2 Deep embedding . . . . .	21
2.6 Discussion . . . . .	21

<b>3 Embedding array computations</b>	<b>23</b>
3.1 The Accelerate EDSL . . . . .	23
3.1.1 Computing a vector dot product . . . . .	25
3.1.2 Arrays, shapes, and indices . . . . .	25
3.1.3 Arrays on the host and device . . . . .	26
3.1.4 Array computations versus scalar expressions . . . . .	27
3.1.5 Computing an $n$ -body gravitational simulation . . . . .	28
3.1.6 Non-parametric array representation . . . . .	30
3.1.7 Richly typed terms . . . . .	31
3.1.8 Tying the recursive knot . . . . .	33
3.2 Manipulating embedded programs . . . . .	34
3.2.1 Typed equality . . . . .	34
3.2.2 Simultaneous substitution . . . . .	36
3.2.3 Inlining . . . . .	38
3.2.4 Function composition . . . . .	38
3.2.5 Environment manipulation . . . . .	39
3.3 Related work . . . . .	40
3.4 Discussion . . . . .	41
<b>4 Accelerated array code</b>	<b>43</b>
4.1 Embedding GPU programs as skeletons . . . . .	43
4.1.1 Array operations as skeletons . . . . .	44
4.1.2 Algorithmic skeletons as template meta programs . . . . .	45
4.2 Instantiating skeletons . . . . .	46
4.2.1 Producer-consumer fusion by template instantiation . . . . .	46
4.2.2 Instantiating skeletons with scalar code . . . . .	48
4.2.3 Representing tuples . . . . .	50
4.2.4 Shared subexpressions . . . . .	52
4.2.5 Eliminating dead code . . . . .	53
4.2.6 Shapes and indices . . . . .	55
4.2.7 Lambda abstractions . . . . .	56
4.2.8 Array references in scalar code . . . . .	57
4.2.9 Conclusion . . . . .	58
4.3 Using foreign libraries . . . . .	59
4.3.1 Importing foreign functions . . . . .	59
4.3.2 Exporting Accelerate programs . . . . .	60
4.4 Dynamic compilation & code memoisation . . . . .	60
4.4.1 External compilation . . . . .	61
4.4.2 Caching compiled kernels . . . . .	62

4.4.3	Conclusion	65
4.5	Data transfer & garbage collection	65
4.5.1	Device-to-host transfers	65
4.5.2	Host-to-device transfers	66
4.5.3	Allocation & Deallocation	67
4.5.4	Memory manager implementation	68
4.5.5	Conclusion	71
4.6	Executing embedded array programs	72
4.6.1	Execution state	73
4.6.2	Annotating array programs	73
4.6.3	Launch configuration & thread occupancy	76
4.6.4	Kernel execution	76
4.6.5	Kernel execution, concurrently	77
4.6.6	Conclusion	81
4.7	Related work	81
4.7.1	Embedded languages	81
4.7.2	Parallel libraries	82
4.7.3	Other approaches	83
4.8	Discussion	83
<b>5</b>	<b>Optimising embedded array programs</b>	<b>85</b>
5.1	Sharing recovery	85
5.1.1	Our approach to sharing recovery	87
5.2	Array fusion	89
5.2.1	Considerations	90
5.2.2	The Main Idea	91
5.2.3	Representing Producers	94
5.2.4	Producer/Producer fusion	96
5.2.5	Producer/Consumer fusion	101
5.2.6	Exploiting all opportunities for fusion	103
5.3	Simplification	108
5.3.1	Shrinking	108
5.3.2	Common subexpression elimination	108
5.3.3	Constant folding	109
5.4	Related work	112
5.4.1	Shortcut fusion	112
5.4.2	Loop fusion in imperative languages	113
5.4.3	Fusion in a parallel context	113
5.5	Discussion	114

5.5.1	Correctness . . . . .	115
5.5.2	Confluence and termination . . . . .	116
5.5.3	Error tests eliminated . . . . .	116
5.5.4	Work duplication . . . . .	117
5.5.5	Ordering and repeated evaluations . . . . .	117
<b>6</b>	<b>Results</b>	<b>119</b>
6.1	Runtime overheads . . . . .	119
6.1.1	Program optimisation . . . . .	120
6.1.2	Memory management & data transfer . . . . .	120
6.1.3	Code generation & compilation . . . . .	121
6.2	Dot product . . . . .	121
6.2.1	Too many kernels . . . . .	123
6.2.2	64-bit arithmetic . . . . .	124
6.2.3	Non-neutral starting elements . . . . .	125
6.2.4	Kernel specialisation . . . . .	126
6.3	Black-Scholes option pricing . . . . .	126
6.3.1	Too little sharing . . . . .	127
6.4	Mandelbrot fractal . . . . .	128
6.4.1	Fixed unrolling . . . . .	129
6.4.2	Loop recovery . . . . .	131
6.4.3	Explicit iteration . . . . .	132
6.4.4	Unbalanced workload . . . . .	133
6.4.5	Passing inputs as arrays . . . . .	133
6.5	N-body gravitational simulation . . . . .	134
6.5.1	Sequential iteration . . . . .	135
6.5.2	Use of shared memory . . . . .	135
6.6	Sparse-matrix vector multiplication . . . . .	136
6.6.1	Segment startup . . . . .	136
6.7	Canny edge detection . . . . .	138
6.7.1	Stencil merging . . . . .	140
6.8	Fluid flow . . . . .	140
6.9	Radix sort . . . . .	142
6.10	Shortest paths in a graph . . . . .	145
6.11	MD5 password recovery . . . . .	146
6.12	K-Means clustering . . . . .	150
6.13	Ray tracer . . . . .	152
6.14	LULESH . . . . .	153
6.15	Discussion . . . . .	157

<b>7 Conclusion</b>	<b>159</b>
7.1 Research contribution . . . . .	161
7.1.1 A high-performance embedded language . . . . .	161
7.1.2 A fusion system for parallel array languages . . . . .	161
7.2 Future work . . . . .	162
7.2.1 Nested data parallelism . . . . .	162
7.2.2 Type-preserving code generation . . . . .	162
7.2.3 Flexibility in the optimisation pipeline . . . . .	163
<b>Bibliography</b>	<b>165</b>
<b>Index</b>	<b>177</b>



---

---

## Figures

---

---

2.1	Block diagram comparing CPU and GPU architectures . . . . .	6
2.2	A parallel tree reduction . . . . .	9
2.3	A parallel inclusive scan . . . . .	11
2.4	Application of a $3 \times 3$ stencil in the border region . . . . .	17
2.5	Overlapping elements in a $3 \times 3$ stencil . . . . .	18
3.1	The overall structure of Accelerate . . . . .	24
4.1	Impact of varying thread block size on multiprocessor occupancy . . . . .	77
4.2	Profiling trace of a program which executes kernels concurrently . . . . .	80
5.1	Recovering sharing in an example term . . . . .	87
5.2	Fusion in Accelerate . . . . .	93
6.1	Program optimisation runtimes . . . . .	121
6.2	Data transfer time and bandwidth measurements . . . . .	122
6.3	Vector dot product kernel benchmarks . . . . .	124
6.4	Black-Scholes kernel benchmarks . . . . .	127
6.5	The Mandelbrot fractal . . . . .	128
6.6	N-body gravitational simulation kernel benchmarks . . . . .	134
6.7	Example of the Canny edge detection algorithm . . . . .	139
6.8	Canny edge detection benchmarks . . . . .	139
6.9	Example of the fluid flow simulation . . . . .	142
6.10	Fluid flow simulation kernel benchmarks . . . . .	144
6.11	Radix sort kernel benchmarks . . . . .	145
6.12	Floyd-Warshall shortest path benchmark . . . . .	147
6.13	A single round of the MD5 hash algorithm . . . . .	148
6.14	K-means clustering kernel benchmarks . . . . .	152
6.15	Ray tracer . . . . .	153
6.16	Ray tracing kernel benchmarks . . . . .	155

XII FIGURES

6.17 Software architecture of hydrodynamics applications . . . . .	156
6.18 LULESH kernel benchmarks . . . . .	156

---

---

## Listings

---

2.1	CUDA kernel for pair wise addition of two vectors . . . . .	7
2.2	Filtering a vector based on a predicate . . . . .	13
2.3	A simple histogram . . . . .	14
3.1	Types of array shapes and indices . . . . .	26
3.2	Core Accelerate array operations . . . . .	27
3.3	A simultaneous substitution to inline terms . . . . .	38
3.4	A simultaneous substitution to compose unary function terms . . . . .	38
3.5	Extending an array environment . . . . .	39
3.6	Sinking terms to a larger environment . . . . .	40
4.1	Accelerate CUDA skeleton for the <code>map</code> operation . . . . .	45
4.2	Accelerate CUDA skeleton for the <code>foldAll</code> operation . . . . .	48
4.3	Generated CUDA code for the fused vector dot-product operation . . . . .	49
4.4	Method to lookup device arrays . . . . .	69
4.5	Method to allocate a new device array . . . . .	70
4.6	Device memory finaliser . . . . .	71
4.7	Example program that executes kernels concurrently . . . . .	80
5.1	Black-Scholes option pricing . . . . .	86
5.2	Producer and consumer operations . . . . .	92
5.3	Representation of fusible producer arrays . . . . .	95
5.4	Representation of fused producer arrays . . . . .	95
5.5	Computing the delayed representation to a manifest array . . . . .	97
5.6	Smart constructor for fusing the <code>map</code> operation . . . . .	97
5.7	Smart constructor for fusing the <code>zipWith</code> operation . . . . .	99
5.8	Producer fusion via bottom-up contraction of the AST . . . . .	100
5.9	Representation of delayed arrays . . . . .	101
5.10	Consumer fusion via top-down annotation of the AST . . . . .	102
5.11	Smart constructor for let bindings . . . . .	106

5.12 Determining when a let binding should be eliminated . . . . .	108
5.13 The shrinking reduction . . . . .	109
5.14 Example of constant expression evaluation . . . . .	112
6.1 Vector dot-product . . . . .	121
6.2 Mandelbrot set generator, using fixed unrolling . . . . .	130
6.3 Mandelbrot set generator, using explicit iteration . . . . .	132
6.4 $N$ -body gravitational simulation, using parallel reduction . . . . .	135
6.5 $N$ -body gravitational simulation, using sequential reduction . . . . .	135
6.6 Sparse-matrix vector multiplication . . . . .	136
6.7 The Canny edge detection algorithm . . . . .	140
6.8 The data-parallel kernels of the Canny edge detection algorithm . . . . .	141
6.9 Fluid flow simulation . . . . .	143
6.10 Radix sort algorithm . . . . .	144
6.11 Floyd-Warshall shortest-paths algorithm . . . . .	146
6.12 A single round of the MD5 hash algorithm . . . . .	149
6.13 $K$ -means clustering for 2D points . . . . .	151
6.14 The core ray casting implementation . . . . .	154

---

---

## Tables

---

---

6.1	Benchmark summary . . . . .	120
6.2	Benchmark code generation and compilation times . . . . .	123
6.3	Mandelbrot fractal kernel benchmarks . . . . .	129
6.4	Sparse-matrix vector multiplication benchmarks . . . . .	137
6.5	MD5 password recovery benchmarks . . . . .	150



---

---

## Abstract

---

---

It is well acknowledged that the dominant mechanism for scaling processor performance has become to increase the number of cores on a chip, rather than improve the performance of a single core. However, harnessing these extra cores to improve single application performance remains an extremely challenging task. A recent trend has been to use commodity graphics processing units to explore new algorithms and programming languages that can address the challenges of parallelism and scalability for devices containing hundreds or thousands of cores.

The research documented in this dissertation builds upon the *Accelerate* language, an embedded domain specific language of purely functional collective operations over dense, multidimensional arrays. Overall, this dissertation explains how to efficiently implement such a language, ranging from optimisations to the input program and the generation of efficient target code, to optimisations in the runtime system which executes the resulting programs.

In particular, I add backend-agnostic optimisations to the embedded array language, primarily in the form of a novel approach to array fusion. These optimisations are completely type preserving, amounting to a partial proof of correctness of the optimisations, checked by the type checker at compilation time.

In addition, I develop the CUDA backend to Accelerate, which targets parallel execution on multicore graphics processing units (GPUs). I implement a dynamic quasi quotation-based code generation system for algorithmic skeletons that are instantiated at runtime. In order to minimise the overheads of runtime compilation, generated codes are compiled in parallel and asynchronously with other operations. Here, compiled binaries are cached for later reuse, including across program invocations. As a significant overhead in GPU programs is the transfer of data to and from the device, a garbage collection-based memory management system was implemented which minimises data transfers. Finally, I implement an execution engine that optimises the loading, configuration, and concurrent execution of the compiled GPU kernels.

My results show that with these additions, programs written in Accelerate can be competitive in performance to programs written in traditional lower-level languages, while enjoying a much higher level of abstraction. Through the work done in this thesis, Accelerate has become the dominant method for GPGPU programming in Haskell.



---

---

## Acknowledgements

---

---

My supervisors Manuel and Gabi, for their (usually) gentle guidance, and for allowing me the opportunity and (most importantly) the time to explore possibilities.

My friends and family, for putting up with me when things got out of hand  
(as they inevitably do).

And of course, to Josephine, for her unwavering support and encouragement.



---

## Introduction

---

Unless someone like you cares a whole awful lot,  
nothing is going to get better. It's not.

—DR. SEUSS

*The Lorax*

The beginning of the twenty first century has seen an interesting trend in computing emerge. General purpose CPUs, such as those provided by Intel, IBM and AMD, have increased performance substantially, but with nowhere near the increases seen in the late 1980’s and early 1990’s. To a large extent, single threaded performance increases have tapered off, due to the low level of inter-process communication in general purpose workloads, and the physical limitations of power dissipation for integrated circuits. The additional millions and billions of transistors afforded by Moore’s rule<sup>1</sup> are simply not very productive in increasing the performance of single-threaded code.

At the same time, the commodity *graphics processing unit* (GPU) has been able to use this ever increasing transistor budget effectively, geometrically increasing rendering performance, as rasterisation is an inherently parallel operation. Modern GPUs are massively parallel multicore processors, offering instruction throughput and memory bandwidth rates much higher than those found on traditional CPUs [98]. However, despite the advertised potential of 100× speedups, development of high-performance *general-purpose GPU* (GPGPU) programs requires highly idiomatic programs, whose development is work intensive and requires a substantial degree of expert knowledge.

Several researchers have proposed methods to ameliorate the status quo by either using a library to compose GPU code or by compiling a subset of a high-level language to low-level

---

<sup>1</sup>Moore’s actual prediction in 1965 referred only to the number of devices that could be fabricated on a single die, and that this quantity would increase by fifty percent each year [95]. I refrain from using the commonly held name of “Moore’s Law” as this trend is not a law in the sense defined by the physical sciences, such as the Laws of Gravity and Thermodynamics — it is simply a description of observed facts, rather than a principle driven by some fundamental controlling influence.

GPU code [3, 23, 25, 59, 81, 88, 96]. This thesis is in the same spirit: we propose a domain-specific high-level language of array computations, called *Accelerate*, that captures appropriate GPGPU programming idioms in the form of parameterised, collective array operations. Our choice of operations was informed primarily by the *scan-vector model* [31], which is suitable for a wide range of algorithms and can be efficiently implemented on modern GPUs [120].

The beginning of the Accelerate project predates this thesis, and several researchers have contributed to it in that time. The following summarises the main areas I have contributed to, and I have explicitly noted which are my own individual work:

- We have developed an embedded language, *Accelerate*, of parameterised collective array computations that is more expressive than previous GPGPU proposals (Section 3.1).
- I implement a dynamic code generator based on CUDA skeletons of collective array operations that are instantiated at runtime (Sections 4.1 & 4.2).
- I implement an execution engine that caches previously compiled skeleton instances and host-to-device data transfers, as well as parallelises code generation, data transfer, and GPU kernel loading, configuration, and execution (Sections 4.4, 4.5, & 4.6).
- We introduce a novel sharing recovery algorithm for type-safe abstract syntax trees, preserving the structure of the deeply embedded program (Section 5.1).
- I introduce type preserving optimisations of embedded language array programs, including a novel approach to array fusion (Sections 3.2, 5.2, & 5.3).
- Benchmarks assessing runtime code generation overheads and kernel performance for a range of programs (Chapter 6).

Although we motivate and evaluate our work in the context of the Accelerate language, our contributions are not limited to this context. Specifically, our sharing recovery algorithm applies to any embedded language based on the typed lambda calculus, and my approach to array fusion applies to any dynamic compiler targeting bulk-parallel SIMD hardware. The current implementation targets CUDA, but the same approach works for OpenCL, as well as traditional CPU programming languages such as C and LLVM.

Our results show that Accelerate programs can be competitive with programs hand-written in CUDA, while having a much higher level of abstraction. Because of the work undertaken in this thesis, we have seen Accelerate gain traction within academia and industry, including:

- Simon Marlow has recently published a book on *Parallel and Concurrent Programming in Haskell* [83], which includes a chapter on GPU programming using Accelerate.
- Peter Braam of Parallel Scientific is building a high level DSL in Haskell based on Accelerate, for image reconstruction computations for the Square Kilometer Array, the largest radio telescope in the world.

- Andy Gill has recently been awarded an NSF grant to research resource-aware DSLs for high performance computing applications, and is building on top of Accelerate.<sup>2</sup>
- Flowbox<sup>3</sup> is a startup developing tools for composition and visual effects for images and video that is built on top of Accelerate. Flowbox is generating interest amongst several Hollywood studios as a promising new technology.
- We commonly see research papers on array-based computations and languages, particularly in a functional programming context, include a comparison to Accelerate in their own benchmarks.<sup>4</sup>

We will discuss related work as each topic arises. The source code for Accelerate including all benchmark code is available from <https://github.com/AccelerateHS>.

---

<sup>2</sup>[http://www.nsf.gov/awardsearch/showAward?AWD\\_ID=1350901](http://www.nsf.gov/awardsearch/showAward?AWD_ID=1350901)

<sup>3</sup><http://www.flowbox.io>

<sup>4</sup>However, I modestly refrain from going so far as to claim that my work in this thesis has set a new standard by which functional array-based programming languages are measured.



You see a wile, you thwart. Am I right?

—TERRY PRATCHETT AND NEIL GAIMAN

*Good Omens*

## 2.1 Data parallelism

The major processor manufacturers and architectures have long since run out of room with most of the traditional approaches to boosting CPU performance. Instead of driving clock speed and straight-line instruction throughput, the inclusion of multiple cores on a single chip has become the dominant mechanism for scaling processor performance. Despite this trend being apparent for a number of years, programming parallel computers still remains an extremely challenging task — even for expert computer programmers, let alone for scientists in other disciplines.

One programming model that has shown to make good use of parallel hardware is *data parallelism*. Data parallelism focuses on distributing the data over the available processors, and for each processor to perform the *same* task on the *different* pieces of distributed data. To the programmer, data parallelism exposes a single logical thread of execution that is fairly easy to understand and reason about. In addition to its simplicity, the data parallel approach to programming parallel computers has several advantages:

1. The model is independent of the number of processors, so it scales to any number of processors by decomposing data into the appropriate number of chunks.
2. All synchronisation is implicit, so the programming model is safe from race conditions, eliminating a major source of errors in parallel programs.
3. As memory bandwidth is often a limiting factor in modern computers, the emphasis on data layout can assist with both data flow as well as parallelisation.



**Figure 2.1:** Block diagram comparing the relative use of transistors in a CPU (left) compared to a GPU (right). The GPU is specialised for highly parallel, compute intensive workloads, and so is designed such that the majority of its transistors are devoted to data processing, rather than data caching and control flow. Images from [98].

In the world of massively parallel computing with strong locality requirements, data parallelism is the well established, demonstrably successful brand leader. Examples of data parallel programming environments include High Performance Fortran (HPF) [58], the collective operations of the Message Passing Interface (MPI) [94], Google’s Map/Reduce framework [38], and NVIDIA’s CUDA API for graphics processors [98].

## 2.2 GPU computing

General-purpose computing on graphics processing units (GPGPU) is the utilisation of a graphics processing unit (GPU) — which typically handles the computations for computer graphics — to perform computations in applications typically handled by the central processing unit (CPU). Since general purpose CPUs are designed to optimise the performance of a single thread of execution, much of the processor’s resources (die area and power) are dedicated to non-computational tasks such as caching and branch prediction. Conversely, the highly parallel nature of graphics processing (rasterisation) means the GPU architecture can instead use these resources to execute many tasks in parallel, improving the computational throughput for parallel workloads at the expense of decreased single threaded performance. This difference in architectures between the CPU and GPU is illustrated in Figure 2.1.

More specifically, the GPU is especially well suited to executing problems that can be expressed as data parallel computations, where the same program is executed by many processors on many different data elements in parallel (§2.1). Because the same program is executed by many processors, there is no need for sophisticated control flow, and because it is executed on many data elements, memory access latency can be hidden with calculations rather than by big caches. Many applications that process large data sets can use the data parallel programming model to speed up calculations, and these applications can often be coded to execute on the GPU.

```

// Kernel definition
__global__ void VecAdd(float *A, float *B, float *C, int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) {
        C[i] = A[i] + B[i];
    }
}

int main()
{
    ...
// Kernel invocation with N threads
    int threadsPerBlock = 128;
    int numBlocks      = (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<numBlocks, threadsPerBlock>>>(A, B, C, N);
    ...
}

```

**Listing 2.1:** A CUDA kernel that illustrates pair wise addition of two vectors. The `__global__` keyword marks a function as a kernel that should be executed on the GPU in data parallel. The execution configuration syntax `<<<...>>>` specifies the number of threads that will each execute the function in data parallel.

## 2.3 CUDA

The CUDA architecture is the parallel computing platform and programming model created by NVIDIA and implemented by the graphics processing units (GPUs) that they produce. Using CUDA, programmers gain direct access to the memory and massively parallel multicore architecture of the GPU, which can then be used for general purpose computing applications, not just graphics.

CUDA extends the C programming language by allowing the programmer to define functions, called *kernels*, that, when called, are executed  $n$  times in  $n$  data parallel threads on the available processing elements, rather than only once like regular C functions. These functions are executed by threads in SIMD groups called *warps* arranged in a multidimensional structure of *thread blocks*. Within a warp, all threads execute the same instructions in lock-step, so divergent control flow must be handled via predicated execution. Threads within a block can cooperate by sharing data through some *shared memory* and by synchronising their execution to coordinate memory access. More precisely, the programmer can specify synchronisation points in the kernel by calling `__syncthreads()`, which acts as a barrier at which all threads in the block must wait before any are allowed to proceed. Each thread block executes independently of each other, so a GPU with a higher number of CUDA cores — or *streaming multiprocessors* (SM) — will execute the program in less time than a GPU with fewer multiprocessors.

As an example, Listing 2.1 illustrates the pair wise addition of two vectors  $A$  and  $B$  of size  $N$  each, and stores the result in a vector  $C$ . The kernel is defined with the `__global__`

declaration specifier. The number of CUDA thread blocks and the number of threads in each block that execute the kernel, at this invocation, is specified in the `<<<...>>>` syntax. Moreover, Listing 2.1 demonstrates that the GPU programming model as exposed by CUDA is a data parallel programming model —  $N$  threads execute  $N$  individual instances of the kernel function `VecAdd`, and each thread operates on a single element of each input array to create a single value in the result. See the CUDA C programming guide for details [98].

## 2.4 Data parallel algorithms in CUDA

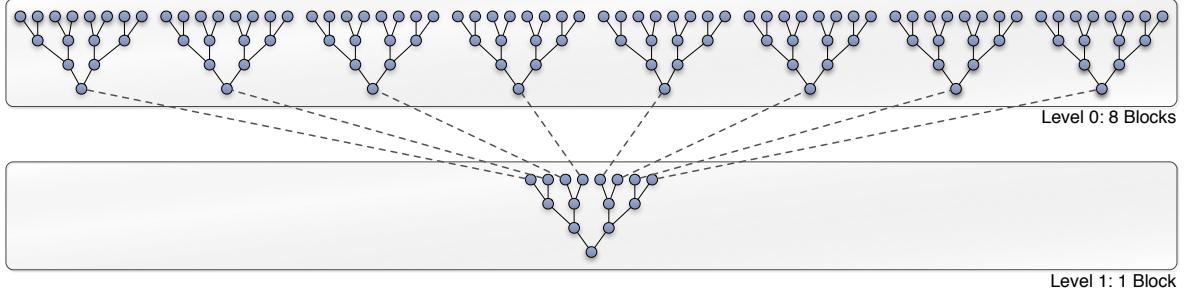
CUDA is a parallel computing platform for programming NVIDIA GPUs. Using CUDA, the GPU becomes accessible not just for graphics applications, but for computational tasks much like a CPU. Unlike CPUs, however, GPUs have a parallel throughput architecture that emphasises executing many concurrent threads slowly, rather than executing a single thread very quickly. In a GPU, the large caches and complex instruction scheduling logic of the CPU that increase single-threaded performance are instead replaced with many additional arithmetic units. See Section 2.3 for more information.

Some collective operations such as `map` have an obvious mapping to the highly parallel CUDA architecture, so we elide discussion of their implementation. Other collective operations such as `scan`, where the value of each element depends on the value of the previous last, may at first seem to not admit a parallel interpretation at all. This section outlines the implementation of this second kind of collective operation, where efficient parallel implementation in CUDA may not be obvious.

### 2.4.1 Reduction

Reductions are a common and important data-parallel primitive [30]. Figure 2.2 illustrates the basic strategy for reducing an array in parallel. Within each thread block a tree-based approach is used to reduce elements to a single value, and multiple thread blocks each process a portion of the array in parallel. However, there is no global synchronisation primitive in CUDA that can be used to combine the partial results. This is because global synchronisation would be (a) expensive to build in hardware for large numbers of multiprocessors, and (b) difficult for programmers to use without introducing sources of deadlock. Instead, kernel launches serve as a global synchronisation point, so thread blocks instead communicate their partial results by committing them to memory, and the kernel is launched recursively until the final reduction value is computed.

Note that if we wish to fuse other operations into the fold kernel, only the first line of reductions at level zero perform the fused operation. All subsequent reductions in that kernel, as well as the recursive steps at level one and beyond, are pure reductions. We discuss fusion further in Section 5.2.



**Figure 2.2:** Illustration of a tree reduction, performed in two steps. In the first step 8 thread blocks in parallel reduce eight segments of the array to a single element each. The thread blocks synchronise by writing their result to memory, and the kernel is called recursively until the final scalar result is computed.

### Parallel Reduction Complexity

If each thread combines two elements at a time, then a vector of length  $N$  will be reduced in  $\mathcal{O}(\log N)$  parallel steps. Each step  $S$  does  $N/2^S$  independent operations, so the *step complexity* of the algorithm is:

$$\mathcal{O}(\log N)$$

For  $N = 2^D$ , the algorithm thus performs  $\sum_{S=1}^D 2^{D-S} = N - 1$  operations. This means that the *work complexity* of the algorithm is:

$$\mathcal{O}(N)$$

and therefore does not perform more work than a sequential algorithm. For  $P$  threads running physically in parallel on  $P$  processors, the *time complexity* is  $\mathcal{O}(N/P + \log N)$ . In a thread block  $N = P$ , so the time complexity is:

$$\mathcal{O}(\log N)$$

Compare this to a sequential reduction, which has a time complexity of  $\mathcal{O}(N)$ .

### Algorithm Cascading

The *cost* of a parallel algorithm is the number of processors  $\times$  time complexity. This implies that the cost of the algorithm is  $\mathcal{O}(N \log N)$ , which is *not* cost efficient.

Brent's theorem [30] suggests that instead of each thread summing two elements, *algorithm cascading* can be used to combine a sequential and parallel reduction. If each thread does  $\mathcal{O}(\log N)$  sequential work, this will reduce the cost of the algorithm to  $\mathcal{O}(N/\log N \log N)$ , or rather:

$$\mathcal{O}(N)$$

while keeping the work complexity  $\mathcal{O}(N)$  and step complexity  $\mathcal{O}(\log N)$ .

As an example, this result implies that each block of 256 threads should sum a total of  $256 \times \log_2(256) = 2048$  elements. In practice it is beneficial to do even more sequential work per thread, as this will reduce the number of levels in the recursive tree reduction, and provide better latency hiding by minimising the number of thread blocks launched.

### Mapping to CUDA Threads

Reduction of a one dimensional array uses multiple thread blocks to cooperatively reduce the array, as described above. The number of thread blocks used is limited to the maximum number which can be simultaneously resident on the current device. This may require each thread to do more sequential work than that indicated by the algorithm cascading analysis, but limits the total kernel startup cost associated with launching thread blocks. Since the maximum number of resident thread blocks is typically much less than the number of threads in a block, the reduction will require at most two kernel invocations.<sup>1</sup>

Higher-dimensional reductions reduce the array along the innermost dimension only. Instead of all thread blocks cooperatively reducing each segment sequentially, each segment is reduced by a single thread block which operates independently of all other thread blocks. A consequence of this is that proper device utilisation depends on the shape of the array and not simply the total number of elements in the array. For example, reduction of an array stored as a  $1 \times n$  matrix will use one thread block to reduce the single matrix row, no matter how large  $n$  is. This simplifies the implementation but is clearly not always ideal.<sup>2</sup>

#### 2.4.2 Scan

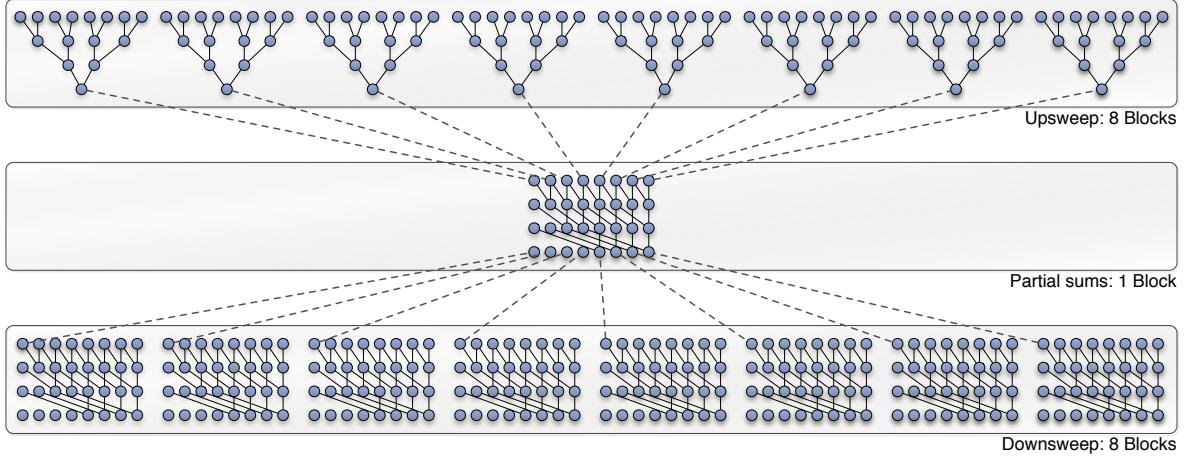
Parallel scan and segmented scan algorithms are a crucial building block for a great many data-parallel algorithms [18, 31]. They also form the basis for efficiently mapping nested data-parallel languages such as NESL [20, 21] on to flat data-parallel machines. Because of their fundamental importance to many algorithms, implementing efficient scan operations in CUDA has received much attention [39, 55, 92, 120, 121].

The basic binary-tree based scan algorithm proceeds in  $\log N$  stages. In order to compute scans efficiently on the GPU, the input array is decomposed into a number of blocks of size  $B$ , such that an inclusive scan of  $B$  elements can be computed by an individual thread block,

---

<sup>1</sup>Accelerate selects the thread block size in order to maximise thread occupancy. As this depends on both the specific GPU being used as well as the user function the array is reduced with, an upper bound of two parallel steps can not be guaranteed. See Section 4.6.3 for more information.

<sup>2</sup>The number of multiprocessors on a device varies between architecture generation and performance of a given card. For example, a Tesla T10 processor (compute capability 1.3) has 240 cores split over 30 multiprocessors, while a Kepler K20X processor (compute capability 3.5) has 2688 cores split over only 14 multiprocessors. A given architecture generation will have the same number of cores per multiprocessor, and lower performance processors in a generation are produced by incorporating (or activating) fewer multiprocessors, thereby reducing the total core count.



**Figure 2.3:** Illustration of a parallel inclusive left-to-right scan, which is performed in three steps. In the first step, 8 thread blocks compute the partial scan result for each eight element section of the array. The partial results are then scanned by a single thread block in the second step. In the final phase, the 8 thread blocks use the partial results as the initial element when computing the final scan result for this section of the array.

locally in shared memory. The inclusive scan algorithm depicted in Figure 2.3 proceeds in three phases:

1. The input array is divided into chunks of size  $B$  elements and distributed to the thread blocks, which each scan their portion of the array in parallel. The final scan value (partial result) from each thread block  $i$  is stored into an intermediate array `block_results[i]`.<sup>3</sup>
2. A single thread block performs a scan of the helper array `block_results`. Accelerate is set up such that the number of blocks in the first step is less than the maximum thread block size (§4.6), so this step does not need to repeatedly apply the global scan algorithm.
3. All blocks scan their section of the input array in parallel again, where thread block  $i$  uses element  $i$  of the result of step 2 as the initial element of the scan.

For an exclusive scan, it is also necessary to ensure that the computation of the partial block sums includes the seed element.

---

<sup>3</sup>We note that computation of the final value requires that elements are processed in the correct left-to-right or right-to-left order. For simplicity, the current implementation reuses the block scan kernel implemented here, and therefore, as described, all but the final result of the scan is discarded. Replacing this step with a directional reduce operation would likely improve performance, but this investigation is left to future work.

### Inclusive vs. Exclusive scan

The above description mentions two different flavours of scans. The output of an exclusive scan at each location is a function of all the elements that came before it, but does *not* include the current element. On the other hand, the inclusive scan is a function of all elements up to and *including* the current element. At an example, the two methods produce the following output for a left-to-right prefix sum:

```
input      = [ 13,  7, 16, 21,  8, 20, 13, 12 ]
exclusive scan = [ 0, 13, 20, 36, 57, 65, 85, 98 ]
inclusive scan = [ 13, 20, 36, 57, 65, 85, 98, 110 ]
```

These methods are well known in data-parallel programming, and some algorithms are best expressed in terms of one or the other. Accelerate supports both of these flavours of `scan`, as well as others that follow the behaviour of the standard Haskell family of `scan` operations.

### Interaction with array fusion

In the scan algorithm described above, the input array to be scanned is read twice: once when computing the partial block results (step 1), and a second time when computing the final scanned array (step 3). If the input does not refer to manifest array data, but instead an array in a symbolic form resulting from array fusion (§5.2), then the elements of the symbolic array — which might contain arbitrarily expensive computations — will be computed twice also.

In order to avoid this, in some circumstances it might be beneficial for either (a) the first phase of the scan to store the entire partial scan result, rather than just the final element; or (b) to always read in manifest data, effectively disallowing producer/consumer fusion (§5.2) for scans in the CUDA backend. The latter option is undesirable, and the former may not work for non-commutative operators, such as the one we used in the operator-transformation-based definition of segmented scan (§2.4.6). It is left to future work to fully investigate this interaction between array fusion and operations such as `scan`, where the underlying implementation requires multiple passes over the input data.

#### 2.4.3 Backpermute

The `backpermute` collective operation is specified by a backwards permutation  $f$  as an index mapping from the result array  $Y$  into a source array  $X$ , which we can write as  $f : X \leftarrow Y$ . That is, for every index  $y$  in  $Y$ , the value at index  $y$  is taken from  $x = f(y)$  in  $X$ . Since each element of the result can be computed independently of all others, the `backpermute` operation is straightforward to implement. However, it is worth defining here it is not commonly found outside of data parallel programming.

```

filter :: Elt a => (Exp a -> Exp Bool) -> Acc (Vector a) -> Acc (Vector a)
filter p vec
= let flags          = map (boolToInt . p) vec
  (targetIdx, len) = scanl' (+) 0 flags
  defaults         = backpermute (index1 $ the len) id vec
in
  permute const defaults (λix → flags!ix ==* 0 ? (ignore, index1 $ targetIdx!ix)) vec

```

**Listing 2.2:** Filtering returns only those elements of a vector which satisfy a predicate. This operation is included as part of Accelerate’s standard prelude.

#### 2.4.4 Permute

The `permute` collective operation defines a forward permutation  $f$  as an index mapping from one array  $X$  onto the result array  $Y$ , which we can write as  $f : X \rightarrow Y$ . Implementation of the `permute` function in Accelerate is complicated because we have not placed any particular restrictions on  $f$ , namely:

1.  $f$  is not surjective: the range of  $f$  may not cover the codomain  $Y$ . For every  $x$  in  $X$ ,  $f(x)$  need not yield every index  $y$  in  $Y$ . This means that the result array must first be initialised with a set of default values.
2.  $f$  is not injective: distinct elements of the domain may map to the same element in the codomain. For all  $x$  and  $x'$  in  $X$ , if  $f(x) = f(x')$ , we may have that  $x \neq x'$ . This means we require an associative combination function to combine elements from the domain that map to the same index in the codomain.
3.  $f$  is partial: elements of the domain may be ignored, and thus do not map to elements in the codomain.

That the permutation function admits partial functions in the index mapping is not particularly challenging for the implementation, and indeed is useful for implementing the `filter` operation, shown in Listing 2.2. The special value `ignore` is used to drop elements of the input vector that do not satisfy the predicate, by not mapping those indices to an index in the codomain.

On the other hand, because we can not prove that `filter` is surjective, we require the result vector to be first initialised with default values. Since we do not know anything about the element type `a`, the only recourse is to copy elements from the input array `vec`. This is doubly wasteful, because we must first execute the `backpermute` kernel to compute the `defaults` array, and then copy those values into the results vector before executing the `permute` kernel, even though we know the initialised values will be completely overwritten.

As a second example, Listing 2.3 demonstrates the computation of a simple ten bin histogram from a vector of floating point elements in the range  $[0, 100)$ . In this case the permutation function is neither surjective nor injective, as some bins may contain no elements and

```

histogram :: Acc (Vector Float) → Acc (Vector Int)
histogram vec
  = let bins      = 10
    zeros     = fill (constant (Z :: bins)) 0
    ones      = fill (shape vec)           1
  in
  permute (+) zeros (λix → index1 (A.floor ((vec ! ix) / P.fromIntegral bins))) ones

```

**Listing 2.3:** A simple histogram written in Accelerate. We assume the input vector contains elements in the range [0, 100) and accumulate into ten equally sized bins.

thus take the default value, while other bins may contain multiple elements which need to be combined (accumulated) correctly.

### Implementation using compare-and-swap

The semantics of the `permute` operation is that every permutation from source to result array is applied at the same time in a single parallel step. If multiple CUDA threads attempt a non-atomic write to the same memory location at the same time, the writes will be serialised but the thread which performs the final write is undefined, and so the final value at that memory slot is also undefined [98]. To support non-injective permutation functions, which are required to evaluate the histogram program correctly, the atomic compare-and-swap operation is used to implement write combining.<sup>4</sup> For a combination function `f` on elements of type `T`, the following atomically combines a value from the source array `x` with the value of the result array `y`:

```

T x      = source[i];
T y, old_y = result[j];
do {
  y      = old_y;
  old_y = atomicCAS(&result[j], y, f x y);
}
while (y != old_y);

```

Any atomic operation on simple types can be implemented in terms of compare-and-swap in this manner, although it would also be possible to use more specific atomic operations such as `atomicAdd` in special cases.

### Implementation using explicit locking

The implementation for combining elements of the permutation function works well for simple atomic types, but as Accelerate uses a struct-of-arrays representation (§4.2.3), combining tuple types must apply the procedure to each component of the tuple separately. Because

---

<sup>4</sup>The atomic compare-and-swap operation on 32-bit values into global memory is only available for devices of compute capability 1.1 and higher, and for 64-bit values on devices of compute capability 1.2 and higher.

the critical section does not apply to all components of the tuple at once, this can lead to inconsistencies in the result.<sup>5</sup>

One alternative for combining complex types is to explicitly lock each element of the output array before threads can access it. Once a thread acquires the lock on a particular element, it commits all components of the array as a single atomic operation. A straightforward method to implement this is via a *spin lock*, where threads trying to acquire the lock simply wait in a loop (“spin”), repeatedly checking whether the lock is available. Since the thread remains active but is not performing any useful work, the use of this kind of busy waiting is inefficient if threads are blocked for an extended period of time — for example, because the lock is contended by many threads, or the atomic section executed while the lock is held is lengthy.

Implementing a standard spin-lock can be done as follows:

```
do {
    old = atomicExch(&lock[i], 1);                                // (1)
} while (old == 1);                                              // (2)
/* atomic section */                                              // (3)
atomicExch(&lock[i], 0);                                         // (4)
```

1. The spin lock procedure requires a temporary array to represent the lock state for each element of the output array. We use 1 to represent the locked state, and 0 to indicate the element is unlocked. To lock element *i* of the output array, we atomically store 1 into the `lock` slot for this element, returning the `old` state of the lock.
2. If the `old` state of the lock was unlocked (0) then we have just acquired the lock. Otherwise, it was already locked, so we must retry.
3. Once the lock is acquired, the atomic section can be computed.
4. To release the lock, write zero back into the lock slot for this element. A memory barrier or atomic operation, as used here, is required for this step to ensure memory consistency on architectures which do not have a strong memory model, such as CUDA devices.

However, a subtle problem emerges when we attempt to use the above algorithm. In a CUDA device, all threads within a warp execute instructions in lockstep, and divergent control flow is handled via predicated execution (§2.3). Once a thread acquires a lock and exits the loop (2), it must wait for all other threads in the warp to reach the same point, before execution of the critical section (3) can begin. If two threads in a warp are attempting to acquire the same lock, once the lock is acquired by one thread, that thread will sit idle while the second thread spins attempting to grab a lock that will never be released, because the first thread can not progress. The system is deadlocked.

In order to solve this problem, we must ensure that threads always make progress once they have acquired the lock:

---

<sup>5</sup><https://github.com/AccelerateHS/accelerate/issues/137>

```

done = 0;
do {
    if (atomicExch(&lock[i], 1) == 0) { // (2)
        /* atomic section */
        done = 1;
        atomicExch(&lock[i], 0);
    }
} while (done == 0); // (1)

```

1. The key change is that threads loop repeatedly until they have executed the entire critical section, rather than looping only to acquire the lock.
2. If a thread fails to acquire the lock, it is disabled until the end of the branch. Threads that successfully acquire the lock enter the body of the conditional, where they execute the critical section and finally release the lock.

#### 2.4.5 Stencil

Stencil operations are a fundamental building block of many scientific and image processing algorithms. A stencil computation is similar to a `map`, that has access to the elements in the local neighbourhood surrounding the element. For example, at the core of many image processing algorithms is the convolution operator  $*$ , whose definition is as follows:

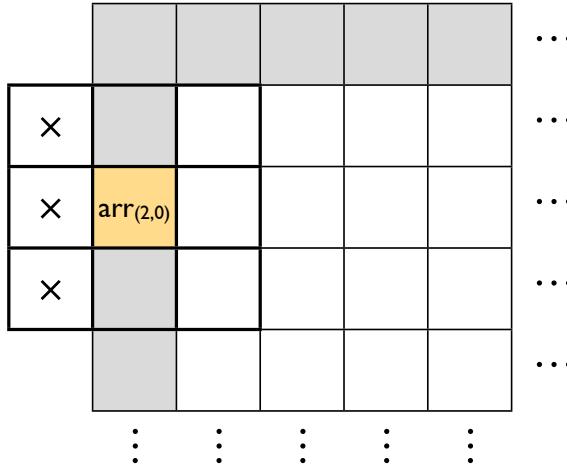
$$(A * K)(x, y) = \sum_i \sum_j A(x + i, y + j)K(i, j)$$

Here  $A$  is the image being processed and  $K$  is the *convolution kernel* or *stencil*. A *stencil* is a small matrix that defines a transformation on the image. Typical transformations include Gaussian blur and the Sobel differentiation operator, both of which are used in the Canny edge detection algorithm (§6.7).

#### Bounds checking

Since the stencil has access to elements in the surrounding neighbourhood, an immediate concern is what to do when the stencil “falls off” the edge of the array. Figure 2.4 shows the application of a  $3 \times 3$  stencil in this circumstance. The white squares indicate the *internal* region where the stencil is entirely within the array, and the grey squares indicate the *border* region, where part of the stencil lies outside of the array boundary.

Boundary conditions determine how to handle out-of-bounds neighbours, by specifying either a constant value to use instead, or by reading the source array at a different index. However, with the array sizes typical of GPU programs, the border region represents only a tiny fraction of the entire array. For example, a 512-pixel square image has less than one percent of its pixels along the edge. This fact implies that for optimal performance, we should avoid testing of the boundary each time we read an element from the source array. For simple



**Figure 2.4:** Application of a  $3 \times 3$  stencil in the border region. If the focal point of the stencil, such as the indicated element at index  $(Z: .2: .0)$ , lies in the border region (grey), the neighbouring elements might fall outside the array bounds, whereas for elements in the internal region (white) the stencil lies entirely within the array.

convolutions such as those used by Canny (§6.7), this adds significant overhead [75]. The current implementation does not separate computation of the border and internal regions.

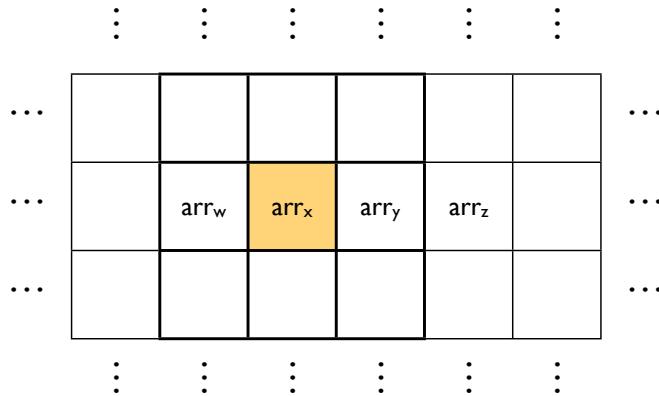
### Overlapping elements

Suppose we wish to apply a dense  $3 \times 3$  stencil to a single internal point in the array, where every element of the stencil is utilised. Application of the stencil requires at least nine elements of the array to be loaded, and one store for the result.

Figure 2.5 shows the evaluation of four horizontally adjacent elements. If we evaluate these points independently, we would need  $4 \times 9 = 36$  loads of the source array, plus the four stores to the result array. However, if the calculation of these four elements can cooperate and share loads from the source array into a local temporary array, this would require only  $3 \times 6 = 18$  loads, plus the four stores. Sharing of stencil elements can thus significantly reduce memory bandwidth requirements.

Sharing of stencil elements in CUDA can be achieved by having the thread block first cooperatively read the stencil elements into shared memory [98], and each thread then computes its stencil function using these shared values. In Accelerate, sharing of stencil elements is complicated by two main factors:

1. Stencil patterns can be large: up to 9 elements in each dimension. Since the shared memory region on the GPU is usually very small, this can significantly limit the maximum thread block size. If the occupancy of the device is too low, there may be insufficient parallelism to hide global memory transfer latency (§4.6.3).



**Figure 2.5:** Overlapping elements in a dense  $3 \times 3$  stencil. Computation of each element requires nine reads from the source array. If the four elements cooperate to share the source data, the number of loads to the source array is halved.

2. A stencil function does not need to use all elements defined in the pattern, so reading all elements into shared memory could waste more bandwidth than it saves. The pattern also does not need to be square nor symmetric, which complicates efficient cooperative loading of the common elements into shared memory.

The implementation currently ignores this issue and does not explicitly share elements between threads, and instead, relies on the hardware cache. For older (compute 1.x) devices that lack a traditional cache mechanism to global memory, we read the source array using the texture cache.<sup>6</sup> It is left to future work to investigate sharing of stencil elements between threads, either in general or specialised for certain cases such as dense stencils or those commonly used for convolutions.

### Interaction with array fusion

Since the stencil operation has access to the elements surrounding the focal point of the stencil, if the input is represented by a symbolic array resulting from array fusion (§5.2), rather than manifest array data, accessing these neighbouring elements implies recomputing the symbolic computations on every element access. For some operation `stencil s . map f`, if the results of `f` can be shared between threads of the stencil (previous section), or if `f` is cheap, then the duplicate computations required to fuse the two operations might be preferable to the extra memory bandwidth required to store the result of `map f` to memory. A full exploration of stencil-aware array fusion is left to future work.

---

<sup>6</sup>A holdover from the graphical heritage of the GPU.

### 2.4.6 Segmented operations

The Accelerate language is restricted to programs which express only *flat* data parallelism. In this model, the computation executed in parallel on each data element must itself be sequential. Although this model admits an efficient implementation on highly parallel architectures and is ideally suited for computations on dense, regular arrays, flat data parallelism is not as well suited for algorithms that operate over *irregular* structures, such as sparse matrices, graphs, or trees.

On the other hand, *nested data-parallel* languages [20, 74] allowing recursive data structures and the application of parallel functions in data-parallel. The key insight is that because the nested calls are to the same function, the *flattening transformation* [22] can convert this nested data-parallelism into an equivalent flat data-parallel program. An essential part of the flattening transformation is the introduction of *segmented* operations, which apply a primitive operation in parallel to subcomponents of a nested structure.

At an example, consider the following nested list of floating-point numbers:

```
xss :: [ [ Float ] ]
xss = [ [1,2], [3,4,5], [], [6] ]
```

The insight of the flattening transformation is to instead view this data structure as a flat array of values, coupled with a *segment descriptor* encoding the length of each of the subsequences:

```
xss' :: [ Float ]
xss' = [ 1, 2, 3, 4, 5, 6 ]

seg :: [ Int ]
seg = [ 2, 3, 0, 1 ]
```

Operations on the nested data structure can then be transformed into segmented operations over the flattened data:

```
map (fold f z xss) ↪ foldSeg f z xss' seg
```

Although we include support for several segmented operations in Accelerate, it is left for significant future work to add support for the expression of nested parallelism.

#### Segmented reduction

Segmented reduction is a primitive operations in Accelerate. It is conceptually similar to the multidimensional reduction, except that the length of each segment is provided by the segment descriptor, rather than determined from the extent of the input array.

In the current implementation, a segmented reduction uses a single warp (SIMD group) to reduce each segment, although it would also be possible to use the entire thread block, as is done for regular multidimensional reductions. It remains to fully investigate which mapping is more efficient.

### Segmented scan

Segmented scans are not implemented as primitive operations in Accelerate. Instead, segmented scans are implemented in terms of unsegmented scans by operator transformation [18, 119]. Given the operator  $\oplus$  we can construct a new operator  $\oplus^s$  that operates on a flag-value pairs  $(f_x, x)$  as follows:

$$(f_x, x) \oplus^s (f_y, y) = (f_x | f_y, \text{ if } f_y \text{ then } y \text{ else } x \oplus y)$$

It is expected that implementing segmented scans directly [39, 121], rather than by operator transformation, will improve performance.

## 2.5 Embedded domain-specific languages

A *domain-specific language* (DSL) is a computer language specialised to a specific problem domain. The DOT language [1, 46] is an example of a DSL for describing graphs. This is in contrast to a general purpose language such as Haskell [100], which is broadly applicable across many domains but may lack specialised features for a particular domain. A domain specific language is created to solve problems in a particular domain, and is not intended to solve problems outside it (although this may be technically possible, for example, the PostScript [7] page description language). Restricting the problem domain the language is intended to solve may allow for more optimisations during compilation, or increased expressivity when representing problems or defining solutions in that domain.

An *embedded* (or *internal*) *domain-specific language* (EDSL) is a domain-specific language that is designed to be used from within another host language [61]. The embedded language is implemented as a library, so is able to reuse the syntax and features of the general purpose host language compiler, including compilation phases such as lexing, parsing, and semantic analyses such as type checking, while adding domain specific elements such as data types, or domain specific optimisations and code generation.

There are two major degrees in which an embedded language can be implemented; as either a *shallow* or *deep* embedding.<sup>7</sup>

### 2.5.1 Shallow embedding

In a shallow embedding, operations in the embedded language translate directly into operations in the target language. A shallow embedding captures the semantics of the data of that domain in a data type, and provides a *fixed* interpretation of that data. Thus, it is easy

---

<sup>7</sup>One might further categorise *combinations* of shallow and deep embeddings [126]. This amounts to a shallow embedding which targets operations in a deeply embedded core language, and is equivalent to, for example, the definition of `filter` (Listing 2.2) being a composition of several operations, rather than a core language primitive.

to add new constructs to the language, so long as they can be interpreted in the semantic domain. The embedded language can also reuse features of the host language, such as variable bindings.

### 2.5.2 Deep embedding

A deep embedding captures the semantics of the domain by reflecting the operations of the *object language* (or *meta language*) into a data structure. This data structure — an *abstract syntax tree* (AST) — then permits transformations, such as optimisations, before being translated into the *target language*. Deeply embedded languages therefore enable a *variable* interpretation of operations in the domain. This makes it is easy to add new interpretations of the embedding, for example, by compiling to a different target language. However, adding new language constructs requires extending the AST data type, and the implementation must deal explicitly with binding of variables. Sharing and recursion are common problems when implementing deeply embedded languages [48].

As the deeply embedded language runs inside of the *host language*, reflecting operations into an AST, an interpreter for the embedded language must be embedded within the host application. As the interpreter executes code for the embedded language at program runtime, code in the embedded language may either be written by the user or generated dynamically by the application. If the target language is different to that of the host language, this entails the interpreter generating, compiling, and executing code for the target language at program runtime.

## 2.6 Discussion

This chapter has outlined the basic concepts driving the design and implementation of our embedded language Accelerate. We have introduced the concept of data parallelism, a programming model that is simple to reason about yet can efficiently make use of large numbers of processors. We introduced the idea of using graphics processors for general purpose computing tasks, and show that GPU computing, and CUDA in particular, are data-parallel programming models. Finally, we have introduced the idea of embedded domain-specific languages.

The next chapter ties these ideas together to introduce Accelerate: an embedded language of array computations that has been designed with massive data parallel processors such as GPUs in mind.



## Embedding array computations

---

---

The trouble with opportunity is that it always comes disguised as hard work.

—HERBERT V. PROCHNOW

Starting with Fortran, array-based code has played a central role in high-performance computing. Array-based programming has proved to be successful not only on data-parallel (SIMD) hardware such as the CM-2, vector computers, and modern graphics processing units (GPUs), but also on task parallel (MIMD) hardware such as distributed memory, symmetric multiprocessor (SMP), and multicore machines.

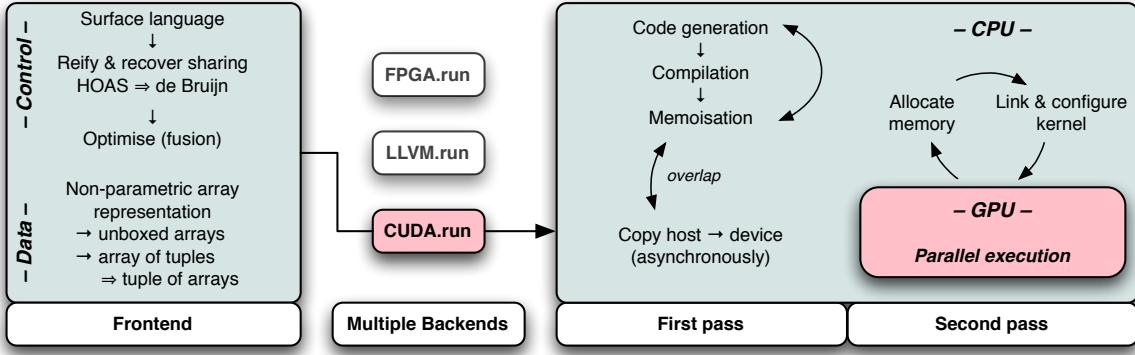
Recent work on stream fusion in the Haskell vector library [37, 82] and the parallel array library Repa [68, 75, 76] have demonstrated that purely functional array code in Haskell (1) can be competitive with the performance of imperative array programs; and (2) lends itself to an efficient parallel implementation on control-parallel multicore CPUs.

So far, the use of purely functional array programs for programming data-parallel SIMD hardware has been less successful. This chapter describes the Accelerate language, our purely functional language over regular, multidimensional arrays, which has been designed to allow efficient execution on massively data-parallel hardware such as GPUs.

### 3.1 The Accelerate EDSL

The current generation of graphical processing units are massively multicore processors (§2.2). They are optimised for workloads with a large degree of data parallelism (§2.1) and good performance depends on highly idiomatic programs with low SIMD divergence and regular memory-access patterns [98]. These unusual characteristics of GPUs means that the development of applications that use the graphics processors for general purpose computations (§2.2) is work intensive and requires a substantial degree of expert knowledge.

Accelerate is our approach to reducing the complexity of GPU programming: a high-level, deeply embedded domain-specific language (§2.5) of array computations that captures



**Figure 3.1:** The overall structure of Accelerate. It comprises a frontend and supports multiple backends that can target a variety of architectures.

appropriate GPGPU idioms in the form of parameterised, *rank-polymorphic* [68], collective operations on arrays [29]. Our choice of operations was informed by the *scan-vector model* [31], which is suitable for a wide range of algorithms, and demonstrated that these operations can be efficiently implemented on modern GPUs [120].

Accelerate's collective array operations are implemented as algorithmic skeletons (§4.1) that capture appropriate GPU programming idioms. The dynamic code generator instantiates CUDA implementations of these skeletons (§2.4) to implement embedded array programs (§4.2). Dynamic code generation exploits runtime information of the target hardware to optimise GPU code, and enables on-the-fly generation of embedded array programs by the host program (§4.4). The evaluator minimises the overhead of dynamic code generation by caching binaries of previously compiled skeleton instantiations and by parallelising code generation, data transfer (§4.5), and GPU kernel loading, configuration, & execution (§4.6).

To achieve performance competitive with hand-written CUDA [89], we implement two primary optimisation techniques: sharing recovery (§5.1) tackles code explosion due to the embedding in Haskell, while array fusion (§5.2) eliminates superfluous intermediate structures, which often arise due to the high-level nature of the language. Both of these techniques are well known from other contexts, but they present unique challenges for an embedded language compiled for execution on a GPU.

Figure 3.1 summarises the overall structure of Accelerate. It comprises a frontend and support for multiple backends that can target a variety of architectures. Here, we are only concerned with the CUDA generating GPU backend, but the approach is amenable to targeting multicore CPU backends exploiting SIMD instructions,<sup>1</sup> backends for OpenCL,<sup>2,3</sup> and for reconfigurable hardware such as FPGAs. In the following, we briefly introduce some features of the design and use of Accelerate.

<sup>1</sup><https://github.com/AccelerateHS/accelerate-llvm>

<sup>2</sup><https://github.com/AccelerateHS/accelerate-backend-kit/tree/master/icc-opencl>

<sup>3</sup><https://github.com/HIPERFIT/accelerate-opencl/>

### 3.1.1 Computing a vector dot product

Consider computing the dot product of two vectors, using standard Haskell lists:

```
dotp_list :: [Float] → [Float] → Float
dotp_list xs ys = foldl (+) 0 (zipWith (*) xs ys)
```

The two input vectors `xs` and `ys` are pointwise multiplied, and the resulting list of products are summed, yielding the scalar result.

Using Accelerate, we implement this computation on arrays as follows:

```
dotp :: Vector Float → Vector Float → Acc (Scalar Float) — (1)
dotp xs ys
  = let xs' = use xs
    ys' = use ys
  in
    fold (+) 0 (zipWith (*) xs' ys') — (2)
                                              — (3)
```

Here, `fold` and `zipWith` are taken from the Accelerate library `Data.Array.Accelerate`, rather than from the standard Haskell library `Data.List`. The Accelerate code consumes two one dimensional arrays (`Vector`) of values, and produces a single (`Scalar`) result as output. It differs from the list version in three main respects:

1. The result is an embedded Accelerate computation, indicated by the type constructor `Acc`. It will be evaluated in the *target* language of dynamically generated parallel code, rather than the *meta* language, which is vanilla Haskell (§2.5).
2. We lift the two plain vectors `xs` and `ys` into `Acc` terms with `use`. This makes the arrays available to the embedded computation (§3.1.3).
3. We use `fold` instead of `foldl`.

The first two points are artefacts of lifting the computation into an embedded language (§2.5), effectively delaying the computation. Concerning the final point, the list traversal function `foldl` guarantees a left-to-right traversal of the elements, whereas `fold` leaves the order in which the elements are combined unspecified. This requires that the binary operator supplied to `fold` is associative,<sup>4</sup> but allows for an implementation using a parallel tree reduction [31, 120].

### 3.1.2 Arrays, shapes, and indices

Parallelism in Accelerate takes the form of collective operations such as `zipWith` on arrays of type `Array sh e`, where `sh` is the *shape* and `e` the *element type* of the array. Following the approach taken in Repa [68], we represent both the shapes and indices of an array using an

---

<sup>4</sup>Although floating-point arithmetic is not strictly associative, it is common to accept the resulting error in parallel applications.

```

data Z           = Z           — rank zero
data tail :: head = tail :: head — increase rank by one

type DIM0 = Z           — synonyms for common array dimensionalities
type DIM1 = DIM0 :: Int
type DIM2 = DIM1 :: Int
type DIM3 = DIM2 :: Int
— and so on

type Array DIM0 e = Scalar e — synonyms for common array types
type Array DIM1 e = Vector e

```

**Listing 3.1:** Types of array shapes and indices

inductive notation of tuples as heterogeneous *snoc* lists to enable rank-polymorphic definitions of array functions.

As shown in Listing 3.1, on both the type and value level, the constructor `Z` is used to represent the shape of a rank-0 array, and the infix operator `(.:.)` is used to increase the rank by adding a new (innermost) dimension to the right of the shape. Thus, a rank-3 index with components `x`, `y`, and `z`, is written `(Z:.z:.y:.x)` and has type `(Z:.Int:.Int:.Int)`, where the component `x` is innermost and varies most rapidly, while the component `z` varies least rapidly.

Overall, an array of type `Array (Z:.Int:.Int) Float` is a rank-2 array of single precision floating-point numbers. Listing 3.1 also defines synonyms for common array types: a singleton array of shape `DIM0` represent a single scalar value, while an array of shape `DIM1` represents a vector, and so on. While it may appear that the explicit mentioning of `Int` in each dimension is redundant, we require indices over types other than `Int` for rank polymorphic functions, such as replicating an array into one or more additional dimensions or slicing a lower-dimensional subarray out of a larger array [29, 68].

### 3.1.3 Arrays on the host and device

Accelerate is an *embedded language* (§2.5) that distinguishes between vanilla Haskell arrays and arrays in the embedded language, as well as computations on both flavours of arrays. Embedded array computations are identified by types formed from the type constructor `Acc`, and must be explicitly executed before taking effect. To make these arguments available to the Accelerate computation they must be embedded with the `use` function, which is overloaded so that it can accept tuples of arrays:

```
use :: Arrays arrays ⇒ arrays → Acc arrays
```

In the context of GPU programming, GPUs typically have their own high-performance memory which is separate from the host CPU’s main memory, and data must be explicitly transferred to the GPU’s memory before it can be used. The distinction between regular arrays of type `Array sh e` and arrays of the embedded language `Acc (Array sh e)` has the added

<code>use</code>	<code>:: Arrays arrs <math>\Rightarrow</math> arrs <math>\rightarrow</math> Acc arrs</code>	embed an array
<code>unit</code>	<code>:: Elt e <math>\Rightarrow</math> Exp e <math>\rightarrow</math> Acc (Scalar e)</code>	create singleton array
<code>reshape</code>	<code>:: Exp sh <math>\rightarrow</math> Acc (Array sh' e) <math>\rightarrow</math> Acc (Array sh e)</code>	impose a new shape
<code>map</code>	<code>:: (Exp a <math>\rightarrow</math> Exp b) <math>\rightarrow</math> Acc (Array sh a) <math>\rightarrow</math> Acc (Array sh b)</code>	map a function over an array
<code>zipWith</code>	<code>:: (Exp a <math>\rightarrow</math> Exp b <math>\rightarrow</math> Exp c) <math>\rightarrow</math> Acc (Array sh a) <math>\rightarrow</math> Acc (Array sh b) <math>\rightarrow</math> Acc (Array sh c)</code>	apply funciton to a... ...pair of arrays
<code>generate</code>	<code>:: Exp sh <math>\rightarrow</math> (Exp sh <math>\rightarrow</math> Exp a) <math>\rightarrow</math> Acc (Array sh a)</code>	array from index mapping
<code>replicate</code>	<code>:: Slice slix <math>\Rightarrow</math> Exp slix <math>\rightarrow</math> Acc (Array (SliceShape slix) e) <math>\rightarrow</math> Acc (Array (FullShape slix) e)</code>	extend array across... ...new dimensions
<code>slice</code>	<code>:: Slice slix <math>\Rightarrow</math> Acc (Array (FullShape slix) e) <math>\rightarrow</math> Exp slix <math>\rightarrow</math> Acc (Array (SliceShape slix) e)</code>	remove existing dimensions
<code>fold</code>	<code>:: (Exp a <math>\rightarrow</math> Exp a <math>\rightarrow</math> Exp a) <math>\rightarrow</math> Exp a <math>\rightarrow</math> Acc (Array (sh:.Int) a) <math>\rightarrow</math> Acc (Array sh a)</code>	tree reduction along... ...innermost dimension
<code>scan{l,r}</code>	<code>:: (Exp a <math>\rightarrow</math> Exp a <math>\rightarrow</math> Exp a) <math>\rightarrow</math> Exp a <math>\rightarrow</math> Acc (Vector a) <math>\rightarrow</math> Acc (Vector a)</code>	left-to-right or right-to-left... ...vector pre-scan
<code>backpermute</code>	<code>:: Exp sh' <math>\rightarrow</math> (Exp sh' <math>\rightarrow</math> Exp sh) <math>\rightarrow</math> Acc (Array sh a) <math>\rightarrow</math> Acc (Array sh' e)</code>	backwards permutation
<code>permute</code>	<code>:: (Exp a <math>\rightarrow</math> Exp a <math>\rightarrow</math> Exp a) <math>\rightarrow</math> Acc (Array sh' a) <math>\rightarrow</math> (Exp sh <math>\rightarrow</math> Exp sh') <math>\rightarrow</math> Acc (Array sh a) <math>\rightarrow</math> Acc (Array sh' a)</code>	forward permutation
<code>stencil</code>	<code>:: Stencil sh a stencil <math>\Rightarrow</math> (stencil <math>\rightarrow</math> Exp b) <math>\rightarrow</math> Boundary a <math>\rightarrow</math> Acc (Array sh a) <math>\rightarrow</math> Acc (Array sh b)</code>	map a function with local... ...neighbourhood context

**Listing 3.2:** Summary of Accelerate’s core collective array operations, omitting `Shape` and `Elt` class constraints for brevity. In addition, there are other flavours of folds and scans as well as segmented versions of these.

benefit of differentiating between arrays allocated in host memory and arrays allocated in GPU device memory. Thus, `use` also implies a host-to-device data transfer. See Section 4.5 for more information on how device memory is managed. Similarly, expressions in `Acc` are computations that are executed on the device (the GPU), whereas regular Haskell code runs on the host (the CPU).

### 3.1.4 Array computations versus scalar expressions

The signatures of the two operations `zipWith` and `fold`, used in the definition of `dotp`, are shown in Listing 3.2. These operations are multidimensional variants of the corresponding list functions, but with arrays wrapped in `Acc`. In addition to `Acc`, which marks *embedded array* computations, we also have `Exp`, which marks *embedded scalar* computations: a term of type `Exp Int` represents an embedded expression yielding a value of type `Int`, and similarly `Exp Float  $\rightarrow$  Exp (Float,Float)` characterises an embedded scalar function that takes an argument of type `Float` and yields a pair of `Floats` as the result. As with computations embedded in `Acc`, computations in `Exp` are executed in the target language of GPU code.

Accelerate distinguishes the types of collective operations and scalar computations to achieve a stratified language. Collective operations in `Acc` consist of many scalar computations in `Exp` that are executed in data-parallel. However, scalar computations *can not* contain collective operations. This stratification excludes *nested, irregular* data parallelism statically — instead, Accelerate is restricted to only *flat data-parallelism* involving regular, multi-dimensional arrays. This stratification, which restricts the Accelerate language to only flat data parallelism, ensures that computations can be efficiently mapped to the hardware.

Compared to regular Haskell, `Exp` computations are rather limited in order to meet the restrictions of what can be efficiently executed on GPUs. In particular, we do not support recursion, and provide only a limited form of explicit value iteration. Scalar expressions support Haskell’s standard arithmetic operations by overloading the standard type classes such as `Num` and `Integral`, as well as bitwise operations from `Bits`. Although we can not overload functions on `Bool` in standard Haskell, we support equality and comparison operators as well as logical connectives using the operators `(==*)`, `(/=*)`, `(<*)`, `(<=*)`, and so on. We also have conditional expressions in the form of `c ? (t, e)`, which evaluates to `t` if `c` yields `True`, otherwise to `e`. There are also scalar operations that take array valued arguments; namely, `arr!ix` which indexes an array and `shape` which queries the extent of an array. The argument to these operations is guaranteed to only be a previously let-bound variable, ensuring that each invocation of the scalar function can not initiate additional parallel work. Finally, we have tupling and projection, and auxiliary functions for computing with indices.

### 3.1.5 Computing an *n*-body gravitational simulation

As a second example, consider simulating the Newtonian gravitational forces on a set of massive bodies in 3D space, using a precise (but expensive)  $\mathcal{O}(n^2)$  algorithm. We use the following type synonyms to represent bodies in the simulation:

```
type Vec a      = (a, a, a)
type Position   = Vec Float
type Accel     = Vec Float

type Mass       = Float
type PointMass  = (Position, Mass)
```

Here, `Vec` is the type of 3-component vectors, which are used to store the *x*-, *y*-, and *z*-components of the particle’s position or acceleration in three dimensional space, along each dimension in `Position` and `Accel` respectively. A `PointMass` is a tuple containing the body’s `Mass` and `Position`.

In a data-parallel setting, the natural implementation of the all-pairs *n*-body algorithm first computes the forces between every pair of bodies, before reducing the components applied to each body using a multidimensional (segmented) reduction. We can calculate the acceleration each body experiences as it interacts with all other bodies in the system as:

```

calcAccels :: Exp Float → Acc (Vector PointMass) → Acc (Vector Accel)
calcAccels epsilon bodies
  = let n      = A.size bodies
    cols   = A.replicate (lift $ z :: n :: All) bodies           — (1)
    rows   = A.replicate (lift $ z :: All :: n) bodies
  in
    A.fold (.+) (vec 0)                                         — (3)
      $ A.zipWith (accel epsilon) rows cols                      — (2)

```

1. `replicate` expands the vector of `n` bodies into an  $n \times n$  matrix, where we replicate the original elements of the source vector, represented by `All`, such that every element down the columns or along the rows of the matrix are identical, respectively. The operation `lift` is used to push the shape constructors into the expression; that is, in this instance we have `lift :: (Z :: Exp Int :: All) → Exp (Z :: Int :: All)`.
2. `zipWith` calculates the interaction between each pair of particles by element wise combining the  $n \times n$  matrices, which were replicated in opposite directions.
3. `fold` reduces the result along the innermost dimension only, yielding a vector containing the sum of interactions for each particle between all others. The auxiliary function `(.+.)` performs component-wise addition of the 3-vector.

This is an example of using rank-polymorphic operations in Accelerate. The function `fold` requires an argument array of at least rank-1 (such as a matrix), and the result is computed by reducing the array along the innermost dimension, reducing the rank of the array by one (in this case yielding a vector). The functions `replicate` and `slice` are rank-polymorphic as well, but require more complex shape constraints that we characterise with the type family `FullShape` and `SliceShape` that statically track changing array dimensions. For details on the various forms of shape polymorphism, see Keller et al. [68]. Overall, as shown in Listing 3.2, the collective operations of Accelerate are a multi-dimensional variant of those underlying the scan-vector model [31, 120].

### Computing gravitational potential

I briefly explain how to compute the gravitational potential between a pair of bodies. Given  $n$  bodies with position  $\mathbf{x}_i$  and velocity  $\mathbf{v}_i$  for  $i \in [1, n]$  (bold face indicates a 3-component vector), the force  $\mathbf{f}_{ij}$  on a body  $i$  caused by its gravitational attraction to body  $j$  is given by:

$$\mathbf{f}_{ij} = G \frac{m_i m_j}{|\mathbf{r}_{ij}|^2} \cdot \frac{\mathbf{r}_{ij}}{|\mathbf{r}_{ij}|}$$

where  $m_i$  and  $m_j$  are the masses of the bodies,  $\mathbf{r}_{ij} = \mathbf{x}_j - \mathbf{x}_i$  is the vector from body  $i$  to body  $j$ , and  $G$  is the gravitational constant. The factor on the left represents the *magnitude* of the acceleration, and is proportional to the product of the masses and diminishes as the

square of the separation of the masses. The right factor is the *direction* of the force, and is the unit vector from  $i$  in the direction of  $j$ .

The total force  $\mathbf{F}_i$  on a body  $i$  due to its interactions with the other bodies is thus:

$$\mathbf{F}_i = \sum_{\substack{1 \leq j \leq n \\ j \neq i}} \mathbf{f}_{ij} = Gm_i \cdot \sum_{\substack{1 \leq j \leq n \\ j \neq i}} \frac{m_j \mathbf{r}_{ij}}{|\mathbf{r}_{ij}|^3}$$

Note that as the bodies approach each other, the force between them grows without bound, which is undesirable for numerical simulations. In astrophysical simulation, collisions between bodies are generally precluded, which is reasonable if the bodies represent galaxies that may pass through each other. A *softening factor*  $\epsilon^2 > 0$  is added, which models the interaction of two Plummer point masses [5, 40]. In effect, softening limits the magnitude of forces between bodies. The denominator is rewritten as follows:

$$\mathbf{F}_i \approx Gm_i \cdot \sum_{1 \leq j \leq n} \frac{m_j \mathbf{r}_{ij}}{(|\mathbf{r}_{ij}| + \epsilon^2)^{3/2}}$$

The condition  $j \neq i$  is no longer needed, because  $\mathbf{f}_{ii} = 0$  when  $\epsilon^2 > 0$ .

Finally, the acceleration experienced by a body is  $\mathbf{a}_i = \mathbf{F}_i/m_i$ , and we can now present the Accelerate routine to compute the acceleration between two bodies.

```
accel :: Exp R → Exp PointMass → Exp PointMass → Exp Accel
accel epsilon pmi pmj = (mj * invr3) *. r
  where
    mj      = massOfPointMass pmj
    r       = positionOfPointMass pmj .-. positionOfPointMass pmi
    rsqr   = dot r r + epsilon * epsilon
    invr   = 1 / sqrt rsqr
    invr3  = invr * invr * invr
```

Here the functions `positionOfPointMass` and `massOfPointMass` extract each component of the `PointMass`, respectively. The function `(.-.)` is component-wise subtraction applied to the 3-vector, and `(*.)` multiplies each component of a 3-vector by a scalar value.

### 3.1.6 Non-parametric array representation

Accelerate computations are parameterised by the type classes `Shape` and `Elt`, which characterise the types that may be used as array indices and array elements respectively. For example, the type of `map` is:

```
map :: (Shape sh, Elt a, Elt b) ⇒ (Exp a → Exp b) → Acc (Array sh a) → Acc (Array sh b)
```

We discussed the nature of array indices in Section 3.1.2. The `Elt` class characterises the types of values that can be used as array elements, and hence appear in scalar Accelerate expressions. The supported `Elt` types are signed & unsigned integers (8, 16, 32, and 64-bit

wide), floating point numbers (single & double precision), as well as `Char`, `Bool`, and array indices formed from `Z` and `(::)`. Furthermore, there are tuples of all of those, including nested tuples, as was seen in the *n*-body example program (§3.1.5).

Accelerate arrays of primitive type, such as `Int` and `Float`, are easily represented as unboxed arrays of the corresponding primitive type. More interesting is the case of arrays of tuples, where we use a non-parametric array representation so that arrays of tuples are represented in memory as a tuple of arrays, one array for each primitive type in the structure. By virtue of this representation, Accelerate is able to make optimal use of the available memory bandwidth. See Section 4.2.3 for details.

Moreover, consider the following types:

```
data Point = P Int Int

point :: Exp Point
sh    :: Exp (Z :: Int :: Int)
tup   :: Exp (Int, Int)
```

The salient feature of each type is that it carries two integers, albeit each wrapped in different constructors. We call this the *surface* type of a term. However, an Accelerate backend implementation should not need to know about the surface type representation. Indeed, we do not want to have to extend the backend to support every new user-defined type such as `Point`. The `Elt` class tells us how to convert between the surface type of an expression that a user programs in, into the *representation* type that an implementation stores and computes data in. We use a type family [28, 118] of nested tuples to represent types as heterogenous snoc-lists of primitive types:

```
type family EltRepr a :: *
type instance EltRepr ()      = ()
type instance EltRepr Int     = (((), Int)
type instance EltRepr (a, b)  = (EltRepr a, EltRepr' b)
type instance EltRepr (a, b, c) = (EltRepr (a, b), EltRepr' c)
— and so on...
```

The type `EltRepr'` is similar, except that it uses a flattened representation for primitive types in order to avoid overly nested pairs, for example, `EltRepr' Int = Int`. Arrays of tuples have a similar mapping into nested tuples of arrays, with the `Arrays` class acting analogously to `Elt`.

### 3.1.7 Richly typed terms

As a deeply embedded language, the operations of both the array and scalar sub-language do not directly issue computations; instead, they build term trees to represent embedded computations (§2.5). The terms in the surface language use *higher-order abstract syntax* (HOAS) to embed function-valued scalar expressions as well as type class overloading to reflect arithmetic expressions. For example, the body of the `dotp` function is translated into:

```

Fold add (Const 0) (ZipWith mul xs' ys')
  where
    add = λx y → PrimAdd (FloatingNumType (TypeFloat FloatingDict)) — (1)
      `PrimApp`
      Tuple (NilTup `SnocTup` x
             `SnocTup` y)
    mul = — as add, but using PrimMul ...

```

where the subterms `add` and `mul` have been extracted into the `where`-clause to improve readability. This is very much like the approach taken by Elliott [45], Gill et al. [50], and Mainland and Morrisett [81]. The difference is that in our approach we use GADTs [104] and type families [28, 118] to preserve the embedded program’s type information in the term tree (1), and use type-preserving transformations in the front end (§3.2).

The HOAS representation, while convenient for a human reader, is awkward for program transformations as it complicates looking under lambdas; i.e. inspecting and manipulating the bodies of function abstractions. After the frontend reifies the embedded program written in HOAS, it recovers sharing (§5.1) while converting the HOAS representation into a *nameless* representation using *typed de Bruijn indices* in the style of Altenkirch and Reus [8]. The type preserving conversion from HOAS to a nameless de Bruijn representation using GADTs [26] was simultaneously discovered by Atkey et al. [10].

Overall, the nameless form of `dotp` is:

```

Fold add (Const 0) (ZipWith mul xs' ys')
  where
    add = Lam (Lam (Body (
      PrimAdd (FloatingNumType (TypeFloat FloatingDict))
        `PrimApp`
        Tuple (NilTup `SnocTup` (Var (SuccIdx ZeroIdx))
               `SnocTup` (Var ZeroIdx))))))
    mul = — as add, but using PrimMul ...

```

The `Lam` constructor introduces nameless abstractions and `Var` wraps a de Bruijn index. At this point, the program is further optimised by the frontend, for example by applying the fusion transformation (§5.2).

Terms in the nameless Accelerate AST are encoded by GADTs using open recursion (`acc`, §3.1.8) and parameterised over the surface type of the return value (`a`, `t`) and the type of the environment (`env` and `aenv`, for scalar and array valued free variables, respectively):

<code>data PreOpenAcc acc aenv a</code>	— collective operations
<code>data PreOpenExp acc env aenv t</code>	— scalar operations

We represent the environment as a heterogenous snoc-list, encoded on the type level as nested pairs and building this list on the value level with the two constructors:

```

data Val env where
  Empty :: Val ()
  Push  :: Val env → t → Val (env, t)

```

A nameless de Bruijn index projects out a specific type from the environment:

```
data Idx env t where
  ZeroIdx :: Idx (env, t) t
  SuccIdx :: Idx env t → Idx (env, s) t
```

— *an index is either...*  
— *...at the top of the stack, or...*  
— *...under some junk*

Indices have type `Idx env t`, where the type `env` keeps track of what variables are in scope using the same encoding of nested tuples used by the environment, while `t` represents the type of a distinguished element in this list. At the value level, the constructors of `Idx` encode the index of that distinguished element in the list.

As terms are pushed into the environment, the type of the new term becomes wrapped in the type of the environment; i.e. `t` is existentially quantified in `Push`. Our de Bruijn indices recover the wrapped type of individual elements by projecting the index through the environment:

```
prj :: Idx env t → Val env → t
prj ZeroIdx      (Push _ v) = v
prj (SuccIdx idx) (Push val _) = prj idx val
prj _             -           = error "inconsistent□valuation"
```

Overall, this is an example of using nested datatypes and polymorphic recursion to precisely enforce constraints and invariants in the type of a term; in this case the type and scope of bound variables. Since an embedded Accelerate program is constructed and evaluated at program *runtime*, encoding properties in the type means that these invariants are checked at *compile* time, reducing the number of possible runtime errors in type-correct programs.

### 3.1.8 Tying the recursive knot

As mentioned in the previous section, terms in the Accelerate language are defined using open recursion. That is, the GADT used to represent nodes of the program AST is defined non-recursively, and instead is parameterised by the recursive knot.

Consider the following definition of a binary tree, which stores all data at the leaves of the tree and is parameterised by the type of the leaf element:

```
data Tree a
  = Leaf a
  | Node (Tree a) (Tree a)
```

— *Leaves of the tree hold the data*  
— *Internal nodes record only left and right branches*

Note that the internal `Nodes` of the `Tree` are defined in terms of itself — that is, `Tree` is a recursively defined data type. Alternatively, the tree can be defined non-recursively, by parameterising the definition by the recursive step:

```
data PreTree tree a
  | Leaf a
  | Node (tree a) (tree a)
```

A recursive tree type can be recovered from this definition. This however introduces an additional step of needing to unwrap the `Tree` constructor before exposing the `Leaf` and `Node` constructors of the `PreTree` itself:

```
data Tree a = Tree (PreTree Tree a)
```

This distinction between the recursive step and the tree constructors is important because it allows us to use the same basic tree definition, but include additional information as part of each recursive step. For example, we can define a new binary tree type that records the size of the subtree at each point, and any existing functions operating over the basic `PreTree` structure can still be used:

```
data SizedTree a = SizedTree Int (PreTree SizedTree a)
```

## 3.2 Manipulating embedded programs

The Accelerate core language is richly typed, maintaining full type information of the embedded language in the term tree. In order to apply transformations to these terms while maintaining the correctness of the program as encoded in its type, we require methods to manipulate these richly typed terms in a type- and scope-preserving manner. This section covers several techniques I developed for manipulating embedded Accelerate programs, that are required to implement the optimisations discussed in the following chapters.

### 3.2.1 Typed equality

If we want to test whether two Accelerate terms are equal — for example, to determine whether the subexpressions can be shared — we immediately run into the problem that terms in the core Accelerate language are existentially typed. That is, how should we implement:

```
Exp s == Exp t = ??
```

There is no reason that `s` and `t` should be expressions of the same type. In some instances we might not care, and as such can define standard heterogeneous equality:

```
instance Eq (PreOpenExp acc env aenv t) where
  (==) = heq
  where heq :: PreOpenExp acc env aenv a → PreOpenExp acc env aenv b → Bool
        heq = ...
```

where we do not care about the result type of each expression, and only require that the environment type (regarding size) of free scalar and array variables are the same so that we can test equality of de Bruijn indices.

However, we often *do* care about the specific types of our existentially quantified terms. Consider the case of defining equality for let-bindings. The Accelerate scalar language defines

let-bindings of the embedded program as a constructor of a GADT [104], whose type tracks the type and scope of the bound term (§3.1.7):

```
data PreOpenExp acc env aenv t where
  Let :: (Elt bnd, Elt body)
    => PreOpenExp acc env           aenv bnd
    => PreOpenExp acc (env, bnd)   aenv body
    => PreOpenExp acc env         aenv body

  — Local binding of a scalar expression
  — bound term
  — body/scope of binding

— other language constructs...
```

To apply `heq` to the body expression, we need to know something about the type of the bound terms to ensure that the scalar environments are the same, namely  $s \sim bnd \sim t$ .<sup>5</sup> To do this we must equip terms with some runtime witness to the existentially quantified type. Our reified dictionaries allow us to do exactly this, so we can define heterogeneous equality for reified dictionaries:

```
heqIntegralType :: IntegralType s → IntegralType t → Bool
heqIntegralType (TypeInt _) (TypeInt _) = True
heqIntegralType (TypeWord _) (TypeWord _) = True
...
heqIntegralType _ _ = False
```

However, doing this is perfectly useless as it only gives us a value of type `Bool`, with no idea what that value means or to what its truth might entitle us. The type checker does not gain any useful knowledge about the types the dictionaries `s` and `t` witness simply by knowing that `heqIntegralType s t = True`. A boolean is a bit uninformative.

Instead, we can write essentially the same test, but in the positive case deliver some *evidence* that the types are equal:

```
data s :=: t where
  REFL :: s :=: s

matchIntegralType :: IntegralType s → IntegralType t → Maybe (s :=: t)
matchIntegralType (TypeInt _) (TypeInt _) = Just REFL
matchIntegralType (TypeWord _) (TypeWord _) = Just REFL
...
matchIntegralType _ _ = Nothing
```

Matching on `Just REFL` will inject the knowledge into the type checker that the types `s` and `t` are the same. Now with our evidence-producing heterogeneous equality test for reified dictionary families, we can compare two terms and gain type-level knowledge when they witness the same value-level types. These witnesses for existentially quantified types then allow us to test for equality *homogeneously*, ensuring that positive results from singleton tests give the bonus of unifying types for other tests:

---

<sup>5</sup> Alternatively we can use `Data.Typeable.gcast` to provide a type-safe cast, but this quickly becomes unwieldy, and is a little unsatisfactory as this amounts to a runtime, rather than compile time, test.

```

matchPreOpenExp
  :: PreOpenExp acc env aenv s
  → PreOpenExp acc env aenv t
  → Maybe (s :=: t)
matchPreOpenExp (Let x1 e1) (Let x2 e2)
  | Just REFL ← matchOpenExp x1 x2      — if the bound expressions match
  , Just REFL ← matchOpenExp e1 e2      — then the environment of the body term will also match
  = Just REFL
matchPreOpenExp ...

```

## Congruence

If we are interested in the facility of matching terms for the purposes of optimisations such as common subexpression elimination (§5.3.2), it is beneficial to define not equality but instead *congruence* of terms. Two nodes are considered congruent if (1) the nodes they label are constants and the constants are equal; or (2) they have the same operator and the operands are congruent. The crux of the latter condition refers to commutative relations, such as scalar addition, where an operator yields the same result when the order of the operands are reversed. When checking congruence of primitive applications, we discriminate binary functions whose arguments commute, and return those arguments in a stable ordering by comparing a hash of each of the sub-terms.

```

commutes
  :: PrimFun (a → r)
  → PreOpenExp acc env aenv a
  → Maybe (PreOpenExp acc env aenv a)
commutes f x = case f of
  PrimAdd _      → Just (swizzle x)
  ...           → Nothing
where
  swizzle :: PreOpenExp acc env aenv (a,a) → PreOpenExp acc env aenv (a,a)
  swizzle exp
    | Tuple (NilTup `SnocTup` a `SnocTup` b) ← exp
    , hashPreOpenExp a > hashPreOpenExp b      = Tuple (NilTup `SnocTup` b `SnocTup` a)
    | otherwise                                = exp

```

Future work on the term matching system could look into the work of rewriting modulo associative and commutative (AC) theories [12, 41–43, 70].

### 3.2.2 Simultaneous substitution

In order to do things like renaming and substitution, we require a value-level substitution algorithm for the richly typed terms. The implementation follows the method of McBride [85, 86], where it is seen that renaming and substitution are both instances of a *single* traversal operation, pushing functions from variables to “stuff” through terms, for a suitable notion of stuff. The trick is to push a *type-preserving* but *environment changing* operation structurally through terms:

```
v :: ∀ t'. Idx env t' → f env' t'
```

Where the operation differs is in the image of variables: renaming maps variables to variables while substitution maps variables to terms. We then lift this to an operation which traverses terms, with appropriate lifting to push under binders, and rebuild the term after applying `v` to the variables:

```
rebuild :: Syntactic f
  ⇒ (∀ t'. Idx env t' → f env' t')
  ⇒ Term env t
  ⇒ Term env' t
```

The `Syntactic` class<sup>6</sup> tells us everything we need to know about `f` — our notion of “stuff” — in order to rebuild terms: a mapping in from variables, a mapping out to terms, and a weakening map which extends the context. Renaming will instantiate `f` with `Idx`, whereas for substitutions we may choose `Term` instead.

```
class Syntactic f where
  varIn   :: Idx env t → f env t           — variables embed in f
  termOut :: f env t → Term env t          — f embeds in terms
  weaken  :: f env t → f (env, s) t         — f allows weakening

instance Syntactic Idx
instance Syntactic Term
```

A key component of this toolkit, *weakening* describes an operation we usually take for granted: every time we learn a new word, old sentences still make sense; if a conclusion is justified by a hypothesis, it is still justified if we add more hypotheses; a term remains in scope if we bind new (fresh) variables. Weakening is the process of shifting things from one scope to a larger scope in which new things have become meaningful, but no old things have vanished. When we use a named representation (or HOAS) we get weakening for free, but in the de Bruijn representation weakening takes work: we need to shift all the variable references to make room for new bindings. To do this we need to explain how to shift the `v` operation into an extended environment; saying what to do with the new variable and how to account for it on the output side.

```
shift :: Syntactic f
  ⇒ (∀ t'. Idx env t' → f env' t')
  ⇒ Idx (env, s) t
  ⇒ f (env', s) t
```

Overall, the crucial functionality of simultaneous substitution is to propagate a class of operations on variables closed under shifting, which is what `Syntactic` and `rebuild` offer.

---

<sup>6</sup>Since Accelerate is a stratified language there are separate implementations of this toolkit on syntactic elements for the scalar and collective operations, but the details are identical. The discussion uses the generic `Term` to mean either of these operations.

```

subTop :: Term env s → Idx (env, s) t → Term env t
subTop s ZeroIdx      = s
subTop _ (SuccIdx ix) = Var ix

inline :: Term (env, s) t → Term env s → Term env t
inline body bnd = rebuild (subTop bnd) body

```

**Listing 3.3:** A simultaneous substitution to inline terms

```

dot :: Term (env, b) c → Term (env, a) b → Term (env, a) c
dot f g = Let g (rebuild split f)
  where
    split :: Idx (env, b) c → Term ((env, a), b) c
    split ZeroIdx      = Var ZeroIdx
    split (SuccIdx ix) = Var (SuccIdx (SuccIdx ix))

compose :: Fun env (b → c) → Fun env (a → b) → Fun env (a → c)
compose (Lam (Body f)) (Lam (Body g)) = Lam (Body (f `dot` g))
compose _ _ = error "compose:@impossible@case"

```

**Listing 3.4:** A simultaneous substitution to compose unary function terms

### 3.2.3 Inlining

The final question, then, is how to write the function `v` which we push through terms, as the type  $\forall t'$  could be anything. After all, what is the point of having a simultaneous substitution algorithm if we can't specialise it to one-at-a-time substitution.

The function `subTop` in Listing 3.3 takes a replacement for the top variable and returns a simultaneous substitution which eliminates `ZeroIdx`. If we have a term with one free variable, we can use this to replace that variable with a term. That is, `inline` a term into all use sites of the free variable.

The demonic  $\forall$  — which is to say that the quantifier is in a position which gives us obligation, not opportunity — of the operator `v` forces us to respect type: when pattern matching detects the variable we care about, we happily discover that it has the type we must respect. The demon is not so free to mess with us as one might at first fear.

### 3.2.4 Function composition

The equivalent of unary function composition `(.)` on terms can be defined in a similar manner. A function `(a → b)` is equivalent to a `Term` of type `b` and free variable `a`. Similar to inlining, we replace an expression that uses the top variable with one that uses another, remembering to bind the result of the first expression so that it can be reused in the second without duplication.

Note the type of `split` in Listing 3.4, which creates space in the environment at the second index (`SuccIdx ZeroIdx`) for the free variable `a` to appear in the resultant `Term`, as the first index will be bound to the result of the first expression `g`.

```
data Extend acc aenv aenv' where
  BaseEnv :: Extend acc aenv aenv
  PushEnv :: Arrays a
    => Extend acc aenv aenv'
    -> PreOpenAcc acc aenv' a
    -> Extend acc aenv (aenv', a)
```

**Listing 3.5:** Extending an array environment

Finally, `compose` lifts this to a function on terms by unwrapping the lambda abstractions. Unfortunately, GHC’s type checker can not determine that the function type prohibits any valid term except those with a single lambda, so we need a second catch-all case to suppress a compilation warning. In the remainder of the text, we elide such impossible cases for brevity.

### 3.2.5 Environment manipulation

As part of the fusion transformation (§5.2) we often need to lift array valued inputs to be let-bound at higher points in the term tree, but at the same time we can not immediately add these bindings to the output term because they will interfere with further fusion steps. Instead, we collect these extra bindings separately from the main term, and splice them back into the output term at some later point. This ensures that we correctly track the types of the environments between the fused terms and the collected let-bindings.

Listing 3.5 defines a heterogeneous snoc-list of array terms that witnesses how an array environment is extended by the application of these operations. In the style of the de Bruijn indices used for projecting environment variables (§3.1.7), `Extend` uses an inductive notion of tuples as heterogeneous *snoc* lists. In contrast to environment indices, the base case does not refer to an empty environment, but instead the environment `aenv` which we are extending. This witnesses how to construct an extended environment `aenv'` by bringing new terms into scope, one at a time.

```
bind :: Arrays a
      => Extend acc aenv aenv'
      -> PreOpenAcc acc aenv' a
      -> PreOpenAcc acc aenv a
bind BaseEnv = id
bind (PushEnv env a) = bind env . Alet (inject a) . inject
```

The auxiliary function `inject :: PreOpenAcc acc aenv a → acc aenv a` completes the definition of each of the array terms.

### Sinking

We define *sinking* as the operation of shifting a term from one (array) environment into another, where new things have come into scope according to some witness, and no old things have disappeared.

```

type env :> env' =  $\forall t'. \text{Idx env } t' \rightarrow \text{Idx env}' t'$ 

class Sink f where
  weaken :: env :> env'  $\rightarrow f \text{ env } t \rightarrow f \text{ env}' t$ 

  sink :: Sink f  $\rightarrow$  Extend acc env env'  $\rightarrow f \text{ env } t \rightarrow f \text{ env}' t$ 
  sink env = weaken (k env)
    where
      k :: Extend acc env env'  $\rightarrow \text{Idx env } t \rightarrow \text{Idx env}' t$ 
      k BaseEnv = id
      k (PushEnv e _) = SuccIdx . k e

```

**Listing 3.6:** Sinking terms to a larger environment

Listing 3.6 generalises `weaken` from Section 3.2.2 to increase the environment of an object `f` based on some function on indices rather than shifting by a single de Bruijn index. Thus we only need to traverse a term once no matter how many positions the variable indices shift by. Using `sink`, we can weaken terms based on the witness provided by `Extend`.

### 3.3 Related work

This section compares several Haskell-based domain specific embedded languages, particularly focusing on the expressivity of the object language.

Vertigo [45] is an embedded language in Haskell that exposes a functional language for 3D graphics, together with an optimising compiler that generates DirectX shader code for the graphics processor. As Vertigo is a first order language, it does not provide higher-order combinators such as `map` and `fold`.

Paraiso [96] is a deeply embedded language in Haskell for solving systems of partial differential equations, such as hydrodynamics equations. While Accelerate supports `stencil` operations that can be used to express such programs, it lacks the more domain-specific language features or compiler optimisations offered by Paraiso for this class of problem. Programs in Paraiso are written using a monadic style, whereas Accelerate has a purely functional interface.

Obsidian [127, 128] and Nikola [81] are embedded languages in Haskell for array programming, both of which are similar in intent to Accelerate. Compared to Accelerate, the programming languages of both Obsidian and Nikola are severely restricted. Both are limited to a single computation over one dimensional vectors of primitive type, whereas Accelerate supports multidimensional arrays, product types, and computations can span multiple operations and kernels. Due to its compilation method, Nikola only supports `map`-like functions, while Obsidian is able to encode more complex operations such as `scan` because its lower level interface exposes more details of the target hardware.

None of these embedded languages encode the types of the source program into the embedded language terms to the degree with which Accelerate does. Encoding properties of the

source language in types means that these invariants are checked at compile time, reducing the number of possible runtime errors in type-correct programs (§3.1.7).

### 3.4 Discussion

This chapter has covered the design of our Haskell-based purely functional embedded language, which can express a wide range of programs over regular multidimensional arrays. We have also discussed methods for the type safe representation of programs, and several base techniques for type preserving manipulation of those programs.

The next chapter covers my implementation of the CUDA based Accelerate backend which executes Accelerate programs on massively parallel GPUs. The next chapter also covers issues such as caching, which must be considered in order to ensure that the embedded language exhibits performance competitive to traditional offline compiled languages.



---

## Accelerated array code

---

At the beginning of a novel, a writer needs confidence, but after that what's required is persistence. These traits sound similar. They aren't. Confidence is what politicians, seducers and currency speculators have, but persistence is a quality found in termites.

It's the blind drive to keep on working that persists after confidence breaks down.

—WALTER KIRN

The previous chapter described the design of the Accelerate language, an embedded DSL of purely functional operations over regular multidimensional arrays. Accelerate is a rich language based on the scan-vector model, and is able to express some interesting real-world problems. The Accelerate language is designed with the idea of executing purely functional array programs on massively data-parallel hardware. This chapter details my implementation of the Accelerate backend targeting parallel execution on CUDA graphics processing units.

### 4.1 Embedding GPU programs as skeletons

Accelerate offers a range of aggregate operations on multi-dimensional arrays. They include operations modelled after Haskell's list library, such as `map` and `fold`, but also array-oriented operations, such as `permute` and stencil convolutions.

As a simple example, consider the dot product of two vectors:

```
dotp :: Acc (Vector Float) → Acc (Vector Float) → Acc (Scalar Float)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

The crucial difference to vanilla Haskell is the `Acc` type constructor representing *embedded array-valued computations*. The types `Vector e` and `Scalar e` represent one-dimensional and zero-dimensional (singleton) arrays, respectively.

The expression `zipWith (*) xs ys` implements point-wise multiplication of the two argument vectors, and `fold (+) 0` sums the resulting products up to yield the final, scalar result, wrapped into a singleton array. The type of `fold` is:

```

fold :: (Shape sh, Elt a)
  ⇒ (Exp a → Exp a → Exp a)
  → Exp a
  → Acc (Array (sh:.Int) a)
  → Acc (Array sh a)

```

It uses a binary folding function operating on *embedded scalar computations* of type `Exp a` to implement a parallel reduction along the innermost dimension of an  $n$ -dimensional, embedded array of type `Array (sh:.Int) a`. The *shape* `sh:.Int` consist of a polymorphic shape `sh` with one added (innermost) dimension, which is missing from the shape of the result array.

#### 4.1.1 Array operations as skeletons

The Accelerate CUDA backend is based on the idea of *algorithmic skeletons* [34], where a parallel program is composed from one or more parameterised skeletons, or templates, encapsulating specific parallel behaviour. In our case, the backend implements each of the aggregate array operations, such as `map`, by way of a CUDA (§2.3) *code template* that is parameterised with array types and worker functions, such as the function to be mapped, to be injected at predefined points.

This generative approach is attractive for specialised hardware, such as GPUs, as the code templates can be hand-tuned to avoid expensive control flow, ensure efficient global-memory access, and use fast on-chip shared memory for local communication — all of which is necessary to ensure good performance on the GPU [98].

In the first version of Accelerate, I implemented CUDA code templates and template instantiation with a mixture of C++ templates and C preprocessor (CPP) macros, which was described in [29]. While workable, this approach turned out to have a number of problems:

- The use of CPP was fragile and difficult to maintain. Template instantiation by inlining of CPP macros required the use of fixed variables with no static checking to ensure the consistent use of names, or that used names were defined before their use. Moreover, it was easy to generate code that wasn't even syntactically valid.
- The approach led to the generation of dead code whenever specific template instances did not use all of their parameters or fields of structured data, which the CUDA compiler was unable to remove as dead code.
- Most importantly, the use of CPP did not scale to support the implementation of producer-consumer skeleton fusion, which is a crucial optimisation, even for code as simple as dot product.

The next sections discuss the new approach to template instantiation that avoids these problems, and appeared in [33].

```
[cunit]
$esc:(#include<accelerate_cuda.h>)
$edecls:texIn

extern "C" __global__ void map
(
    $params:argIn,
    $params:argOut
){
    const int shapeSize      = size(shOut);
    const int gridSize       = $exp:(gridSize dev);
    int ix;

    for ( ix = $exp:(threadIdx dev); ix < shapeSize; ix += gridSize )
    {
        $items:(dce x      .=. get ix)           — (3)
        $items:(setOut "ix" .=. f x)              — (2)
    }
}
]
```

**Listing 4.1:** Accelerate CUDA skeleton for the `map` operation

#### 4.1.2 Algorithmic skeletons as template meta programs

The code generator includes CUDA code skeletons implementing each of the collective operations in Accelerate. Each skeleton encodes the behaviour for each thread for this collective operation, fixing the parallel algorithmic structure of the operation. Each skeleton contains placeholders for the inputs that parameterise the skeleton, such as the array types and worker functions. Due to the shortcomings of C++ templates and CPP, we instead use Mainland’s *quasiquotation* extensions [80] to Template Haskell [122] to define skeletons as quoted CUDA C templates with splices for the template parameters.

Listing 4.1 shows the skeleton code for the `map` operation. The `[cunit|...|]` brackets enclose CUDA C definitions comprising this skeleton template. CUDA uses the `__global__` keyword to indicate that `map` is a GPU kernel — a single data-parallel computation launch on the GPU by the CPU (§2.3). *Antiquotations* `$params:var`, `$exp:var`, `$items:var`, and `$stms:var` denote template parameters using a Haskell expression `var` to splice CUDA C parameters, expressions, items, and statements, respectively, into the skeleton.

In order to instantiate the `map` skeleton, the code generator needs to generate concrete parameters for all of the holes in the skeleton template. In particular:

1. The function `f` which is applied to each individual array element;
2. The function `get`, which extracts the appropriate input array element for the array index `ix`; and finally
3. The types of the input and output array arguments that are collected and spliced into `argIn` and `argOut`, respectively.

The meaning of the auxiliary combinators `get`, `setOut`, `dce`, and `(.=.)` will be explained in the following sections.

As the quasiquoter `[cunit|···|]` is executed at Haskell compile time, syntactic errors in the quotations and antiquotations, as well as their composition, are detected at compilation time. Thus, we can be sure that the generated skeleton code will be syntactically correct if the backend compiles. See [80] for more information on quasiquoters.

## 4.2 Instantiating skeletons

Accelerate is a deeply embedded language (§2.5), meaning that while we write programs using Haskell syntax, we do not compile arbitrary Haskell programs directly to GPU machine code. Rather, an Accelerate program generates at program runtime an *abstract syntax tree* (AST) that represents the embedded program. The job of Accelerate’s CUDA backend code generator is to instantiate CUDA implementations of these collective operations, which are then compiled, loaded onto the GPU, and executed.

For the most part, translating Accelerate scalar expressions to plain CUDA C code is a straightforward syntactic translation of the de Bruijn AST to C. The quasiquoting library `language-c-quote`<sup>1</sup> allows the code generator to construct a term representation of the CUDA code in concrete syntax, which is then spliced into our skeleton templates at predefined points (§4.1). The following sections describe some particular features of code generation.

### 4.2.1 Producer-consumer fusion by template instantiation

In the first version of Accelerate’s CUDA backend, which was based on C++ templates and CPP macros [29], I generated one or more CUDA GPU kernels for each aggregate array operation. This schema is undesirable as it leads to many intermediate arrays and array traversals. Recall the definition of vector dot product:

```
dotp :: Acc (Vector Float) → Acc (Vector Float) → Acc (Scalar Float)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

The `zipWith` and `fold` operations would each compile to separate skeletons, and as a result the execution of `zipWith` produces an intermediate array that the `fold` kernels immediately consume.

This is not what a CUDA programmer would manually implement. It is more efficient to inline the `zipWith` computation into the kernel of the `fold`. This strategy eliminates one GPU kernel, as well as one intermediate array that is of the same extent as the [intersection of the] two input arrays. To achieve the same performance as handwritten CUDA code, we developed the shortcut fusion technique described in chapter 5, and which appeared in [89].

---

<sup>1</sup><http://hackage.haskell.org/package/language-c-quote>

The fusion system distinguishes producer-producer and consumer-producer fusion. The former combines two skeletons where threads produce array elements independently, such as `zipWith`, from skeletons where threads cooperate to produce an array, such as `fold`. Central to this approach is a representation of arrays as functions, which we called *delayed arrays* (in contrast to *manifest arrays*) which are represented as follows:

```
Delayed      :: (Shape sh, Elt e) ⇒
{ extentD   :: PreExp DelayedOpenAcc aenv sh           — array extent
, indexD    :: PreFun DelayedOpenAcc aenv (sh → e)     — generate element at index...
, linearIndexD :: PreFun DelayedOpenAcc aenv (Int → e) — ...at linear index
}
→ DelayedOpenAcc aenv (Array sh e)
```

Instead of generating a skeleton instance for `zipWith` straight away, we represent the computation implemented by `zipWith` as a scalar function (actually, a pair of functions) that generates an element of the array at a given array index, together with the extent (domain) of the array, as a value of type `DelayedOpenAcc`. For details of the representation see Section 5.2.

The crucial step in enabling producer-consumer fusion in Accelerate’s CUDA backend is in the function `codegenAcc`, which turns an AST node of type `DelayedOpenAcc aenv a` producing a manifest array, into an instantiated CUDA kernel of type `CUSkeleton aenv a`:

```
codegenAcc
:: DeviceProperties
→ DelayedOpenAcc aenv a
→ Gamma aenv
→ CUSkeleton aenv a
codegenAcc dev (Manifest pacc) aenv =
  case pacc of
    Fold f z a → mkFold dev aenv (codegenFun2 f) (codegenExp z) (codegenDelayed a)
    ...

```

Here we see that `mkFold`, which generates an instance of the `fold` template, gets as its last argument the code generated for a delayed array resulting from the call to `codegenDelayed`. The use of template meta programming to implement CUDA skeletons is crucial to enable producer-consumer fusion by way of template instantiation. In the dot product example, the delayed producer is equivalent to the scalar function  $\lambda ix \rightarrow (xs!ix) * (ys!ix)$ . The call to `mkFold` receives a CUDA version of this function, which can be inlined directly into the `fold` skeleton. The delayed producer function is bound to the argument `getIn` in the `fold` skeleton given in Listing 4.2, which is used in the line marked (1) and expands to the following CUDA code:

```
const Int64 v1 = ({ assert(ix ≥ 0 && ix < min(shIn0_0, shIn1_0)); ix; });
y0 = arrIn0_0[v1] * arrIn1_0[v1];
```

Here the `assert` checks whether the array index is in bounds, and is only active when Accel-

```
[cunit]
$esc:(#include<accelerate_cuda.h>")
$edecls:texIn

extern "C" __global__ void foldAll
(
    $params:argIn,
    $params:argOut
)
{
    // omitted variable declarations

    $items:(sh .=. shIn)
    const int shapeSize      = $exp:(cszie sh);
    const int gridSize       = $exp:(gridSize dev);
    int ix                  = $exp:(threadIdx dev);

    if ( ix < shapeSize ) {
        $items:(y .=. getIn ix)

        for ( ix += gridSize; ix < shapeSize; ix += gridSize ) {
            $items:(x .=. getIn ix)
            $items:(y .=. combine x y)
        }
    }

    ix = min(shapeSize - blockIdx.x * blockDim.x, blockDim.x);
    $items:(sdata "threadIdx.x" .=. y)                                — (1)
    $items:(reduceBlock dev combine x y sdata ix)                      — (2)

    // first thread writes the final result to global memory
}
]
```

**Listing 4.2:** Accelerate CUDA skeleton for the `foldAll` operation

erate is compiling kernels in debugging mode.<sup>2</sup> In this case the embedded `zipWith` operation is over two vectors, so no conversion to and from multidimensional indices was required to generate the array elements.

In contrast to the `map` skeleton shown in Listing 4.1, the code generated by `mkFold` proceeds in two parallel phases. The first phase is a sequential `for` loop including the use of the `getIn` embedded producer function. The second phase begins at the line marked (2) and implements a parallel tree reduction. See Section 2.4.1 for further details on the implementation of parallel reductions in CUDA. The instantiated skeleton for the fused `dotp` operation is shown in Listing 4.3.

#### 4.2.2 Instantiating skeletons with scalar code

Most aggregate array operations in Accelerate are parameterised by scalar functions, such as the mapping function for `map` or the combination function for `fold`. Thus, a crucial part of template instantiation is inlining the CUDA code implementing scalar Accelerate functions

---

<sup>2</sup>Kernel debug mode is activated when the Accelerate CUDA backend is installed in debug mode, and the command line flag `-ddebug-cc` is provided.

```

#include <accelerate_cuda.h>
extern "C" __global__ void foldAll
(
    const Int64 shIn0_0, const float* __restrict__ arrIn0_0,
    const Int64 shIn1_0, const float* __restrict__ arrIn1_0,
    const Int64 shOut_0,      float* __restrict__ arrOut_0
)
{
    // omitted variable declarations

    const Int64 sh0      = min(shIn1_0, shIn0_0);
    const int shapeSize = sh0;
    const int gridSize = blockDim.x * gridDim.x;
    int ix             = blockDim.x * blockIdx.x + threadIdx.x;

    // Phase 1: Each thread performs a local, sequential reduction
    if (ix < shapeSize) {
        const Int64 v1 = ({ assert(ix >= 0 && ix < min(shIn1_0, shIn0_0)); ix; });
        y0 = arrIn1_0[v1] * arrIn0_0[v1];

        for (ix += gridSize; ix < shapeSize; ix += gridSize) {
            const Int64 v1 = ({ assert(ix >= 0 && ix < min(shIn1_0, shIn0_0)); ix; });
            x0 = arrIn1_0[v1] * arrIn0_0[v1];
            z0 = y0 + x0;
            y0 = z0;
        }
    }

    // Phase 2: Threads cooperatively reduce the local sums to a single value
    ix = min(shapeSize - blockIdx.x * blockDim.x, blockDim.x);
    sdata0[threadIdx.x] = y0;
    __syncthreads();
    if (threadIdx.x + 512 < ix) {
        x0 = sdata0[threadIdx.x + 512];
        z0 = y0 + x0;
        y0 = z0;
    }
    __syncthreads();
    sdata0[threadIdx.x] = y0;
    __syncthreads();
    if (threadIdx.x + 256 < ix) {
        x0 = sdata0[threadIdx.x + 256];
        z0 = y0 + x0;
        y0 = z0;
    }
    __syncthreads();
    sdata0[threadIdx.x] = y0;
    // cooperative tree reduction in shared memory continues...

    // first thread writes the final result to global memory
    if (threadIdx.x == 0) {
        if (shapeSize > 0) {
            if (gridDim.x == 1) {
                x0 = 0.0f;
                z0 = y0 + x0;
                y0 = z0;
            }
            arrOut_0[blockIdx.x] = y0;
        } else {
            arrOut_0[blockIdx.x] = 0.0f;
        }
    }
}

```

**Listing 4.3:** Generated CUDA code for the fused vector dot-product operation

into the predefined points of the template code. Inlining of scalar functions is always possible as the scalar sub-language of Accelerate is first-order and does not support recursion. These restrictions are necessary in order to generate GPU code, as GPU hardware neither supports large stacks (for recursion) nor closures (for higher-order functions).

To splice scalar code fragments into the skeleton code of array operations, we define a typeclass of l-values and r-values to define a generic assignment operator (`(.=.)`). For example, the operator was used in lines (1) and (2) of Listing 4.1. This representation abstracts over whether our skeleton uses l-values in single static assignment-style to `const` declarations, or as a statement updating a mutable variable. The class declarations are the following:

```
class Lvalue a where
    lvalue :: a → C.Exp → C.BlockItem

class Rvalue a where
    rvalue :: a → C.Exp

class Assign l r where
    (.=.) :: l → r → [C.BlockItem]

instance (Lvalue l, Rvalue r) ⇒ Assign l r where
    lhs .=. rhs = [ lvalue lhs (rvalue rhs) ]
```

The `Assign` class allows for flexibility when dealing with tuples (§4.2.3), shared subexpressions (§4.2.4), and the removal of unused expressions (§4.2.5).

### 4.2.3 Representing tuples

Accelerate arrays of primitive type, such as `Float` and `Int`, are easily represented in CUDA using the corresponding floating point and integral types. More interesting is the case of arrays of tuples. A naïve implementation of tuples in CUDA might use arrays of `struct` type — that is, we might consider representing values of type `Array DIM1 (Int, Float)` in CUDA C by a value of type:

```
typedef struct { int a; float b; } arrayIntFloat[];
```

Known as the *array-of-struct* (AoS) representation, this strategy is in general not efficient as it easily violates the strict memory access rules imposed on CUDA devices, decreasing effective bandwidth by as much as an order of magnitude [98].

#### Non-parametric array representation

To avoid this inefficiency, Accelerate uses a non-parametric array representation: arrays of tuples are represented as tuples of arrays of primitive type. This *struct-of-array* (SoA) representation stores arrays of type `Array DIM1 (Int, Float)` by values of the type:

```
typedef struct { int a[]; float b[]; } arrayIntFloat;
```

By virtue of this non-parametric array representation, Accelerate (1) maintains global memory access coalescing rules; and (2) is able to avoid redundant reads of array elements that are never used. The second point will be discussed further in the following section.

### Non-parametric scalar expressions

This non-parametric array representation also existed in the old version of Accelerate [29]. However, since code generation was based on C++ templates and CPP macros, and moreover, the skeleton code was required to abstract over array element types, C **structs** were used to represent tuples of scalar values. A family of getter and setter functions were then generated to create and consume these **structs** when reading and writing elements of the non-parametric array representation.

However, this leads to the generation of dead code whenever specific template instances don't use some of their parameters or fields of the structured data. As an example, consider the following Accelerate function that projects the first component of each element of a vector of quadruples:

```
fst4 :: Acc (Vector (a,b,c,d)) → Acc (Vector a)
fst4 = map (λv → let (x,_,_,_) = unlift v in x)
```

The function **unlift** turns an embedded scalar expression that yields a quadruple, into a quadruple comprising four embedded scalar expressions — hence, we can perform pattern matching in the let binding in order to unpack the expression. Moreover, in **fst4** under the old code generation strategy, array elements are copied into a **struct**, only for the first element to be extracted again and the **struct** to be discarded. One might hope that the CUDA compiler (1) spots the redundant copying of array elements; and (2) that the elements of three of the four arrays are never used. Alas, this does not happen, and as a result **fst4** generates considerable memory traffic.

With template meta programming and the **Assign** type class introduced previously, we fare much better. As the entire skeleton template is quasiquoted, we can inline the definitions of scalar expressions, including array accesses, directly into the skeleton template. Instead of packaging the tuple components into a **struct**, we represent it by a list of primitive values, one per component. During code generation, we keep track of the values constituting a tuple by maintaining a list of expressions, one for each component of the tuple. The **(.=.)** operator allows us to assign all values constituting the tuple with one assignment in the meta programming system; i.e., we have lists of l-values and r-values:

```
instance Assign l r ⇒ Assign [l] [r] where
  []     .=. []      = []
  (x:xs) .=. (y:ys) = assign x y ++ assign xs ys
```

#### 4.2.4 Shared subexpressions

A well known problem with deeply embedded languages is the issue of *sharing*, where, without *sharing recovery*, each occurrence of a let-bound variable in the source program creates a separate unfolding of the bound expression in the reified program. To avoid this problem, we implemented a sharing recovery algorithm that recovers exactly those let-bindings used in the source language. See Section 5.1 for details on our approach to sharing recovery, which appeared in [89] and is a variant of Gill’s observable sharing [48] — his paper also discusses the issue of sharing in embedded languages in greater detail.

Following sharing recovery and conversion of the source program into the internal de Bruijn representation, sharing of scalar subcomputations is represented explicitly by the term:

```
Let :: (Elt bnd, Elt body)
  ⇒ PreOpenExp acc env      aenv bnd           — (1) bound expression
  → PreOpenExp acc (env, bnd) aenv body        — (2) body expression, scope of local binding
  → PreOpenExp acc env      aenv body
```

Note that the result of evaluating the let-bound expression (1) is only in scope while evaluating the body of the let-binding (2). This property is encoded in the type of the environment of the body expression.

To support code generation including shared subexpressions, the CUDA code generator uses a monad to generate fresh names. The monad is also used to store the expressions representing let-bound terms, which must be evaluated before evaluation of the body.

```
data Gen = Gen { unique      :: Int           — fresh name generation
                , localBindings :: [C.BlockItem]   — let-bound terms
                , usedTerms    :: Set C.Exp }     — see Section 4.2.5
```

Although we use generated names to avoid accidental name capture, since the code generation state is refreshed for each kernel that is generated, the same names will be generated for equivalent kernels, so we will still be able to reuse generated kernels (§4.4.2).

When code generation encounters a let binding, we first generate code for the bound expression, assign this result to a new (fresh) variable, then use this stored variable throughout the evaluation of the body:

```
codegenOpenExp (Let bnd body) env = do
  bnd' ← codegenOpenExp bnd env >>= pushEnv bnd
  body' ← codegenOpenExp body (env `Push` bnd')
  return body'
```

Recall that Accelerate’s CUDA code generator uses a non-parametric representation of tuples, where a scalar expression is represented as a list of C expressions, one per component of the (flattened) tuple (§4.2.3). The function `pushEnv` generates a fresh name for each expression in the tuple, assigns the expression to a new variable, and returns the new variable names to be used throughout evaluation of the body expression.

```

pushEnv :: exp env aenv a → [C.Exp] → Gen [C.Exp]
pushEnv dummy exps = zipWithM go (expType dummy) exps
  where
    go t c = case c of
      C.Var{} → return c
      _ → do name ← fresh
             let next = [citem| const $ty:t $id:name = $exp:c; []]
                 modify (λs → s { localBindings = next : localBindings s })
             return (cvar name)

```

— (1)

Note that there is no need to create a fresh name if the input term is a variable (1). That is, we avoid creating additional declarations in the generated code such as:

```

const int x0 = ...
const int x1 = x0;

```

This observation also simplifies the implementation of dead-code analysis (§4.2.5). The output of code generation is the list of C expressions representing the final computation, together with a list of local declarations to be evaluated first (`localBindings`). With the `Assign` type class introduced previously, we can transparently bring these extra terms into scope before evaluating the body, with the addition of the following instance:

```

instance Assign l r ⇒ Assign l ([C.BlockItem], r) where
  lhs .=. (env, rhs) = env ++ (lhs .=. rhs)

```

## Future work

One disadvantage of the current implementation is that the scope of let bindings is lost: once a new variable is declared, it remains in scope to the end of the skeleton. This is in contrast to the definition of the `Let` AST node, where the binding is only in scope during evaluation of the body. Adding explicit scoping to specify the lifetime of expressions may help the CUDA compiler improve register allocation.

Generating code with explicit scoping may also remove the need for a monad to generate fresh names, since unique names can be derived based on the de Bruijn *level* of the expression, which can be determined directly from the size of the environment.

### 4.2.5 Eliminating dead code

As discussed in Section 4.2.3, one problem of the original code generator based on CPP and C++ templates was its inability to remove some forms of dead code. This was seen in the `fst4` example of Section 4.2.3, which generates significant memory traffic if tuples are represented as `structs`, even though three of the four values read from memory were never used. With the use of template meta programming to inline the definition of scalar expressions directly into the skeleton template, the situation is greatly improved.

Unfortunately, the CUDA compiler does not always eliminate memory reads, as it does not always detect if the values are used. Hence, rather than rely on the CUDA compiler, the code generation process explicitly tracks which values are used in the generated scalar code, and elides statements whose results are not used when splicing assignments into a skeleton template. The following instance of the `Assign` class uses a flag that is `False` to indicate that the assigned value is not used.

```
instance Assign l r => Assign (Bool,l) r where
  (used,lhs) .=. rhs
  | used      = assign lhs rhs
  | otherwise = []
```

The `map` skeleton of Listing 4.1 exploits this: when generating code for the mapped function `f`, the function `dce :: [a] → [(Bool,a)]` on the line marked (2) is also generated, and determines for each term whether it is subsequently used. Then, when the code generated by `get` reads data from the input array, it does not read in any unused values. Consequently, the function `fst4` will only touch the array representing the first component of the quadruple of arrays.

To generate the function `dce`, which ultimately computes the `used` flags, we need to determine which inputs to the generated scalar function are ultimately used in the calculation. Code generation for a function such as `fst4` begins by initialising the scalar environment of the function body with input variables of a known name (1):

```
codegenFun1
  :: DeviceProperties
  → Gamma aenv
  → DelayedFun aenv (a → b)
  → CUFun1    aenv (a → b)
codegenFun1 dev aenv (Lam (Body f)) =
  let
    dummy      = locals "undefined_x" (undefined :: a)                                — (1)
    Gen _ _ used = execGen $ codegenOpenExp dev aenv f (Empty `Push` dummy)           — (2)
  ...

```

During code generation we build a set containing the variables that are used in the function body. In (2) code generation proceeds with the dummy variables, returning the final state which contains the set of used variables. Dead-code analysis then builds the function `dce` by testing whether each input variable is present in the set:

```
deadCodeElim :: Set C.Exp → [C.Exp] → [a] → [(Bool,a)]
deadCodeElim used vars =
  let flags = map (λx → x `Set.member` used) vars
  in zipWith (,) flags
```

During code generation, we must decide when to add variables to the set. In general, we check the result from each step of `codegenOpenExp` with the function `visit`, and test if the result was a scalar C variable:

```

visit :: [C.Exp] → Gen [C.Exp]
visit exp
| [x] ← exp = markAsUsed x >> return exp
| otherwise    = return exp

markAsUsed :: C.Exp → Gen ()
markAsUsed x@C.Exp{} = modify (λs → s { usedTerms = Set.insert x (usedTerms s) } )
markAsUsed _         = return ()

```

## Future work

Completing our definition of `codegenFun1`, we return the result of dead-code analysis produced by `deadCodeElim`, together with a function that *re-runs* code generation with the actual input variables. Thus, our method requires performing code generation at least twice for every scalar function: once to generates usage information, and then again each time the function is instantiated with a particular set of variables.

```

codegenFun1 dev aenv (Lam (Body f)) =
  let
    dummy      = ...
    Gen _ _ used = execGen $ codegenOpenExp ...
  in
  CUFun1 (deadCodeElim used dummy)
    (λxs → evalGen $ codegenOpenExp dev aenv f (Empty `Push` xs))

```

Furthermore, this method only eliminates unused inputs to a scalar function. If the generated code produced intermediate values (§4.2.4) based on these unused inputs, these expressions will still appear in the generated code. We leave addressing both of these issues to future work.

### 4.2.6 Shapes and indices

Parallelism in Accelerate takes the form of collective operations on arrays of type `Array sh e`, where `sh` is the *shape* and `e` is the *element type* of the array. As discussed in Section 3.1.2, we use an inductive notation of snoc lists to enable rank-polymorphic definitions of array functions. Listing 3.1 shows the types of array shapes and indices.

During code generation, the old version of Accelerate [29] represented multidimensional shapes and indices as a `struct`. This representation additionally came with a family of overloaded C functions for operating on shapes and indices, such as `toIndex` and `fromIndex`, so that skeletons were not specialised to a particular dimensionality.

The new code generation method<sup>3</sup> does not require the use of `structs` to represent shapes. The new approach represents shapes during code generation in the same manner as tuples; as a list of expressions representing each of the individual components of the shape. Operations such as `toIndex` are implemented directly in the skeleton, rather than packing the shape

---

<sup>3</sup>Subsequent to, but not affecting the result of [33]

components into a `struct` and then calling the family of overloaded functions which were provided separately in a header file.

Recall the vector dot product example from Section 4.2.1, where the pairwise multiplication of the two arrays resulted in the scalar function  $\lambda ix \rightarrow (xs!ix) * (ys!ix)$  being embedded into the consumer template. If a `struct`-based representation of shapes is used, this results in the following C code [33]:

```
const Int64 v2 = ix;
const int v3 = toIndex(shIn0, shape(v2));
const int v4 = toIndex(shIn1, shape(v2));
y0 = arrIn0_a0[v3] * arrIn1_a0[v4];
```

Here, `shape` and `toIndex` are C functions provided by the library, which map multidimensional indices to the linear array representation. Since the source arrays are vectors, they do not contribute anything to the example, in particular, their definitions are:

```
static __inline__ __device__ int toIndex(const DIM1 sh, const DIM1 ix)
{
    assert(ix >= 0 && ix < sh);
    return ix;
}

static __inline__ __device__ DIM1 shape(const int a)
{
    return a;
}
```

Thus, `v3` and `v4` map, indirectly, to `v2`. Luckily, in this case the CUDA compiler is able to remove the superfluous assignments. With the new code generator method, we produce the following code which does not introduce the redundant assignment, and thus does not rely on optimisations performed by the CUDA compiler:

```
const Int64 v1 = ({ assert(ix >= 0 && ix < min(shIn0_0, shIn1_0)); ix; });
y0 = arrIn0_0[v1] * arrIn1_0[v1]; // (1)
```

The r-value at (1) is a statement expression:<sup>4</sup> a compound statement enclosed in parentheses that may be used as an expression, where the last thing in the compound statement is an expression followed by a semicolon, and serves as the value for the entire construct.

#### 4.2.7 Lambda abstractions

Concerning lambda abstractions, Accelerate's scalar expression language is first order in the sense that, although it includes lambda abstractions, it does not include a general application form. This ensures that lambda abstractions of scalar expressions can only be used as the arguments to collective operations, such as `zipWith`. This restriction is necessary in order to generate code for GPU hardware. As a consequence, lambda abstractions are always

---

<sup>4</sup><http://gcc.gnu.org/onlinedocs/gcc/Statement-Exprs.html>

outermost (in type correct programs) and we can always translate them into either (a) plain C functions, or (b) inline them directly into the use site. The old code generator had no choice but to select the first alternative, whereas the new method can make use of several optimisations by using the second approach, such as def-use analysis (my implementation of restricted dead code elimination; see Section 4.2.5).

A recent change to the array and scalar language has been the introduction of explicit value iteration. In the scalar language, this is encoded with the following AST node, which behaves analogously to a standard C while loop:

```
While :: Elt a
  ⇒ PreOpenFun acc env aenv (a → Bool)      — (1) continue while true
  → PreOpenFun acc env aenv (a → a)          — (2) function to iterate
  → PreOpenExp acc env aenv a                — initial value
  → PreOpenExp acc env aenv a
```

Although we use scalar functions to represent the iteration test (1) and loop body (2), the operation is suitably restricted such that, so long as some care is taken when generating the code for the loop, we do not need to perform lambda lifting of the function valued arguments. Unfortunately, we can not use the standard `codegenFun1` operation to generate the CUDA code for the unary function, as this interferes with our implementation of def-use analysis.

The following function is used to generate C code for the loop operations, which generates code for the function body in a clean environment (1), and restores the outer environment on exit (2), including any newly introduced bindings as part of the return value (3).

```
cvtF1 :: DelayedOpenExp env aenv t → Val env → Gen ([C.BlockItem], [C.Exp])
cvtF1 (Lam (Body e)) env = do
  old ← state (λs → ( localBindings s, s { localBindings = [] } ))
  e' ← codegenOpenExp e env
  env' ← state (λs → ( localBindings s, s { localBindings = old } ))
  return (reverse env', e')
```

Thus, any shared subexpressions (§4.2.4) encountered during code generation are evaluated before executing the function body, using the `Assign` class discussed earlier.

#### 4.2.8 Array references in scalar code

Accelerate includes two scalar operations that receive an array-valued argument, namely array indexing (!) and determining the `shape` of an array. These are, for example, used in the sparse-matrix vector multiplication operation shown in Listing 6.6. Specifically, this code includes the following use of `backpermute` (§2.4.3) to extract the values of the vector that are to be multiplied with the non-zero elements of the sparse matrix:

```
A.backpermute (shape inds) (λi → index1 $ inds!i) vec
```

Here the array computation `inds :: Acc (Array DIM1 Int)` is used in the first and second argument of `backpermute`. In the code for `smvm`, `inds` is a previously let-bound variable. If

instead, a collective operation would have been used in place of `inds` in the scalar function  $\lambda i \rightarrow \text{index1 } \$ \text{ inds!}i$ , we would lift it out of the scalar function and let bind it. After all, we do not want to execute an arbitrarily complex array computation once for every invocation of a scalar function. In fact, CUDA does not permit us to do this, as the scalar function is translated into GPU kernel code, which can not include further parallel operations.<sup>5</sup>

Nevertheless, even though array valued arguments to scalar functions will always be let-bound variables, code generation is not straightforward. Recall that in the old version of Accelerate [29], skeleton templates were static, and were instantiated by the code generator by combining them with scalar code fragments at CUDA compilation time. Since the skeleton template — and in particular the parameters to the skeleton function — were static, we could not add a new argument to the prototype of the `backpermute` function. This problem was worked around by using *texture references*, a CUDA feature that comes from its graphics heritage, to define what are essentially read-only arrays as global tables on a per skeleton instantiation basis. The use of texture references also had another happy consequence: skeletons such as `backpermute` allow unconstrained indexing patterns, which may not follow the strict requirements for coalesced and aligned access to global memory. Not following these memory access rules can incur a severe performance penalty [98], but since texture memory access is cached, it may be more efficient in these circumstances.

In contrast to traditional caches, although the texture cache reduces bandwidth requirements to global memory, it does not reduce access *latency* [98, §5.3.2]. However, beginning with compute capability 2.0, the GPU also comes equipped with a more traditional L2 cache. Since the new code generator based on quasiquoting [33] is able to dynamically rewrite the entire skeleton — including the function prototype — we are no longer forced to access array-valued arguments via global variables. In order to use the inbuilt L2 cache available devices of compute capability 2.0 and higher, arrays referenced from scalar code are marshalled to the kernel via the kernel function parameters, and read using standard array indexing. For older series 1.x devices that do not have a L2 cache, the texture cache is still used instead.

#### 4.2.9 Conclusion

The use of template meta programming for skeleton definition and instantiation enables us to combine the advantages of conventional code generators, such as def-use analysis for dead code elimination, with those of generative skeleton-based code generators, such as handwritten idiomatic code for special-purpose architectures. Finally, and most importantly, the use of quasiquotation for template instantiation permits the implementation of producer-consumer skeleton fusion, which is a crucial optimisation even for code as simple as dot product (§5.2).

---

<sup>5</sup>The most recent Kepler architecture (compute capability 3.5) introduced *dynamic parallelism*, whereby GPU threads are able to launch new parallel operations. At this time Accelerate does not make use of this feature. Moreover, in this case this is not what we want, because each thread would spawn a new instance of the *same* parallel operation, rather than computing the array once and sharing the result.

## 4.3 Using foreign libraries

Accelerate is a high-level language capturing idioms suitable for massively parallel GPU architectures, without requiring the expert knowledge required to achieve good performance at the level of CUDA. On the other hand, there exist highly optimised libraries that implement well known algorithms, such as operations from linear algebra and fast Fourier transforms. For Accelerate to be practically useful, we need to provide a means to use those libraries. Moreover, access to native code also provides the developer the opportunity to drop down to raw CUDA C in those parts of an application where the code generated by Accelerate is not fast enough. We achieve this with the *Accelerate Foreign Function Interface* (or FFI).

The Accelerate FFI is a two-way street: (1) it enables calling native code from embedded Accelerate computations; and (2) it facilitates calling Accelerate computations from C code. A developer can implement an application in a mixture of Accelerate and foreign libraries while maintaining that the source code is portable across multiple Accelerate backends.

Given that Accelerate is an embedded language in Haskell, it might seem that Haskell's standard FFI should be sufficient to enable interoperability with foreign code. However, this is not the case. With Haskell's standard FFI, we can call C functions that in turn invoke GPU computations from Haskell host code. However, we want to call GPU computations from within embedded Accelerate code and pass data structures located in GPU memory directly to native CUDA code and vice versa. The latter is crucial, as transferring data from CPU memory to GPU memory and back is very expensive.

The Accelerate FFI is the, to our knowledge, first foreign function interface for an embedded language. The design of the Accelerate FFI was largely completed by another student, Robert Clifton-Everest, and so I mention only the main points here. See [33] for details.

### 4.3.1 Importing foreign functions

Calling foreign code in an embedded Accelerate expression requires two steps: (1) the foreign function must be made accessible to the host Haskell program; and (2) the foreign function must be lifted into an Accelerate computation to be available to embedded code. For the first step, we use the standard Haskell FFI. The second step requires an extension to Accelerate.

To address the second step, we extend the Accelerate language with a new AST node type `Aforeign` representing foreign function calls. One instance of an `Aforeign` node encodes the code for one backend, but also contains a fallback implementation in case a different backend is being used. The AST data constructor is defined as:

```
Aforeign :: (Arrays as, Arrays bs, Foreign f)
    => f as bs
    -> (Acc as → Acc bs)
    -> Acc as
    -> Acc bs
```

— (1) *foreign function*  
 — (2) *fallback implementation*  
 — *input array*

When code generation encounters an `Aforeign` node, it dynamically checks whether it can execute the foreign function (1). If it can't, it executes the fallback implementation (2). The fallback implementation might be another `Aforeign` node with native code for another backend (for example, for C instead of CUDA), or it could simply be a vanilla Accelerate implementation of the same function. In this way, a cascade of `Aforeign` nodes can provide an optimised native implementation of a function for a range of backends.

Similarly, we extend the Accelerate scalar language with a `Foreign` node, to insert calls to foreign CUDA functions directly into the generated skeleton code.

### 4.3.2 Exporting Accelerate programs

Accelerate simplifies writing GPU code as it obviates the need to understand most low-level details of GPU programming. Hence, we would like to use Accelerate from within other languages. As with importing foreign code into Accelerate, the foreign export functionality of the standard Haskell FFI is not sufficient for efficiently using Accelerate from other languages.

To export Accelerate functions as C functions, we provide a Template Haskell [122] function `exportAfun` that generates the necessary export declarations for a given Accelerate function. Compiling a module, say, `M.hs`, that exports Accelerate computations:

```
dotp :: Acc (Vector Float, Vector Float) → Acc (Scalar Float)
dotp = uncurry $ λxs ys → A.fold (+) 0 (A.zipWith (*) xs ys)

exportAfun 'dotp "dotp_compile"
```

generates the additional file `M_stub.h` containing the C prototype for the foreign exported function:

```
#include "HsFFI.h"
#include "AccFFI.h"
extern AccProgram dotp_compile(AccContext a1);
```

The C function provided by `exportAfun` compiles the Accelerate code, returning a reference to the compiled code. The provided C function `runProgram` is then used to marshal input arrays, execute the compiled program, and marshal output arrays. This design mirrors the behaviour of separating the compilation and execution phases provided by the `run1` function from Haskell. See Section 4.6 for details.

## 4.4 Dynamic compilation & code memoisation

When you write a program in Accelerate, what you are really doing is writing a Haskell program that *generates* a CUDA program, that is compiled, loaded, and executed on the GPU. Thus, one major difference between using Accelerate for general purpose GPU programming compared to programming in CUDA directly, is that kernels are generated dynamically, at application *runtime*, whereas plain CUDA code is pre-compiled.

Dynamic code generation has significant advantages, especially for embedded languages. In particular, the code generator can query the capabilities of the hardware on which the code will be executed, and optimise the generated program accordingly, as well as specialise the code to the available input data. The host program can also be used to generate the embedded program.

The main drawback of dynamically generating and compiling GPU kernels is the additional execution time required to do so, so attempts must be made to mitigate these overheads. In general purpose GPU programming, it is only worthwhile to offload computations to the GPU if they are computationally intensive. This implies a significant runtime and usually the use of significant amounts of input and/or output data. Hence, the overhead of dynamic kernel compilation is not necessarily problematic in the face of long kernel runtimes and long data-transfer times between host and device memory, especially if those kernels are compiled once and executed many times.

#### 4.4.1 External compilation

In unison with the AST traversal that extracts `Use` subterms that initiates data transfers (§4.5.2), the CUDA backend initiates code generation for each collective array operation it encounters, by template instantiation (§4.1). After the CUDA code is generated, it must be compiled with the standard, external `nvcc` CUDA compilation tool-chain to produce binary object code for each of the kernels implementing the collective operation.

As with data transfer, the compilation of the generated CUDA code proceeds asynchronously and in parallel with other operations, including compilation of other kernels. For each generated kernel, the CUDA compiler is invoked by spawning an external process that executes the raw system command. However, if the program requires the compilation of many kernels, carelessly spawning a new process for every instantiated skeleton can cause the IO bus to become saturated, increasing overall compilation time, or even exhaust the host operating system's available process handles. Rather than spawn each external process directly, a worker pool is created that is used to process jobs in first-in first-out (FIFO) order:<sup>6</sup>

```
{# NOINLINE worker #-}
worker :: Queue.MSem Int
worker = unsafePerformIO $ Queue.new =<< getNumProcessors
```

The worker pool is defined as the Haskell equivalent of a global variable<sup>7</sup> and is initialised once per program execution. The number of slots in the queue is set to the number of physical CPU cores present in the host system<sup>8</sup> which specifies the maximum number of kernels to compile concurrently. Instead of initiating the compilation process directly, it is instead added to the work queue:

---

<sup>6</sup>Queue denotes the `SafeSemaphore` package: <http://hackage.haskell.org/package/SafeSemaphore>.

<sup>7</sup>The idiom popularly known as “the unsafePerformIO hack”.

<sup>8</sup>Note that this is not limited by the RTS option `-N` that specifies the number of processors to use.

```

enqueueProcess :: FilePath → [String] → IO (MVar ())
enqueueProcess nvcc flags = do
    mvar ← newEmptyMVar
    ← forkIO $ do
        Queue.with worker $ do
            (_,_,_	pid) ← createProcess (proc nvcc flags)
            waitFor pid
            putMVar mvar ()
    return mvar

```

— (2)  
— (1)  
— (3)  
— (4)

1. The process is spawned from a separate thread so that the thread can block waiting for both the worker queue and the external process, without interrupting the main thread. Thus compilation proceeds asynchronously with other host processing tasks.
2. An empty `MVar` is created that will act as a signal to the main thread that the external process has completed. Since `takeMVar` on an empty `MVar` blocks the thread until that `MVar` is filled, the main thread can wait for the external process to complete if the compiled code is not yet available by the time it is required during the execution phase.
3. The thread waits for a worker to become available from the queue. Blocked threads are woken up by the runtime system and are serviced in a first-in first-out order; that is, threads do not actively poll the queue continually testing for an open slot. Once a worker becomes available, the external process is launched and the thread once again blocks until that process completes.
4. Once the process completes the worker is returned to the queue where it can be assigned to handle any further tasks. Finally, the thread fills the `MVar` to signal the main thread that the compilation has completed.

We return to the question of linking in the compiled code when we discuss the process of executing the program (§4.6).

#### 4.4.2 Caching compiled kernels

The CUDA backend associates each CUDA binary with the skeleton instantiation whose computation that binary implements. The binary object code is keyed on a skeleton and the parameters of its instantiation, not to the specific AST node used to instantiate the skeleton. As a result, compiled binaries are reusable when the same skeleton instantiation is required again, whether that is in the same Accelerate computation applied to a different set of input arrays, or an entirely different computation. For example, operations such as `scanl (+) 0` are common, and it would be wasteful to dynamically generate and compile the same code multiple times.

The *program cache* is a hash table which is keyed by an MD5 [110] digest of the generated CUDA code which instantiates the skeleton, as well as the compute capability the generated code was specialised and compiled for.

```
type ProgramCache = MVar ( HashTable KernelKey KernelEntry )

type KernelKey     = (CUDA.Compute, ByteString)
data KernelEntry
  = CompileProcess FilePath (MVar ())
  | KernelObject ByteString (FullList CUDA.Context CUDA.Module) — (1)
  | KernelObject ByteString (FullList CUDA.Context CUDA.Module) — (2)
```

The hash table value represents the state of compilation, which can be either:

1. A currently compiling external process. This records the path of the file currently being compiled, which will be rooted in a temporary directory, as well as the `MVar` that will signal when the external process has completed (§4.4.1). If a kernel is requested that is currently compiling, the thread blocks on the `MVar` until the result is available and then updates the hash table with;
2. The raw compiled data with a non-empty list of CUDA contexts that the object code has been linked into. The entry may have been added to the cache by an alternate context of the same compute capability, for example on a system with multiple CUDA devices, in which case the code simply needs to be re-linked for the current context.

This scheme effectively caches kernels both within the same Accelerate computation and across different Accelerate expressions. However, it must be regenerated every time the program is executed, which increases program startup time. To alleviate this, once a kernel is compiled, the binary object code is saved in the user’s home directory so that it is available across separate runs of the program, or indeed different programs. This information is stored in a *persistent cache*:

```
type PersistentCache = MVar ( HashTable KernelKey () )
```

The persistent cache is a hash table indexed by the same key used by the in-memory cache described above, but without a value component. Instead, the presence of a key in the persistent cache indicates that the compiled binary represented by that key is available to be loaded from disk. The location of the object code is determined from the MD5 digest by creating a ASCII representation of the digest, replacing special characters with z-encoded strings.

```
encode_ch :: Char → String
encode_ch c | unencodedChar c = [c] — regular characters except ‘z’ and ‘Z’
encode_ch '(' = "ZL" — punctuation using two-character sequences
encode_ch ')' = "ZR"
...
encode_ch c    = encode_as_unicode_char c — other characters using hexadecimal
```

Initialising the persistent cache populates the hash table by reading an index file from disk, which is simply a binary file containing the number of entries at the start of the file, followed by the keys.

```
restore :: FilePath → IO PersistentCache
restore db = do
    exist ← doesFileExist db
    pt   ← case exist of
        False      → encodeFile db (0:Int) >> HashTable.new
        True       → do
            store      ← L.readFile db
            let (n,rest,_) = runGetState get store 0
            pt         ← HashTable.newSized n

    let go []      = return ()
        go (!k:xs) = HashTable.insert pt k () >> go xs

    go (runGet (getMany n) rest)

newMVar pt
```

The `binary` package is used to deserialise the data from file. Note that the default implementation for serialising lists of elements provided by the `binary` package preserves the order of elements in the list, which requires creating the entire list structure in memory.<sup>9</sup> Since we do not require this property, we provide our own deserialisation routine that allows the elements to be lazily consumed and added directly to the hash table as they are read from file.

```
getMany :: Binary a ⇒ Int → Get [a]
getMany n = go n []
where
    go 0 xs = return xs
    go i xs = do x ← get
                 go (i-1) (x:xs)
```

Finally, we present the entire *kernel table* data structure used to cache compiled kernels, consisting of in-memory kernels tracked by the program cache as well as the on-disk kernels tracked by the persistent cache.

<code>data KernelTable = KT ProgramCache</code> <code>                  PersistentCache</code>	<small>— first level, in-memory cache</small> <small>— second level, on-disk cache</small>
---	---

Looking for entries in the kernel table entails first searching the in-memory program cache, which represents kernels that are (a) currently compiling; (b) compiled, but not linked into the current context; or (c) compiled and linked into the current context. If the kernel is not available in the program cache, then (d) the persistent cache is searched. If found in the

---

<sup>9</sup>A tail-recursive loop that accumulates the list structure as the file is read sequentially from disk produces list elements in reverse order. Thus, the list must be reversed before being returned, making the structure spine-strict and preventing elements being consumed as they are read from file.

persistent cache, the associated object file is linked into the current context, and the program cache is updated to include this entry. As each new kernel is compiled, it is added to the program cache, and the on-disk persistent cache file is updated once compilation completes and the file is linked. Note that the implementation does *not* need to keep the in-memory and on-disk representations of the persistent cache synchronised.

#### 4.4.3 Conclusion

This section discusses the implementation of a dynamic compilation system for an embedded language. To make such a system practicable, much attention must be paid to amortising the additional overheads of runtime code generation and compilation. In the Accelerate CUDA backend, this is achieved through a system of (a) kernel caching, which eliminates duplicate compilations; (b) asynchronous compilation, which compiles multiple kernels simultaneously and overlaps compilation with other execution tasks; and (c) a method of annotating the Accelerate program which associates each collective operation in the program with the compiled kernels that implement the operation. This last feature is described in Section 4.6.2.

### 4.5 Data transfer & garbage collection

In a standard Haskell program, memory is managed entirely automatically by the runtime system. This relieves a large burden from the programmer and eliminates a source of potential runtime errors. In contrast, in the CUDA programming model all memory management is explicit and handled by the programmer. CUDA devices typically have their own memory, physically separate from the memory of the host CPU, which data must be explicitly transferred to and from.

In order to keep Accelerate programs as close to idiomatic Haskell as possible, the Accelerate runtime system must do the job of managing the CUDA memory space. Moreover, host-device data transfers are expensive, given the relatively high latency and low bandwidth of the PCI-E bus, so minimising data transfer is an important pressure point for achieving high performance [98]. There are several considerations in order to realise this.

#### 4.5.1 Device-to-host transfers

Evaluating Accelerate expressions is done with the following function, which encapsulates the entire process of compiling and executing a program on the GPU. See Section 4.6 for more information.

```
CUDA.run :: Arrays a => Acc a -> a
```

The result of evaluating this program is made available for use in vanilla Haskell, so `run` must copy the final array result back to the host. However, any intermediate results computed during evaluation of the program are *not* copied back to the host.

### 4.5.2 Host-to-device transfers

Accelerate distinguishes between vanilla Haskell arrays (in CPU host memory) from embedded arrays (in GPU device memory). Every array on the host is uniquely associated to an array on the device, which defines where data is copied when transferred between the host and device. The function `use` embeds a Haskell array into an embedded array computation, and thus implies host-to-device data transfer:

```
use :: Arrays arrays ⇒ arrays → Acc arrays
```

At a first approximation, we simply copy the data to the GPU as soon as we encounter the `Use` node in an expression. This occurs during the first phase of program execution, and is done asynchronously so that it may overlap code generation and compilation (§4.4).<sup>10</sup> However, we can do better.

### Intra-expression sharing

Consider the following example, where we `use` a single array twice. Since Accelerate operations are pure and thus do not mutate arrays, the device arrays can be safely shared. Thus, we would like these two arrays to refer to the same data in GPU memory. This reduces the amount of memory required to store the data as well as reducing memory traffic.<sup>11</sup>

```
square :: (Elt e, IsNum e, Shape sh) ⇒ Array sh e → Acc (Array sh e)
square xs = zipWith (*) (use xs) (use xs)
```

As we shall see, since the two `use` nodes refer to the same array on the heap, they will share the same array in GPU memory, and thus the data is only copied once. This is a happy consequence of the method we use to uniquely associate host and device arrays.

### Inter-expression sharing

As a second example, given an array describing the particle density `df` and the direction and magnitude of a velocity field `vf` at each point, the following computes how the density and velocity fields evolve under advection and diffusion:

<pre>fluid :: Timestep       → Viscosity       → Diffusion       → DensityField       → VelocityField</pre>	<i>— time to evolve the simulation</i> <i>— viscous damping factor</i> <i>— mass diffusion rate</i> <i>— particle density at each point in the field</i> <i>— velocity at each point in the field</i>
---	---

---

<sup>10</sup>The implementation could be improved, but this is left for future work. See Section 4.5.5.

<sup>11</sup>Why doesn't sharing recovery (§5.1) notice that the two uses of `xs` are the same and combine them? The sharing recovery algorithm observes the sharing of *Accelerate* terms, and because we have called `use` twice, we are effectively constructing two *separate* Accelerate terms (`Use` nodes). Furthermore, since GHC does not do common subexpression elimination, as it can affect the strictness/laziness of the program and potentially introduce space leaks (<http://ghc.haskell.org/trac/ghc/ticket/701>), these common subexpressions are not combined, two separate values are created on the heap, and there is no sharing to be observed.

```

→ Acc (DensityField, VelocityField)
fluid dt dp dn df vf =
  let vf'      = velocity steps dt dp vf      — dampen velocity field
      df'      = density  steps dt dn vf' df   — move particles based on forces
  in
  A.lift (df', vf')

```

This is the entry function to the program that implements Jos Stam's stable fluid algorithm [123] (§6.8). As is typical with numerical simulations, to evolve the simulation over a period of time the function is called repeatedly over many small time steps, with the output of each step of the simulation forming the input to the next step in the sequence. That is, given some initial values `df0` and `vf0` the simulation proceeds as:

```

simulation dt dp dn df0 vf0 =
  let (df1, vf1)      = run $ fluid dt dp dn df0 vf0  — first step using initial conditions
      (df2, vf2)      = run $ fluid dt dp dn df1 vf1  — second step progresses result of the first
      (df3, vf3)      = run $ fluid dt dp dn df2 vf2  — ...and so on
  ...

```

Thus, the results of the first step `df1` and `vf1` will be the inputs to the second step of the simulation, and so on. Since these arrays were just copied *from* the GPU, we should avoid immediately copying the *same data* back again in the subsequent step.

To be able to avoid this redundant transfer, the Accelerate memory manager needs to operate over the lifetime of the entire Haskell program, and not be constrained to work within a single `run` invocation. Otherwise, all device memory needs to be deallocated after evaluating the expression, or the program will leak memory. Thus, localised memory management techniques such as reference counting or syntax directed allocations are insufficient.

The key observation is that the Haskell runtime system knows when objects will be reused (the result of the first step `df1` is used in the second step), and when they are no longer required and can be garbage collected (`df1` is no longer needed after evaluating the second step and can be safely deallocated). Since every array in GPU memory is uniquely associated with an array on the host, which is managed by the Haskell runtime system, we can attach finalisers [102] to the host array that will deallocate the GPU memory at the same time the Haskell heap object is garbage collected.

#### 4.5.3 Allocation & Deallocation

The Accelerate language defines only pure operations on arrays, so every kernel stores its result into a fresh device array. This style of programming results in a very high memory allocation and deallocation rate. For programs that execute many kernels, such as the fluid simulator, this can have a significant contribution to overall performance (§6.8) because allocations cause a device synchronisation.

Instead of immediately deallocating device memory when an array is garbage collected, we instead remove its association to a specific host array but retain a reference to the device

memory for later reuse. This memory area is the *nursery*, which is simply a finite mapping from array sizes to a chunk of memory of that size. When a new array is allocated, we first check if a block of the appropriate size is available in the nursery. If so, that block is reused, otherwise fresh memory is allocated.

Finally, we note that since garbage collection efficiency often requires postponing the identification of dead objects, relying on finalisation for timely resource recovery is known to be problematic. Accelerate also supports explicit deallocation, which simply forces the finalisers attached to an array to run immediately.

#### 4.5.4 Memory manager implementation

The memory management system is implemented in several stages. The first step is to decompose operations on the surface `Array` data type, into operations on each array of primitive type in the struct-of-array representation (§4.2.3). Implementation of this stage is mostly straightforward. However, care should be taken with the handling of the `use` operation. Since Accelerate is a pure language that does not allow mutation, the array data only needs to be transferred to the device on the first use of that array. If the `use` operation is separated into its constituent `malloc` and `poke` (host to device transfer) operations — as good programmers are wont to do — this is easily forgotten, resulting in the array data being needlessly copied every time `use` is called. As explained in Section 4.5.2, the mapping of host to device arrays ensures that the memory allocation will only ever be performed once, so we must ensure that the data for a `use` operation is only transferred to the device at the time of first allocation.

The *memory table* records the association between host and device arrays. Implemented as a weak hash table [102], the table can purge itself of unneeded key/value pairs, releasing the device memory at the same time by moving it to the nursery. The table will also release itself when no longer required, initialising the finalisers for all remaining key/value pairs as it goes. Furthermore, the memory table is wrapped in an `MVar`, so that it can be shared between several Accelerate operations evaluated in parallel.

```

type MT          = MVar ( HashTable HostArray DeviceArray )
data MemoryTable = MemoryTable MT (Weak MT) Nursery

data HostArray where
  HostArray :: ∃ e. Typeable e
    => CUDA.Context           — a specific device execution context
    => StableName (ArrayData e) — array data on the host CPU
    => HostArray

data DeviceArray where
  DeviceArray :: ∃ e. Typeable e
    => Weak (DevicePtr e)      — data on the GPU
    => DeviceArray
  
```

```

lookup :: (Typeable a, Typeable b)
  ⇒ Context
  → MemoryTable
  → ArrayData a
  → IO (Maybe (DevicePtr b))
lookup ctx (MemoryTable ref _) arr = do
  sa ← makeStableArray ctx arr
  mw ← withMVar ref (`HashTable.lookup` sa) — (1)
  case mw of
    Nothing → return Nothing
    Just (DeviceArray w) → do
      mv ← deRefWeak w
      case mv of
        Just v | Just p ← gcast v → return (Just p)
        | otherwise → error "lookup:type mismatch"
        Nothing → do
          sa' ← makeStableArray ctx arr
          error ("lookup:dead weak pair:" ++ show sa') — (2)

```

— (3)

**Listing 4.4:** Method to lookup the device array corresponding to a given host array.

The key and value parts of the hash table are packaged into untyped containers so that all arrays are tracked using a single memory table. The `Typeable` constraint permits a type-safe cast to later recover the existentially quantified array type. Furthermore, the host arrays are tagged with a specific CUDA context, which is used to support mapping the host array to multiple devices and execution contexts. Looking up an array in the memory table demonstrates the procedure for working with weak pointers. The implementation of the `lookup` operation is shown in Listing 4.4, which follows the method laid out by Peyton Jones et al. [102]. The following features are of note:

1. The memory table maps a stable name of the argument host-side array to a weak pointer to the corresponding device array of a given context.
2. If the lookup is successful, we can attempt to dereference the weak pointer to find the actual value; an array in device memory. On success, a type-safe cast is used to recover the existentially quantified type of the array data.
3. However, there is an awkward race condition, because at the moment `deRefWeak` is called there might, conceivably, be no further references to the host array `arr`. If that is so, and a garbage collection intervenes following (1), the weak pointer will be tombstoned (made unreachable) before `deRefWeak` gets to it. In this unusual case we throw an error, but by using `arr` in the failure case, this ensures `arr` is reachable and thus ensures `deRefWeak` will always succeed. This sort of weirdness, typical of the world of weak pointers, is why we can not reuse the stable name `sa` computed at (1) in the error message.

In addition to the memory table which tracks associations between host and device arrays, the *nursery* is a hash table from array sizes to a non-empty list of pointers to arrays of that

```

malloc :: ∀ a b. (Typeable a, Typeable b, Storable b)
  ⇒ Context
  → MemoryTable
  → ArrayData a
  → Int
  → IO (DevicePtr b)

malloc ctx mt`MemoryTable _ _ nursery ad n = do
  let multiple x f      = floor ((x + (f-1)) / f :: Double) — (1)
      chunk             = 1024
      n'                = chunk * multiple (fromIntegral n) (fromIntegral chunk)
      bytes             = n' * sizeOf (undefined :: b)

  mp ← Nursery.malloc bytes (deviceContext ctx) nursery
  ptr ← case mp of — (2)
    Just p           → return (CUDA.castDevPtr p)
    Nothing         →
      CUDA.mallocArray n' `catch` λ(e :: CUDAException) → — (3)
        case e of
          ExitCode OutOfMemory → reclaim mt >> CUDA.mallocArray n'
          → throwIO e
  insert ctx mt ad ptr bytes
  return ptr — (4)

```

**Listing 4.5:** Allocate a new device array to be associated with the given host side array. This will attempt to use an old array from the Nursery, but will otherwise allocate fresh data.

size. Keys are additionally tagged with the device execution context within which they are allocated.

```

type NRS      = MVar (HashTable (CUDA.Context, Int) (FullList () (DevicePtr ())))
data Nursery = Nursery NRS (Weak NRS)

```

Querying the nursery returns a pointer to an array of the given size if one is available, removing it from the nursery. If no suitably sized chunks are available, fresh memory is allocated instead. Overall, the method for allocating memory is shown in Listing 4.5.

1. Instead of allocating the array size to be exactly as many elements as needed, we round up to the next highest multiple of a set chunk size. This improves the reuse rate for the nursery. We note that the CUDA runtime also allocates in page-sized chunks.
2. Check the nursery if a block of the right size can be reused. This avoids calling out to the CUDA API to allocate fresh data, which requires a device synchronisation and thus may block until any currently executing kernels complete.
3. If nothing is available from the nursery, allocate a new array. If this fails, attempt to recover any free memory by running any pending finalisers as well as clearing the nursery, then attempt the allocation again.
4. Update the memory table with this new host/device array association.

Finally, the finaliser that is invoked when the host array is garbage collected by the Haskell runtime is shown in Listing 4.6. Note that the finaliser must refer to its surrounding execution

```

finalizer :: Weak CUDA.Context      — execution context this array is allocated in
→ Weak MT                         — memory table
→ Weak NRS                        — nursery
→ HostArray → DevicePtr b        — key/value pair
→ Int                             — array size in bytes
→ IO ()                           — (1)

finalizer weak_ctx weak_tbl weak_nrs key ptr bytes = do
  mr ← deRefWeak weak_tbl
  case mr of
    Nothing → return ()
    Just tbl → withMVar tbl (`HashTable.delete` key)
  — (1)

  mc ← deRefWeak weak_ctx
  case mc of
    Nothing → return ()
    Just ctx → do
      mn ← deRefWeak weak_nrs
      case mn of
        Nothing → bracket_ (CUDA.push ctx) CUDA.pop (CUDA.free ptr)
        Just nrs → Nursery.stash bytes ctx nrs ptr
  — (2)

  — (3)

```

**Listing 4.6:** The finaliser that is attached to a host array, to be executed when that array is garbage collected. This attempts to move the device memory into the Nursery for later reuse, and failing that directly deallocates the memory.

context via weak pointers, otherwise these values will be kept alive until *all* arrays on the host (the keys to our memory table) have become unreachable.

1. If the memory table is still alive, remove this association between host and device arrays.
2. If the CUDA execution context has been garbage collected, then the device memory has implicitly been deallocated already and there is nothing further to do.
3. Otherwise, if the nursery is still active stash the array there for possible later reuse, otherwise deallocate it immediately. Note that since the finaliser might run at any time, we need to reactivate this specific context for the duration of the deallocation.

#### 4.5.5 Conclusion

The Accelerate runtime automatically manages the CUDA memory space through the use of finalisers to hook into the garbage collector of the runtime system of the host program. This approach is able to avoid several types of redundant data transfer between the host CPU and attached GPU, which is important for achieving high performance.

The amount of memory available on the device often limits the size of problems that a GPU can be used to solve. The current system is designed to minimise host-device memory transfers, which can increase the lifetime of arrays stored on the device. Future work could improve handling of programs which exceed the available memory, for example in the case where this is due to the use of many intermediate or input arrays, but where individual sub-problems fit within the available memory. In this case, arrays that are not required for

the current calculation could be evacuated from the device by transferring data back to the associated host-side array, and re-transferred back to the device only once required. Further inspiration for reducing the intermediate memory footprint of computations could be found in the areas of stream processing [131] or the piecewise execution of nested data-parallel programs [66, 79, 106].

While it is possible to overlap memory transfers with both kernel executions and tasks such as compilation and optimisation, the current implementation does not completely implement this. To support asynchronous memory transfer requires the use of non-default streams and event waiting, the same as required for concurrent kernel execution (§4.6). The missing piece is that the array on the host needs to be *page-locked* or *pinned* so that it can be accessed directly by the GPU using direct memory access (DMA). Regular memory allocations are *pageable* by default, which means the host operating system is free to change the physical location of the data, such as moving the data to disk in order to free up memory for other applications. Since the location of pageable memory can change, and may not exist in physical RAM at all, this memory can not be directly accessed by the GPU. Instead, when copying data to the GPU from pageable memory, the CUDA runtime must first copy the data to a staging area allocated using pinned memory, and then transfer the data to the device from the staging area (via DMA). Only the second step of this process proceeds asynchronously. Allocations on the host are currently handled by the core Accelerate package, which contains no CUDA specific operations such as the use of pinned memory memory.<sup>12</sup> Note that page-locked memory is a scarce resource, and using it reduces the amount of physical memory available to the operating system, so consuming too much page-locked memory can reduce overall system performance [98].

## 4.6 Executing embedded array programs

The previous sections have discussed how to generate code for collective operations in an Accelerate program (§4.2), compile the generated code asynchronously and in parallel with other operations such as data transfer to the GPU (§4.4.1), as well as our approach to kernel caching in order to reduce the overheads of dynamic compilation (§4.4.2). These operations occur during the first traversal of the program AST (see Figure 3.1 for an overview). This section discusses the second phase of evaluating an Accelerate computation: loading the compiled code onto the GPU and executing the generated kernel(s) for each collective operation, in a second bottom-up traversal of the program AST.

---

<sup>12</sup>The memory allocated by Accelerate is pinned only with respect to the Haskell garbage collector, not with respect to the host operating system kernel. Regardless, the CUDA runtime maintains its own notion of pinned memory separate from the kernel's notion of pinned memory as specified using `mlock()`, since DMA requires not only the virtual address but the *physical* address to remain constant.

### 4.6.1 Execution state

The CUDA backend provides the following function which encapsulates the entire process of compiling and evaluating an embedded array program denoted in the Accelerate language:

```
CUDA.run :: Arrays a ⇒ Acc a → a
```

As the previous sections have demonstrated, running an Accelerate computation in the CUDA backend entails the use of much internal state, in particular a `StateT` monad stacked over `IO`. This is necessary to use the Haskell foreign function interface (FFI) to communicate with the CUDA runtime, maintain the CUDA environment’s device context, and to keep track of internal resources such as GPU memory and the compiled kernel code.

Nevertheless, that we are under the hood manipulating CUDA through the FFI and that we need to maintain internal state should not distract from the property that `run` is a pure function at the level of the user-level Accelerate API. In fact, Accelerate provides an alternative backend implemented as a purely functional interpreter. The interpreter forms an executable specification of the denotational semantics of any Accelerate backend, and can be used to verify accelerated backends.

### 4.6.2 Annotating array programs

Terms in the Accelerate language are defined non-recursively (§3.1.7). That is, the GADT used to represent nodes of the program AST are parameterised by the recursive knot (§3.1.8):

```
data PreOpenAcc acc aenv a where
    Map :: (Shape sh, Elt a, Elt b)
        ⇒ PreFun      acc aenv (a → b)
        → acc          aenv (Array sh a)
        → PreOpenAcc acc aenv (Array sh b)
    ...
    ...
```

*— the type parameter `acc` is the “recursive knot”*

This characteristic is used in order to improve the performance of executing array programs in the CUDA backend. In addition to the function `run` mentioned earlier for executing array programs, we also provide the following function, which prepares and executes an embedded array program of one argument:

```
CUDA.run1 :: (Arrays a, Arrays b) ⇒ (Acc a → Acc b) → a → b
```

The idea is that, if we have an array program (first argument) that is executed many times, we want to avoid having to repeat the first phase of execution every time the function is applied to new input data (second argument). That is, rather than at each invocation of the function, generating code for every operation and finding each time that the compiled code already exists in the kernel cache, we would instead like to know precisely which kernel object needs to be executed without even consulting the caches. The Accelerate CUDA backend uses the knot-tying technique to achieve this.

Following the fusion transformation, the Accelerate array program is parameterised by a series of `Manifest` and `Delayed` nodes, representing operations that will eventually be executed on the device, and computations that have been fused (embedded) into other operations, respectively. See Section 5.2.3 for details of the representation. To execute a CUDA computation, manifest nodes are annotated with the compiled kernel code implementing the computation, while delayed computations recall only the shape of the computation that was embedded within its consumer.

```
data ExecOpenAcc aenv a where
  ExecAcc   :: FL.FullList () (AccKernel a)
    → Gamma aenv
    → PreOpenAcc ExecOpenAcc aenv a
    → ExecOpenAcc aenv a

  EmbedAcc :: (Shape sh, Elt e)
    → PreExp ExecOpenAcc aenv sh
    → ExecOpenAcc aenv (Array sh e)
```

— (1) *compiled kernel code*  
 — (2) *arrays referenced from scalar code*  
 — (3) *shape of embedded array*

1. A non-empty list of the kernels required to execute this node of the computation.
2. The free array variables required to execute the kernel. In particular, this maps environment indexes (de Bruijn indices) to some token identifying that array in the generated code. Rather than using the de Bruijn index directly in the generated code, the extra indirection results in generating code that is less sensitive to the placement of let bindings, ultimately leading to better kernel caching.
3. The size of the input array that was embedded into the consumer, which is often required when executing the kernel the operation was embedded within.

In order to execute a single kernel the following data is required:

```
data AccKernel a where
  AccKernel :: String → CUDA.Fun → CUDA.Module
    → CUDA.Occupancy → Int → Int
    → (Int → Int)
    → AccKernel a
```

— (1) *compiled code*  
 — (2) *thread occupancy information*  
 — (3) *launch configuration*

1. The name of the `__global__` kernel that the function object (second parameter) implements. The compiled object code is linked into the CUDA context as part of the binary module (third parameter).
2. Once the kernel is compiled, details such as the number of registers required for each thread can be used to determine the optimal launch configuration of the kernel. In particular, the configuration is chosen in order to maximise *thread occupancy*, which is discussed further in Section 4.6.3. The analysis (fourth parameter) determines the

optimal number of threads per block to use (fifth parameter) together with the number of bytes of shared memory required when initiating the kernel launch (sixth parameter).

3. A function that computes the number of thread blocks to launch the kernel given the size of the computation, which is generally only known at program runtime. How this function is used exactly depends on the kernel being executed, but typically takes as input the size of the input array.

This formulation does not require searching the kernel tables during the execution phase in order to locate binary code: any kernels required to execute a computation are attached directly to the computation they instantiate. This is an important optimisation, which we provide through use of the `run1` operator, which we can now complete:

```
CUDA.run1 :: (Arrays a, Arrays b) ⇒ (Acc a → Acc b) → a → b
CUDA.run1 f = λa → unsafePerformIO (execute a)
  where
    acc      = convertAfun f
    afun     = unsafePerformIO $ evalCUDA (compileAfun acc)
    execute a = evalCUDA (executeAfun1 afun a >>= collect)
```

Here, `compileAfun :: DelayedAfun f → ExecAfun f` instantiates and compiles skeleton code implementing each of the collective array operations in the program (§4.2), returning a new program AST where each node is annotated in the manner described above. The function `executeAfun1 :: ExecAfun (a → b) → a → CIO b` is used to execute this annotated AST. Since `executeAfun1` is used via a new closure, the first phase of execution that compiles the annotated AST `afun` will only be executed once.

Note that there is a subtle tension here: the task of the compilation phase is to produce an annotated AST carrying all of the information required to execute the computation, but that in turn requires the *compiled* kernel modules. So, how can compilation proceed asynchronously? We use a clever trick and take advantage of Haskell's non-strict execution semantics: once the external compilation has been initiated in a separate thread, `unsafePerformIO` is used (in a safe manner) to delay waiting for the external process to complete compilation and link the binary object until it is demanded by the running program.

```
build1 :: DelayedOpenAcc aenv a → CIO (AccKernel a)
build1 acc = do
  ...
  (name, key) ← compile code                                — initiate external compilation
  let (cta, blocks, smem)        = launchConfig acc occ
      (mdl, fun, occ)          = unsafePerformIO $ do          — delay linking until actually required
        m           ← link key
        f           ← CUDA.getFun m name
        o           ← determineOccupancy acc device f
  return (m, f, o)

  return $ AccKernel name fun mdl occ cta smem blocks
```

### 4.6.3 Launch configuration & thread occupancy

In the CUDA execution model, GPU kernels are executed by a multi-dimensional hierarchy of threads. In particular, threads are grouped into thread blocks, which are distributed over the available streaming multiprocessors (SMs). In contrast to a traditional CPU core, a GPU core (SM) does not have features such as branch prediction or speculative execution, features which increase single threaded performance. In exchange, a GPU core is capable of executing hundreds of threads concurrently. In particular, a SM dispatches work in small groups of threads called *warps*, and by executing warps concurrently and exchanging warps when one is paused or stalled, the GPU is able to keep all of the ALUs busy. The ratio of active warps (i.e. warps that could run given a kernel's resource requirements such as number of registers and bytes of shared memory per thread) to the maximum possible number of active warps (i.e. number of in-flight threads that can be tracked by the hardware) is called *occupancy*.

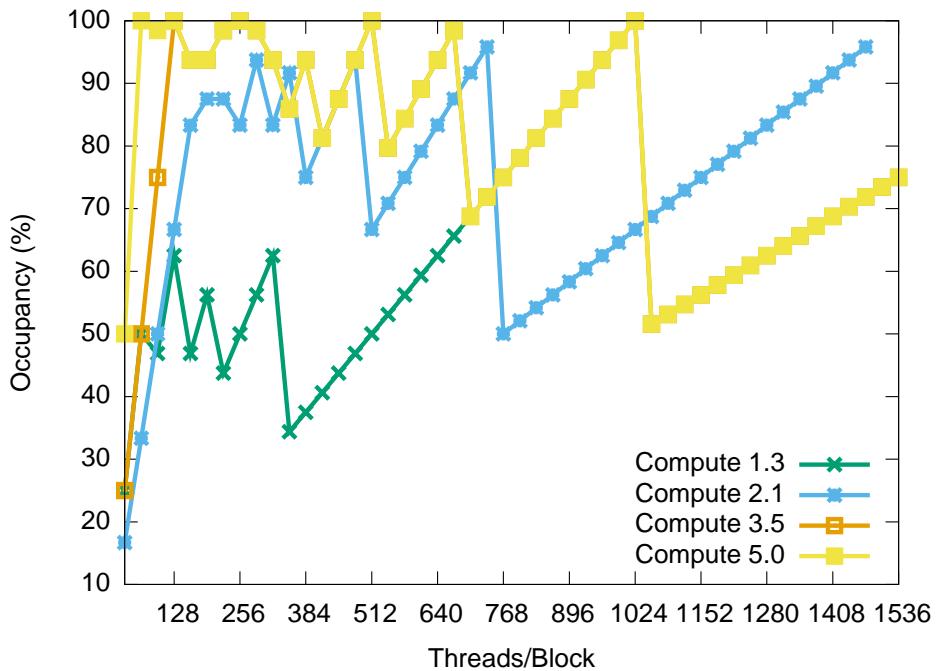
Higher occupancy does not always equate to higher performance — for example, if a kernel is limited by instruction throughput rather than memory bandwidth — but low occupancy always interferes with the ability of the GPU to hide memory and instruction latency by swapping stalled warps, resulting in suboptimal performance. Several factors influence the configuration required for maximum possible occupancy, but once a kernel has been compiled and its resource usage known, we can calculate an ideal launch configuration. The CUDA backend does this for every kernel, to make optimal use of physical resources which vary from GPU to GPU. Additionally, we limit the number of thread blocks per multiprocessor to that which can be physically resident. This requires the generated kernels to be able to process multiple elements per thread, but reduces the overhead of launching thread blocks. Figure 4.1 demonstrates the impact of varying thread block size on thread occupancy for the different hardware generations.

These calculations are a reimplementation of the CUDA occupancy calculator spreadsheet that ships as part of the CUDA toolkit. Accelerate uses this information to launch kernels at either the smallest or largest thread block size that yields maximum occupancy, depending on the operation in question.

### 4.6.4 Kernel execution

To evaluate the array computation on the CUDA backend, we perform a bottom-up traversal of the annotated program (§4.6.2) executing the attached kernels. For each node of the AST, we distinguish three cases:

1. If it is a `Use` node, return a reference to the device memory holding the array data.
2. If it is a non-skeleton node, such as a let binding or shape conversion, the evaluator executes the operation directly by adjusting the environment or similar as required.



**Figure 4.1:** Illustration of the impact of varying the thread block size on the overall multiprocessor occupancy. The example kernel requires 22 registers per thread and 16 bytes of shared memory per thread. Each device generation (compute capability) achieves peak occupancy at different points in the sequence, so selecting a static thread block size for all kernels and devices is clearly suboptimal. Older devices are unable to achieve 100% occupancy due to limitations in the register file size and available shared memory. The first two generations of devices can not execute the kernel with more than 704 and 1472 threads per block respectively.

3. If it is a skeleton node, memory is allocated to hold the result of evaluating the skeleton, and finally invokes the one or more GPU kernels that implement the operation.

In summary, the execution pass interleaves host-side evaluation and the invocation of GPU kernels, while keeping track of device memory, allocating intermediates, and releasing memory once no longer required. See Section 4.5 for details on memory management. GPU kernels execute asynchronously, so the evaluator is able to immediately begin to set up the next stage of the computation while the current kernel executes. The CUDA runtime maintains the sequence of kernels to be executed on the device, known as the execution *stream*.

#### 4.6.5 Kernel execution, concurrently

Beginning with compute capability 2.0, some CUDA devices are able to execute multiple kernels concurrently, rather than always executing kernels one after the other. This can improve overall performance by increasing the number of active threads on the device, particularly when executing many small kernels. Accelerate's collective operations have a purely functional semantics, so concurrent expression evaluation is always sound. The current implementation

executes subtrees of the expression concurrently on the single device. We leave for future work additionally executing programs concurrently on separate GPUs, as is available in some Tesla configurations.

In order to execute kernels concurrently, the kernels must be issued in different non-default streams. A *stream* is a sequence of commands that execute in order; different streams, on the other hand, may execute their commands out of order with respect to one another or concurrently. To allow streams to synchronise, an *event* may be inserted into a stream. Any stream may delay execution until a given event has been posted, which allows efficient cross-stream synchronisation that is managed by the device.

The Accelerate CUDA backend introduces concurrency when evaluating the bound expression at a let binding, and synchronises when looking up a bound variable in the environment. Following conversion and optimisation of the program by the Accelerate frontend, any operation that is to be evaluated will appear as a let-bound expression in the program, and so sequences of let bindings may be able to execute their subtrees concurrently. The array expression evaluator has the type:

```
executeOpenAcc
  :: ExecOpenAcc aenv arrs
  → Aval aenv
  → Stream
  → CIO arrs
  — annotated array program (§4.6.2)
  — array environment with synchronisation points
  — current execution stream
```

The array environment is augmented to carry both the array object as well as an event that execution streams can synchronise against, which signals that the array has been fully evaluated:

```
data Async a = Async Event a

data Aval env where
  Aempty :: Aval ()
  Apush  :: Aval env → Async t → Aval (env, t)
```

The only interesting cases for the evaluator are those of let bindings and environment lookup. When looking up an array in the environment, we ensure that all future work submitted to the current stream will occur after the asynchronous event for the array in the environment has been fulfilled. Synchronisation occurs on the device, so the following does not block waiting for the result:

```
after :: Stream → Async a → CIO a
after stream (Async event arr) = Event.wait event (Just stream) [] >> return arr
```

Evaluating the binding of a let requires evaluating the expression in a new asynchronous execution stream. The body expression is then evaluated with the bound array wrapped in an `Async`, which contains the event indicating when the last kernel issued to the stream has completed.

```

streaming :: Context
  → Reservoir
  → (Stream → CIO a)
  → (Async a → CIO b)
  → CIO b
streaming ctx rsv`(`Reservoir _ weak_rsv) bnd body = do
  stream ← create ctx rsv                                — (1)
  bnd' ← bnd stream                                     — (2)
  end ← Event.waypoint stream
  body' ← body (Async end bnd')
  destroy (weakContext ctx) weak_rsv stream             — (3)
  Event.destroy end                                     — (4)
  return body'

```

1. The function `create` allocates a new execution stream. It uses a technique similar to the `Nursery` (§4.5.4) to reuse execution streams that are currently idle, taking an inactive stream from the `Reservoir` if one is available, or allocating a new execution stream otherwise.
2. The bound expression is executed asynchronously, assigned to the new `Stream`.
3. The body of the expression is evaluated. The extended array environment includes the event `end`, which will be signalled once the result of the binding is available.
4. In contrast to the memory manager, events and streams never remain active beyond the evaluation of the Accelerate program, so the stream and event event associated with evaluating the binding are explicitly deallocated after evaluation of the body completes.

As an example, Listing 4.7 contains an example program that is able to execute the nine `map` operations concurrently on devices of compute capability 2.0 and greater. Note that, to keep the example simple, fusion is disabled to prevent the `map` operations from fusing into the `zip9`. The corresponding execution trace from the NVIDIA Visual Profiler application is shown in Figure 4.2, which demonstrates each of the `map` operations executing in separate streams and overlapping with each other.

### Interaction with Array Fusion

In the example shown in Listing 4.7, it was necessary to disable the array fusion optimisation (§5.2) in order to prevent the kernels being combined into a single kernel, within which each of the fused operations execute sequentially. While a simple example was chosen on purpose in order to clearly demonstrate kernels executing concurrently, the same situation nevertheless arises in real programs. That is, by combining sequences of operations into a single operation, fusion has the effect of removing opportunities for concurrent kernel execution.

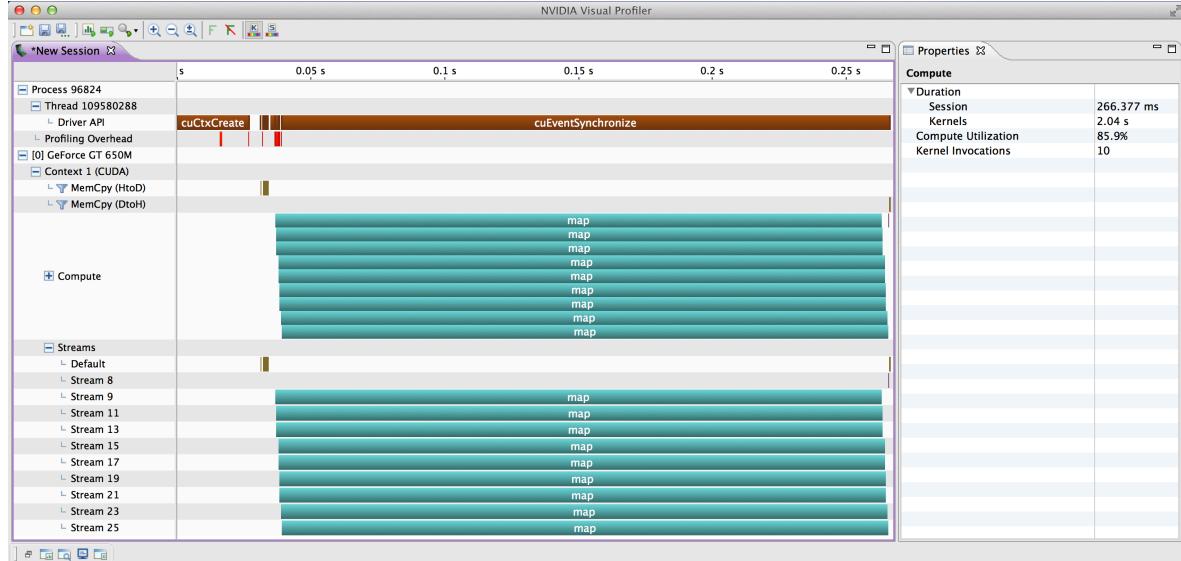
It is left to future work to provide some analysis of when it would be beneficial to avoid fusing operations, on the basis that it would be more efficient to instead execute the two sub-computations concurrently. Such an analysis would need to take into account the hardware

```

loop :: Exp Int → Exp Int
loop ticks = A.while (λi → i <* clockRate * ticks) (+1) 0
  where
    clockRate = 900000

concurrentTest :: Acc (Vector (Int,Int,Int,Int,Int,Int,Int,Int))
concurrentTest
= A.zip9 (A.map loop (use $ fromList Z [1]))
  (A.map loop (use $ fromList Z [1]))
  ...
  
```

**Listing 4.7:** An example program that executes each of the `map` kernels concurrently, on devices of compute capability 2.0 and greater. Note that in order to keep the example simple, the displayed behaviour is only seen when fusion is disabled, to prevent the operations from being combined into a singel kernel.



**Figure 4.2:** Trace from the NVIDIA Visual Profiler application for the program shown in Listing 4.7, where the nine `map` kernels execute concurrently on devices of compute capability 2.0 and greater. Each `map` operation takes 226.7 ms to execute. As shown, an equivalent of 2.04 seconds of GPU time is execute in a wall clock time of 266.4 ms, which includes program initialisation.

that the program is to be executed on, as well as an estimate of how much of the GPU’s resources each kernel requires. This is required in order to determine if the two kernels could be scheduled concurrently.

#### 4.6.6 Conclusion

Runtime system operations are important, as they partially compensate for the overhead of being an embedded language; that is, of needing to traverse the program AST in order to evaluate it. The Accelerate runtime system is designed to execute programs as efficiently as possible, which in turn helps maximise the computational utilisation of the device, and ensure all processors are kept busy.

The runtime system attempts to optimise both individual kernel executions, as well as execution of the entire program. In the former case, each kernel launch is configured to maximise thread occupancy for the particular device the kernel is being executed on, and kernels are scheduled such that independent operations can be executed concurrently, on devices which support this. The single most important optimisation, however, is that each node of the program AST is decorated with all of the data required to execute that operation. For programs which are executed more than once, this means that the costly front-end analysis and code generation steps of program execution can be skipped on the second and subsequent invocations of the program, and kernels can begin executing immediately. This process happens automatically, and almost<sup>13</sup> entirely transparently to the user.

With these optimisations, the Accelerate runtime system is efficient enough that we can execute numerically intensive programs such as fluid flow simulations (§6.8) and real-time image processing applications (§6.7) and achieve performance comparable to hand-written CUDA programs.

## 4.7 Related work

The development of general-purpose applications that run on the GPU is highly work intensive, and has been estimated to require at least  $10\times$  as much effort as developing an efficient single-threaded application [129]. Several researchers have proposed to ameliorate the status quo by either using a high-level library to compose GPU code or by compiling a subset of a high-level language to low-level GPU code. This section explores some of these approaches.

### 4.7.1 Embedded languages

Obsidian [127, 128] and Nikola [81] are also Haskell EDSLs for GPGPU programming, and are in aim and spirit the embeddings closest to Accelerate. Both Nikola and Obsidian produce

---

<sup>13</sup>The user must express their program in the form of a function of one array argument, and use the corresponding `run1` function to execute the program. This is not so onerous, as any operation that the user intends to execute multiple times is likely to be easily written in the form of a function from arrays to arrays.

CUDA code as we do, however while we use algorithmic skeletons (§4.1), Nikola explicitly schedules loops, limiting it to `map`-like operations, and Obsidian is a lower level language where more details of the GPU hardware are exposed to the programmer. Both Nikola and Obsidian lack any system of kernel caching, memory management, or any significant runtime system or compilation optimisations. In both Obsidian and Nikola, algorithms spanning multiple kernels need to be explicitly scheduled by the application programmer and incur additional host-device data transfer overhead, which incurs a severe performance penalty.

Paraiso [96] is a Haskell EDSL for solving systems of partial differential equations (PDEs), such as hydrodynamics equations, and generates both CUDA code for GPUs as well as OpenMP [99] code for multicore CPUs. Accelerate supports `stencil` operations for expressing this class of operation, but lacks the domain specific language features and compiler optimisations implemented by Paraiso for this type of problem.

Baracuda [72] is a Haskell EDSL producing CUDA GPU kernels, though it is intended to be used offline, with the kernels called directly from a C++ application. It supports only `map` and `fold` over vectors.

RapidMind [140], which targets the GPU using OpenCL, and its predecessor Sh [87, 88] which used pixel shaders of the graphics API, are C++ meta programming libraries for data parallel programming. RapidMind was merged into Intel’s Ct compiler to create the Array Building Blocks (ArBB) library. However, the ArBB project was retired before support for GPUs was integrated. GPU++ [62] is an embedded language in C++ using similar techniques to RapidMind and Sh, but provides a more abstract interface than the other C++ based methods listed here.

#### 4.7.2 Parallel libraries

Accelerator [23? ] is a C++ library-based approach with less syntactic sugar for the programmer, but does have support for bindings to functional languages. In contrast to our current work, it already targets multiple architectures, namely GPUs, FPGAs, and multicore CPUs. However, the code generated for GPUs uses the DirectX 9 API, which does not provide access to several modern GPU features, negatively impacting performance and preventing some operations such as scattered writes.

Thrust [59] is a library of algorithms written in CUDA, with an interface similar to the C++ Standard Template Library. Sato and Iwasaki [117] describe a C++ library for GPGPU programming based on algorithmic skeletons. Both of these libraries can be used to generate both CUDA code for the GPU as well as OpenMP code targeting multicore CPUs.

Dandelion [113] is a library of data-parallel operations, specified as comprehensions on arrays using the LINQ framework for .NET (programmers write in either C# or F#). Operations are compiled for the GPU as well as the CPU, and the runtime system distributes the computation over all of the available processors in a heterogeneous cluster.

PyCUDA [71] provide low-level access to the CUDA driver API from within Python, and facilitates runtime code generation for the GPU. A similar approach is taken by CLyther [3] which targets OpenCL. Copperhead [25] uses PyCUDA and Thrust internally to provide a higher level of abstraction to compile, link, cache, and execute CUDA code. Both Parakeet [114] and Anaconda Accelerate [36] are Python libraries that attempt to just-in-time compile a subset of Python code that uses the NumPy [2] array library to instead target the GPU. These approaches can all fail to produce GPU code at program runtime, falling back to interpreted execution.

Jacket [6] is a Matlab extension that allows matrix computations to be offloaded to the GPU, by introducing new datatypes and operations for transferring matrices to GPU memory, and overloading operations on those matrices to trigger execution of suitable GPU kernels.

#### 4.7.3 Other approaches

Delite/LMS [112] is a compiler framework for writing parallel domain specific languages in Scala, and is perhaps Accelerate’s main competitor. Delite/LMS includes several backend code generation targets including C++ and CUDA. Its principle embodiment is OptiML [124], a DSL for machine learning which shows good runtime performance. More recent work attempts to demonstrate that individual DSLs created with the Delite/LMS framework can still be used together efficiently [125]. Like Accelerate, Delite/LMS is a staged language, however unlike Accelerate, Delite/LMS generates and compiles embedded code offline, at Scala compile time, whereas we generate and compile target code at Haskell runtime. This distinction affords us greater opportunity to specialise the generated code, at the expense of needing to amortise these additional runtime overheads. At least in the case of OptiML, the Delite/LMS compiler framework is extremely slow, even for very simple programs, and hard-coded configuration and linking paths make it difficult to compile or use.

Nesl/GPU [17] compiles NESL [20] code to CUDA, a system which was independently developed as CuNesl [142]. Performance suffers because the implementation relies on the legacy NESL compiler, which produces a significant number of intermediate computations.

NOVA [35] is a high-level lisp-like functional language and compiler for parallel operations, targeting both GPUs and CPUs. Compared to Accelerate, NOVA supports fewer parallel operations, but claims to support nested parallelism, recursion, and sum data types. The NOVA compiler has not been released, so these claims can not be tested.

## 4.8 Discussion

This chapter has detailed the implementation of the CUDA generating Accelerate backend targeting parallel execution on NVIDIA GPUs. We discussed our approach to skeleton-based code generation, as well as the caching mechanisms that are implemented in order to reduce

the overheads of the embedding, such as of runtime code generation and compilation. This chapter also dealt with the issue of memory management and data transfer, which is a key consideration to achieving high performance. Finally, we discussed the execution of the generated GPU kernels, including launch configuration and concurrent execution optimisations, in order to maximise device utilisation and help reduce overall program run times.

Now that we can compile and run Accelerate programs on the GPU, the next chapter covers how to optimise Accelerate programs. In particular, we introduce our type-safe approaches to sharing recovery and array fusion, which we identify as the two most pressing performance limitations.

## Optimising embedded array programs

---

---

Relax. As usual, I will bore you with the details.

—CHRIS LEE

The previous chapter discussed the architecture of the Accelerate language embedding and execution on CUDA hardware. Through a set of benchmarks, our previous work [29] identified the two most pressing performance limitations: operator fusion and data sharing. This chapter describes the methods used to overcome these issues, focusing primarily on my novel approach to operator fusion in the context of the stratified Accelerate language and skeleton-based implementation of the CUDA backend. This chapter expands upon the ideas that appeared in [89].

It should be noted that “optimisation” is a misnomer; only rarely does applying optimisations to a program result in object code whose performance is optimal, by any measure. Rather, the goal is to *improve* the performance of the object code generated by a backend, although it is entirely possible that they may decrease it or make no difference at all. As with many interesting problems in [computer] science, in most cases it is formally undecidable whether a particular optimisation improves, or at least does not worsen, performance.

In general, we would like to be as aggressive as possible in improving code, but not at the expense of making it incorrect.

### 5.1 Sharing recovery

Accelerate is a *deeply embedded* language, meaning that evaluating an Accelerate program does not directly issue computations; instead, it builds an *abstract syntax tree* (AST) that represent the embedded computation (§2.5). This AST is later executed by an Accelerate backend to compute the result on a given backend target (§4.6).

A well known problem of defining deeply embedded languages in this way is the issue of *sharing*. The deeply embedded language implementation of Accelerate reifies the abstract

```

riskfree, volatility :: Float
riskfree = 0.02
volatility = 0.30

horner :: Num a => [a] -> a -> a
horner coeff x = x * foldr1 madd coeff
  where
    madd a b = a + x*b

cnd' :: Floating a => a -> a
cnd' d =
  let poly      = horner coeff
      coeff    = [0.31938153, -0.356563782, 1.781477937, -1.821255978, 1.330274429]
      rsqrt2pi = 0.39894228040143267793994605993438
      k        = 1.0 / (1.0 + 0.2316419 * abs d)
  in
    rsqrt2pi * exp (-0.5*d*d) * poly k

blackscholes :: Vector (Float, Float, Float) -> Acc (Vector (Float, Float))
blackscholes = map callput . use
  where
    callput x =
      let (price, strike, years) = unlift x
          r           = constant riskfree
          v           = constant volatility
          v_sqrtT     = v * sqrt years
          d1          = (log (price / strike) + (r + 0.5 * v * v) * years) / v_sqrtT
          d2          = d1 - v_sqrtT
          cnd d      = let c = cnd' d in d >* 0 ? (1.0 - c, c)
          cndD1      = cnd d1
          cndD2      = cnd d2
          x_expRT    = strike * exp (-r * years)
      in
        lift ( price * cndD1 - x_expRT * cndD2           — call price
              , x_expRT * (1.0 - cndD2) - price * (1.0 - cndD1))           — put price

```

**Listing 5.1:** Black-Scholes option pricing

syntax of the deeply embedded language in Haskell. However, a straightforward reification of the surface language program results in each occurrence of a let-bound variable in the source program creating a separate unfolding of the bound expression in the compiled code.

As an example, consider the pricing of European-style options using the Black-Scholes formula, the Accelerate program for which is shown in Listing 5.1. Given a vector with triples of underlying stock price, strike price, and time to maturity (in years), the Black-Scholes formula computes the price of a call and put option. The function `callput` evaluates the Black-Scholes formula for a single triple, and `blackscholes` maps it over a vector of triples, such that all individual applications of the formula are executed in parallel.

The function `callput` introduces a significant amount of sharing: the helper functions `cnd'` and hence also `horner` are used twice — for `d1` and `d2` — and its argument `d` is used multiple times in the body. Worse, the conditional expression leads to a growing number of predicated instructions which incurs a large penalty on the SIMD architecture of a GPU. A lack of sharing recovery was a significant shortcoming of our initial implementation of Accelerate [29].

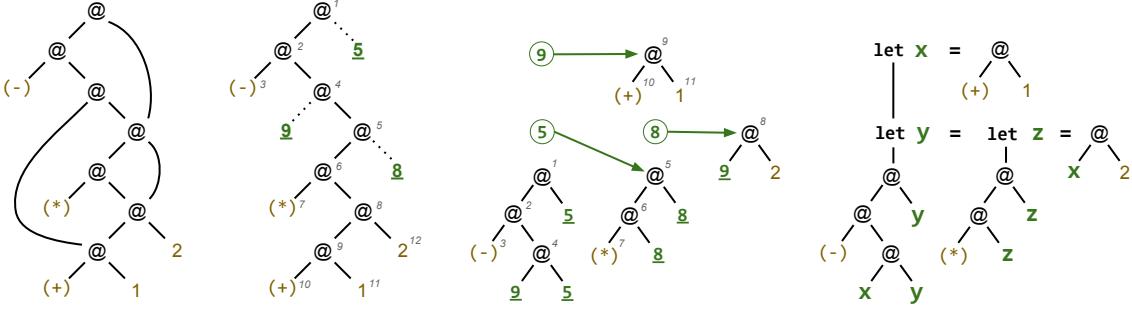


Figure 5.1: Recovering sharing in an example term

### 5.1.1 Our approach to sharing recovery

This problem was solved elegantly by Gill [48] who proposed the use of *stable names* [102] to recover sharing of source terms in a deeply embedded language. Unfortunately, Gill’s original approach (1) reifies the abstract syntax in *graph* form, and (2) assumes an *untyped* syntax representation. We use a variant of Gill’s technique that instead preserves types and produces a tree with minimal flattening, which is able to recover exactly those let bindings which appear in the source program [89].

The higher-order abstract syntax (HOAS) representation of the surface language, while convenient for the human reader, is awkward for program transformations as it complicates looking under lambdas. We convert the source representation to a type-safe internal representation based on nameless de Bruijn indices in the style of Altenkirch and Reus [8], using GADTs [104] and type families [28, 118] to preserve the embedded program’s type information. While developed independently [26, 89], this conversion is similar to the *unembedding* of Atkey et al. [10]. Unembedding and sharing recovery are necessarily intertwined. Sharing recovery must be performed on the source representation, otherwise sharing will have already been lost, but we can not perform sharing recovery on higher-order abstract syntax as we need to traverse below the lambda abstractions. Hence, both operations go hand in hand.

The contribution of the sharing recovery algorithm is work done primarily by my supervisor Manuel Chakravarty, so I present here only a brief overview of the method; see [89] for details. Consider the following source term:

```

let inc = (+1) 1
in let nine = let three = inc 2
    in
        (* three three
in
(-) (inc nine) nine
  
```

The term’s abstract syntax DAG is shown as the left-most diagram in Figure 5.1, which uses @ nodes to represent application. Typed conversion from HOAS with sharing recovery proceeds in three stages:

### Phase 1: Prune shared terms:

Phase one is a depth-first traversal of the AST which annotates each node with its unique stable name, and builds an occurrence map for the number of times each node is seen in the overall program. The stable names of two Haskell terms are equal only when the terms are represented by the same heap structure in memory. Likewise, when the AST of two terms of an embedded language program have the same stable name, we know that they represent the same term. If we encounter a node already in the occurrence map, it represents a previously visited node and is thus a *shared subterm*. We replace the entire subterm at that node with a placeholder containing its stable name. The second diagram in Figure 5.1 shows the outcome of this stage. Each node is labelled by a number that represents its stable name, and the dotted edges indicate where we encountered a previously visited, shared node. The placeholders are indicated by underlined stable names. In this way all but the first occurrence of a shared subterm are pruned, and we do not descend into terms that have been previously encountered. This avoids completely unfolding the embedded expression, so the complexity of sharing observation is proportional to the number of nodes in the tree *with* sharing.<sup>1,2</sup>

As the stable name of an expression is an intensional property, it can only be determined in Haskell’s `IO` monad, and strictly speaking, because of this it is not deterministic. The stable name API does not guarantee completeness: for two stable names `sn1` and `sn2`, if `sn1 == sn2` then the two stable names were created from the same heap object. However, the reverse is not necessarily true; if the two stable names are not equal the objects they come from may still be equal. Put another way, equality on stable names may return a false negative, which means that we fail to discover some sharing, but can never return a false positive since stable names from different heap objects are not considered equal. Luckily, sharing does not affect the denotational meaning of the program, and hence a lack of sharing does not compromise denotational correctness.

### Phase 2: Float shared terms:

The second phase is a bottom-up traversal of the AST that determines the scope for every shared binding that will be introduced. It uses the occurrence map to determine, for every shared subterm, the meet of all the shared subterm occurrences — the lowest AST node at which the binding for the subterm can be placed. This is why the occurrence map generated in the previous phase can not be simplified to a set of occurring names: we need the actual occurrence count to determine where shared subterms should be let-bound.

The third diagram of Figure 5.1 shows the result of this process. Floated subterms are referenced by circled stable names located *above* the node that they floated to. If the node

---

<sup>1</sup>Since computing stable names requires the use of Haskell’s `IO` monad, we can use a hash table to store the occurrence counts and thereby keep the complexity of Phase 1 as  $\mathcal{O}(n)$ .

<sup>2</sup>This is also why we do not use redundancy elimination techniques such as global value numbering or common subexpression elimination, as those methods are applied to the completely unfolded expression.

collects more than one shared subterm, the subterm whose origin is deeper in the original term goes on top; here, 9 on top of 5. Nested sharing leads to subterms floating up inside other floated subterms; here, 8 stays inside the subterm rooted at 5.

### Phase 3: Binder introduction:

Finally, each floated subterm gets let-bound right above the node it floated to, as shown in the rightmost diagram of Figure 5.1. At the same time, the AST is converted into nameless de Bruijn form by introducing de Bruijn indices at the same time as introducing the lets.

## 5.2 Array fusion

Fusion, or deforestation, is a term used to describe techniques for having a compiler automatically eliminate intermediate data structures in a computation by combining successive traversals over these data structures. For example, to compute the sum of squares of all integers from one to a given number in Haskell [100], one could write:

```
sum_of_squares :: Int → Int
sum_of_squares n
  = sum
    $ map (λx → x * x)           — add all numbers in the list
    $ enumFromTo 1 n              — traverse list doubling each element
                                  — generate list of numbers [1..n]
```

While the meaning of the program is clear, it is inefficient, as this code produces two intermediate lists of numbers which each require  $O(n)$  memory to store and data transfers to manipulate. Instead, one could write the program as a single tail-recursive loop as such:

```
sum_of_squares :: Int → Int
sum_of_squares n = go 1 0
  where
    go i acc | i > n      = acc           — return final tally
              | otherwise = go (i+1) (acc + i*i) — add to accumulator and step to next element
```

The second program is much more efficient than the first because it does not involve the production of any intermediate lists and executes in constant space. Unfortunately, the clarity of the original program has been lost. What we *really* want is to write the first program, and have the compiler *automatically* transform it into the second, or something morally equivalent.

This example also demonstrates a subtle behavioural tendency of optimising program transformations: while the second (target) program does not produce any intermediate data structures as desired, we can no longer interpret the program as a sequence of combinators, such as `map` and `sum`. This observation is critical if the combinators represent collective operations expressed as algorithmic skeletons, as they do in Accelerate: while the second program compiles to an efficient *scalar* loop, its *parallel* interpretation has been lost.

### 5.2.1 Considerations

Fusion in a massively data-parallel (embedded) language such as Accelerate requires several uncommon considerations. We discuss related work further in Section 5.4.

#### Parallelism

While fusing parallel collective operations, we must be careful not to lose information essential to parallel execution. For example, `foldr/build` [49] and `stream` [37] fusion are not applicable, because they produce sequential tail-recursive loops rather than massively parallel GPU kernels. Similarly, the `split/join` [67] approach used by data-parallel Haskell (DPH) is not helpful. Although fused operations are split into sequential and parallel subcomputations, the granularity of the parallel operations is rather coarse and the sequential component again consists of tail-recursive loops, both of which are ill suited for a massively parallel target such as GPUs. Accelerate compiles massively parallel array combinators to CUDA code via template skeleton instantiation (§4.1), so any fusion system must preserve the combinator representation of the intermediate code.

#### Sharing

Shortcut fusion transforms rely on inlining to move producer and consumer expressions next to each other, which allows adjacent constructor/destructor pairs to be detected and eliminated. When let-bound variables are used multiple times in the body of an expression, unrestrained inlining can lead to duplication of work. Compilers such as GHC handle this situation by inlining the definitions of let-bound variables that have a single use site, or by relying on some heuristic about the size of the resulting code to decide what to inline [101]. In typical Accelerate programs, each array is used at least twice: once to access the shape information and once to access the array data, so we must handle at least this case specially.

#### Fusion at runtime

As the Accelerate language is embedded in Haskell, compilation of the Accelerate program happens at Haskell *runtime* rather than when compiling the Haskell program (§4.4). For this reason, optimisations applied to an Accelerate program contribute to its overall runtime, so we must be mindful of the cost of analysis and code transformations. On the flip-side, we are able to make use of information that is only available at runtime, such as constant values, and code generation (§4.1) selects instructions based on the capabilities of the target hardware.

#### Filtering

General array fusion transformations must deal with filter-like operations, for which the size of the result structure depends on the *values* of the input array, as well as its size. For

example stream fusion includes the `Skip` constructor in order to support filtering operations (among other uses). Although easily implementable as a combination of the core primitives (Listing 2.2), filtering is difficult to implement as a single-step parallel operation. Since filter-like operations are not part of the core language, we do not need to consider them further.

### Fusion on typed de Bruijn indices

We fuse Accelerate programs by rewriting typed de Bruijn terms in a type preserving manner. Maintaining type information adds complexity to the definitions and rules, but amounts to a partial proof of correctness checked by the type checker (§3.1.7).

#### 5.2.2 The Main Idea

All collective operations in Accelerate are array-to-array transformations. Reductions, such as `fold`, which reduce an array to a single element, yield a singleton array rather than a scalar expression. We partition array operations into two categories:

1. Operations where each element of the output array can be computed independently of all others. We refer to these operations as *producers*.
2. Operations where each array element can not be computed independently, or requires traversal over multiple array elements rather than access to a single element. We call these operations *consumers*, in spite of the fact that, as with all collective operations in Accelerate, they also produce an array as output.

Listing 5.2 summarises the collective array operations that we support. In a parallel context, producers are much more pleasant to deal with because independent element-wise operations have an obvious mapping to massively parallel processors such as GPUs. Consumers are a different story, as we need to know exactly how the elements depend on each other in order to implement them efficiently in parallel. For example, a reduction (with associative operator) can be implemented efficiently in parallel as a recursive tree reduction, but a parallel scan requires two distinct phases (§2.4). Unfortunately, this is the sort of information that is obfuscated by most fusion techniques. To support the different properties of producers and consumers, the fusion transform is split into two distinct phases:

1. *Producer/producer*: A bottom-up contraction of the AST that fuses sequences of producers into a single producer. This is implemented as a source-to-source transformation on the AST.
2. *Producer/consumer*: A top-down transformation that annotates the AST as to which nodes should be computed to manifest data and which should be embedded into their consumers. This process is ultimately completed during code generation, where we

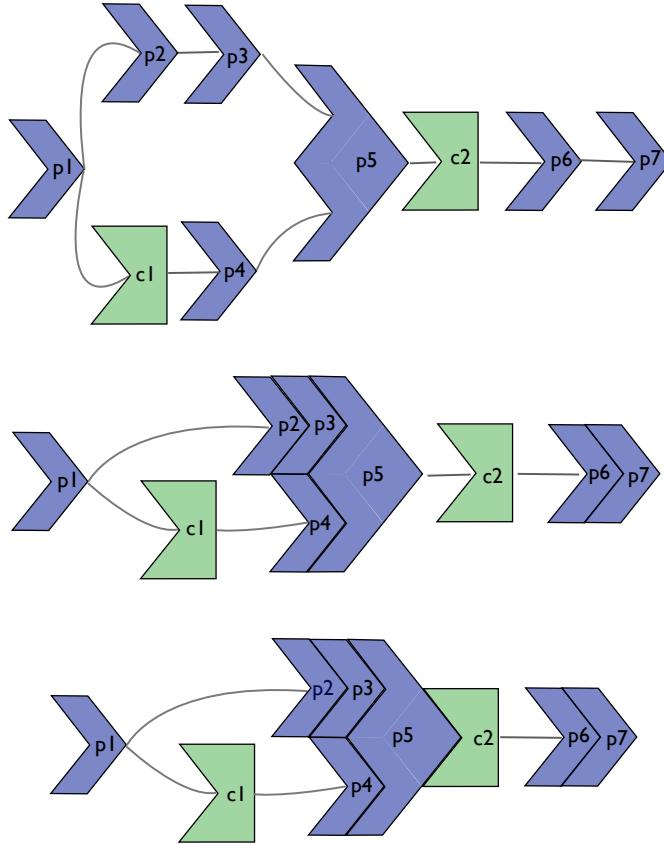
Producers		
map	:: (Exp a → Exp b) → Acc (Array sh a) → Acc (Array sh b)	map a function over an array
zipWith	:: (Exp a → Exp b → Exp c) → Acc (Array sh a) → Acc (Array sh b) → Acc (Array sh c)	apply funciton to a... ...pair of arrays
backpermute	:: Exp sh' → (Exp sh' → Exp sh) → Acc (Array sh a) → Acc (Array sh' e)	backwards permutation
replicate	:: Slice slix ⇒ Exp slix → Acc (Array (SliceShape slix) e) → Acc (Array (FullShape slix) e)	extend array across... ...new dimensions
slice	:: Slice slix ⇒ Acc (Array (FullShape slix) e) → Exp slix → Acc (Array (SliceShape slix) e)	remove existing dimensions
reshape	:: Exp sh' → Acc (Array sh e) → Acc (Array sh' e)	reshape an array
generate	:: Exp sh → (Exp sh → Exp a) → Acc (Array sh a)	array from index mapping
Consumers		
fold	:: (Exp a → Exp a → Exp a) → Exp a	tree reduction along...
scan{l,r}	→ Acc (Array (sh:.Int) a) → Acc (Array sh a) :: (Exp a → Exp a → Exp a) → Exp a → Acc (Vector a) → Acc (Vector a)	...innermost dimension left-to-right or right-to-left ...vector pre-scan
permute	:: (Exp a → Exp a → Exp a) → Acc (Array sh' a) → (Exp sh → Exp sh') → Acc (Array sh a) → Acc (Array sh' a)	forward permutation
stencil	:: Stencil sh a stencil ⇒ (stencil → Exp b) → Boundary a → Acc (Array sh a) → Acc (Array sh b)	map a function with local... ...neighbourhood context

**Listing 5.2:** Summary of Accelerate’s core collective array operations, classified as either producer or consumer operations. We omit the `Shape` and `Elt` class constraints for brevity. In addition, there are other flavours of folds and scans as well as segmented versions of these.

specialise the consumer skeleton by directly embedding the code for producing each element into the skeleton.

Figure 5.2 shows how the fusion technique affects the AST: blue boxes labelled  $p_1$  to  $p_7$  represent producers, where  $p_5$  is a producer like `zipWith` that takes two arrays as input. The consumers are labelled  $c_1$  and  $c_2$ . Each box represents an operation that will be computed to memory, so the goal is to combine the boxes into the smallest number of clusters as possible. In the first phase (centre) successive producer operations are fused together ( $p_2$ ,  $p_3$ ,  $p_4$  and  $p_5$ ). In the second phase (bottom) fused producers are embedded into consumers ( $p_{2,3,4,5}$  into  $c_2$ ). The resulting fused producer/consumer terms are not fused into successive operations. Overall, each box in Figure 5.2 represents a term that will be computed and stored to memory.

Throughout the transformation  $p_1$  is left as is, as its result is used by both  $c_1$  and  $p_2$ . It would be straightforward to change the implementation such that the work of  $p_1$  is duplicated into both  $p_2$  and  $c_1$ . Despite reducing memory traffic, this is not always advantageous, so the current implementation is conservative and never duplicates work. Since Accelerate is a restricted, deeply embedded language, we can compute accurate cost estimates and make an informed decision, but this is left for future work. Moreover, this is an instance of a funda-



**Figure 5.2:** The stages of Accelerate’s fusion algorithm. (1) Before fusion. (2) After producer/producer fusion. (3) After producer/consumer fusion.

mental limitation with shortcut fusion techniques such as `foldr/build` [49] and stream [37] fusion. Since shortcut fusion is a local transformation that depends on inlining, it is applicable only to fusing operations which have a single use site. To implement fusion where an array is efficiently consumed by multiple operations without work duplication would require fundamental changes to the algorithm. Consumer/producer fusion of  $c_1$  into  $p_4$ , or  $c_2$  into  $p_6$ , would also require substantial changes to the fusion implementation, and is only applicable for `map`-like producer terms, as these producers do not require arbitrary array indexing of the source array.

In order to implement the transformation, we require a representation that will facilitate fusing and embedding producer terms into consumers. We do not wish to manipulate source terms directly, as this would require a number of rewrite rules quadratic in the number of operations that are to be fused. This was the problem with Wadler’s original deforestation algorithm [134, 136]. The representation must also be sufficiently expressive to represent all operations we want to fuse, which was a limitation of `foldr/build` fusion [49].

### 5.2.3 Representing Producers

The basic idea behind the representation of producer arrays in Accelerate is well known: simply represent an array by its size and a function mapping array indices to their corresponding values. This method has been used successfully to optimise purely functional array programs in Repa [68], and has also been used by others [32].

The major benefit of the use of delayed arrays in Repa is that it offers by-default automatic fusion. Moreover, this form of fusion does not require sophisticated compiler transformations or for the two fusible terms to be statically juxtaposed — fusion is an inherent property of the data representation. Automatic fusion sounds too good to be true, and indeed it is. There are at least two reasons why it is not always beneficial to represent all array terms uniformly as functions. The first issue is *sharing*. Consider the following:

```
let b = map f a
in  mmMult b b
```

Every access to an element `b` will apply the (arbitrarily expensive) function `f` to the corresponding element of `a`. It follows that these computations will be done *at least twice*, once for each argument of `mmMult`, quite contrary to the programmer's intent. Indeed, if `mmMult` consumes its elements in a non-linear manner, accessing elements more than once, the computation of `f` will be performed every time. Thus it is important to be able to represent some terms as manifest arrays so that a delayed-by-default representation can not lead to an arbitrary loss of sharing. This is a well known problem with Repa [76].

The other consideration is *efficiency*. Since we are targeting a massively parallel architecture designed for performance, it is better to use more specific operations whenever possible. An opaque indexing function is too general, conveying no information about the pattern in which the underlying array is accessed, and hence provides no opportunities for optimisation.

Listing 5.3 shows the ways in which the fusion transformation represents intermediate arrays. The fusion proceeds by recasting producer array computations in terms of a set of scalar functions used to construct an element at each index, and fusing successive producers by combining these scalar functions. The representation<sup>3</sup> is parameterised by the recursive closure of array computations `acc` and the array environment `aenv`, and has three constructors:

1. `Done` injects a manifest array into the type.
2. `Yield` defines a new array in terms of its shape and a function that maps indices to elements.
3. `Step` encodes a special case of `Yield`, that defines a new array by applying an index and value transformation to an argument array.

---

<sup>3</sup>cunc-ta-tion | k'NGk'tāSH'n kəŋk'tel ən | (n) the action or an instance of delaying; tardy action.

```

data Cunction acc aenv a where
  Done :: Arrays arrs
    => Idx           aenv arrs
    => Cunction acc aenv arrs

  Yield :: (Shape sh, Elt e)
    => PreExp      acc aenv sh
    => PreFun      acc aenv (sh → e)
    => Cunction acc aenv (Array sh e)

  Step :: (Shape sh, Shape sh', Elt a, Elt b)
    => PreExp      acc aenv sh'
    => PreFun      acc aenv (sh' → sh)
    => PreFun      acc aenv (a → b)
    => Idx          aenv (Array sh a)
    => Cunction acc aenv (Array sh' b)

```

— (1)  
— manifest array(s)

— (2)  
— output shape  
— compute an element at each index

— (3)  
— output shape  
— backwards permutation into input array  
— function to apply to input value  
— manifest input array

**Listing 5.3:** Representation of fusible producer arrays

```

data Embed acc aenv a where
  Embed :: Extend acc aenv aenv'           — additional bindings to bring into scope, to evaluate the...
    => Cunction acc aenv' a                 — ...representation of (fused) producer terms
    => Embed     acc aenv     a

```

**Listing 5.4:** Representation of fused producer arrays

Note that the argument arrays to **Done** and **Step** are not expressed as terms in **acc**, the type of collective array operations. Instead, requiring a de Bruijn index (**Idx**) into the array environment ensures that the argument array is manifest. Thus the definition of **Cunction** is non-recursive in the type of array computations. This allows the representation to be embedded within producer terms (§5.2.5) with the guarantee that an embedded scalar computation will not invoke further parallel computations (§3.1.4).

In order to bring additional array terms into scope, which may be required for **Done** and **Step**, we combine this representation of producer arrays with the method of Section 3.2.5 for collecting supplementary environment bindings. Listing 5.4 shows the complete data structure used for representing producer terms in Accelerate. Note the types of the array environments. The **Embed** type is expressed in relation to the type **aenv**, which represents all let-bindings currently in scope at this point of the AST. The data constructor contains an existentially typed **aenv'**. This existential is witnessed by the first argument **Extend**, that contains any additional array terms collected during the fusion process (§3.2.5). The delayed array representation in the second argument is expressed in terms of this extended environment, so has access to any terms from the AST (**aenv**) as well as those additional bindings introduced by **Extend** (**aenv'**).

Why do we separate the auxiliary bindings from the representation of fused producers? The constructors of **Cunction** could carry this information, and then we would not require this additional data structure. For example, the **Yield** constructor would then be defined as:

```

Yield' :: (Shape sh, Elt e)
  — Alternative definition (bad)
  ⇒ Extend    acc aenv aenv'
  — NEW: supplementary environment bindings
  → PreExp     acc      aenv' sh
  — CHANGED: inner terms now defined w.r.t. aenv'
  → PreFun     acc      aenv' (sh → e)
  → Cunction  acc aenv      (Array sh e)

```

To create a real AST node from the `Embed` representation, we need both the environment bindings as well as the array description. However, analysis of how to fuse terms requires only the array description. If the additional bindings are bundled as part of the array representation, as in `Yield'`, the existentially quantified environment type is only available after we pattern match on the constructor. This is problematic because to inspect terms and do things like function composition (§3.2.4), the types of the environments of the two terms must be the same. Therefore, after we pattern match on the constructor we must `sink` (§3.2.5) the second term into `aenv'`. Even worse, for terms in the delayed state the only way to do this sinking is to first convert them into a real AST node and then back into the delayed state. If for some reason we can not fuse into this representation — for example we do not meet the requirements for the more restrictive `Step` — then this additional work is wasted.

Why can we not analyse terms with respect to the exposed environment type `aenv`? At some point we will need to merge the extended environments. What would be the type of this operation? We have no way to express this, even if the existential types are exposed.

```
cat :: Extend env env1 → Extend env env2 → Extend env ???
```

Because of the limited scope in which the existential type is available in the `Yield'` style representation, we ultimately perform this process of converting terms and sinking into a suitable type many times. If the extended environments are placed on the constructors of the delayed representations, the complexity of the fusion algorithm for an AST of  $n$  nodes scales as  $\mathcal{O}(r^n)$ , where  $r$  is the number of different rules we have for combining delayed terms. While the semantics of the algorithm are the same, for performance this separation is critical.

Finally, we convert the delayed representations into manifest data using the `compute` function shown in Listing 5.5. It inspects the argument terms of the representation to identify special cases, such as `map` and `backpermute`, as this may allow a backend to emit better code. The helper functions `match` and `isIdentity` check for congruence of expressions. As discussed in Section 3.2.1, we match on `Just REFL` to inject information on the positive case to both the type and value level. For example, `isIdentity` checks whether a function corresponds to the term  $\lambda x.x$ , and in the positive case witnesses that its type must be  $a \rightarrow a$ .

### 5.2.4 Producer/Producer fusion

The first phase of the fusion transformation is to merge sequences of producer functions. Consider the canonical example of map/map fusion, where we would like to convert the operation `map f (map g xs)` into the equivalent but more efficient representation of `map (f . g) xs`,

```

compute :: Arrays arrs => Embed acc aenv arrs -> PreOpenAcc acc aenv arrs
compute (Embed env cc) = bind env (compute' cc)

compute' :: Arrays arrs => Cunction acc aenv arrs -> PreOpenAcc acc aenv arrs
compute' cc = case simplify cc of
  Done v           -> Avar v
  Yield sh f      -> Generate sh f
  Step sh p f v   ->
    | Just REFL <- match sh (arrayShape v)           — a manifest array
    , Just REFL <- isIdentity p                   — array from indexing function
    , Just REFL <- isIdentity f &gt; Avar v        — special cases...
    | Just REFL <- match sh (arrayShape v)
    , Just REFL <- isIdentity p &gt; Map f (avarIn v) — an unmodified manifest array
    | Just REFL <- isIdentity f &gt; Backpermute sh p (avarIn v) — linear indexing OK
    | otherwise           -> Transform sh p f (avarIn v) — pure index transform

```

**Listing 5.5:** Computing the delayed representation to a manifest array

```

mapD :: Elt b
  => PreFun acc aenv (a -> b)
  => Cunction acc aenv (Array sh a)
  => Cunction acc aenv (Array sh b)
mapD f (step -> Just (Step sh ix g v)) = Step sh ix (f `compose` g) v
mapD f (yield -> Yield sh g)           = Yield sh (f `compose` g)

```

**Listing 5.6:** Smart constructor for fusing the `map` operation

which only makes a single traversal over the array. While it would be relatively straightforward to traverse the AST and replace the `map`/`map` sequence with the combined expression, manipulating the source terms directly in this manner requires a number of rewrite rules quadratic in the number of combinators we wish to fuse. This approach does not scale.

Instead, producer/producer fusion is achieved by converting terms into the array representation discussed in Section 5.2.3, and merging sequences of these terms into a single operation. Smart constructors for each producer manage the integration with predecessor terms. For example, Listing 5.6 shows the smart constructor used to generate the fused version of `map`.

The smart constructor `mapD` uses two helper functions, `step` and `yield`, to expose the only two interesting cases of the delayed representation for this operation, and compose (§3.2.4) the input function `f` into the appropriate place in the constructor. Index transformation functions such as `backpermute` proceed in the same manner. The helper functions `step` and `yield` operate by converting the input `Cunction` into the constructor of the same name:

```

step :: Cunction acc aenv (Array sh e) -> Maybe (Cunction acc aenv (Array sh e))
step cc = case cc of
  Yield{}           -> Nothing
  Step{}           -> Just cc
  Done v | ArraysRarray <- accType cc -> Just $ Step (arrayShape v) identity identity v

```

```

yield :: Cunction acc aenv (Array sh e) → Cunction acc aenv (Array sh e)
yield cc = case cc of
  Yield{}                                → cc
  Step sh p f v                          → Yield sh (f `compose` indexArray v `compose` p)
  Done v | ArraysRarray ← accType cc → Yield (arrayShape v) (indexArray v)

```

A producer such as `map` can always be expressed as a function from indices to elements in the style of `Yield`, but this is not the case for the more restrictive `Step`, which returns a `Maybe Cunction`. The latter case is used because we want to keep operations as specific as possible to retain contextual information, rather than always converting into the most general form of an indexing function. For the `Done` case in both instances, GHC can not infer that the result type is a single `Array`, even though this is specified in the type signature, so we must use the `accType` function to reify the `Arrays` class representation.

The only producer case that must be handled specially is that of `zipWith`, as this is the only producer which consumes multiple arrays (see Listing 5.2). In contrast to, for example, `foldr/build`, we are able to fuse the operation in both of the input arguments. The `zipWith` smart constructor is shown in Listing 5.7. We note the following characteristics:

1. Because it is advantageous to express operations in the most specific way possible, `zipWith` considers the special case of when the two input arrays derive from the same manifest source data with the same indexing pattern.
2. The default behaviour is to convert both input arrays into functions from indices to elements, and `combine` these into a single indexing function.
3. The function `combine` completes the task of drawing an element from each of the input arrays and combining them with the binary function `f`, being sure to let-bind the intermediate results along the way.
4. The first guard requires a type signature, otherwise the type variable `e` introduced by the weakening step will “escape”, and can not be unified correctly. This new parameter `e` represents for (1) the array element type, and at (2) the array index type.

The first phase of fusion can now be completed via bottom-up tree contraction of the AST. Smart constructors for each of the producer functions are used to convert terms into the intermediate representation, and adjacent producer/producer terms are merged. Listing 5.8 demonstrates the procedure.

1. Non-computation forms such as control flow and let bindings may also employ smart constructors. We discuss fusion of let bindings in Section 5.2.6.
2. Array introduction forms convert their argument into the internal representation by adding the term as manifest data into the extended environment. This also illustrates

```

zipWithD :: (Shape sh, Elt a, Elt b, Elt c)
    ⇒ PreFun acc aenv (a → b → c)
    → Cunction acc aenv (Array sh a)
    → Cunction acc aenv (Array sh b)
    → Cunction acc aenv (Array sh c)

zipWithD f cc1 cc0
| Just (Step sh1 p1 f1 v1)   ← step cc1
, Just (Step sh0 p0 f0 v0)   ← step cc0
, Just REFL                 ← match v1 v0
, Just REFL                 ← match p1 p0
= Step (sh1 `Intersect` sh0) p0 (combine f f1 f0) v0 — (1)

| Yield sh1 f1              ← yield cc1
, Yield sh0 f0              ← yield cc0
= Yield (sh1 `Intersect` sh0) (combine f f1 f0) — (2)

where
combine :: ∀ acc aenv a b c e. (Elt a, Elt b, Elt c)
    ⇒ PreFun acc aenv (a → b → c)
    → PreFun acc aenv (e → a)
    → PreFun acc aenv (e → b)
    → PreFun acc aenv (e → c)

combine c ixa ixb
| Lam (Lam (Body c')) ← weakenFE SuccIdx c
    :: PreOpenFun acc ((,),e) aenv (a→b→c) — (3)
, Lam (Body ixa')      ← ixa
, Lam (Body ixb')      ← ixb
= Lam $ Body $ Let ixa' $ Let (weakenE SuccIdx ix b') c' — (4)

```

Listing 5.7: Smart constructor for fusing the `zipWith` operation

why we require both `Done` and `Step` constructors in the delayed representation (Listing 5.3). It would be convenient to represent manifest data as a `Step` node with identity transformation functions, but the type of `Step` is restricted to a single `Array`, whereas `Done` is generalised to the `Arrays` class.

3. Producer terms such as `map` and `zipWith` are converted into the internal representation (Listing 5.3) using their smart constructors. In order to do this, all terms must be defined with respect to the same environment type. The helper function `fuse` encodes the bottom-up traversal by first converting the argument array `a` into the `Embed` representation. The extended environment type and delayed representation are then passed to the continuation function `into`, which sinks (§3.2.5) the argument `f` into the extended environment type of the delayed representation, before applying the smart constructor `mapD` (Listing 5.6). This results into a new delayed representation that can be further fused into later terms. At the `zipWith` case `fuse2` acts analogously, but must ensure that the two input array terms are lowered into the same extended environment before applying the smart constructor `zipWithD` (Listing 5.7).
4. Consumer operations such as `fold` force the operation to be evaluated to a manifest array at this point in the program, by converting the `Embed` representation back into AST terms using `compute'` (Listing 5.5) and pushing the result onto the extended

```

embedPreAcc
  :: ∀ acc aenv arrs. Arrays arrs
  ⇒ PreOpenAcc acc aenv arrs
  → Embed acc aenv arrs
embedPreAcc pacc =
  case pacc of
    Alet bnd body      → aletD bnd body
    Use arrs          → done (Use arrs)           — (1)
    ...               — (2)

  — Producers
  Map f a            → fuse (into mapD (cvtF f)) a
  ZipWith f a b     → fuse2 (into zipWithD (cvtF f)) a b
  ...
  — Consumers
  Fold f z a        → embed (into2 Fold (cvtF f) (cvtE z)) a           — (3)
  ...
  where
    done :: Arrays a ⇒ PreOpenAcc acc aenv a → Embed acc aenv a
    done (Avar v) = Embed BaseEnv                                (Done v)
    done pacc      = Embed (BaseEnv `PushEnv` pacc) (Done ZeroIdx)

    into :: Sink f ⇒ (f env' a → b) → f env a → Extend acc env env' → b
    into op a env = op (sink env a)

    fuse :: Arrays as
      ⇒ (∀ aenv'. Extend acc aenv aenv'
          → Cunction acc aenv' as
          → Cunction acc aenv' bs)
      → acc aenv as
      → Embed acc aenv bs
    fuse op (embedAcc → Embed env cc) = Embed env (op env cc)

    embed :: (Arrays as, Arrays bs)
      ⇒ (∀ aenv'. Extend acc aenv aenv'
          → acc aenv' as
          → PreOpenAcc acc aenv' bs)
      → acc aenv as
      → Embed acc aenv bs
    embed op (embedAcc → Embed env cc)
      = Embed (env `PushEnv` op env (injectAcc (compute' cc))) (Done ZeroIdx)

    injectAcc :: PreOpenAcc acc aenv a → acc aenv a
    embedAcc :: Arrays arrs ⇒ acc aenv arrs → Embed acc aenv arrs

```

**Listing 5.8:** Producer fusion via bottom-up contraction of the AST

```

data DelayedOpenAcc aenv a where
  Manifest      :: PreOpenAcc DelayedOpenAcc aenv a
  → DelayedOpenAcc aenv a

  Delayed       :: (Shape sh, Elt e) ⇒
  { extentD    :: PreExp DelayedOpenAcc aenv sh
  , indexD     :: PreFun DelayedOpenAcc aenv (sh → e)
  , linearIndexD :: PreFun DelayedOpenAcc aenv (Int → e)
  }            → DelayedOpenAcc aenv (Array sh e)

```

**Listing 5.9:** Representation of delayed arrays

environment. Consumers fusion will be treated in Section 5.2.5, although note that the producer is placed adjacent to the term that consumes it.

The result of this process is an AST where all adjacent producer/consumer operations have been combined into a single producer. The next step of the fusion pipeline is to fuse producers into consumers.

### 5.2.5 Producer/Consumer fusion

Now that we have a story for producer/consumer fusion, we discuss how to deal with consumers. Recall the classic array fusion example that is vector dot product:

```

dotp :: Acc (Vector Float) → Acc (Vector Float) → Acc (Scalar Float)
dotp xs ys = A.fold (+) 0
              — sum result of...
              $ A.zipWith (*) xs ys
              — ...element-wise multiplying inputs

```

This consists of two collective operations: the first multiplies values from the two input arrays element-wise, and the second sums this result. The fusion system should translate this into a single collective operation, in much the same way as a human would write if working directly in a low-level language such as C or CUDA.

For the dot product example, the delayed producer is equivalent to the scalar function  $\lambda ix \rightarrow (xs!ix) * (ys!ix)$ . This function must be embedded directly into the implementation for `fold` so that the values are generated online, without an intermediate array. However, these two steps happen at different phases of the compilation pipeline: producers are fused using smart constructors, as described in the previous section, but producers are only finally embedded into consumers during code generation (§4.1).

To work around this stage mismatch, the second phase of the fusion transformation annotates the program AST (see Section 3.1.8) with the representation shown in Listing 5.9. This encodes the information as to which nodes should be computed to manifest arrays, and which should be embedded into their consumers. During code generation, the code for the embedded producer terms, such as `indexD`, is generated and integrated directly into the skeleton for the consumer (§4.2). The end result is that no intermediate array needs to be created.

```

convertOpenAcc
  :: Arrays arrs
  => OpenAcc aenv arrs
  -> DelayedOpenAcc aenv arrs
convertOpenAcc = manifest . computeOpenAcc . embedOpenAcc — (1)
where
  manifest :: OpenAcc aenv a → DelayedOpenAcc aenv a
  manifest (OpenAcc pacc) =
    Manifest $ case pacc of
      Use arr          → Use arr
      Alet a b         → alet (manifest a) (manifest b)
      Map f a          → Map (cvtF f) (delayed a)
      Fold f z a       → Fold (cvtF f) (cvtE z) (delayed a)
      Stencil f b a   → Stencil (cvtF f) b (manifest a)
      ...
      ...
      delayed :: (Shape sh, Elt e)
      => OpenAcc      aenv (Array sh e)
      => DelayedOpenAcc aenv (Array sh e)
  delayed (embedOpenAcc fuseAcc → Embed BaseEnv cc) =
    case cc of
      Done v           → Delayed (arrayShape v) (indexArray v) (linearIndex v)
      ...
      ...
      cvtF :: OpenFun env aenv f → DelayedOpenFun env aenv f
      cvtE :: OpenExp env aenv t → DelayedOpenExp env aenv t

```

**Listing 5.10:** Consumer fusion via top-down annotation of the AST

The second phase of fusion is implemented as a top-down translation of the `Embed` representation from the first phase (§5.2.4) into an AST that uses `DelayedOpenAcc` to make the representation of fused producer/consumer pairs explicit. This procedure, shown in Listing 5.10, has the following points of note:

1. The fusion pipeline proceeds in two phases. A bottom-up traversal that merges adjacent producer/producer pairs, which is implemented by the function `embedOpenAcc`, described in Section 5.2.4. The second phase is the top-down traversal which annotates the AST as to which nodes should be computed to manifest data and which should be embedded into their consumers. This phase is implemented by the function `manifest`.
2. For the most part, the function `manifest` simply wraps each node of the AST in the `Manifest` constructor, indicating that this term should be computed to memory.
3. Similarly, let bindings compute both the binding and body to manifest arrays. The smart constructor `alet` flattens needless let bindings of the form `let var = x in var` to `x`. This form is common, as the delayed array constructors always refer to manifest data via de Bruijn indices (see Listing 5.3), meaning that all manifest terms will be let bound regardless of the number of use sites.
4. A consumer such as `fold` specifies that its array valued argument should be `delayed`, so the representation of producer terms is used to annotate the AST node directly. Code

generation will later embed this functional representation directly into the consumer skeleton, thereby completing the producer/consumer fusion process (§4.2.1).

5. In contrast, stencil operations (§2.4.5) — where the computation of each element has access to elements within a local neighbourhood — first fully computes the argument to a manifest array in order to avoid duplicating the work of the delayed representation at each array access. The transformation thus makes it clear for which operations the argument arrays will be delayed, and for which operations the arguments are manifest.
6. The function `delayed` generates the delayed array representation of how to produce each array element that can be directly embedded into consumers. The function takes care to recover optimisation opportunities such as linear indexing of the underlying array data. Note that it is safe to match on `BaseEnv` — indicating no additional terms are being brought into scope — because the first phase always places producers adjacent to the term that will consume it; see the definition of `embed` in Listing 5.8.

Overall, adjacent producer/producer pairs have been merged, and consumers can integrate directly into the skeleton the code for on-the-fly producing each element so that no intermediate array needs to be created.

### 5.2.6 Exploiting all opportunities for fusion

The approach to array fusion employed by Accelerate is similar to that used by Repa [68, 75, 76], although the idea of representing arrays as functions is well known [44, 52]. It was discovered that one of the most immediately pressing problems with delayed arrays in Repa was that it did not preserve sharing. In Repa, the conversion between delayed and manifest arrays is done manually, where by default shared expressions will be recomputed rather than stored. Depending on the algorithm, such recomputation can be beneficial, but excessive recomputation quickly ruins performance.

However, since Accelerate has explicit sharing information encoded in the term tree as let bindings, we can avoid the problem of unrestrained inlining that can be a problem when using Repa. While we want to be careful about fusing shared array computations to avoid duplicating work, *scalar* Accelerate computations that manipulate *array shapes*, as opposed to the bulk array data, can lead to terms that employ sharing but can never duplicate work. Such terms are common in Accelerate code and so it is important that they are treated specially so that they do not inhibit fusion.

#### **let-elimination**

Consider the following example that first reverses a vector with `backpermute`, and then maps a function `f` over the result. Being a function of two producer term (see Listing 5.2) we would hope these are fused into a single operation:

```
reverseMap f xs
= map f
\$ backpermute (shape xs) (ilift1 \$ λi → length xs - i - 1) xs
```

Unfortunately, sharing recovery, using the algorithm from Section 5.1, causes a problem. The variable `xs` is used three times in the arguments to `backpermute`, so sharing recovery introduces a let binding at the lowest common meet point for all uses of the array. This places it between the `map` and `backpermute` operations:

```
reverseMap f xs
= map f
\$ let v = xs
in backpermute (shape v) (λi → length v - i - 1) v
```

The binding, although trivial, prevents fusion of the two producer terms. Moreover, it does so unnecessarily. The argument array is used three times: twice to access shape information, but only once to access the array data — in the final argument to `backpermute`.

Fortunately there is a simple workaround. Since the delayed array constructors representing producer arrays, `Step` and `Yield` (Listing 5.3), carry the shape of the arrays they represent as part of the constructor, we can disregard all uses of the array that only access shape information. For this example, that leaves us with just a single reference to the array's payload. That single reference allows us remove the unnecessary let-binding and re-enable fusion.

Let-elimination can also be used to *introduce* work duplication, which may be beneficial if we can estimate that the cost of the recomputation is less than the cost of completely evaluating the array to memory and subsequently retrieving the values. While Accelerate is currently conservative and never introduces work duplication, since we have the entire term tree available, we could compute an accurate cost model and make an informed decision. Such analysis is left to future work.

### let-floating

Similarly, the semantics of our program do not change if we float let bindings of manifest data across producer chains. This helps to expose further opportunities for producer/producer fusion. For example, we can allow the binding of `xs` to floating above the `map` so that the two producers can be fused:

```
map g \$ let xs = use (Array ... )
      in zipWith f xs xs
```

While floating let bindings opens up the potential for further optimisations, we must be careful to not change the *scope* of let bindings, as that would increase the lifetime of bound variables, and hence increase live memory usage.

Both these tasks, let-floating and let-elimination, are handled by a smart constructor for let bindings during the first fusion phase that contracts the AST and merges adjacent producer/producer pairs (§5.2.4). The smart constructor for let bindings must consider several cases, as well as maintain type correctness and the complexity of the algorithm, by ensuring each term is inspected exactly once. The code is shown in Listing 5.11.

1. If the binding is a manifest array, immediately inline the variable referring to the bound expression into the body, instead of adding the binding to the environment and thereby creating an indirection that must be later eliminated. This effectively floats let-bound terms across producer-producer chains.
2. Note that on entry to the `aletD` smart constructor, the binding term is converted to the internal representation (via the view pattern), but the body expression is left as is. This is important, as the inlining process in (1) can only be applied to *concrete* AST terms, not to the delayed `Embed` representation. Only if the binding is not a manifest term do we convert the body to the delayed representation in order to analyse it. This ensures the body is only traversed once, maintaining complexity of the algorithm. The subsequent cases, where we wish to analyse both the binding and body expressions in delayed form, is handled by the auxiliary function `aletD'`.
3. The first case checks whether it is possible to eliminate the let binding at all, using the continuation `elimAcc`. The implementation of this function is shown in Listing 5.12. Note that the analysis must be applied to the binding `cc1` plus its local environment `env1`, otherwise we can be left with dead terms that are not subsequently eliminated.

If the let binding will not be eliminated, it is added to the extended environment, indicating that it is to be evaluated to a manifest term. Note that it is important to add the term to a *new* environment. Otherwise, nested let bindings are effectively flattened, resulting in terms required for the bound term being lifted out into the same scope as the body. That is, if we don't place the binding into a fresh environment, we get terms such as:

```
let a0 = ( terms for binding ) in
let bnd = ( bound term ) in
( body term )
```

Rather than the following, where the scope of `a0` is restricted to the evaluation of the bound term only:

```
let bnd =
  let a0 = ( terms for binding )
    in ( bound term )
  in ( body term )
```

```

type ElimAcc acc = ∀ aenv s t. acc aenv s → acc (aenv,s) t → Bool

aletD :: (Arrays arrs, Arrays brrs)
    ⇒ ElimAcc acc
    → acc aenv arrs
    → acc (aenv, arrs) brrs
    → Embed acc aenv brrs
aletD elimAcc (embedAcc → Embed env1 cc1) acc0
| Done v1 ← cc1
, Embed env0 cc0 ← embedAcc $ rebuildAcc (subAtop (Avar v1) . sink1 env1) acc0
= Embed (env1 `append` env0) cc0
— (1)

| otherwise
= aletD' elimAcc (Embed env1 cc1) (embedAcc acc0)
— (2)

aletD' :: ∀ acc aenv arrs brrs. (Arrays arrs, Arrays brrs)
    ⇒ ElimAcc acc
    → Embed acc aenv arrs
    → Embed acc (aenv, arrs) brrs
    → Embed acc aenv brrs
aletD' elimAcc (Embed env1 cc1) (Embed env0 cc0)
| acc1 ← compute (Embed env1 cc1)
, False ← elimAcc (inject acc1) acc0
= Embed (BaseEnv `PushEnv` acc1 `append` env0) cc0
— (3)

| acc0' ← sink1 env1 acc0
= case cc1 of
  Step{} → eliminate env1 cc1 acc0'
  Yield{} → eliminate env1 cc1 acc0'
— (4)

where
acc0 :: acc (aenv, arrs) brrs
acc0 = computeAcc (Embed env0 cc0)

eliminate :: ∀ aenv aenv' sh e brrs. (Shape sh, Elt e, Arrays brrs)
    ⇒ Extend acc aenv aenv'
    → Cunction acc aenv' (Array sh e)
    → acc (aenv', Array sh e) brrs
    → Embed acc aenv brrs
eliminate env1 cc1 body
| Done v1 ← cc1 = elim (arrayShape v1) (indexArray v1)
| Step sh1 p1 f1 v1 ← cc1 = elim sh1 (f1 `compose` indexArray v1 `compose` p1)
| Yield sh1 f1 ← cc1 = elim sh1 f1
where
bnd :: PreOpenAcc acc aenv' (Array sh e)
bnd = compute' cc1

elim :: PreExp acc aenv' sh → PreFun acc aenv' (sh → e) → Embed acc aenv brrs
elim sh1 f1
| sh1' ← weakenEA rebuildAcc SuccIdx sh1
, f1' ← weakenFA rebuildAcc SuccIdx f1
, Embed env0' cc0' ← embedAcc
$ rebuildAcc (subAtop bnd)
$ replaceAcc sh1' f1' ZeroIdx body
= Embed (env1 `append` env0') cc0'
— (8)
— (7)
— (6)

replaceAcc :: ∀ aenv sh e a. (Shape sh, Elt e)
    ⇒ PreExp acc aenv sh → PreFun acc aenv (sh → e) → Idx aenv (Array sh e)
    → acc aenv a
    → acc aenv a

subAtop :: Arrays t ⇒ PreOpenAcc acc aenv s → Idx (aenv, s) t → PreOpenAcc acc aenv t
sink1 :: Sink f ⇒ Extend acc env env' → f (env, s) t → f (env', s) t
— (5)

```

Listing 5.11: Smart constructor for let bindings

Removing the nested structure of let bindings increases the scope of bound terms, and hence has the side effect of increasing the maximum memory usage.

4. If the let binding can be eliminated, the delayed representation for the binding is passed to the auxiliary function `eliminate`. This separation is once again important to avoid excessive sinking and delaying of terms, as well as to expose the (existential) type of the extended environment. Pattern matching on `cc1` is necessary to convince the type checker that the type `arrs` is actually restricted to a single `Array`.
5. To eliminate the let binding, all uses of the array variable referring to the bound expression must be replaced with an equivalent scalar expression that instead generates the array shape or data element directly. The appropriate replacements are determined by inspecting the delayed representation of the bound term `cc1`.
6. The first step of eliminating the let binding is to traverse the `body` expression and replace uses of the bound array through the array environment with the provided scalar shape and element generations functions instead. The array environment of the replacement terms `sh1'` and `f1'` was first weakened to open a new array variable, representing the bound array at index zero.
7. The bound term is then substituted directly into all occurrences of the top most index. Since all such occurrences were replaced with scalar expression fragments in the previous step, this effectively just shrinks the array environment in a type-preserving manner.
8. The result is converted into the delayed representation, effectively re-traversing the term. Since eliminating the let binding can open up further opportunities for optimisation, this will enable those optimisations.

In order to determine when let bindings should be eliminated, the smart constructor calls the argument function `elimAcc`. The current implementation is shown in Listing 5.12.

1. Although duplicating work is sometimes beneficial in order to reduce memory traffic in exchange for increased computation, the current version of Accelerate is conservative and never duplicates work. The bound expression is only eliminated when there is a single use of the array data component of the bound array. The function `count` traverses the term and returns the number of uses of the data component at the bound variable, while ignoring uses of the array shape.
2. As a special case, the function looks for the definition of `unzip` applied to manifest array data, which is defined by the Accelerate prelude as a `map` projecting out the appropriate element of the tuple.

It is left to future work to implement a cost model analysis or similar, that can determine when it is beneficial to introduce work duplication, and to identify other special cases.

```

elimOpenAcc :: ElimAcc OpenAcc
elimOpenAcc bnd body
| OpenAcc (Map f a)      ← bnd
, OpenAcc (Avar _)      ← a
, Lam (Body (Prj _ _)) ← f
= True

| otherwise
= count ZeroIdx body ≤ LIMIT
where
LIMIT = 1

count :: Idx aenv s → OpenAcc aenv t → Int
count idx (OpenAcc pacc) = usesOfPreAcc count idx pacc

```

**Listing 5.12:** Determining when a let binding should be eliminated

## 5.3 Simplification

Functional language compilers often perform optimisations. To avoid speculative optimisations that can blow up code size, we only use optimisations guaranteed to make the program smaller: these include dead-variable elimination, constant folding, and a restricted  $\beta$ -reduction rule that only inlines functions which are called once. This leads to a simplification system guaranteed not to lead to code blowup or non-terminating compilation.

### 5.3.1 Shrinking

The *shrinking reduction* arises as a restriction of  $\beta$ -reduction to cases where the bound variable is used zero (dead-code elimination) or one (linear inlining) times. By simplifying terms, the shrinking reduction exposes opportunities for further optimisation such as more aggressive inlining, constant folding and common sub-expression elimination.

The difficulty with implementing the shrinking reduction is that dead-code elimination at one redex can expose further shrinking reductions at completely different portions of the term, so attempts at writing a straightforward compositional algorithm fail. The current implementation uses a naïve algorithm that re-traverses the whole reduct whenever a redex is reduced, although more efficient imperative algorithms exist [9, 16, 69].

The implementation of the `shrink` function is shown in Listing 5.13. The only interesting case is that of a  $\beta$ -redex: when the number of uses is less than or equal to one, the bound term is inlined directly into the body. As Accelerate does not have separate forms for let bindings and lambda abstractions, this serves to implement both inlining and dead code elimination.

### 5.3.2 Common subexpression elimination

*Common subexpression elimination* finds computations that are performed at least twice on a given execution path and eliminates the second and later occurrences, replacing them with

```

usesOf :: Idx env s → Term env t → Int
usesOf idx (Var this)
| Just REFL ← match this idx           = 1
| otherwise                            = 0
usesOf idx (Let bnd body)             = usesOf idx bnd + usesOf (SuccIdx idx) body
usesOf ...

shrink :: Term env t → Term env t
shrink (Let bnd body)
| usesOf ZeroIdx bnd' ≤ 1            = shrink (inline body' bnd')
| otherwise                          = Let bnd' body'
where
  bnd' = shrink bnd
  body' = shrink body
shrink ...

```

**Listing 5.13:** The shrinking reduction

uses of saved values. The current implementation performs a simplified version of common subexpression elimination, where we look for expressions of the form:

```

⟨ common subexpression elimination ⟩
let x = e1
in [x/e1]e2

```

and replace all occurrences of `e1` in `e2` with `x`. Thus, this implementation can only eliminate subexpressions where the common term is already let-bound. This might seem like a severe limitation, however the focus is to eliminate those bindings introduced during the array fusion optimisation, where we always introduce let-bindings for intermediate values when composing terms, rather than eliminating redundancy from the input user program, which is instead addressed via sharing observation (§5.1).

While it may seem that common subexpression elimination is always worthwhile, as it reduces the number of arithmetic operations performed, this is not necessarily advantageous. The simplest case in which it may not be desirable is if it causes a register to be occupied for a long time in order to hold the shared expression's value, which hence reduces the number of registers available for other uses. Even worse is if that value has to be spilled to memory because there are insufficient registers available, or if the number of active threads is too low to be able to hide global memory access latency (§4.6.3). Investigation of this tricky target-dependent issue is left for future work.

### 5.3.3 Constant folding

*Constant-expression evaluation*, or *constant folding*, refers to the evaluation at compile time of scalar expressions whose operands are known to be constant. Essentially, constant-expression evaluation involves determining that all the operands in an expression are constant valued and performing the evaluation of the expression at compile time, replacing the expression by this result.

The applicability of the transformation depends on the type of the expression under consideration. For Boolean values, the optimisation is always applicable. For integers, it is almost always applicable, with the exceptions being cases that would produce run-time exceptions if they were executed, such as division by zero and underflow or overflow. For floating-point values, the situation is even more complicated; the compiler's floating point arithmetic must match that of the processor being compiled for, otherwise floating-point operations may produce different results. Furthermore, the IEEE-754 standard specifies many types of exceptions and exceptional values — including infinities, NaNs and denormalised values — all of which should be taken into account.

## Constant Folding

The current implementation performs constant folding of scalar expressions for all primitive functions and all element types. No explicit checks for underflow or overflow are made, nor for invalid or exceptional values. For example, the following rewrite is always applied:

```

⟨ constant folding ⟩
PrimAdd ((elided type info))
  `PrimApp`
    Tuple (NilTup `SnocTup` Const x `SnocTup` Const y)
  ↪
  Const (x+y)

```

The attentive reader will note that it is straightforward to choose a positive non-zero floating-point value `eps` such that `eps + 1.0 > 1.0` is `False`. Issues arising from simplification of floating-point expressions are currently ignored, but this is not so outrageous as it would be for a traditional offline compiler: since Accelerate programs are just-in-time compiled, any such issues still occur only at program runtime. The only distinction is from which phase of program execution the problem manifests, not that it does.

## Constant Propagation

The effectiveness of constant folding can be increased by combining it with other data-flow optimisations, particularly constant propagation. When determining whether the arguments to a primitive function application are constants, we consider let-bound constants and constant-valued tuple components in addition to literal constants.

## Algebraic Simplification

*Algebraic simplifications* use algebraic properties of operators, or particular operator/operand combinations to simplify expressions. The most obvious algebraic simplifications involve combining a binary operator with an operand that is the algebraic identity of that operator, or with an operand that always yields a constant, independent of the other value of the operand. For example, for a term `x` the following are always true:

```

⟨ algebraic simplification ⟩
x + 0 = 0 + x = x - 0 = x
0 - x = -x
x * 1 = 1 * x = x / 1 = x
x * 0 = 0 * x = 0

```

Similarly, there are simplifications that apply to unary operators and combinations of unary and binary operators. Some operations can also be viewed as strength reductions, that is, replacing an operator by one that is faster to compute, such as:

```

⟨ strength reduction ⟩
x ↑ 2 = x * x
2 * x = x + x

```

Likewise, multiplications by small constants can frequently be computed faster by sequences of shifts and adds and/or subtractions. These techniques are often more effective if applied during code generation rather than optimisation, so strength reductions are not currently applied.

### Algebraic Reassociation

*Reassociation* refers to using specific algebraic properties — namely associativity, commutativity and distributivity — to divide an expression into parts that are constant and parts that are variable. This is particularly relevant when only one operand of a binary operator is identified as being constant valued. For example, the expression  $x + 1 + 2$  would only be simplified if the second addition occurred first. For the case of commutative binary operators where only one operand is constant valued, the term is rewritten so that the constant expression is the first operand. The previous term would then be rewritten as:

```

⟨ algebraic reassociation ⟩
x + 1 + 2 ↪ 1 + x + 2
      ↪ 1 + 2 + x
      ↪ 3 + x

```

### Summary

Constant folding and algebraic simplifications are applied to scalar and array terms. The list of algebraic simplifications presented here and currently implemented is not necessarily exhaustive, and adding additional simplifications might be eased through the use of a rewrite rule system expressed in terms of the source language, rather than by direct manipulation of de Bruijn terms. Furthermore, it may be important to assess the validity of constant folding based on the type of the expression, particularly for floating point terms. Both these concerns are left for future work. An example of the current implementation of constant folding is shown in Listing 5.14.

```

f :: Exp Float → Exp Float
f x = let a = lift (constant 30, x)
      b = 9 - fst a / 5
      c = b * b * 4
      d = c >* pi + 10 ? (c - 15, x)
      in x * d * (60 / fst a)
↪
42.0 * x

```

**Listing 5.14:** Example of constant expression evaluation

## 5.4 Related work

The desire to program in a functional style while still achieving the performance level of an imperative language has been around for as long as functional programming itself, receiving plenty of attention in this context [37, 49, 60, 91, 112, 130, 135, 138]. While most prior work focuses on removing intermediate lists, some methods apply to arrays [27, 32, 51, 68]. This section briefly summarises some of the related work in this area.

### 5.4.1 Shortcut fusion

Most practically successful fusion systems are *shortcut fusion* methods that rely on local program transformations, and are implemented as simple but specific rewrite rules combined with general purpose program transformations. The fusion system described in this work and implemented in Accelerate is in this vein.

`foldr/build` fusion [47, 49] implements library operations out of two basic functions: `build` constructs a list while `foldr` consumes it, and a single rewrite rule suffices to remove intermediate structures. However, not all operations can be expressed in terms of these two operations, and the transformation does not handle functions with accumulators or that consume multiple inputs.

Stream fusion [37] addresses the shortcomings of `foldr/build` fusion, while remaining simple and useful in practice. Operations in the library are written in terms of the `Stream` data type, which consists of some abstract state and a stepper function that advances the stream to the next state. A single rule eliminates matching pairs of constructor and destructor.

Shortcut fusion systems typically only function when there is a single use site of the data structure. Work in the intersection of data flow languages and array fusion has yielded systems that do not have this restriction. Tupling transformations [51, 60] can fuse a restricted form of branching data flow, where two results are computed by directly traversing the same input structure. More recent work was able to fuse programs in a restricted data flow language into a sequence of loops, where each loop can each produce multiple arrays or compute multiple values as output [77, 111].

### 5.4.2 Loop fusion in imperative languages

In contrast, *loop fusion* methods used in imperative languages merge multiple loop nests, typically using loop dependence graphs [137] to determine whether fusion is legal and beneficial. When a producer and consumer loop are merged, array contraction [115] can then remove or reduce the size of the intermediate arrays.

Most recent work in imperative languages focuses on the polyhedral model, an algebraic representation of imperative loop nests and the transformations on them, including fusion transformations. Polyhedral systems are able to express all possible distinct loop transformations where the array indices, conditionals, and loop bounds are affine functions of the surrounding loop indices [90, 107]. Recent work extends the polyhedral model to support arbitrary indexing [133], as well as conditional control flow that is predicated on arbitrary, non-affine functions of the loop indices [15]. However, the indices used to write into the destination array must still be computed with affine functions.

These systems require fusion-specific compiler support and more global reasoning than shortcut fusion techniques.

### 5.4.3 Fusion in a parallel context

Repa [68] is a Haskell library for parallel array programming on shared-memory multicore machines. Repa uses a split representation of arrays as either *delayed* or *manifest*, on which the `DelayedOpenAcc` type is based (§5.2). Representing arrays as functions avoids creating unnecessary intermediate structures, rather than relying on a subsequent fusion transformation to remove them. With Repa, the conversion between array representations is done manually, where by default subexpressions are recomputed rather than stored. In Accelerate, the conversion is automatic and conservative, so that shared scalar and array subexpressions are never recomputed.

Obsidian [128] is an EDSL in Haskell for GPGPU programming, where more details of the GPU hardware are exposed to the programmer. Recent versions of Obsidian [32, 127] implement Repa-style delayed *pull arrays* as well as *push arrays*. Whereas a pull array represents a general producer, a push array represents a general consumer. Push arrays allow intermediate programs to be written in continuation passing style (CPS), and help to compile and fuse append-like operations. Baracuda [72] is another Haskell EDSL that produces CUDA kernels but is intended to be used offline. The paper mentions a fusion system that appears to be based on pull arrays, but the mechanism is not discussed in detail.

Delite/LMS [112] is a parallelisation framework for DSLs in Scala that uses library-based multi-pass staging to specify complex optimisations in a modular manner. Delite supports loop fusion using rewrite rules on the graph-based intermediate language. Similar to Accelerate, Delite/LMS express all operations in terms of a small set of combinators (loop generators) and applies fusion rules to that representation. In contrast to our work, users are responsible

for terminating the fusion process to avoid unrestrained inlining and code explosion, whereas in Accelerate this is automatic. OptiML [124], the primary embodiment of a Delite/LMS based DSL, demonstrates good performance in its target domain of machine learning applications.

Nesl/GPU [17] compiles NESL code to CUDA. Nesl/GPU performs `map/map` fusion, but can not fuse any other operations. NOVA [35] is a high-level lisp-like functional language and compiler for nested data-parallel array operations. It has a fusion system based on rewrite rules that combines common patterns such as `map/map` and `map/fold`.

Sato and Iwasaki [117] describe a C++ skeleton-based library for GPGPU programming that additionally includes a source-to-source fusion mechanism based on list homomorphisms. SkeTo [84] is a C++ library that provides parallel skeletons for CPUs. SkeTo's use of C++ templates provides a fusion system similar to delayed arrays, which would be equivalently implemented using CUDA templates. In contrast, the popular Thrust [59] library of C++ templates requires users to fuse operations manually through the construction of iterators.

Paraiso [96] is a Haskell EDSL for solving systems of partial differential equations (PDEs), such as hydrodynamics equations. It features an automated tuning framework that will search for faster implementations of the program, guided by user annotations.

## 5.5 Discussion

This chapter has detailed our approach to optimising programs written in Accelerate, a language of parallel array operations suitable for execution on bulk-parallel SIMD architectures such as GPUs and multicore CPUs. Our previous work identified sharing and array fusion as the key bottlenecks to achieving high performance. This chapter discussed our novel approaches that tackle these problems. While we motivate our approaches in the context of Accelerate, both methods are more widely applicable. In particular, our sharing recovery algorithm applies to any embedded language based on the typed lambda calculus, and our array fusion optimisation applies to any dynamic compiler targeting SIMD hardware.

The set of optimisations that have been described in this chapter can be seen as a set of term rewriting rules. We would like this set of transformations to be:

- *Correct*: The transformed code retains the same semantics as the original code.
- *Improving efficiency*: The transformed code should require less time and/or fewer resources to execute than the original. We address this through a series of benchmarks in chapter 6.

In addition, it would be of significant practical advantage if the transformations were:

- *Confluent*: When more than one transformation is applicable to a given term, applying the transformations in any ordering should yield the same result. This is important

to ensure that opportunities to apply transformations are not lost, or worse code is generated, due to the choice of applying one transformation before another.

- *Terminating:* We reach a point when no transformation is applicable so the simplification process stops. One must be careful that one transformation [sequence] can not generate terms that can be transformed back to the original; that no transformation undoes the work of another.

### 5.5.1 Correctness

Fusion systems such as `foldr/build` [49] and the system presented here, work by eliminating intermediate data structures. Since these transformations are performed automatically by the compiler, we would like to be sure that the left and right hand sides of the rewrite are equivalences. In particular, we are concerned as to whether:

- A safely terminating program can be transformed into a failing one.
- A safely terminating program can be transformed into another safely terminating program that gives a different value as the result.

For `foldr/build` fusion applied to regular Haskell lists, it is possible to distinguish cases where the two sides of the transformation are not equivalent. Breaking semantic equivalence for `foldr/build` requires the use of the special operator `seq`, which evaluates its first argument to head normal form.<sup>4</sup> We avoid this issue because the Accelerate language itself is strict: every term in the language always evaluates to a value. Thus an operator to force evaluation is unnecessary, and we can not distinguish fused and unfused programs in this way.

The Accelerate language does not have an explicit term to represent exceptional values. However, it is possible to compute undefined values, such as division by zero. Consider the following program:

```
xs :: Acc (Vector Float)
xs = map (λx → 131.95 / x)
$ generate (constant (Z:.10))
  (λix → let i = unindex1 ix in i ==* 0 ? (π, 0))
```

— [42, ∞, ∞, ...]  
— [3.14159, 0, 0, ...]

Evaluating this program, the second and subsequent entries in the array evaluate to infinity, as the program attempts to divide by zero. However, if we were to use this term in the following program that extracts only the first value:

```
unit (xs ! constant (Z:.0)) ↪ unit 42
```

---

<sup>4</sup>Recall that Haskell has non-strict evaluation semantics. This means, for example, that it is possible to evaluate the length of the following list, even though the second list element raises an error, because the individual list elements are never demanded: `length [1, error "boom!", 3] ↪ 3`. However, if we were to first apply `seq` to each list element in order to demand its evaluation, then the error will be raised, and our program will fail.

The program is fused into a single operation that no longer performs the division by zero. If we classify invalid arithmetic as a kind of program failure, then the fusion system presented in this chapter has just transformed a failing program into a safely terminating program, because the intermediate states that performed the invalid operations have been eliminated. It is left to future work to provide a formal proof that the converse can not happen.

### 5.5.2 Confluence and termination

The array fusion transformation (§5.2) is designed to be a single-pass algorithm that specifies only a single rewrite to be applied at each node. Thus, the transformation does not need to select from a set of possible rewrites or the order in which to apply them, based on heuristics or otherwise, so the same code is always generated. Since the delayed representation of producers is non-recursive (Listing 5.3), referencing input arrays via environment indices rather than as general terms, a subsequent shrinking pass (§5.3.1) is not required to eliminate or combine the uses of bound variables. Hence, it is not necessary to iterate the fusion and shrinking steps, so that shrinking can expose all opportunities for fusion.

Informally, then, we are confident that the fusion transformation is both confluent and terminating. Furthermore, since we fuse programs by rewriting typed de Bruijn terms in a type preserving manner, we are confident that each individual rewrite is correct, since the type information amounts to a partial proof of correctness checked by the type checker.

### 5.5.3 Error tests eliminated

Collective operations in Accelerate are rank polymorphic. For example, `fold` reduces along the innermost dimension of an array, reducing the rank of that array by one. The `reshape` operation allows us to change the shape of an array without altering its contents. This is useful to, for example, reduce an array of arbitrary rank to a single value:

```
foldAll :: (Shape sh, Elt e)
    ⇒ (Exp e → Exp e → Exp e)
    → Exp e
    → Acc (Array sh e)
    → Acc (Scalar e)
foldAll f z a = fold f z $ reshape (index1 (size a)) a
```

In order to reshape arrays efficiently, we require that the size of the source and result arrays are identical. This allows the runtime to simply change the interpretation of the array, without touching the actual data, and is thus a constant-time operation. This precondition must be checked at runtime, and an error generated if the sizes do not match.

However, the `reshape` operation is also a general producer (Listing 5.2), and can thus be fused into other operations. If this happens, the runtime can no longer check the precondition on the host — because the term has been removed from the source program — and since it

is not possible to throw exceptions from massively parallel GPU code, fusion has effectively eliminated the runtime error check.

It is left for future work to add the ability to represent error terms in the embedded language. For `reshape` — currently the only term which includes a runtime test — we minimise the loss of the error test by only fusing the operation into delayed terms, while still emitting the `reshape` term whenever it is applied to manifest data, in order to preserve the error test for this case.

#### 5.5.4 Work duplication

While the fusion transformation is cautious and only fuses producer terms that have a single use site, it is still possible to introduce some amount of work duplication. For example, fusing an operation such as `replicate sh (map f xs)` results in the function `f` being applied a number of times equal to the size of the resulting shape `sh`, rather than the size of the source array `xs`. If `f` is expensive or `sh` is large, even though there is a single use site for the source array `xs`, it might still be beneficial to not fuse these terms.

The second source of work duplication can arise through the use of array indexing. As an example, consider the following function, where each element of the array `xs` is repeated four times sequentially in the result, i.e.  $[a, b, c] \mapsto [a, a, a, a, b, b, b, b, c, c, c, c]$ :

```
quad = generate sh (λ ix → xs ! ilift1 (`div` 4) ix)
  where
    sh = index1 (size xs * 4)
    xs = map f (use (Array ...))
```

Since `xs` has a single use site in `generate`, the function `f` will be fused directly into this location. The fusion transformation has thus just committed to evaluating `f` four times more than was specified in the original program.

For GPUs, which support thousands of active threads, recomputing values to reduce memory bandwidth is usually beneficial, but for a CPU with relatively few cores, this might not be the case. Allowing users to explicitly specify in the source language which terms should always be computed to memory, or conversely fused unconditionally, is left to future work. Adding an analysis that estimates when such work duplication is beneficial is also left to future work.

#### 5.5.5 Ordering and repeated evaluations

There are several possible orderings for the transformation passes that have been discussed in this chapter, and it would be possible to invent examples to show that no one order can be optimal for all programs.

As described above, the array fusion pass is designed so that it only needs to be applied to the source program once, and not iterated with the shrinking step in order to expose

all opportunities for fusion. However, for the case of let-elimination, if the bound term is eliminated by inlining it directly into the body, then the new redex must be re-traversed. For the fusion transformation, let-elimination is the only reason a term might be inspected more than once.

On the other hand, simplification of scalar expressions alternates between the simplification and shrinking steps. Thus, we must decide in which order to apply the two operations. Because the shrinking step changes the structure of the AST by removing let bindings, which might be useful for the simplification analysis,<sup>5</sup> we perform scalar simplifications before shrinking. Additionally, both the simplification and shrinking operations return a boolean indicating whether or not the operation made any changes to the AST. The scalar optimisation pipeline applies one round of simplification, then iterates shrinking and simplification until the expression no longer changes, or the limit of two applications of each operation is reached. Moreover, although shrinking and simplification must be iterated, all of the operations performed only work to reduce the size of the term tree (such as  $\beta$ -reduction and common subexpression elimination), so no operation can undo the work of another to transform the term back into the original.

---

<sup>5</sup>In particular, before the introduction of an explicit value recursion term into the source language, uses encoded iteration by defining a single step of the loop body as a collective operation, and used Haskell to repeatedly apply it, effectively unrolling the loop a fixed number of times. Fusion would then combine these  $n$  collective operations into a single operation containing  $n$  instances of the loop body. However, it was found that the CUDA compiler was not well suited to compiling the manually unrolled loop. The loop recovery simplification (§6.4.2) was thus designed to convert the unrolled loop body back into an explicit iteration form, and in part this relied on the structure of let terms, which could be obfuscated by shrinking.

# 6

---

## Results

---

Logic merely enables one to be wrong with authority.

—THE DOCTOR

The previous chapters discussed the design, implementation and optimisation of the Accelerate language and CUDA backend.<sup>1</sup> This chapter analyses the performance of the implementation. Benchmarks were conducted on a single Tesla T10 processor (compute capability 1.3, 30 multiprocessors = 240 cores at 1.3GHz, 4GB RAM) backed by two quad core Xeon E5405 CPUs (64-bit, 2GHz, 8GB RAM) running GNU/Linux (Ubuntu 12.04 LTS). Measurements are the average of 100 runs. Programs are compiled with ghc-7.8.2,<sup>2</sup> CUDA 6.0, gcc-4.6.3 and llvm-3.4. Parallel CPU programs are run on all eight cores.

Haskell programs are compiled with the following options, which is the recommended set to use when compiling Repa programs:

```
-Wall -threaded -rtsopts -fllvm -optlo-O3 -Odpf -fno-liberate-case  
-funfolding-use-threshold100 -funfolding-keeness-factor100
```

CUDA programs are compiled with the options:

```
-O3 -m64 -arch=sm_13
```

A summary of the benchmark results is shown in Table 6.1.

### 6.1 Runtime overheads

Runtime program optimisation, code generation, kernel loading, data transfer, and so on can contribute significant overhead to short lived GPU computations. Accelerate mitigates these overheads via caching and memoisation. For example, the first time a particular expression is executed it is compiled to CUDA code, which is reused for subsequent invocations. This

<sup>1</sup>Version 0.15 release candidate: [de156af5f6d839bc6e0597d33a7a151f0e492a66](https://gitlab.com/accelerate-project/accelerate/-/blob/de156af5f6d839bc6e0597d33a7a151f0e492a66)

<sup>2</sup>Including fix for stable name bug #9078: <https://ghc.haskell.org/trac/ghc/ticket/9078>

Benchmark	Input size	Contender (ms)	Accelerate (ms)	Accelerate no optim. (ms)
Dot product	20M	1.88 (CUBLAS)	2.34 (124%)	4.31 (230%)
Black-Scholes	20M	6.69 (CUDA)	6.19 (92.5%)	115.6 (1727%)
Mandelbrot	2M	4.99 (CUDA)	7.61 (153%)	430.8 (8632%)
N-body	32k	54.4 (CUDA)	102.9 (189%)	(out of memory)
SMVM (protein)	2M	1.05 (CUSP)	0.58 (54.8%)	1.13 (107%)
Canny	16M	46.0 (OpenCV)	100 (217%)	105 (227%)
Fluid flow	2M	974.4 (Repa)	208.5 (21.4%)	220.9 (22.7%)
Radix sort	2M	404 (Nikola)	221.5 (54.8%)	283.3 (70.1%)
Floyd-Warshall	1k	7766 (Repa)	836 (10.8%)	840 (10.8%)
MD5	1M	1.29 (Hashcat)	3.80 (294%)	(error)
K-Means	100k	24.7 (MonadPar)	1.01 (4.1%)	0.77 (3.1%)
Ray tracer	16M	2027 (Repa)	2174 (107%)	(error)
LULESH	91k	1.11 (CUDA)	2.09 (189%)	(error)

**Table 6.1:** Summary of the performance of the benchmark programs, before and after optimisations. Note that array-level sharing recovery is always enabled, otherwise most programs simply do not run in a reasonable amount of time. The MD5, Ray tracer, and LULESH programs also fail to run without scalar sharing recovery.

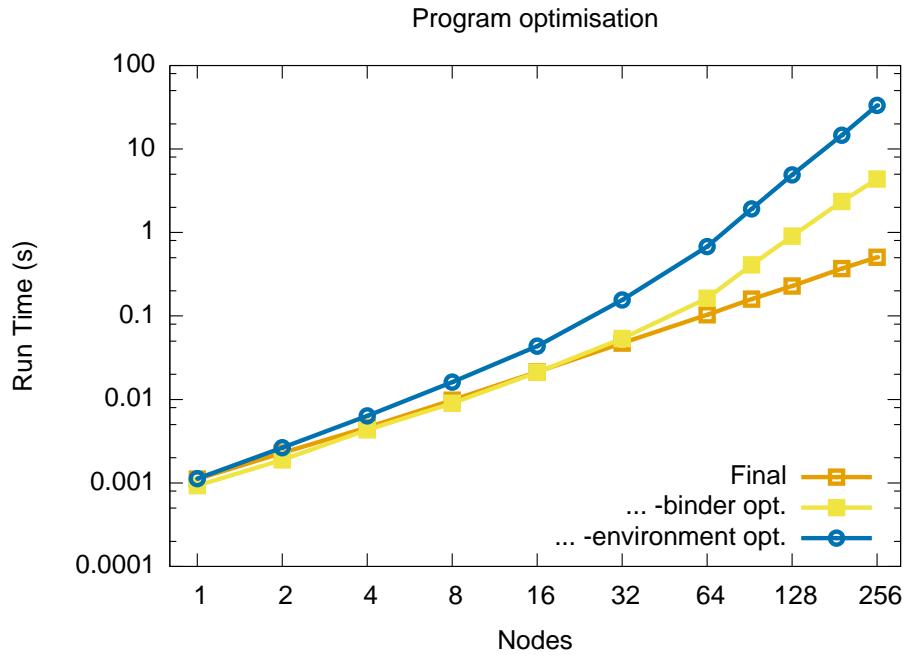
section analyses those overheads, so that they can be factored out later when we discuss the effects of the program optimisations on kernel runtimes of the benchmark programs.

### 6.1.1 Program optimisation

The Floyd-Warshall algorithm for finding the shortest paths in a directed graph, discussed in Section 6.10, was found to cause serious problems with the original implementation of the optimisations discussed in this thesis. Thus, changes were made to ensure that the complexity of the optimisations is not exponential in the number of terms in the program. In particular, the reified program consists of a sequence of let bindings to producer operations, so the handling of let bindings (§5.2.6) and the representation of producers that separates the delayed array representation from the captured environment terms (§5.2.3) was found to be crucial. Figure 6.1 shows the time to optimise the Floyd-Warshall program with and without these changes of the optimisation pipeline.

### 6.1.2 Memory management & data transfer

Figure 6.2 shows the time to transfer data to and from the device. Observe that the data transfer time can account for a significant portion of the overall execution time for some benchmarks.



**Figure 6.1:** Time to optimise the Floyd-Warshall program, comparing the final version of the fusion algorithm to earlier implementations which exhibit exponential scaling in the number of AST terms. Note the log-log scale.

```
dotp :: Acc (Vector Float) → Acc (Vector Float) → Acc (Scalar Float)
dotp xs ys = A.fold (+) 0
              —— sum result of...
              $ A.zipWith (*) xs ys
              —— ...element-wise multiplying inputs
```

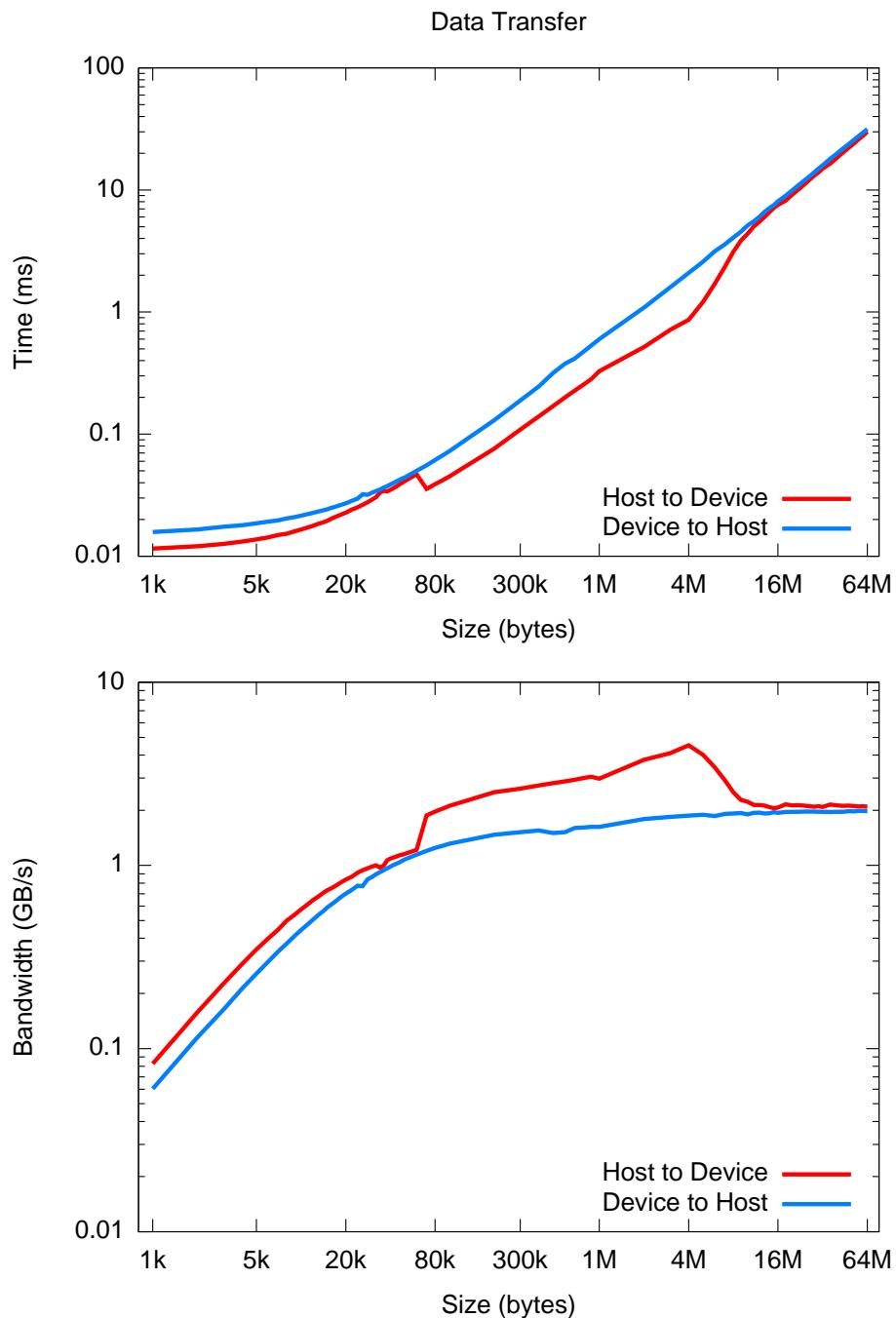
**Listing 6.1:** Vector dot-product

### 6.1.3 Code generation & compilation

To demonstrate the effectiveness of our persistent kernel caching techniques (§4.4.2), Table 6.2 lists the code generation and compilation times for each of the benchmark programs considered in this chapter. Compilation times are measured by compiling the generated code directly with the `nvcc` command line tool. Times are the average of 10 evaluations. In contrast, loading the pre-compiled kernels from the on-disk cache typically takes less than one millisecond.

## 6.2 Dot product

Dot product, or scalar product, is an algebraic operation that takes two equal-length sequences of numbers and returns a single number by computing the sum of the products of the corresponding entries in the two sequences. Dot product can be defined in Accelerate using the code shown in Listing 6.1, and seen previously.



**Figure 6.2:** Data transfer time and bandwidth measurements

Benchmark	No. kernels	Generation (ms)	Compilation (s)
Dot Product	2	1.03	1.97 (3.87)
Black-Scholes	1	0.59	1.95 (1.95)
Mandelbrot	1	0.65	1.94 (1.94)
N-Body	2	1.47	1.99 (3.88)
SMVM	4	4.84	2.01 (7.82)
Canny	11	5.91	4.91 (21.4)
Fluid flow	11	10.47	5.07 (21.6)
Radix sort	8	5.21	2.49 (15.6)
Floyd-Warshall	2	0.51	1.96 (3.85)
Hashcat	1	11.60	2.31 (2.31)
K-Means	5	7.33	2.18 (10.0)
Ray	1	40.11	3.73 (3.73)
LULESH	28	118.3	12.6 (71.4)

**Table 6.2:** Code generation and compilation times for the benchmark programs. Kernels are compiled concurrently, one per core, as is done by the Accelerate runtime system. The equivalent sequential compilation time is shown in parenthesis.

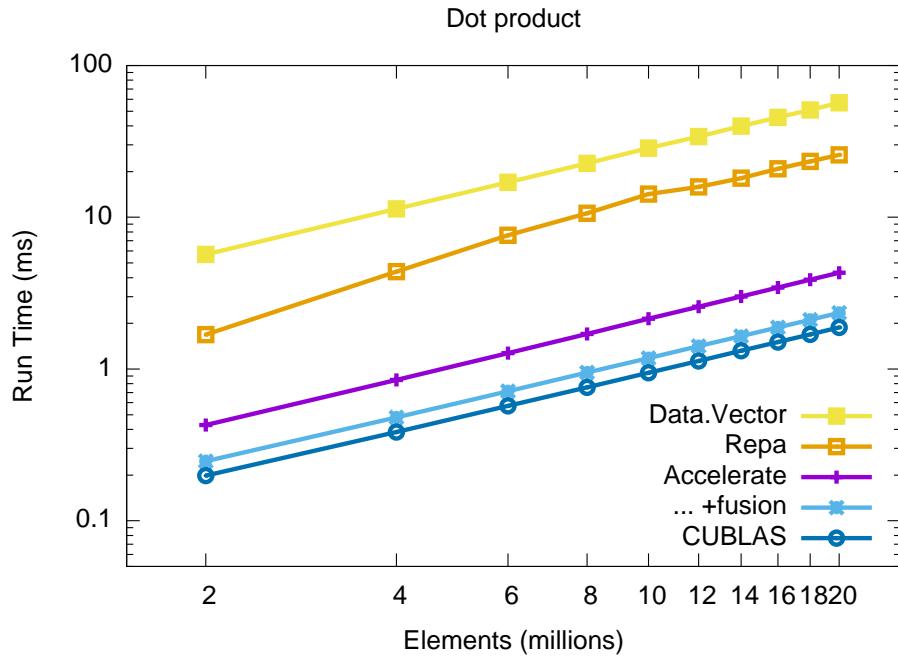
Figure 6.3 show the result of running this code, compared to several other sequential and parallel implementations. The `Data.Vector` baseline is sequential code produced by stream fusion [37], running on the host CPU. The `Repa` version runs in parallel on all eight cores of the host CPU, using the fusion method of delayed arrays [68].

Without optimisations the Accelerate version executes in approximately twice the time of the CUBLAS version. Since the individual aggregate operations consist of only a single arithmetic operation each, the problem can not be a lack of sharing.

### 6.2.1 Too many kernels

The slow-down in the unoptimised version is due to Accelerate generating one GPU kernel function for each aggregate operation in the source program. The use of two separate kernels requires the intermediate array produced by `zipWith` to be constructed in GPU memory, which is then immediately read back by `fold`. In contrast, the CUBLAS version combines these operations into a single kernel. As this is a simple memory bound benchmark, we would expect the lack of fusion to double the runtime, which is what we observe.

The fused version of dot product combines the aggregate computations by embedding a function from array indices to array values of type  $(\text{sh} \rightarrow \text{a})$  into the reduction, represented here as the second argument to the constructor `Delayed`. This scalar function does the work of element wise multiplying the two input vectors, and is performed on-the-fly as part of the reduction kernel instead of requiring an intermediate array. See Section 5.2 for more information. Applied to two manifest arrays, the following embedded program will be executed:



**Figure 6.3:** Kernel runtimes for vector dot product, in Accelerate with and without optimisations, compared to other parallel GPU and CPU implementations. Note the log-log scale.

```

let a0 = use (Array ...) in           — manifest input arrays
let a1 = use (Array ...) in           — extent of the input array
fold (λx0 x1 → x0 + x1) 0.0
  (Delayed (intersect (shape a0) (shape a1))
    (λx0 → (a0!x0) * (a1!x0)))   — function to generate a value at each index

```

While the fused Accelerate version is  $25\times$  faster than the sequential version executed on the CPU, it is still approximately 20% slower than the hand-written CUBLAS version. As dot product is a computationally simple task, any additional overheads in the implementation will be significant. To see why this version is still slightly slower, we continue by analysing the generated code to inspect it for sources of overhead.

### 6.2.2 64-bit arithmetic

The host architecture that the Haskell program executes on is likely to be a 64-bit processor. If this is the case, then manipulating machine word sized types such as `Int` in embedded code must generate instructions to manipulate 64-bit integers in the GPU. On the other hand, CUDA devices are at their core 32-bit processors, and are thus optimised for 32-bit arithmetic. For example, our Tesla GPU with compute capability 1.3 has a throughput of eight 32-bit floating-point add, multiply, or multiply-add operations per clock cycle per multiprocessor, but only a single operation per cycle per multiprocessor of the 64-bit equivalents of these [98].

The delayed producer function that is embedded into the consumer `fold` kernel corresponds to the operation `zipWith (*)`, and has concrete type `(Exp (Z : .Int) → Exp Float)`. Since this is executed on a 64-bit host machine, the indexing operations must generate embedded code for 64-bit integers. See Section 4.2 for more information on code generation, which produces the following CUDA code:

```
const Int64 v1 = ({ assert(ix ≥ 0 && ix < min(shIn1_0, shIn0_0)); ix; });
y0 = arrIn1_0[v1] * arrIn0_0[v1];
```

This is translated by the CUDA compiler into the following PTX assembly code [97], which is the lowest level representation accessible to the developer:

```
// 13      const Int64 v1 = ({ assert(ix >= 0 && ix < min(shIn1_0, shIn0_0)); ix; });
        cvt.s64.s32    %rd5, %r5;
        mov.s64         %rd6, %rd5;
// 15      y0 = arrIn1_0[v1] * arrIn0_0[v1];
        mov.s64         %rd7, %rd6;
        mul.lo.u64     %rd8, %rd7, 4;
        ld.param.u64   %rd9, [__cudaparm_foldAll_arrIn1_0];
        ld.param.u64   %rd10, [__cudaparm_foldAll_arrIn0_0];
        add.u64         %rd11, %rd8, %rd9;
        ld.global.f32  %f1, [%rd11+0];
        add.u64         %rd12, %rd8, %rd10;
        ld.global.f32  %f2, [%rd12+0];
        mul.f32         %f3, %f1, %f2;
```

The first instruction converts the loop variable `ix` — which was not shown but is declared with the C `int` type — to a 64-bit integer using the `cvt.s64.s32` instruction. The `assert` function does not contribute anything in this case as the kernel was not compiled with debugging enabled. Instructions in PTX assembly are appended with the type of their operands, where `s64` and `u64` represent signed and unsigned 64-bit integers respectively, so it is clear that the instructions in this fragment are manipulating 64-bit values.

In this example, 64-bit arithmetic was introduced through the use of a one-dimensional shape index, which has Haskell type `(Z : .Int)` and corresponds to a 64-bit wide integer on our host machine. Working with 32-bit indices on the GPU instead may provide a small performance boost. Further investigation is left for future work.

### 6.2.3 Non-neutral starting elements

In order to support efficient parallel execution, the `fold` function in Accelerate requires the combination function to be an associative operator. However, we do not require the starting value to be a neutral element of the combination function. For example, `fold (+) 42` is valid in Accelerate, even though 42 is not the neutral element of addition.

While convenient for the user, this feature complicates the implementation. In particular, threads can not initialise their local sum with the neutral element during the first sequential reduction phase, and during the second cooperative reduction step must be sure to only

include elements from threads that were properly initialised from the input array. See Section 2.4.1 for more information on the implementation of parallel reductions in CUDA. Both of these restrictions necessitate additional bounds checks, which increases overhead from ancillary instructions that are not loads, stores, or arithmetic for the core computation. As the summation reduction has low arithmetic intensity, which is the limiting factor in the performance of this kernel [54], additional bounds checks further reduce performance.

It is left for future work to have operations such as `fold` and `scan` observe when the combination function and starting element form a monoid, and provide optimised kernels that avoid these extra administrative instructions. This is the approach taken by, for example, Thrust [59].

#### 6.2.4 Kernel specialisation

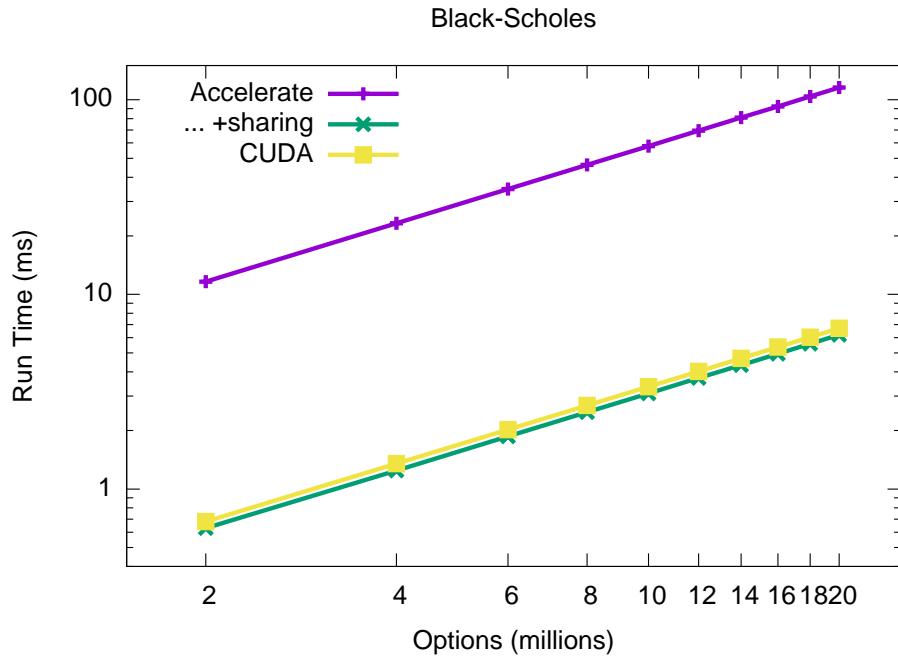
To further reduce instruction overhead, it is possible to completely unroll the reduction by specialising the kernel for a specific block size. In CUDA this can be achieved through the use of C++ templates. Branches referring to the template parameter will be evaluated at compile time, resulting in a very efficient inner loop.

```
template <unsigned int blockSize> __global__ void reduce(...) {
    ...
    if (blockSize > 512) {
    }
    if (blockSize > 256) {
    }
    ...
}
```

This technique has shown to produce significant gains in practice [54], although requires compiling a separate kernel for each thread block size we wish to specialise for. Accelerate can achieve this kind of kernel specialisation because the CUDA code is generated at program runtime. However, since compilation also occurs at program runtime, this adds significant additional runtime overhead. Since compiled kernels are cached and reused, if we know that the reduction will be computed many times, the extra compilation overhead can be amortized by the improved kernel performance, and thus may be worthwhile. See Section 4.4 for more information on the mechanism of compilation and code memoisation. Implementing kernel specialisations, and evaluating their effectiveness, is left to future work.

### 6.3 Black-Scholes option pricing

The Black-Scholes algorithm is a partial differential equation for modelling the evolution of a European-style stock option price under certain assumptions. The corresponding Accelerate program was shown in Listing 5.1. The Black-Scholes formula computes the expected call and put price given the underlying stock price, strike price, and time to maturity of a stock. Many individual applications of the formula can be executed in parallel using the `map` operation.



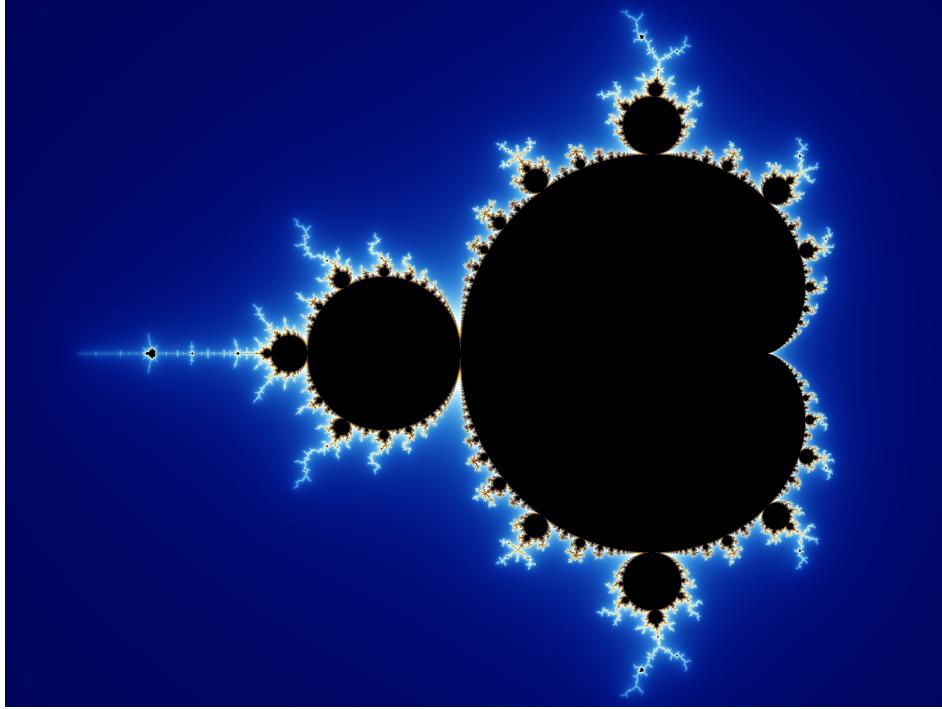
**Figure 6.4:** Kernel runtimes for Black-Scholes options pricing, in Accelerate with and without optimisations, compared to a hand-written CUDA version. Note the log-log scale.

Figure 6.4 shows the result of running this code compared to the implementation that ships with the CUDA SDK. Without optimisations, the Accelerate version is almost twenty times slower than the equivalent implementation in CUDA. As `blackscholes` includes only one collective array operation, the problem can not be a lack of fusion.

### 6.3.1 Too little sharing

The function `callput` from Listing 5.1 includes a significant amount of sharing: the helper functions `cnd'` and hence `horner` are used twice — for `d1` and `d2` — and its argument `d` is used multiple times in the body. Furthermore, the conditional expression `d >* 0 ? (1 - c, c)` results in a branch that, without sharing, results in a growing number of predicated instructions that leads to a large penalty on the SIMD architecture of the GPU.

Without sharing the generated code executes 5156 instructions to calculate 128 options (2 thread blocks of 64 threads each) and results in significant warp divergence which serialises portions of the execution. With sharing recovery only 628 instructions are executed (one thread block of 128 thread) and is actually slightly faster than the reference CUDA version because the latter contains a common subexpression that neither the programmer nor the CUDA compiler spotted. The common subexpression performs a single multiplication. The CUDA version executes 652 instructions (one thread block of 128 thread).



**Figure 6.5:** Image of a Mandelbrot set with a continuously coloured environment. Each pixel corresponds to a point  $c$  in the complex plane, and its colour depends on the number of iterations  $n$  before the relation diverges. Centre coordinate  $(-0.5 + 0i)$ , horizontal width 3.2.

## 6.4 Mandelbrot fractal

The Mandelbrot set is generated by sampling values  $c$  in the complex plane and determining whether under iteration of the complex quadratic polynomial:

$$z_{n+1} = z_n^2 + c$$

that the magnitude of  $z$  (written  $|z_n|$ ) remains bounded however large  $n$  gets. Images of the Mandelbrot set are created such that each pixel corresponds to a point  $c$  in the complex plane, and its colour depends on the number of iterations  $n$  before the relation diverges, where  $z_0 = c$ . The set of points forming the boundary of this relation forms the distinctive and easily recognisable fractal shape shown in Figure 6.5.

Table 6.3 shows the results of the Mandelbrot program. As expected, without fusion the performance of this benchmark is poor because storing each step of the iteration saturates the memory bus, to the point that reducing arithmetic intensity with sharing recovery provides no improvement.

Benchmark	Time (ms)	Bandwidth (GB/s)	Step Speedup	Cumulative Speedup
Accelerate	430.77	0.018		
Accelerate (+sharing)	426.18	0.018	1.01×	1.01×
Accelerate (+fusion)	16.10	0.48	26.5×	26.8×
Accelerate (+loop recovery)	20.17	0.38	0.80×	21.4×
Accelerate (+iteration)	7.61	1.01	2.65×	56.6×
CUDA (limit)	13.97	0.55		
CUDA (avg)	4.99	1.54	2.79×	2.79×
Repa (-N8)	142.87	0.054		

**Table 6.3:** Mandelbrot fractal benchmarks in Accelerate with and without optimisations, compared to a hand written CUDA version. The benchmark computes the Mandelbrot set shown in Figure 6.5 at a resolution of  $1600 \times 1200$ . The CUDA (limit) benchmark computes every pixel to the maximum iteration count.

#### 6.4.1 Fixed unrolling

In order to meet the restrictions of what can be efficiently executed on specialised hardware such as GPUs, Accelerate did not directly support any form of iteration or recursion. To implement the recurrence relation we instead define each step of computing  $z_{n+1}$  given  $z_n$  and  $c$  as a collective operation, and apply the operation a fixed number of times. The trick then is to keep a pair  $(\mathbf{z}, \mathbf{i})$  for every point on the complex plane, where  $\mathbf{i}$  is the iteration at which the point  $\mathbf{z}$  diverged. The code implementing this is shown in Listing 6.2.

1. The function `step` advances the entire complex plane by one iteration of the recurrence relation. This function is repeated `depth` number of times, and the sequence combined by folding together with function application `(\$)`. This effectively unrolls the loop to `depth` iterations.
2. The function `iter` computes the next value in the sequence at a point on the complex plane. If the point has diverged, we return the iteration count at which divergence occurred, otherwise the new value  $\mathbf{z}'$  is returned and the iteration count  $\mathbf{i}$  is increased.
3. The conditional test is performed by the `(?)` operator. Recall that GPUs are designed to do the same thing to lots of different data at the same time, whereas we want to do something different depending on whether or not a particular point has diverged. While we can not avoid having some kind of conditional in the code, we ensure there is only a bounded amount of divergence by having just one conditional per iteration, and a fixed number of iterations.

Array fusion applied to the program in Listing 6.2 results in the `depth` copies of the function `step` being combined into a single kernel. As seen in Table 6.3, while fusion improves the performance of this program, it is still slower than the hand written version.

```

mandelbrot
  :: ∀ a. (Elt a, IsFloating a)
  ⇒ Int
  → Int
  → Int
  → Acc (Scalar (View a))
  → Acc (Array DIM2 Int32)
mandelbrot screenX screenY depth view
= P.snd . A.unzip
$ P.foldr ($) zs0
$ P.take depth (repeat step)
where
  — The view plane
  (xmin,ymin,xmax,ymax)      = unlift (the view)
  sizex                      = xmax - xmin
  sizey                      = ymax - ymin
  viewx                      = constant (P.fromIntegral screenX)
  viewy                      = constant (P.fromIntegral screenY)

step :: Acc (Array DIM2 (Complex a, Int32))
      → Acc (Array DIM2 (Complex a, Int32))
step = A.zipWith iter cs

iter :: Exp (Complex a) → Exp (Complex a, Int32) → Exp (Complex a, Int32)           — (1)
iter c zi = next (A.fst zi) (A.snd zi)
where
  next :: Exp (Complex a) → Exp Int32 → Exp (Complex a, Int32)
  next z i =
    let z' = c + z*z
    in (dot z' >* 4) ? ( zi , lift (z', i+1) )                                     — (2)

dot :: Exp (Complex a) → Exp a
dot c = let r :+ i = unlift c
        in r*r + i*i

— initial conditions for a given pixel in the window, translated to the
— corresponding point in the complex plane
cs = A.generate (constant $ Z :: screenY :: screenX) initial
zs0 = A.map (λc → lift (c, constant 0)) cs

initial :: Exp DIM2 → Exp (Complex a)
initial ix = lift ( (xmin + (x * sizex) / viewx) :+ (ymin + (y * sizey) / viewy) )
where
  pr = unindex2 ix
  x  = A.fromIntegral (A.snd pr :: Exp Int)
  y  = A.fromIntegral (A.fst pr :: Exp Int)

```

— (3)

**Listing 6.2:** Mandelbrot set generator, using fixed unrolling

This slowdown is because the fused code generates a completely unrolled loop, with `depth` copies of the function body. Unrolling loops not only increases instruction counts, but can often increase register lifetimes because of the scheduling of loads and stores. In particular, stores are pushed down and loads moved up in the instruction stream, which results in temporary scalar lifetimes being longer.<sup>3</sup> Since the Mandelbrot program is limited by computation, reducing the number of resident threads has a corresponding reduction on maximum performance.

### 6.4.2 Loop recovery

Loop unrolling or loop unwinding is a loop transformation technique that attempts to optimise a program's execution speed by reducing or eliminating instructions that control the loop, such as branching and the end-of-loop test. Loop unrolling does not always improve performance, however, because it may lead to, for example, increased register usage.

Applying the fusion transformation to the Mandelbrot program shown in Listing 6.2 resulted in a single kernel with `depth` copies of the function `iter`. In essence, the loop computing the Mandelbrot recurrence relation has been completely unrolled. Unfortunately, the fused program does not match the performance of the hand written version, because (a) The program has a increased register usage; and (b) the conditional test is still performed at the end of each copy of the loop body, so there is no reduction in the overall number of instructions and branches executed.

We attempted to reduce these overheads by re-introducing explicit scalar loops into the generated code. Recovering scalar loops then enables a backend to generate explicit `for` loops in the target language, and the compiler then makes the decision of whether or not to unroll the loop. The following pair of rewrite rules are used.

```

< loop introduction >
let x =
  let y = e1
  in e2
in e3
⇒
iterate[2] (λy → e2) e1           if e2 ≡ e3

< loop joining >
let x = iterate[n] (λy → e2) e1
in e3
⇒
iterate[n+1] (λy → e2) e1           if e2 ≡ e3

```

---

<sup>3</sup>Register usage depends on which architecture the code is being compiled for. For the compute 1.3 series processor, each thread required 29 registers, whereas on a newer 3.0 processor the same code required 59 registers per thread. I conjecture that this is because older devices use a different underlying compiler architecture (Open64 vs. LLVM).

```

mandelbrot
  :: ∀ a. (Elt a, IsFloating a)
  ⇒ Int
  → Int
  → Int
  → Acc (Scalar (View a))
  → Acc (Array DIM2 Int32)
mandelbrot screenX screenY depth view =
  generate (constant (Z :: screenY :: screenX))
    (λix → let c = initial ix
      in A.snd $ A.while (λzi → A.snd zi <* LIMIT &&* dot (A.fst zi) <* 4)
        (λzi → lift1 (next c) zi)
        (lift (c, constant 0)))
  where
    LIMIT = P.fromIntegral depth
    next :: Exp (Complex a) → (Exp (Complex a), Exp Int32) → (Exp (Complex a), Exp Int32)
    next c (z, i) = (c + (z * z), i+1)

```

**Listing 6.3:** Mandelbrot set generator, using explicit iteration

As seen in Table 6.3, loop recovery did not result in improved performance in this case, because of the introduction of extra instructions to track the iteration count of the loop.<sup>4</sup> Both the unrolled and loop recovery methods are slower than the hand written program, because every thread always executes until the iteration limit.

#### 6.4.3 Explicit iteration

Although not yet appearing in any published materials, we have added explicit value recursion constructs to the source and target languages. Robert Clifton-Everest implemented the necessary changes to the sharing recovery algorithm, while I implemented the backend changes for code generation and execution. The new function `while` applies the given function, starting with the initial value, as long as the test function continues to evaluate to `True`.

```

while :: Elt e
  ⇒ (Exp e → Exp Bool)           — conditional test
  → (Exp e → Exp e)              — function to iterate
  → Exp e                         — initial value
  → Exp e

```

The Mandelbrot program using explicit iteration is shown in Listing 6.3, where the common parts to the `where` clause in Listing 6.2 have been elided. Note that in contrast to the implementation based on fixed unrolling, the function will stop executing as soon as the relation diverges, rather than always continuing to the iteration limit even if the point has already diverged.

---

<sup>4</sup>As noted in footnote 3, the CUDA compiler produces code with different register usage patterns depending on which architecture is being targeted. For newer devices that exhibited higher register usage when compiling completely unrolled code, loop recovery provided good increases in performance, by reducing register usage and thereby increasing multiprocessor occupancy.

With respect to implementation on SIMD architectures such as CUDA, while both the fixed unrolling and explicit iteration methods have a conditional test that introduces thread divergence, it is important to note that the behaviour of the divergence is different for each method. In the fixed unrolling case, both branches of the conditional must be executed to keep track of whether the element has already diverged. This results in threads following separate execution paths, where threads that follow the `True` branch execute while all other threads sit idle, then execution switches to the `False` branch and the first set of threads sit idle. In contrast, in the explicit iteration case, threads exit the loop as soon as their point on the complex plane diverges, and then simply wait for all other threads in their SIMD group to complete their calculation and catch up.

While the introduction of explicit value recursion required changes to the source language program, as seen in Table 6.3, performance improves significantly. Additionally, if all threads in the SIMD group diverge quickly, this can save a significant amount of redundant computation. Table 6.3 also shows the difference in execution time between computing the entire fractal shown in Figure 6.5, where all points diverge at different depths in the sequence (avg), compared to a region where all threads continue to the iteration limit (limit).

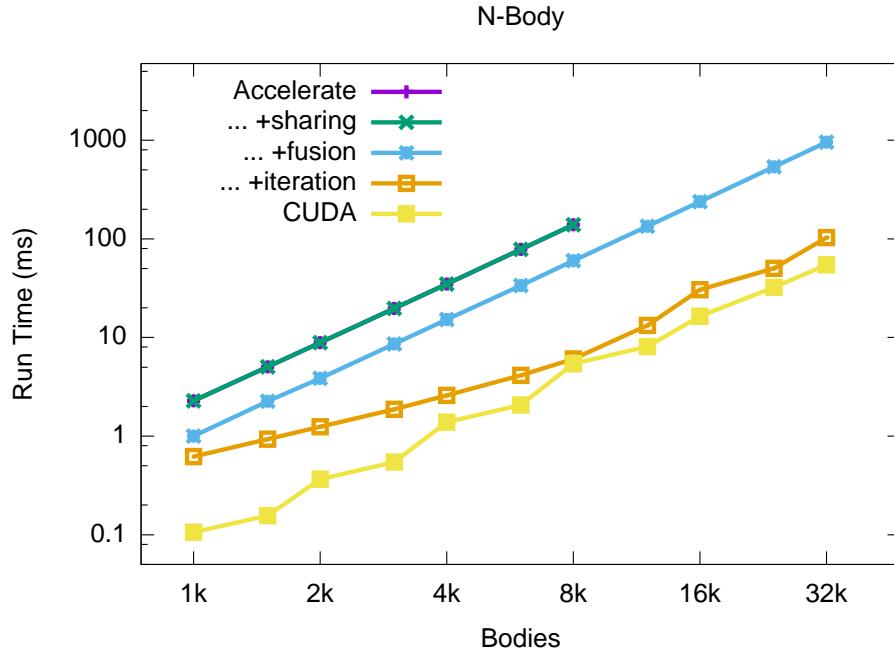
#### 6.4.4 Unbalanced workload

In order to manage the unbalanced workload inherent in the Mandelbrot computation, the hand-written CUDA version includes a custom, software-based thread-block scheduler, an idea which has gained recent popularity under the name of persistent threads [53]. In this vein, future work could investigate adding a software-based load balancing scheduler to the generated kernels, perhaps based on ideas of work stealing [109, 132], instead of statically partitioning the work space. An orthogonal line of work is that of generating kernel code that avoids thread divergence, which is an active area of research [141].

#### 6.4.5 Passing inputs as arrays

Note that in the definition of `mandelbrot`, the current view bounds are passed as a scalar array. Why don't we pass these values as plain `Floats`? When the program runs, Accelerate converts the program into a series of CUDA kernels, each of which must be compiled and loaded onto the GPU (§4.1). Since compilation can take a while, Accelerate remembers the kernels it has seen before and reuses them (§4.4). Our goal with `mandelbrot` is to make a kernel that will be reused, otherwise the overhead of compiling a new kernel each time the view is changed will ruin performance.

By defining the `view` as a scalar array, we ensure that it is defined *outside* the call to `generate`. If we don't do this, a different value for the `view` parameter will be embedded directly into each call to the `generate` function, which will defeat Accelerate's caching of CUDA kernels.



**Figure 6.6:** Kernel runtimes for the  $n$ -body gravitational simulation, in Accelerate with and without optimisations, compared to a hand-written CUDA implementation. Note the log-log scale.

Any scalar values that we wish to be treated as function parameters must be lifted to array variables. This is because the internal representation of array terms is closed with respect to scalar variables, in order to statically exclude nested parallelism. Adding support for nested data parallelism to Accelerate is an active area of research. With it, it may be possible to avoid lifting free scalar variables to free array variables in order to correctly cache kernels.

## 6.5 N-body gravitational simulation

The  $n$ -body example simulates the Newtonian gravitational forces on a set of massive bodies in 3D space, using the naïve  $\mathcal{O}(n^2)$  algorithm. In a data-parallel setting, the natural way to express this algorithm is first to compute the forces between every pair of bodies, before adding the forces applied to each body using a segmented sum. Without fusion this approach also requires  $\mathcal{O}(n^2)$  space for the intermediate array of forces, which exhausts the memory of our device (4GB!) when using more than about five thousand bodies. With fusion, the reduction operation consumes each force value on-the-fly, so that the program only needs  $\mathcal{O}(n)$  space to store the final force values. The core of the  $n$ -body simulation is shown in Listing 6.4, where `accel` calculates the acceleration between two particles, and `(.+. )` component-wise sums the acceleration of a particle along each  $x$ ,  $y$  and  $z$  axis.

```

calcAccels :: Exp Float → Acc (Vector Body) → Acc (Vector Accel)
calcAccels epsilon bodies
= let n      = A.size bodies
   cols    = A.replicate (lift $ z :: n :: All) bodies
   rows    = A.replicate (lift $ z :: All :: n) bodies
in
A.fold (.+.) (vec 0) $ A.zipWith (accel epsilon) rows cols

```

**Listing 6.4:** *N*-body gravitational simulation, using parallel reduction

```

calcAccels :: Exp R → Acc (Vector PointMass) → Acc (Vector Accel)
calcAccels epsilon bodies
= let move body = A.sfoldl (λacc next → acc .+. accel epsilon body next)
   (vec 0)
   (constant z)
   bodies
in
A.map move bodies

```

**Listing 6.5:** *N*-body gravitational simulation, using sequential reduction

### 6.5.1 Sequential iteration

Even with fusion, the reference CUDA version is over  $10\times$  faster. Although the fused program requires only  $\mathcal{O}(n)$  space, it still requires all  $\mathcal{O}(n^2)$  memory accesses, and  $n^2$  threads to cooperatively compute the interactions.

With the addition of sequential iteration, we can emit an alternative implementation. Rather than computing all interactions between each pair of bodies and reducing this array in parallel, we express the program as a parallel map of sequential reductions, as shown in Listing 6.5. Although the program still performs all  $n^2$  memory accesses, since all threads access the `bodies` array in sequence, these values can be cached efficiently by the GPU, and the actual bandwidth requirements to main memory are reduced.

### 6.5.2 Use of shared memory

In CUDA, the memory hierarchy of the device is made explicit to the programmer, including a small on-chip shared memory region threads can use to share data. The shared memory region is essentially a software managed cache [98].

The program shown in Listing 6.4 uses the shared memory region to share computations while computing the parallel reduction.<sup>5</sup> The second implementation shown in Listing 6.5 does not use shared memory at all, but rather relies on hardware caching. Similar to our latter implementation, the CUDA program is also implemented as a parallel map of sequential reductions, but explicitly uses the shared memory region to share the particle mass and position data between threads, rather than rely on hardware caching. Since the shared mem-

---

<sup>5</sup>As part of the second phase of the `fold` skeleton, where threads in a block cooperatively reduce elements to a single value. See §2.4.1 for details.

```

type SparseVector e = Vector (Int, e)           — column index and value
type SparseMatrix e = (Segments Int, SparseVector e) — length of each row

smvm :: (Elt a, IsNum a) => Acc (SparseMatrix a) → Acc (Vector a) → Acc (Vector a)
smvm smat vec
= let (segd, svec)      = unlift smat
  (inds, vals)        = A.unzip svec
  vecVals            = gather inds vec
  products           = A.zipWith (*) vecVals vals
in foldSeg (+) 0 products segd

```

**Listing 6.6:** Sparse-matrix vector multiplication

ory region is very low latency, the CUDA program remains faster. Automatic use of shared memory is a separate consideration to the optimisations discussed in this work, and remains an open research problem [78].

## 6.6 Sparse-matrix vector multiplication

This benchmark considers the multiplication of sparse matrices in compressed row format (CSR) [31] with a dense vector. This matrix format consists of an array of the non-zero elements paired with their column index, together with a segment descriptor recording the number of non-zeros in each row. The corresponding Accelerate code is shown in Listing 6.6. Table 6.4 compares Accelerate to the CUSP library [13, 14], a special purpose library for sparse matrix operations (version 0.4), and the CUSPARSE library which is part of the NVIDIA CUDA distribution. Using a 14 matrix corpus derived from a variety of application domains [139], we compare against using compressed row format matrices. The measured times include only the core sparse matrix multiply operation.

Compared to our previous work [29] the fusion transformation compresses the program into a single segmented reduction. As predicted, the corresponding reduction in memory bandwidth means that Accelerate is on par with the CUSP library for several of the test matrices. In a balanced machine SMVM should be limited by memory throughput, so a dense matrix in sparse format should provide an upper bound on performance because loops are long running and accesses to the source vector are contiguous and have high re-use. We see that Accelerate with array fusion achieves this expected performance limit. For some benchmarks Accelerate is significantly faster than CUSP, while in others it lags behind. It is unknown why the NVIDIA CUSPARSE library is significantly slower on all test matrices.

### 6.6.1 Segment startup

To maximise global memory throughput, the skeleton code ensures that the vector read of each matrix row is coalesced and aligned to the warp boundary. Combined with the behaviour that reduction operations in Accelerate do not require a neutral starting element, this means

spyplot	Description	Dimensions	Nonzeros (nnz/row)	Accelerate	Accelerate no fusion	CUSP	CUSPARSE
	Dense matrix in sparse format	2K × 2K	4.0M (2K)	15.55	5.82	11.8	0.06
	Protein data bank 1HYS	36K × 36K	2.19M (60)	7.59	3.89	4.16	0.03
	FEM/Concentric spheres	83K × 83K	3.05M (37)	4.93	1.92	3.67	0.04
	FEM/Cantilever	62K × 62K	2.03M (33)	4.46	2.37	3.37	0.94
	Pressurised wind tunnel	128K × 128K	5.93M (27)	3.99	1.40	3.80	0.02
	FEM/Charleston harbour	47K × 47K	2.37M (50)	6.70	3.60	7.03	0.02
	Quark propagators (QCD/LGT)	49K × 49K	1.92M (39)	5.07	3.03	5.45	0.02
	FEM/Ship section detail	141K × 141K	3.98M (28)	4.03	1.82	3.61	0.08
	Macroeconomics model	207K × 207K	1.27M (6)	0.97	0.61	2.91	0.04
	2D Markov epidemiology model	526K × 526K	2.10M (4)	0.64	0.29	5.62	0.74
	FEM/Accelerator cavity design	121K × 121K	1.36M (11)	1.72	1.20	2.12	0.04
	Circuit simulation	171K × 171K	959k (6)	0.87	0.60	3.24	0.01
	Web connectivity matrix	1M × 1M	3.11M (3)	0.50	0.20	2.01	0.01
—	Railways set cover constraint matrix	4K × 1.1M	11.3M (2633)	5.08	3.64	5.16	0.07

**Table 6.4:** Overview of sparse matrices tested and results of the benchmark. Measurements are in GFLOPS/s (higher is better).

that there is significant overhead in initialising the local sum for each thread at the beginning of the reduction. See Section 6.2 for further discussion.

For matrices such as Dense, the increase in memory bandwidth gained from aligning global memory reads in this way offsets the additional startup cost of doing so, and as a result Accelerate exhibits higher performance than its competitors. However, matrices such as the FEM/Spheres, which contain only a few non-zero elements per row ( $\lesssim 2 \times$  warp size = 64) exhibit a drop in performance, because this extra startup cost can not be amortised over the row length. Matrices such as Epidemiology, with large vectors and few non-zero elements per row, exhibit low flop:byte ratio and are poorly suited to the CSR format, with all implementations performing well below peak. This highlights the nature of sparse computations and the reason the CUSP library supports several algorithms and matrix formats.

As was found with the dot product benchmark (§6.2), implementing specialised kernels that can avoid this additional startup cost when the combining function and starting element form a monoid is expected to improve performance, but is left to future work.

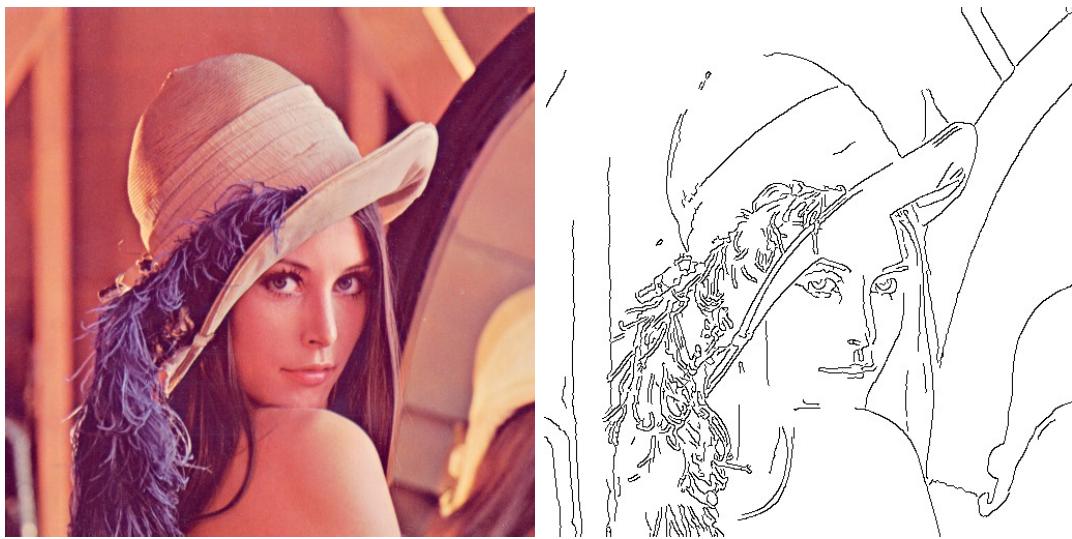
## 6.7 Canny edge detection

The edge detection example applies the Canny algorithm [24] to rectangular images of various sizes. Figure 6.8 compares Accelerate to parallel implementations on the CPU and GPU. The overall algorithm is shown in Listing 6.7 and consists of seven distinct phases, the first six of which, shown in Listing 6.8, are naturally data parallel and are performed on the GPU. The last phase uses the recursive `wildfire` algorithm to “connect” the pixels that make up the output lines. In our implementation this phase is performed on the CPU, which requires the image data to be transferred from the GPU back to the CPU for processing, and accounts for the non-linear slowdown visible with smaller image sizes. In contrast, OpenCV<sup>6</sup> is able to perform the final step on the GPU. The benchmark “Accelerate (whole program)” includes the time to transfer the image data back to the host for the final processing step. Also shown are the run times for just the first six data parallel phases, which exhibit linear speedup and do not include data transfer to the host.

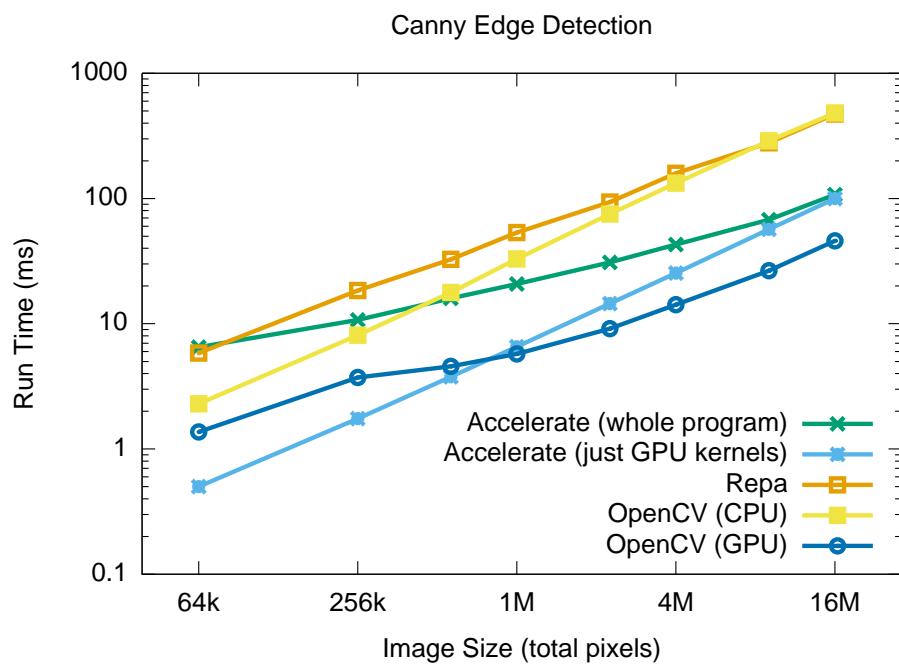
Neglecting the final phase, note that the data parallel phase is still slightly slower than in OpenCV. As discussed in Section 2.4.5, this is because the stencil kernels in Accelerate currently make a test for every array access to see if the element is in bounds, or if it lies outside the array and needs to be handled specially, even though the vast majority of points in the array are far from the boundary.

---

<sup>6</sup>Release version 2.4.9



**Figure 6.7:** An example of the Canny algorithm applied to the Lena standard test image (left) which identifies edge pixels in the image (right).



**Figure 6.8:** Runtimes for the Canny edge detection algorithm, comparing the Accelerate kernels and whole program to sequential and parallel implementations. Note the log-log scale.

```

canny :: Exp Float → Exp Float → Image → IO Image
canny low high =
  let grey          = toGreyscale
      blurred       = gaussianY . gaussianX . grey
      magdir        = gradientMagDir low . blurred
      suppress      = nonMaximumSuppression low high . magdir
      stage1 x      = let s = suppress x in (s, selectStrong s)
      (image, strong) = run $ A.lift (stage1 (use img))
  in
  wildfire (A.toRepa image) (A.toRepa strong)

```

**Listing 6.7:** The Canny edge detection algorithm

### 6.7.1 Stencil merging

Two of the data parallel stages of the Canny algorithm are the calculation of the image intensity gradients by convolving the image with the  $3 \times 3$  kernels shown below. This convolution, known as the Sobel operator, is a discrete differentiation operator that computes an approximation of the horizontal and vertical image gradients.

$$\text{Sobel}_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad \text{Sobel}_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

There is a significant amount of sharing between these two kernels, in terms of the coefficients required to compute the derivative at each point. Thus, it is beneficial to merge these two stencils into a single kernel, rather than computing them separately. This would reduce the number of accesses to the source array from  $6 + 6 = 12$  to only 8. As with all shortcut fusion methods, the system presented here does not combine multiple passes over the same array into a single traversal that computes multiple results. We leave this, as well as other significant stencil optimisations [57, 63, 73], to future work.

## 6.8 Fluid flow

The fluid flow example implements Jos Stam's stable fluid algorithm [123], which is a fast approximate algorithm intended for animation and games, rather than accurate engineering simulation. An example sequence is shown in Figure 6.9.

The core of the algorithm is a finite time step simulation on a grid, implemented as a matrix relaxation involving the discrete Laplace operator ( $\nabla^2$ ). This step, known as the linear solver, is used to diffuse the density and velocity fields throughout the grid, as well as apply a projection operator to the velocity field to ensure it conserves mass. The linear solver is implemented in terms of a stencil convolution, repeatedly computing the following for each grid element to allow the solution to converge:

```

convolve5x1 :: (Elt a, IsNum a) ⇒ [Exp a] → Stencil5x1 a → Exp a
convolve5x1 kernel (_, (a,b,c,d,e)), _)
= P.sum $ P.zipWith (*) kernel [a,b,c,d,e]

gaussianX :: Acc (Image Float) → Acc (Image Float)
gaussianX = stencil (convolve5x1 gaussian) Clamp
  where
    gaussian = P.map (/16) [ 1, 4, 6, 4, 1 ]

gradientX :: Acc (Image Float) → Acc (Image Float)
gradientX = stencil grad Clamp
  where
    grad :: Stencil3x3 Float → Exp Float
    grad ((u, _, x))
      ,(v, _, y)
      ,(w, _, z)) = x + (2*y) + z - u - (2*v) - w

gradientMagDir :: Exp Float → Acc (Image Float) → Acc (Array DIM2 (Float,Int))
gradientMagDir low = stencil magdir Clamp
  where
    magdir :: Stencil3x3 Float → Exp (Float,Int)
    magdir ((v0, v1, v2)
            ,(v3, _, v4)
            ,(v5, v6, v7)) =
      let
        dx      = v2 + (2*v4) + v7 - v0 - (2*v3) - v5           — image gradients
        dy      = v0 + (2*v1) + v2 - v5 - (2*v6) - v7
        mag    = sqrt (dx * dx + dy * dy)                         — gradient magnitude
        theta = atan2 dy dx                                         — angle of the gradient vector
        alpha = (theta - (pi/8)) * (4/pi)                          — make segments easier to classify
        norm  = alpha + 8 * A.fromIntegral (boolToInt (alpha ≤* 0))
        undef = abs dx ≤* low && abs dy ≤* low
        dir   = boolToInt (A.not undef) * ((64 * (1 + A.floor norm `mod` 4)) `min` 255)           — quantise gradient to 8 directions
      in
        lift (mag, dir)

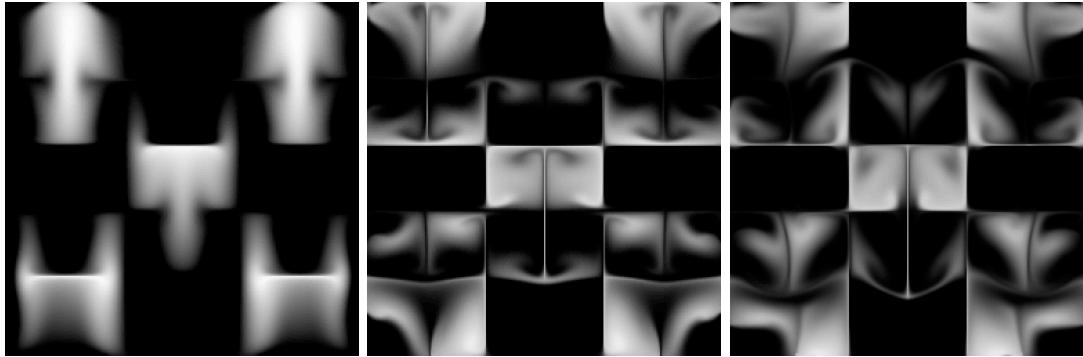
nonMaximumSuppression
  :: Exp Float → Exp Float → Acc (Image (Float,Int)) → Acc (Image Float)
nonMaximumSuppression low high magdir =
  generate (shape magdir) $ λix →
    let (mag, dir)      = unlift (magdir ! ix)
      Z :: h :: w     = unlift (shape magdir)
      Z :: y :: x     = unlift ix

      — Determine the points that lie normal to the image gradient
      offsetX       = dir >* orient' Vert ? (-1, dir <* orient' Vert ? (1, 0))
      offsetY       = dir <* orient' Horiz ? (-1, 0)
      (fwd, _)      = unlift $ magdir ! lift (clamp (Z :: y+offsety :: x+offsetx))
      (rev, _)      = unlift $ magdir ! lift (clamp (Z :: y-offsety :: x-offsetx))
      clamp (Z::u::v) = Z :: 0 `max` u `min` (h-1) :: 0 `max` v `min` (w-1)
      strong        = mag ≥* high
      none          = dir ==* orient' Undef
                    ||* mag <* low ||* mag <* fwd ||* mag <* rev
    in
      A.fromIntegral (boolToInt (A.not none) * (1 + boolToInt strong)) * 0.5

selectStrong :: Acc (Image Float) → Acc (Vector Int)
selectStrong img =
  let strong        = A.map (λx → boolToInt (x ==* edge' Strong)) (flatten img)
      (targetIdx, len) = A.scanl' (+) 0 strong
      indices         = A.enumFromN (index1 $ size img) 0
      zeros           = A.fill (index1 $ the len) 0
  in
    A.permute const zeros (λix → strong!ix ==* 0 ? (ignore, index1 $ targetIdx!ix)) indices

```

**Listing 6.8:** The data-parallel kernels of the Canny edge detection algorithm



**Figure 6.9:** An example showing the progression of the fluid flow simulation for a set of initial conditions after 10 (left), 50 (centre) and 75 (right) steps.

$$u''_{i,j} = (u_{i,j} + a \cdot (u'_{i-1,j} + u'_{i+1,j} + u'_{i,j-1} + u'_{i,j+1})) / c$$

Here,  $u$  is the grid in the previous time step,  $u'$  the grid in the current time step and previous relaxation iteration, and  $u''$  the current time step and iteration. The values  $a$  and  $c$  are constants chosen by the simulation parameters. The core implementation of the algorithm is shown in Listing 6.9.

Figure 6.10 compares Accelerate to a parallel implementation written in Repa and running on the host CPU [76]. The program is extremely memory intensive, performing approximately 160 convolutions of the grid per time step. Given the nature of the computation, we find that the raw bandwidth available on the CUDA device, which is much greater than that available to the CPU, results in correspondingly shorter run times. Since the program consists of a sequence of stencil operations, fusion does not apply to this program. Sharing recovery has a surprisingly minimal impact because the implementation of stencils always shares some parts of the computation: namely access to the grid elements  $u_{xy}$ , precisely because these operations are so expensive.

## 6.9 Radix sort

The radix sort benchmark implements a simple parallel radix sort algorithm as described by Blelloch [19] to sort an array by an integral key, and is shown in Listing 6.10. The `radixPass` function sorts the vector  $v$  based on the value of bit  $k$ ; moving elements with a zero at that bit to the beginning of the vector and those with a one bit to the end. This simple algorithm requires  $b$  iterations to sort an array of elements whose keys are  $b$ -bits wide, and each pass requires multiple traversals of the array.

As seen in Figure 6.11, the absolute performance of this simple algorithm, which sorts the array by a single bit at a time, is quite low. Implementations of radix sort optimised for the

```

type Timestep          = Float
type Diffusion         = Float
type Density           = Float
type Velocity          = (Float, Float)

type Field a            = Array DIM2 a
type DensityField      = Field Density
type VelocityField     = Field Velocity

class Elt e ⇒ FieldElt e where
  ...

instance FieldElt Density
instance FieldElt Velocity

diffuse :: FieldElt e ⇒ Int → Timestep → Diffusion → Acc (Field e) → Acc (Field e)
diffuse steps dt dn df0 =
  a ==# 0
  ?| ( df0 , foldl1 (.) (P.replicate steps diffuse1) df0 )
  where
    a          = A.constant dt * A.constant dn * (A.fromIntegral (A.size df0))
    c          = 1 + 4*a

  diffuse1 df = A.stencil2 relax (A.Constant zero) df0 (A.Constant zero) df

  relax :: FieldElt e ⇒ A.Stencil3x3 e → A.Stencil3x3 e → Exp e
  relax (_,(_,x0,_),_) ((_,t,_), (l,_,r), (_,b,_)) = (x0 .+. a .* (l.+.t.+.r.+.b)) ./ c

project :: Int → Acc VelocityField → Acc VelocityField
project steps vf = A.stencil2 poisson (A.Constant zero) vf (A.Constant zero) p
  where
    grad       = A.stencil divF (A.Constant zero) vf
    p1        = A.stencil2 pF (A.Constant zero) grad (A.Constant zero)
    p         = foldl1 (.) (P.replicate steps p1) grad

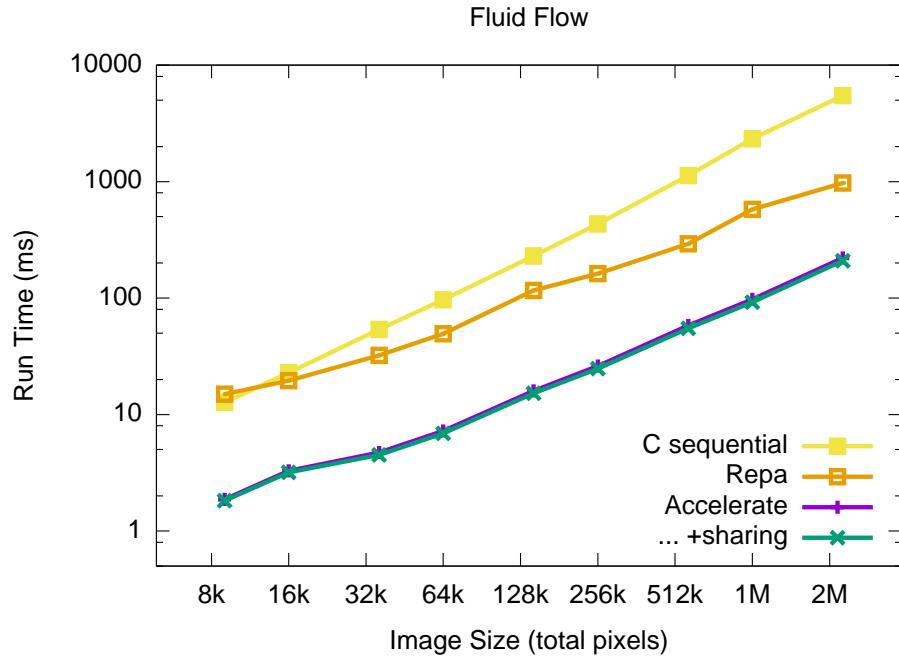
  poisson :: A.Stencil3x3 Velocity → A.Stencil3x3 Float → Exp Velocity
  poisson (_,(_,uv,_),_) ((_,t,_), (l,_,r), (_,b,_)) = uv .-. 0.5 .* A.lift (r-l, b-t)

  divF :: A.Stencil3x3 Velocity → Exp Float
  divF ((_,t,_), (l,_,r), (_,b,_)) = -0.5 * (A.fst r - A.fst l + A.snd b - A.snd t)

  pF :: A.Stencil3x3 Float → A.Stencil3x3 Float → Exp Float
  pF (_,(_,x,_),_) ((_,t,_), (l,_,r), (_,b,_)) = 0.25 * (x + l + t + r + b)

```

Listing 6.9: Fluid flow simulation



**Figure 6.10:** Kernel runtimes for the fluid flow simulation, in Accelerate with and without optimisations, compared to sequential and parallel CPU implementations. Note the log-log scale.

```

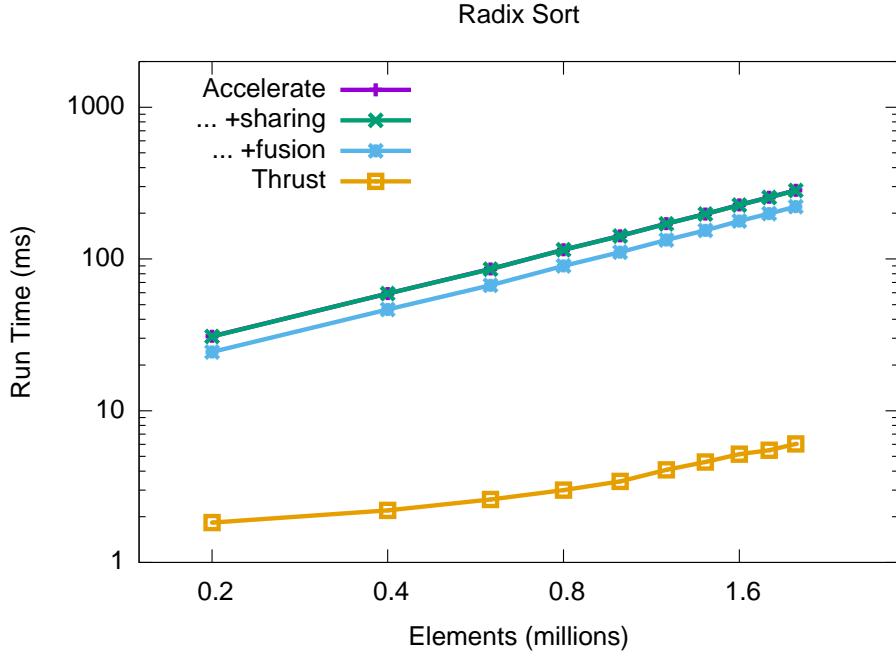
class Elt e => Radix e where
  passes :: e → Int
  radix   :: Exp Int → Exp e → Exp Int
    — number of passes (bits) required to sort this key type
    — extract the nth bit of the key

sortBy :: ∀ a r. (Elt a, Radix r)
  ⇒ (Exp a → Exp r)
  → Acc (Vector a)
  → Acc (Vector a)
sortBy rdx arr = foldr1 (>->) (P.map radixPass [0..p-1]) arr
  — loop over bits of the key, low...
  — ...bit first to maintain sort order
  where
    p      = passes (undefined :: r)
    deal f x = let (a,b) = unlift x in (f ==* 0) ? (a,b)

radixPass k v =
  let k'   = unit (constant k)
  flags = A.map (radix (the k')) . rdx v
  idown = prescanl (+) 0 . A.map (xor 1)           $ flags
  iup   = A.map (size v - 1 -) . prescanr (+) 0 $ flags
  index = A.zipWith deal flags (A.zip idown iup)
  in
  permute const v (λ ix → index1 (index!ix)) v
    — to create reusable kernels
    — extract the sort key
    — move elements to the beginning...
    — ...end of the vector
    — move elements to new position

```

**Listing 6.10:** Radix sort algorithm



**Figure 6.11:** Kernel runtimes for the radix sort benchmark of signed 32-bit integers, comparing the Accelerate version to other parallel implementations. Note the log-log scale.

CUDA architecture [59, 93, 116], are approximately 10× faster because they make efficient use of the on-chip shared memory to sort keys by 8-bits at a time. Such implementations are, essentially, different algorithms to Blelloch’s algorithm we have implemented here, but serve to illustrate the absolute performance of the device. To address this we have implemented a foreign function interface for Accelerate to take advantage of existing high-performance libraries (§4.3). The foreign function interface is orthogonal to the work presented here of optimising programs written in the Accelerate language. Useful future work would be to combine a fast-yet-fixed foreign implementation with a more generic interface, using for example the method of Henglein and Hinze [56].

## 6.10 Shortest paths in a graph

The Floyd-Warshall algorithm finds the lengths of the shortest paths between all pairs of points in a weighted directed graph. The implementation in Accelerate is shown in Listing 6.11, and appeared in the book *Parallel and Concurrent Programming in Haskell* [83]. The implementation in Accelerate applies the algorithm over a dense adjacency matrix, building the solution bottom-up so that earlier results are used when computing later ones. Each step of the algorithm is  $O(n^2)$  in the number of vertices, so the whole algorithm is  $O(n^3)$ . See [83] for further information.

```

type Weight = Int32
type Graph = Array DIM2 Weight
                                         — distance between vertices as an adjacency matrix

shortestPaths :: Graph → Graph
shortestPaths g0 = run (shortestPathsAcc n (use g0))
  where
    Z :: _ :: n = arrayShape g0

shortestPathsAcc :: Int → Acc Graph → Acc Graph
shortestPathsAcc n g0 = foldl1 (.) steps g0
  where
    steps :: [ Acc Graph → Acc Graph ]           — apply step in the sequence 0..n - 1
    steps = [ step (unit (constant k)) | k ← [0 .. n-1] ]

step :: Acc (Scalar Int) → Acc Graph → Acc Graph
step k g = generate (shape g) sp
  where
    k' = the k
    sp :: Exp DIM2 → Exp Weight
    sp ix = let Z :: i :: j = unlift ix
            in min (g ! (index2 i j))
                  (g ! (index2 i k') + g ! (index2 k' j)) — the shortest path from i to j...
                                              — is either the path i → j, or...
                                              — the path i → k then k → j

```

**Listing 6.11:** Floyd-Warshall shortest-paths algorithm [83]

The core of the algorithm is the function `step`, which computes the lengths of the shortest paths between using two elements, using only vertices up to `k`, given the lengths of the shortest paths using vertices up to `k - 1`. The length of the shortest path between two vertices `i` and `j` is then the minimum of the previous shortest path from `i` to `j`, or the path that goes from `i` to `k` and then from `k` to `j`. The final adjacency graph is constructed by applying `step` to each value of `k` in the sequence `0 .. n-1`. Figure 6.12 compares the performance of the algorithm in Accelerate to several parallel CPU implementations. Note that without array level sharing, the number of terms in the program scales exponentially to the number of nodes in the graph, and so does not complete in a reasonable time even for small graphs.

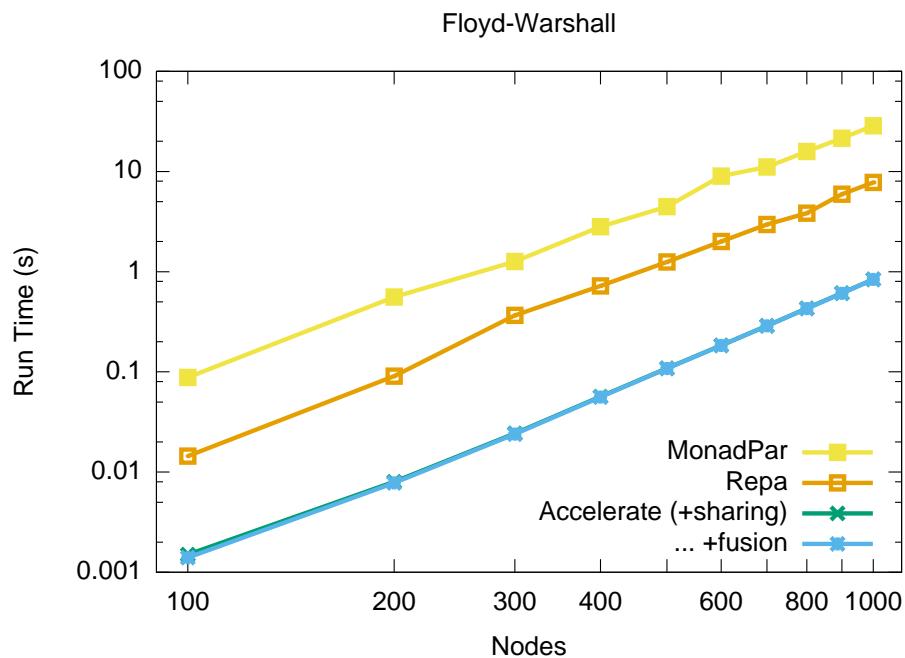
## 6.11 MD5 password recovery

The MD5 message-digest algorithm [110] is a cryptographic hash function producing a 128-bit hash value from a variable length message. MD5 has been used in a wide variety of cryptographic and data integrity applications, although cryptographers have since found flaws in the algorithm and now recommend the use of other hash functions for cryptographic applications.<sup>7</sup>

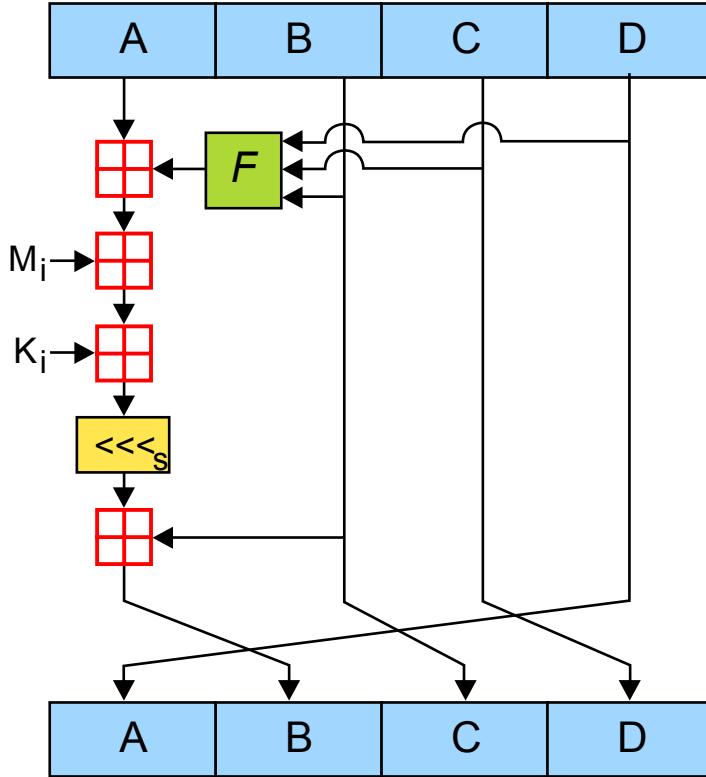
Figure 6.13 illustrates the main MD5 algorithm, whose implementation in Accelerate is shown in Listing 6.12. This implementation only processes a single MD5 round (512-bits of input as  $16 \times 32$ -bit words), and the input message is padded appropriately on the CPU before being passed to the `md5round` function. The algorithm operates on a 128-bit state, divided

---

<sup>7</sup><http://www.kb.cert.org/vuls/id/836068>



**Figure 6.12:** Kernel runtimes for Floyd-Warshall shortest path algorithm, compared to other parallel implementations. The test data is randomly generated graph of  $n$  nodes and (up to)  $2n$  edges. The Monad-Par program uses a sparse representation which is sensitive to the total number of edges, whereas the Accelerate and Repa versions use a dense adjacency matrix representation which is not. Note the log-log scale.



**Figure 6.13:** A single round of the MD5 hash algorithm. It consists of 64 iterations of this operation, split into 4 rounds of 16 iterations each.  $F$  is a nonlinear function that is different for each round,  $M_i$  denotes a 32-bit block of the input message, and  $K_i$  a 32-bit constant.  $\lll_s$  is left rotation by  $s$  bits, and  $\boxplus$  addition modulo  $2^{32}$ . Image from <http://en.wikipedia.org/wiki/MD5>.

into  $4 \times 32$ -bit words, denoted  $A$ ,  $B$ ,  $C$ , and  $D$ , which are initialised to fixed constants. The algorithm uses the 512-bit message block to modify the state in four rounds of 16 operations each, where each round is based on a nonlinear function  $F$ , modulo addition, and left rotation. The nonlinear mixing operation used by each round, where  $\oplus$ ,  $\wedge$ ,  $\vee$ , and  $\neg$  are the bitwise XOR, AND, OR and NOT operations, are respectively:

$$\begin{aligned} F(B, C, D) &= (B \wedge C) \vee (\neg B \wedge D) \\ G(B, C, D) &= (B \wedge D) \vee (C \wedge \neg D) \\ H(B, C, D) &= B \oplus C \oplus D \\ I(B, C, D) &= C \oplus (B \vee \neg D) \end{aligned}$$

The input to the `md5round` function consists of a `(Z .. 16 .. n)` input array containing  $n$  512-bit input chunks to hash, which are processed by  $n$  threads in parallel. Note that the input array is stored in column major order, so that accesses to the input `dict` are coalesced. See [98] for details on CUDA memory access rules. In contrast, a CPU prefers the data to be laid out in row-major order, so that the input chunk is read on a single cache line. Exploring

```

type MD5          = (Word32, Word32, Word32, Word32)      — output 128-bit digest
type Dictionary = Array DIM2 Word32                         — input 16×32-bit words per block

md5Round :: Acc Dictionary → Exp DIM1 → Exp MD5
md5Round dict (unindex1 → word)
= lift
$ foldl step (a0,b0,c0,d0) [0..64]
where
  step (a,b,c,d) i
  | i < 16    = shfl ((b .&. c) .|. ((complement b) .&. d))      — F
  | i < 32    = shfl ((b .&. d) .|. (c .&. (complement d)))      — G
  | i < 48    = shfl (b `xor` c `xor` d)                                — H
  | i < 64    = shfl (c `xor` (b .|. (complement d)))      — I
  | otherwise = (a+a0,b+b0,c+c0,d+d0)
  where
    shfl f = (d, b + ((a + f + k i + get i) `rotateL` r i), b, c)

get :: Int → Exp Word32
get i
| i < 16    = getWord32le i
| i < 32    = getWord32le ((5*i + 1) `rem` 16)
| i < 48    = getWord32le ((3*i + 5) `rem` 16)
| otherwise = getWord32le ((7*i) `rem` 16)

getWord32le :: Int → Exp Word32
getWord32le (constant → i) = dict ! index2 word i

a0, b0, c0, d0 :: Exp Word32      — initial values (constants)

k :: Int → Exp Word32            — binary integer part of sines & cosines (in radians)
k i = constant (ks P.!! i)       — (constants)
  where ks = [ ... ]

r :: Int → Exp Int               — per-round shift amounts
r i = constant (rs P.!! i)       — (constants)
  where rs = [ ... ]

```

**Listing 6.12:** A single round of the MD5 hash algorithm

Benchmark	Rate (MHash/sec)
Accelerate	263.3
Hashcat (CPU)	57.1
Hashcat (GPU)	773.3

**Table 6.5:** MD5 password recovery benchmarks

the differences in memory architectures between CPUs and GPUs, which focus on task versus data parallelism respectively, is left for future work.

In the MD5 benchmark, each thread computes the hash of a 512-bit input column and compares it to a supplied hash. If the values match, the input corresponds to the plain text of the unknown hash. Table 6.5 compares Accelerate to several other implementations. Note that the Hashcat program is not open source and provides many additional options not supported by the Accelerate implementation, so they are not strictly comparable. However, Hashcat is regarded as the fastest password recovery tool available, so provides a useful baseline.

## 6.12 K-Means clustering

In the  $K$ -means problem, the goal is to partition a set of observations into  $k$  clusters, in which each observation belongs to the cluster with the nearest mean. Finding an optimal solution to the problem is NP-hard, however there exist efficient heuristic algorithms that converge quickly to a local optimum, and in practice give good results. The most well known heuristic technique is Lloyd’s algorithm, which finds a solution by iteratively improving on an initial guess. The algorithm takes as a parameter the number of clusters, makes an initial guess at the centre of each cluster, and proceeds as follows:

1. Assign each point to the cluster to which it is closest. This yields the new set of clusters.
2. Compute the centroid of each cluster.
3. Repeat steps (1) and (2) until the cluster locations stabilise. Processing is also stopped after some chosen maximum number of iterations, as sometimes the algorithm does not converge.

The initial guess can be constructed by randomly assigning each point in the data set to a cluster and then finding the centroids of those clusters. The core of the algorithm is shown in Listing 6.13, which computes the new cluster locations. To complete the algorithm, the function `makeNewClusters` is applied repeatedly until convergence, or some maximum iteration limit is reached. While the algorithm works for any number of dimensions, the implementation shown here is for two dimensional points only.

```

type Point a    = (a, a)
type Id        = Word32
type Cluster a = (Id, (a, a))
type PointSum a = (Word32, (a, a))

distance :: (Elt a, IsNum a) ⇒ Exp (Point a) → Exp (Point a) → Exp a
distance u v = ...

findClosestCluster
  :: ∀ a. (Elt a, IsFloating a, RealFloat a)
  ⇒ Acc (Vector (Cluster a))
  → Acc (Vector (Point a))
  → Acc (Vector Id)
findClosestCluster clusters points =
  A.map (λp → A.fst $ A.foldl (nearest p) z (constant Z) clusters) points
  where
    z = constant (-1, inf)

  nearest :: Exp (Point a) → Exp (Id, a) → Exp (Cluster a) → Exp (Id, a)
  nearest p st c =
    let d = A.snd st
      d' = distance p (centroidOfCluster c)
    in
      d' <* d ? ( lift (idOfCluster c, d') , st )

makeNewClusters
  :: ∀ a. (Elt a, IsFloating a, RealFloat a)
  ⇒ Acc (Vector (Point a))
  → Acc (Vector (Cluster a))
  → Acc (Vector (Cluster a))
makeNewClusters points clusters
  = pointSumToCluster
  . makePointSum
  . findClosestCluster clusters
  $ points
  where
    npts      = size points
    nclusters = size clusters

  pointSumToCluster :: Acc (Vector (PointSum a)) → Acc (Vector (Cluster a))
  pointSumToCluster ps =
    A.generate (A.shape ps)
      (λix → lift (A.fromIntegral (unindex1 ix), average (ps ! ix)))

  makePointSum :: Acc (Vector Id) → Acc (Vector (PointSum a))
  makePointSum = A.foldl addPointSum . compute . pointSum

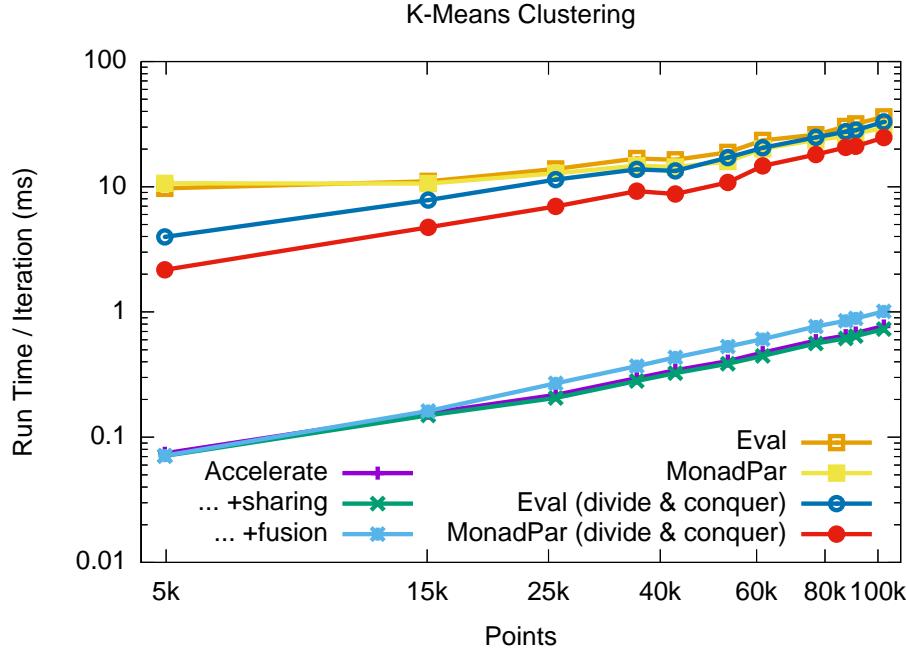
  pointSum :: Acc (Vector Id) → Acc (Array DIM2 (PointSum a))
  pointSum nearest =
    A.generate (lift (Z :. nclusters :. npts))
      (λix → let Z:i:j = unlift ix    :: Z :: Exp Int :: Exp Int
             near   = nearest ! index1 j
             yes    = lift (constant 1, points ! index1 j)
             no     = constant (0, (0,0))
           in
             near ==* A.fromIntegral i ? ( yes, no ))

  average :: Exp (PointSum a) → Exp (Point a)
  average ps = ...          — average of the x- and y-coordinates

  addPointSum :: Exp (PointSum a) → Exp (PointSum a) → Exp (PointSum a)
  addPointSum x y = ...      — component-wise addition

```

Listing 6.13: K-means clustering for 2D points



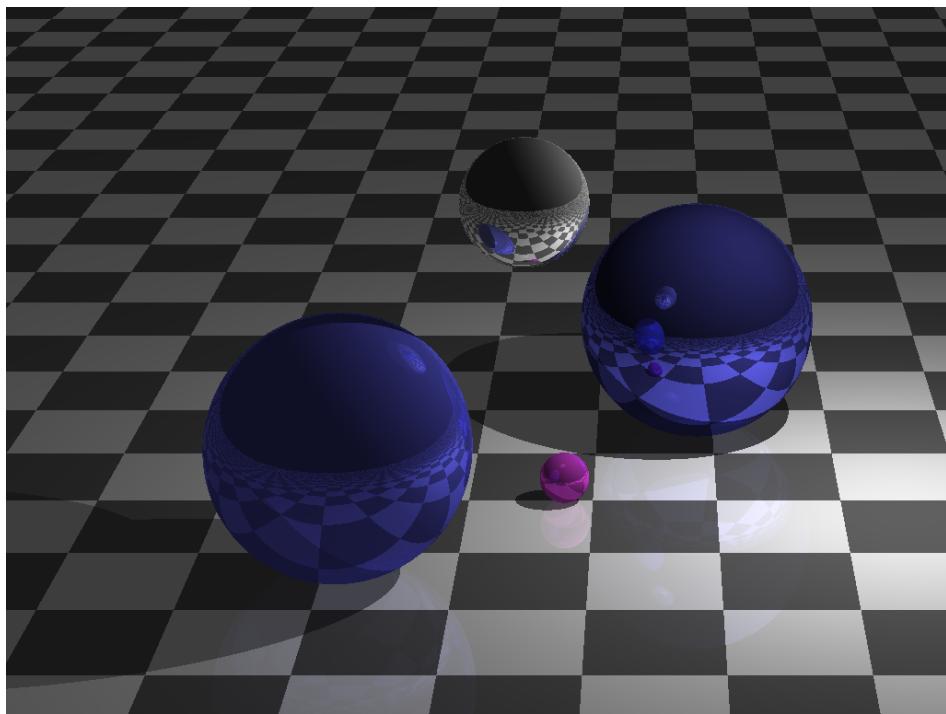
**Figure 6.14:** Kernel runtimes for the  $k$ -means clustering algorithm in Accelerate compared to several parallel CPU implementations. Note the log-log scale.

Figure 6.14 compares the performance of the implementation shown in Listing 6.13 to several other parallel CPU implementations, which are adapted from [83].

Note the use of `compute` in the definition of `makePointSum` of Listing 6.13, which explicitly *prevents* the computation `pointSum` from fusing into the reduction that consumes it. Although fusion is possible in this circumstance, resulting program is 10× slower than the unfused program (kernel time). This slowdown is because the fused kernel requires 41 registers per thread and thus achieves only 30% occupancy, compared to the unfused kernels which uses 17 registers and executes at 75% occupancy. Since the occupancy of the fused kernel is so low, the GPU can not adequately hide data transfer latency by swapping active threads, further reducing performance. Implementing backend-specific and hardware-aware optimisation decisions is left for future work.

## 6.13 Ray tracer

Ray tracing is a technique for generating an image by tracing the path of light through pixels in an image plane and simulating the effects of its encounters with virtual objects. The technique is capable of producing a very high degree of realism compared to typical scanline rendering methods, as ray tracing algorithms are able to simulate optical effects such as reflection and refraction, scattering, and dispersion phenomena. This increased accuracy has a larger computational cost, which is why ray tracing is not typically used for real time



**Figure 6.15:** Image of a ray traced scene rendered on the GPU with Accelerate, featuring multiple reflections. The scene is animated and renders in real time on modest hardware.

rendering applications such as computer games. Figure 6.15 shows the result of rendering a scene using the ray casting algorithm shown in Listing 6.14.

Figure 6.16 compares the performance of Accelerate to an equivalent program written in Repa. The main source of the poor performance of the Accelerate program is that the generated code uses 73 registers per thread, resulting in very low thread occupancy of only 19%. The Repa program is written in continuation passing style, and produces very efficient code for the `traceRay` function. Since Accelerate has no support for general recursion, the structure of the program is unrolled on the Haskell level, resulting in a large amount of generated code as well as additional branching. Additionally, Accelerate does not have any support for sum algebraic data types (tagged unions), so the code must test intersection with each different type of object in the scene separately. Improving code generation and adding support for general recursion and sum data types in a manner that remains efficient on massively parallel SIMD architectures such as GPUs is left for future work.

## 6.14 LULESH

Computer simulations for a wide variety of scientific and engineering problems require modelling of hydrodynamics, which describes the motion of materials relative to each other in the presence of forces. Many important simulation problems involve complex multi-material sys-

```

traceRay
:: Int          — Maximum reflection count
→ Acc Objects   — Objects in the scene
→ Acc Lights    — Direct lighting sources in the scene
→ Exp Color     — Ambient light in the scene
→ Exp Position  — Origin of the ray (pixel location)
→ Exp Direction — Direction of the ray
→ Exp Color

traceRay limit objects lights ambient = go limit
where
(spheres, planes) = unlift objects

dummySphere      = constant (Sphere (XYZ 0 0 0) 0           (RGB 0 0 0) 0)
dummyPlane       = constant (Plane  (XYZ 0 0 0) (XYZ 0 0 1) (RGB 0 0 0) 0)

go 0 _ _          — Stop once there are too many reflections, in case
= black           — we've found two parallel mirrors

go bounces orig dir  — See which objects the ray intersects
= let
  (hit_s, dist_s, s) = castRay distanceToSphere dummySphere spheres orig dir
  (hit_p, dist_p, p) = castRay distanceToPlane dummyPlane planes orig dir
in
A.not (hit_s ||* hit_p) ?
( black           — ray didn't intersect any objects

, let             — ray hit an object
  — determine the intersection point, and surface properties that
  — will contribute to the colour
  next_s          = hitSphere s dist_s orig dir
  next_p          = hitPlaneCheck p dist_p orig dir
  (point, normal, color, shine)
    = unlift (dist_s <* dist_p ? ( next_s, next_p ))

  — result angle of ray after reflection
  newdir          = dir - (2.0 * (normal `dot` dir)) .* normal

  — determine the direct lighting at this point
  direct          = applyLights objects lights point normal

  — see if the ray hits anything else
  refl            = go (bounces - 1) point newdir

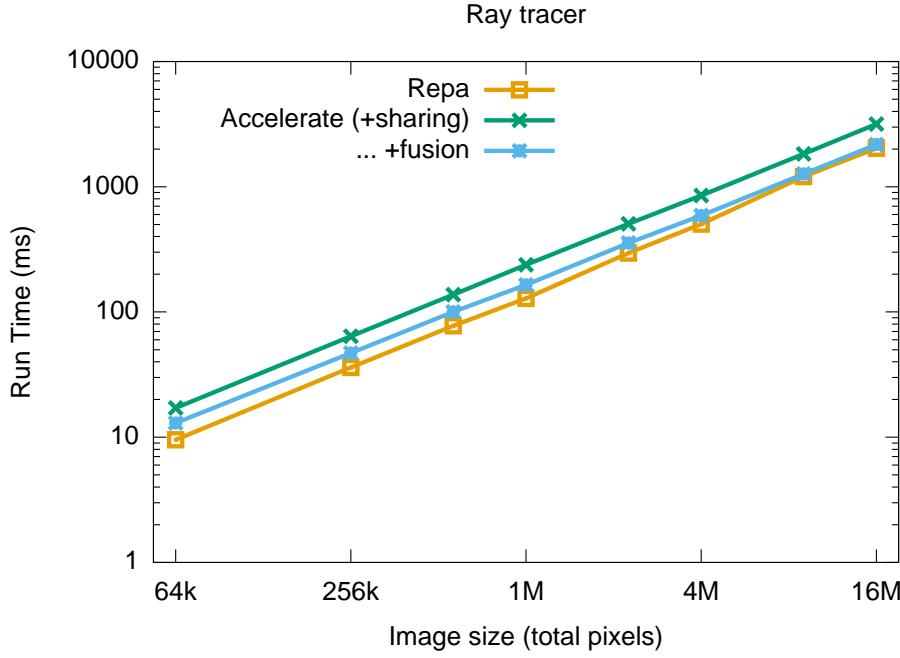
  — total lighting is the direct lighting plus ambient
  lighting        = direct + ambient

  — total incoming light is direct lighting plus reflections
  light_in        = scaleColour shine         refl
                  + scaleColour (1.0 - shine) lighting

  — outgoing light is incoming light modified by surface color, clipped in case
  — the sum of all incoming lights is too bright to display
  light_out       = clampColor (light_in * color)
in
light_out
)

```

**Listing 6.14:** The core ray casting implementation

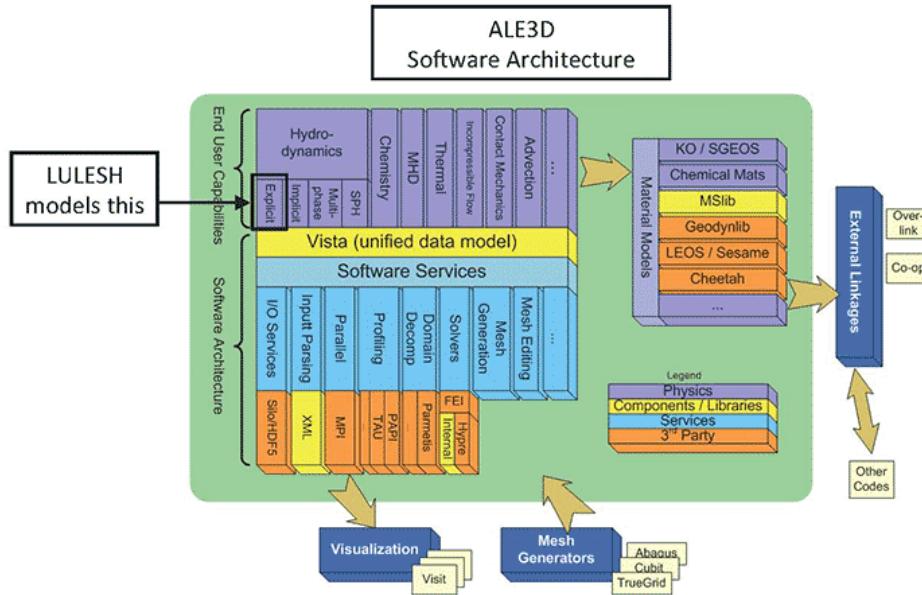


**Figure 6.16:** Kernel runtimes for the ray tracer program, in Accelerate compared to a parallel implementation written in Repa. Note the log-log scale.

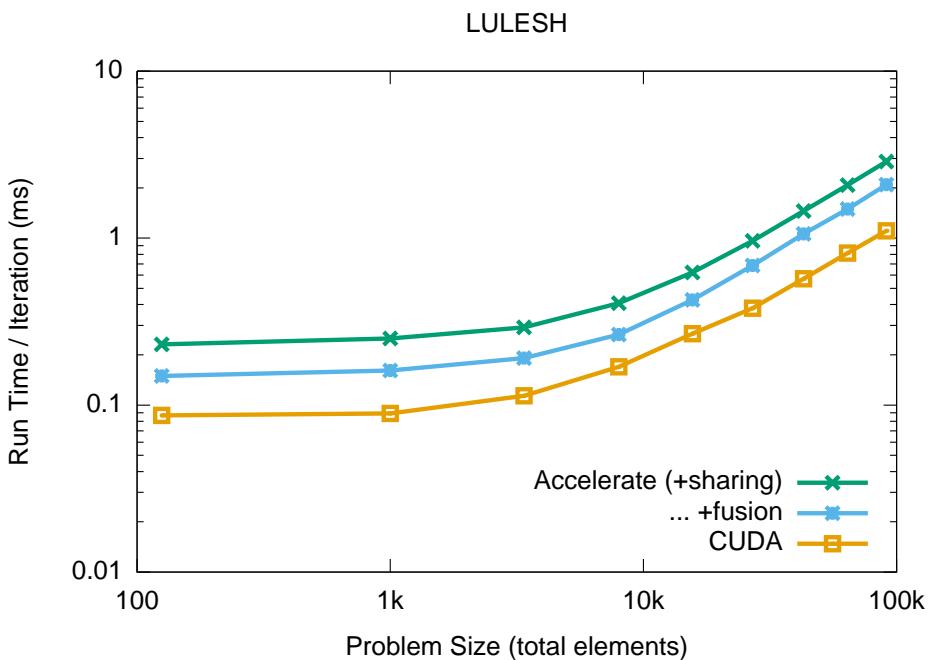
tems that undergo large deformations. The Livermore Unstructured Lagrange Explicit Shock Hydrodynamics (LULESH) application [4, 64, 65] is a proxy application which solves the Sedov blast wave problem, and is designed to be representative of the numerical algorithms, data motion, and programming style in scientific hydrodynamics applications. Figure 6.17 depicts the part of larger hydrodynamic codes that are modelled by the LULESH proxy application.

The LULESH benchmark provides some insight as to how Accelerate compares to real applications. The reference CUDA implementation consists of 3000 lines of code, and is specialised to the Kepler architecture only (compute capability 3.x). A separate implementation is required for the Fermi architecture (compute capability 2.x) and is 4400 lines of code (although additionally supports multiple GPUs). Karlin et al. [65] describe the GPU port of LULESH, and note that several changes over the base (sequential CPU) implementation required significant human effort, and may not be portable between GPU generations. In contrast, the Accelerate implementation is only 800 lines of code, and is portable to multiple architectures and backend targets from a single source.

As no implementation for our compute capability 1.3 GPU is available, this benchmark was run using a Tesla K40c processor (compute capability 3.5, 15 multiprocessors = 2880 cores at 750MHz, 11GB RAM) backed by two 12-core Xeon E5-2670 CPUs (64-bit, 2.3GHz, 32GB RAM) running GNU/Linux (Ubuntu 14.04 LTS). Figure 6.18 compares the performance of Accelerate to the reference implementation for Kepler GPUs. Future work could investigate



**Figure 6.17:** The structure of the full-featured ALE3D hydrodynamics application, and its relation to the LULESH proxy application. Although highly simplified, LULESH is designed to be representative of the numerical algorithms, data motion, and programming style of the overall application. Image from <https://codesign.llnl.gov/lulesh.php>.



**Figure 6.18:** Kernel runtimes for the LULESH program, in Accelerate compared to the reference GPU implementation. Note the log-log scale.

whether any of the optimisations the reference implementation enjoys are useful in other applications, and could be targets for integration into Accelerate.

## 6.15 Discussion

The previous chapters have discussed how to implement and optimise an EDSL with skeleton based code generation targeting bulk parallel SIMD hardware such as GPUs. This chapter presented a series of benchmark applications, including a comparison to other hand-optimised CPU and/or GPU implementations, that demonstrates the positive effect of these optimisations, showing that we can approach the performance of hand-written CUDA code while retaining a much higher level of abstraction.



## Conclusion

---

---

A new scientific truth does not triumph by convincing its opponents and making them see the light, but rather because its opponents eventually die, and a new generation grows up that is familiar with it.

—MAX PLANCK

The present dissertation argues that purely functional embedded languages represent a good programming model for making effective use of massively parallel SIMD hardware. Indeed, all current GPU programming models, including CUDA, implicitly emphasise a purely functional style, where global side-effects must be tightly controlled by the user.

To this end, this thesis has developed a purely functional language, compiler, and runtime system for executing flat data-parallel array computations targeting graphics processing units. When implementing such an embedded language, the naïve compilation of such programs quickly leads to both code explosion and excessive intermediate data structures. The resulting slowdown is not acceptable on target hardware that is usually chosen to achieve high performance. This thesis demonstrates that our embedding in Haskell, combined with the program optimisations and runtime system that I have implemented in this thesis, are a successful approach to implementing an expressive array programming language targeting bulk-parallel SIMD hardware. Moreover, I demonstrate that the Accelerate language and CUDA backend that I have developed simultaneously offers a higher level of abstraction while often approaching the performance of traditional low-level methods for programming GPUs.

Chapter 3 introduces Accelerate, an embedded language of parameterised, collective array operations. Accelerate is our approach to reducing the complexity of programming massively parallel multicore processors such as GPUs. The choice of language operations is informed by the scan-vector model, which has been shown to be suitable for a wide range of algorithms and can be implemented efficiently on the GPU. We extend this model with parameterised, rank-polymorphic operations over multidimensional arrays, making Accelerate more expressive than previous high-level GPU programming language proposals.

Chapter 4 details how I implemented the Accelerate backend which targets programmable graphics processing units. While the current implementation targets CUDA, the same approach works for other target languages such as OpenCL and LLVM. In the implementation of Accelerate’s CUDA backend, I regard the collective operations of the Accelerate language as algorithmic skeletons, where a parallel program is composed of one or more parameterised skeletons, or templates, encapsulating specific parallel behaviour. The dynamically generated code is specialised to the capabilities of the current hardware, and to minimise the overhead of online compilation, kernels are compiled in parallel and asynchronously with other operations, and previously compiled kernels are cached and reused. The runtime system automatically manages data on the GPU, minimising allocations and data transfers and overlapping host-to-device transfers with other operations. The execution engine optimises kernel launches for the current hardware to maximise thread occupancy, and kernels are scheduled so that independent operations are executed concurrently, for devices that support this capability. All this and more is done to ensure that the Accelerate runtime system executes programs as efficiently as possible.

Chapter 5 describes our solution to what we found as the two most pressing performance limitations for executing embedded array computations: sharing recovery and array fusion. In particular, Chapter 5 discusses my novel approach to type-safe array fusion for compilers targeting bulk-parallel SIMD hardware. Fusion for a massively data-parallel embedded language instrumented via algorithmic skeletons requires a few uncommon considerations compared to existing shortcut fusion methods. This problem was tackled by classifying array operations into two categories: producers are operations where individual elements are computed independently, while consumers need to know exactly how the input elements depend on each other in order to implement the operation efficiently in parallel. The problem with existing fusion systems is that this information is obfuscated, and as a result the parallel interpretation of the program is lost. Program fusion is realised in two stages. First, sequences of producer operations are fused using type-preserving source-to-source term rewrites. In the second stage, the fused producer representation is embedded directly into the consumer skeleton during code generation. This phase distinction is necessary in order to support the different properties of producers and consumers.

Following these two main chapters, Chapter 6 presents a series of benchmarks comparing the performance of the Accelerate program to hand-optimised reference implementations. The benchmarks illustrate the positive effect of the optimisations discussed in this thesis, and the accompanying analyses discuss future work that can be undertaken in cases where the performance of the Accelerate program is lower than that of the reference solution. Overall, I have demonstrated that the Accelerate language is suitable for expressing a wide range of programs, and that the runtime system and program optimisations described in this thesis result in programs that are often competitive with low level, hand optimised solutions.

## 7.1 Research contribution

This thesis presents two main research contributions. First, it demonstrates how to implement a high-performance embedded language and runtime system. Second, it establishes a novel method of array fusion for languages targeting bulk-parallel SIMD hardware. Together, these contributions provide an embedded language of array operations, which achieves performance comparable to traditional imperative language approaches, but at a much higher level of abstraction.

### 7.1.1 A high-performance embedded language

While the current generation of graphics processing units are massively parallel processors with peak arithmetic and memory throughput rates much higher than traditional CPUs, attaining the advertised  $100\times$  speedups remains a work intensive exercise that requires a substantial degree of expert knowledge. Several researchers have attempted to ameliorate the status quo, by either using a library to compose GPU code or by compiling a subset of a high-level language to low-level GPU code. However, no other proposal has offered a high-level interface with the breadth and expressivity of operations provided by the Accelerate language, together with an implementation whose performance is comparable to low-level programming languages.

The implementation of Accelerate’s CUDA backend, covered in Chapter 4, is original in that it successfully addresses these problems. While the current implementation targets CUDA, my approach to code generation works for other languages targeting data or control parallel hardware, and the underlying runtime system is applicable to any deeply embedded language which is implemented via runtime code generation and online compilation.

### 7.1.2 A fusion system for parallel array languages

Although the topic of fusion has received significant prior attention, particularly in the context of functional programming languages, none of the existing techniques are adequate for a type-preserving embedded language compiler targeting massively parallel SIMD hardware such as graphics processing units.

The implementation of fusion for embedded array languages implemented via algorithmic skeletons, as developed in Chapter 5, is original. In particular, because the fusion transformation retains the structure of the program as a sequence of collective operations, our compilation techniques developed in Chapter 4 are able to generate efficient parallel implementations of fused operations via skeleton template instantiation. Our fused skeletons are able to exploit all levels of the processor thread and memory hierarchy, which is of crucial importance to achieving high performance.

## 7.2 Future work

Based on the results of this thesis, a wide range of interesting future work is possible. The previous chapters and the analysis of the benchmark results in particular have already given suggestions for future work; these are not repeated here, but some further topics are discussed.

### 7.2.1 Nested data parallelism

This thesis has demonstrated a high level array programming language and runtime environment that is expressive and has performance competitive to much lower level imperative program languages. However, the Accelerate language is restricted to programs which express only *flat* data parallelism, in which the computation of each data element must itself be sequential. In practice, this severely limits the applications of data-parallel computing, especially for sparse and irregular problems [108].

Blelloch's pioneering work on NESL showed that it was possible to combine the more flexible model of *nested* data parallelism, where the computation of each element may spawn further parallel computations, with a flat data parallel execution model [22]. Recent work has extended flattening to support richer data types and higher order functions [105].

Extending the Accelerate language to support the expression of nested parallelism, and implementing the vectorisation transformation to convert this nested parallelism into a flat data parallel program which can be executed on the compiler and runtime system presented in this thesis, is worthwhile future work. This would allow Accelerate to support programs that are very difficult to parallelise in a flat data parallel setting, such as the Barnes-Hut algorithm for  $N$ -body simulation [11].

### 7.2.2 Type-preserving code generation

The Accelerate language is richly typed, maintaining full type information of the embedded language in the term tree and during the transformations described in this thesis. Since Accelerate is an embedded language implemented via online compilation, compilation time of the Accelerate program corresponds to Haskell program *runtime*. By encoding type information into the Accelerate terms that are checked by the Haskell type checker at Haskell compile time, we ensure that only well formed Accelerate programs will be compiled, reducing the number of possible runtime errors.

Although we ensure the source program is well typed, the implementation of code generation presented in this thesis does not reflect the types of the target language into the Haskell type system. Thus, any errors in translating the Accelerate program into CUDA source code will not be detected at Haskell compilation time, and will instead raise a runtime error when the generated code is compiled. Adding a fully type preserving compiler that translates the well-typed Accelerate program into verifiable target code is important future work.

### 7.2.3 Flexibility in the optimisation pipeline

Chapter 6 demonstrated the positive effects of the optimisations that have been presented in this thesis. However, the optimisation pipeline is completely fixed, so changing or extending the set of optimisations requires changing the compiler itself.

Useful future work would be the ability to control the existing optimisation process, for example by specifying the eagerness with which producer operations should be fused. This would allow the user to override the default behaviour of always fusing expressions that are used exactly once in subsequent computations.

Orthogonal to this work would be the ability to extend the optimisation process through the implementation of a system of extensible rewrite rules [103]. Although our host language Haskell compiler supports rewrite rules, since Accelerate programs can be generated at program runtime, we require rewrite rules at program runtime as well. In addition to allowing the compiler to generate rules internally to propagate information obtained from automated analyses, this would allow library authors to express domain specific knowledge that the compiler can not discover for itself. For example, the author of a library of linear algebra routines might like to express the equivalence that  $x^{-1}x \mapsto 1$ . While a programmer may be unlikely to write such expressions themselves, they can easily appear when aggressive inlining brings together code that was written separately. Adding the ability to add rewrite rules to the program could be a very powerful feature, but raises questions such as verifying whether the programmer-specified rules are consistent with the underlying function definitions that they purport to describe, or that the set of rules are confluent or even terminating.



---

---

## Bibliography

---

---

- [1] The DOT Language. URL <http://www.graphviz.org/content/dot-language>.
- [2] NumPy, 2006. URL <http://www.numpy.org>.
- [3] CLyther, 2010. URL <http://srossross.github.io/Clyther/>.
- [4] Hydrodynamics Challenge Problem. Technical Report LLNL-TR-490254, 2011.
- [5] S. J. Aarseth. *Gravitational N-Body Simulations: Tools and Algorithms*. Cambridge Monographs on Mathematical Physics. Cambridge University Press, 1 edition, 2003.
- [6] AccelerEyes. Jacket. 2012.
- [7] Adobe Systems Incorporated. *PostScript Language Reference*. Addison-Wesley, 3 edition, Feb. 1999.
- [8] T. Altenkirch and B. Reus. Monadic Presentation of Lambda Terms Using Generalised Inductive Types. In J. Flum and M. Rodriguez-Artalejo, editors, *CSL '99: Computer Science Logic*, pages 453–468, 1999.
- [9] A. W. Appel and T. Jim. Shrinking lambda expressions in linear time. *Journal of Functional Programming*, 7(5), Sept. 1997.
- [10] R. Atkey, S. Lindley, and J. Yallop. Unembedding domain-specific languages. In *Haskell '09: The 2nd ACM SIGPLAN Symposium on Haskell*, pages 37–48, 2009.
- [11] J. Barnes and P. Hut. A hierarchical  $\mathcal{O}(N \log N)$  force-calculation algorithm. *Nature*, 324(6096):446–449, Dec. 1986.
- [12] W. Belkhir and A. Giorgetti. Lazy AC-Pattern Matching for Rewriting. In *WRS '11: The 10th International Workshop on Reduction Strategies in Rewriting and Programming*, pages 37–51, 2011.
- [13] N. Bell and M. Garland. Efficient Sparse Matrix-Vector Multiplication on CUDA. Technical Report NVR-2008-004, NVIDIA Corporation, 2008.

- [14] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: The 2009 International Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, 2009.
- [15] M.-W. Benabderahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *CC '10: The 2010 International Conference on Compiler Construction*, Mar. 2010.
- [16] N. Benton, A. Kennedy, S. Lindley, and C. Russo. Shrinking reductions in SML.NET. In *IFL '04: The 16th International Symposium on Implementation and Application of Functional Languages*, Sept. 2004.
- [17] L. Bergstrom and J. Reppy. Nested data-parallelism on the GPU. In *ICFP '12: The 17th ACM SIGPLAN international conference on Functional programming*, Sept. 2012.
- [18] G. E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.
- [19] G. E. Blelloch. Prefix Sums and Their Applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Nov. 1990.
- [20] G. E. Blelloch. NESL: A Nested Data-Parallel Language. Technical Report CMU-CS-95-170, School of Computer Science, Carnegie Mellon University, Sept. 1995.
- [21] G. E. Blelloch and J. Greiner. A provable time and space efficient implementation of NESL. In *ICFP '96: The 1st ACM SIGPLAN International Conference on Functional Programming*, pages 213–225, 1996.
- [22] G. E. Blelloch and G. W. Sabot. Compiling collection-oriented languages onto massively parallel computers. In *FRONTIERS '88: Proceedings of the 2nd Symposium on the Frontiers of Massively Parallel Computation*, pages 575–585, Oct. 1988.
- [23] B. Bond, K. Hammil, L. Litchev, and S. Singh. FPGA Circuit Synthesis of Accelerator Data-Parallel Programs. In *FCCM '10: The 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 167–170, May 2010.
- [24] J. F. Canny. A Computational Approach to Edge Detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, pages 679–698, 1986.
- [25] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an embedded data parallel language. In *PPoPP '11: The 16th ACM Symposium on Principles and Practice of Parallel Programming*, Feb. 2011.
- [26] M. M. T. Chakravarty. Converting a HOAS term GADT into a de Bruijn term GADT, 2009. URL <http://www.cse.unsw.edu.au/~chak/haskell/term-conv/>.

- [27] M. M. T. Chakravarty and G. Keller. Functional array fusion. In *ICFP '01: The 6th ACM SIGPLAN international conference on Functional programming*, pages 205–216, 2001.
- [28] M. M. T. Chakravarty, G. Keller, and S. Peyton Jones. Associated type synonyms. In *POPL '05: The 32nd ACM Symposium on Principles of Programming Languages*, pages 241–253, 2005.
- [29] M. M. T. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell array codes with multicore GPUs. In *DAMP '11: The 6th Workshop on Declarative Aspects of Multicore Programming*, Jan. 2011.
- [30] S. Chatterjee and J. Prins. *COMP663: Parallel Computing Algorithms*. Department of Computer Science, University of North Carolina at Chapel Hill, 2009.
- [31] S. Chatterjee, G. E. Blelloch, and M. Zagha. Scan primitives for vector computers. In *SC '90: The 1990 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 666–675, 1990.
- [32] K. Claessen, M. Sheeran, and B. J. Svensson. Expressive Array Constructs in an Embedded GPU Kernel Programming Language. In *DAMP '12: The 7th Workshop on Declarative Aspects of Multicore Programming*, pages 21–31, 2012.
- [33] R. Clifton-Everest, T. L. McDonell, M. M. T. Chakravarty, and G. Keller. Embedding Foreign Code. In *PADL '14: The 16th International Symposium on Practical Aspects of Declarative Languages*, Jan. 2014.
- [34] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. The MIT Press, 1989.
- [35] A. Collins, D. Grewe, V. Grover, S. Lee, and A. Susnea. NOVA: A Functional Language for Data Parallelism. Technical Report NVR-2013-002, NVIDIA Corporation, July 2013.
- [36] Continuum Analytics. Anaconda Accelerate, 2013. URL <https://store.continuum.io/cshop/accelerate/>.
- [37] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: from lists to streams to nothing at all. In *ICFP '07: The 12th ACM SIGPLAN international conference on Functional programming*, pages 315–326, 2007.
- [38] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

- [39] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli. Fast scan algorithms on graphics processors. In *ICS '08: The 22nd International Conference on Supercomputing*, pages 205–213, 2008.
- [40] C. C. Dyer and P. S. S. Ip. Softening in N-body simulations of collisionless systems. *The Astrophysical Journal*, 409(1):60–67, June 1993.
- [41] S. Eker. Associative-commutative matching via bipartite graph matching. *The Computer Journal*, 38(5):381–399, Jan. 1995.
- [42] S. Eker. Single Elementary Associative-Commutative Matching. *Journal of Automated Reasoning*, 28(1):35–51, 2002.
- [43] S. Eker. Associative-Commutative Rewriting on Large Terms. In *RTA '03: The 14th international conference on Rewriting Techniques and Applications*, pages 14–29, June 2003.
- [44] C. Elliott. Functional Images. In *The Fun of Programming*. Palgrave, 2003.
- [45] C. Elliott. Programming graphics processors functionally. In *Haskell '04: The 2004 ACM SIGPLAN workshop on Haskell*, Sept. 2004.
- [46] J. Ellson, E. Gansner, A. E. Lefohn, S. North, and G. Woodhull. Graphviz — open source graph drawing tools, 2001. URL <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.87.9389>.
- [47] A. Gill. *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, Department of Computer Science, University of Glasgow, 1996.
- [48] A. Gill. Type-Safe Observable Sharing in Haskell. In *Haskell '09: The 2nd ACM SIGPLAN Symposium on Haskell*, pages 117–128, Sept. 2009.
- [49] A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In *FPCA '93: The 1993 conference on Functional programming languages and computer architecture*, pages 223–232, 1993.
- [50] A. Gill, T. Bull, G. Kimmell, E. Perrins, E. Komp, and B. Werling. Introducing Kansas Lava - Springer. In *IFL '11: The 23rd International Symposium on Implementation and Application of Functional Languages*, pages 18–35, 2011.
- [51] C. Grelck, K. Hinckfuß, and S.-B. Scholz. With-Loop Fusion for Data Locality and Parallelism. In *IFL '05: The 17th International Symposium on Implementation and Application of Functional Languages*, pages 178–195, 2006.

- [52] L. J. Guibas and D. K. Wyatt. Compilation and Delayed Evaluation in APL. In *POPL '78: The 5th ACM symposium on Principles of Programming Languages*, pages 1–8, 1978.
- [53] K. Gupta, J. A. Stuart, and J. D. Owens. A study of Persistent Threads style GPU programming for GPGPU workloads. In *InPar '12: Innovative Parallel Computing*, pages 1–14. IEEE, May 2014.
- [54] M. Harris. Optimizing Parallel Reduction in CUDA. In *CUDA SDK*, 2007.
- [55] M. Harris and M. Garland. Optimizing Parallel Prefix Operations for the Fermi Architecture. In *GPU Computing Gems: Jade Edition*, pages 29–38. Morgan Kaufmann, Oct. 2012.
- [56] F. Henglein and R. Hinze. Sorting and Searching by Distribution: From Generic Discrimination to Generic Tries. In *Programming Languages and Systems*, pages 315–332. Springer, Dec. 2013.
- [57] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector SIMD architectures. In *ICS '13: The 27th International Conference Supercomputing*, pages 13–24, June 2013.
- [58] High Performance Fortran Forum. High Performance Fortran Language Specification version 2.0. Rice University, Jan. 1997.
- [59] J. Hoberock and N. Bell. Thrust: A Parallel Template Library.
- [60] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple data traversals. In *ICFP '97: The 2nd ACM SIGPLAN International Conference on Functional Programming*, 1997.
- [61] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es):196, 1996.
- [62] T. Jansen. *GPU++: An Embedded GPU Development System for General-Purpose Computations*. PhD thesis, Technische Universität München, 2008.
- [63] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *MSPC '06: The 2006 workshop on Memory System Performance and Correctness*, pages 51–60, Oct. 2006.
- [64] I. Karlin, A. Bhatele, B. L. Chamberlain, J. Cohen, Z. Devito, M. Gokhale, R. Haque, R. Hornung, J. Keasler, D. Laney, E. Luke, S. Lloyd, J. McGraw, R. Neely, D. Richards, M. Schulz, C. H. Still, F. Wang, and D. Wong. LULESH Programming Model and Performance Ports Overview. Technical Report LLNL-TR-608824, Dec. 2012.

- [65] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. Devito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. Still. Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application. In *IPDPS '13: The 2013 IEEE International Symposium on Parallel and Distributed Processing*, Boston, USA, 2013.
- [66] G. Keller. *Transformation-based Implementation of Nested Data Parallelism for Distributed Memory Machines*. PhD thesis, Technische Universität Berlin, Fachbereich Informatik, 1999.
- [67] G. Keller and M. M. T. Chakravarty. On the Distributed Implementation of Aggregate Data Structures by Program Transformation. In J. e. Rolim, editor, *IPPS/SPDP '99: The 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 108–122, 1999.
- [68] G. Keller, M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *ICFP '10: The 15th ACM SIGPLAN international conference on Functional programming*, pages 261–272, Sept. 2010.
- [69] A. Kennedy. Compiling with continuations, continued. In *ICFP '07: The 12th ACM SIGPLAN international conference on Functional programming*, Oct. 2007.
- [70] H. Kirchner and M. Pierre-Etienne. Promoting rewriting to a programming language: a compiler for non-deterministic rewrite programs in associative-commutative theories. *Journal of Functional Programming*, 11(02):207–251, Mar. 2001.
- [71] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Computing*, 38(3), Mar. 2012.
- [72] B. Larsen. Simple Optimizations for an Applicative Array Language for Graphics Processors. In *DAMP '11: The 6th Workshop on Declarative Aspects of Multicore Programming*, Jan. 2011.
- [73] M. Lesniak. PASTHA: Parallelizing stencil calculations in Haskell. In *DAMP '10: The 5th Workshop on Declarative Aspects of Multicore Programming*, pages 5–14, 2010.
- [74] G. M. Levin and L. Nyland. An Introduction to Proteus. Technical Report TR95-025, Clarkson University and University of North Carolina, Aug. 1994.
- [75] B. Lippmeier and G. Keller. Efficient parallel stencil convolution in Haskell. In *Haskell '11: The 4th ACM SIGPLAN Symposium on Haskell*, Sept. 2011.

- [76] B. Lippmeier, M. M. T. Chakravarty, G. Keller, and S. Peyton Jones. Guiding parallel array fusion with indexed types. In *Haskell '12: The 5th ACM SIGPLAN Symposium on Haskell*, Sept. 2012.
- [77] B. Lippmeier, M. M. T. Chakravarty, G. Keller, and A. Robinson. Data Flow Fusion with Series Expressions in Haskell. In *Haskell '13: The 6th ACM SIGPLAN Symposium on Haskell*, Sept. 2013.
- [78] W. Ma and G. Agrawal. An integer programming framework for optimizing shared memory use on GPUs. In *HiPC '10: The International Conference on High Performance Computing*, pages 1–10, Dec. 2010.
- [79] F. M. Madsen and A. Filinski. Towards a streaming model for nested data parallelism. In *FHPC '13: The 2nd ACM SIGPLAN Workshop on Functional High-Performance Computing*, pages 13–24, Sept. 2013.
- [80] G. Mainland. Why it's nice to be quoted. In *Haskell '07: The 2007 ACM SIGPLAN workshop on Haskell*, page 73, 2007.
- [81] G. Mainland and G. Morrisett. Nikola: Embedding Compiled GPU Functions in Haskell. In *Haskell '10: The 3rd ACM SIGPLAN Symposium on Haskell*, 2010.
- [82] G. Mainland, R. Leshchinskiy, and S. Peyton Jones. Exploiting vector instructions with generalized stream fusion. In *ICFP '13: The 18th ACM SIGPLAN international conference on Functional programming*, pages 37–48, Nov. 2013.
- [83] S. Marlow. *Parallel and Concurrent Programming in Haskell*. O'Reilly Media, first edition, July 2013.
- [84] K. Matsuzaki and K. Emoto. Implementing Fusion-Equipped Parallel Skeletons by Expression Templates. In *IFL '09: The 21st international Symposium on Implementation and application of functional languages*, pages 72–89. Springer Berlin Heidelberg, 2011.
- [85] C. McBride. Epigram: Practical Programming with Dependent Types. In *AFP '04: The 5th international conference on Advanced Functional Programming*, pages 130–170, 2005.
- [86] C. McBride. Type-Preserving Renaming and Substitution. *Journal of Functional Programming*, 2006.
- [87] M. D. McCool and S. Du Toit. *Metaprogramming GPUs with Sh*. A. K. Peters/CRC Press, Aug. 2004.
- [88] M. D. McCool, S. Du Toit, T. Popa, B. Chan, and K. Moule. Shader algebra. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 787–795, 2004.

- [89] T. L. McDonell, M. M. T. Chakravarty, G. Keller, and B. Lippmeier. Optimising Purely Functional GPU Programs. In *ICFP '13: The 18th ACM SIGPLAN international conference on Functional programming*, Sept. 2013.
- [90] S. Mehta, P.-H. Lin, and P.-C. Yew. Revisiting loop fusion in the polyhedral framework. In *PPoPP '14: The 19th ACM Symposium on Principles and Practice of Parallel Programming*, Feb. 2014.
- [91] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *The 5th conference on Functional programming languages and computer architecture*, pages 124–144, 1991.
- [92] D. G. Merrill and A. S. Grimshaw. Parallel Scan for Stream Architectures. Technical Report CS2009-14, Department of Computer Science, University of Virginia, Dec. 2009.
- [93] D. G. Merrill and A. S. Grimshaw. High Performance and Scalable Radix Sorting. *Parallel Processing Letters*, 21(2):245–272, June 2011.
- [94] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard version 3.0. High Performance Computing Center Stuttgart (HLRS), Sept. 2012.
- [95] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965.
- [96] T. Muranushi. Paraiso: an automated tuning framework for explicit solvers of partial differential equations. *Computational Science & Discovery*, 5(1):015003, Jan. 2012.
- [97] NVIDIA. *Parallel Thread Execution ISA*, 3.1 edition, Sept. 2012.
- [98] NVIDIA. CUDA C Programming Guide. Technical Report PG-02829-001\_v5.0, NVIDIA Corporation, Oct. 2012.
- [99] OpenMP Architecture Review Board. OpenMP Application Program Interface, v3.0, May 2008.
- [100] S. Peyton Jones, editor. *Haskell 98 Languages and Libraries: The Revised Report*. Haskell 98 Languages and Libraries. Cambridge University Press, Apr. 2003.
- [101] S. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4-5):393–434, July 2003.
- [102] S. Peyton Jones, S. Marlow, and C. Elliott. Stretching the Storage Manager: Weak Pointers and Stable Names in Haskell. In *IFL '99: The 11th International Symposium on Implementation and Application of Functional Languages*, pages 37–58, Sept. 1999.

- [103] S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell '01*, pages 203–233, 2001.
- [104] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ICFP '06: The 11th ACM SIGPLAN international conference on Functional programming*, pages 50–61, 2006.
- [105] S. Peyton Jones, R. Leshchinskiy, G. Keller, and M. M. T. Chakravarty. Harnessing the Multicores: Nested Data Parallelism in Haskell. In R. Hariharan, M. Mukund, and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, Oct. 2008.
- [106] W. Pfaffenstiel. Piecewise Execution of Nested Parallel Programs - A Thread-Based Approach. In *Euro-Par '99: Parallel Processing*, pages 445–448, 1999.
- [107] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache. Loop transformations: convexity, pruning and optimization. In *POPL '11: The 38th ACM symposium on Principles of Programming Languages*, Jan. 2011.
- [108] J. F. Prins, S. Chatterjee, and M. Simons. Irregular Computations in Fortran – Expression and Implementation Strategies. *Scientific Programming*, 7(3):313–326, 1999.
- [109] K. H. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, June 1998.
- [110] R. Rivest. The MD5 Message-Digest Algorithm, Apr. 1992.
- [111] A. Robinson, B. Lippmeier, and G. Keller. Fusing Filters with Integer Linear Programming. In *FHPC '14: The 3rd ACM SIGPLAN Workshop on Functional High-Performance Computing*, Sept. 2014.
- [112] T. Rompf, A. K. Sujeeth, N. Amin, K. J. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs: new directions for extensible compilers based on staging. In *POPL '13: The 40th ACM symposium on Principles of programming languages*, Jan. 2013.
- [113] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: A compiler and runtime for heterogenous sytems. In *SOSP '13: The 24th ACM Symposium on Operating Systems Principles*, pages 49–68, 2013.
- [114] A. Rubinsteyn, E. Hielscher, N. Weinman, and D. Shasha. Parakeet: a just-in-time parallel accelerator for python. In *HotPar '12: The 4th USENIX conference on Hot Topics in Parallelism*, June 2012.

- [115] V. Sarkar and G. R. Gao. Optimization of array accesses by collective loop transformations. In *ICS '91: The 5th International Conference on Supercomputing*, pages 194–205, 1991.
- [116] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for many-core GPUs. In *IPDPS '09: The 2009 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–10, May 2009.
- [117] S. Sato and H. Iwasaki. A Skeletal Parallel Framework with Fusion Optimizer for GPGPU Programming. In *APLAS '09: the 7th Asian Symposium on Programming Languages and Systems*, pages 79–94, 2009.
- [118] T. Schrijvers, S. Peyton Jones, M. M. T. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *ICFP '08: The 13th ACM SIGPLAN international conference on Functional programming*, pages 51–62, 2008.
- [119] J. T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2(4):484–521, Oct. 1980.
- [120] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *GH '07: The 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics Hardware*, pages 97–106, 2007.
- [121] S. Sengupta, M. Harris, and M. Garland. Efficient Parallel Scan Algorithms for GPUs. Technical Report NVR-2008-003, NVIDIA Corporation, Dec. 2008.
- [122] T. Sheard and S. Peyton Jones. Template meta-programming for Haskell. *ACM SIGPLAN Notices*, 37(12):60–75, Dec. 2002.
- [123] J. Stam. Stable fluids. In *SIGGRAPH '99: The 26th annual conference on Computer graphics and interactive techniques*, July 1999.
- [124] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, H. Chafi, M. Wu, A. R. Atreya, M. Odersky, and K. Olukotun. OptiML: An implicitly parallel domain-specific language for machine learning . In *ICML '11: The 28th International Conference on Machine Learning*, June 2011.
- [125] A. K. Sujeeth, T. Rompf, K. J. Brown, H. Lee, H. Chafi, V. Popic, M. Wu, A. Prokopec, V. Jovanovic, M. Odersky, and K. Olukotun. Composition and Reuse with Compiled Domain-Specific Languages. In *ECOOP '13: The 27th European Conference on Object-Oriented Programming*, pages 52–78, 2013.
- [126] J. D. Svenningsson and E. Axelsson. Combining Deep and Shallow Embedding for EDSL. In *Trends in Functional Programming*, pages 21–36. 2013.

- [127] B. J. Svensson. *Embedded Languages for Data-Parallel Programming*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden, 2013.
- [128] B. J. Svensson, M. Sheeran, and K. Claessen. Obsidian: A domain specific embedded language for parallel programming of graphics processors. In *IFL '08: The 20th International Symposium on Implementation and Application of Functional Languages*, 2008.
- [129] T. Sweeney. The End of the GPU Roadmap. In *High Performance Graphics*, Aug. 2009.
- [130] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *FPCA '95: The 7th international conference on Functional programming languages and computer architecture*, pages 306–313, Oct. 1995.
- [131] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *CC '02: The 11th International Conference on Compiler Construction*, pages 179–196, Mar. 2002.
- [132] A. Tzannes, G. C. Caragea, R. Barua, and U. Vishkin. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. In *PPoPP '10: The 15th ACM Symposium on Principles and Practice of Parallel Programming*, pages 179–190, Jan. 2010.
- [133] A. Venkat, M. Shantharam, M. Hall, and M. M. Strout. Non-affine Extensions to Polyhedral Code Generation. In *CGO '14: The 12th IEEE/ACM International Symposium on Code Generation and Optimization*, Feb. 2014.
- [134] P. Wadler. Applicative style programming, program transformation, and list operators. In *FPCA '81: The 1981 conference on Functional programming languages and computer architecture*, Oct. 1981.
- [135] P. Wadler. Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time. In *LFP '84: The 1984 ACM Symposium on LISP and functional programming*, Aug. 1984.
- [136] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, June 1990.
- [137] J. Warren. A hierarchical basis for reordering transformations. In *POPL '84: The 11th ACM symposium on Principles of Programming Languages*, pages 272–282, 1984.
- [138] R. C. Waters. Automatic transformation of series expressions into loops. *ACM Transactions on Programming Languages and Systems*, 13(1):52–98, Jan. 1991.

- [139] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178–194, 2009.
- [140] L. Xu and J. W. L. Wan. Real-time intensity-based rigid 2D-3D medical image registration using RapidMind Multicore Development Platform. In *EMBS '08: The 30th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 5382–5385, Aug. 2008.
- [141] E. Z. Zhang, Y. Jiang, Z. Guo, and X. Shen. Streamlining GPU applications on the fly. In *ICS '10: The 24th International Conference on Supercomputing*, pages 115–126, 2010.
- [142] Y. Zhang and F. Mueller. CuNesl: Compiling Nested Data-Parallel Languages for SIMT Architectures. In *ICPP '12: The 41st International Conference on Parallel Processing*, pages 340–349, Sept. 2012.

---

## Index

---

- algorithmic skeleton, 2, 24, 44–50
- antiquotation, 45, 46
- AoS (array-of-struct), 50
- array
  - delayed, 47
  - manifest, 47
- AST (abstract syntax tree), 21, 46, 85
- cache
  - allocation, 69–71
  - data, 68–69
  - kernel, 62–65
- code generation
  - array indexing, 57–58
  - fusion, consumer, 46–48
  - shapes, 55–56
  - sharing, 52–53
  - tuples, 50–51
- complexity
  - step, 9, 10
  - time, 9
  - work, 9
- congruence, 36
- convolution, 16–18
- CUDA, 2, 6–20, 24
  - `--global__`, 7, 45, 74
  - `--syncthreads()`, 7
  - event, 78
  - kernel, 7, 45
  - shared memory, 7, 11, 17, 135
- stream, 78
- streaming multiprocessor, 7
- texture reference, 58
- thread block, 7, 10, 11, 17, 75
- thread occupancy, 74–76
- warp, 7, 19, 76
- data parallelism, 5
- de Bruijn, 32–41, 95
- device, the, *see* GPU
- DSL, *see* language: domain-specific
- EDSL, *see* language: embedded
- fusion, 8, 12, 18, 24
  - consumer, 91, 101–103
  - delayed array, 94, 113
  - producer, 91, 96–101
  - shortcut, 112
- GPGPU (general-purpose computing on GPUs), 1, 2, 6, 24
- GPU (graphics processing unit), 1, 2, 6–20, 23, 24, 26, 27
- HOAS (higher-order abstract syntax), 31, 32, 37, 87
- injective, 13
- language
  - domain-specific, 20
  - embedded, 20, 21, 26–28, 40

host, 20, 21  
internal, *see* language: embedded  
meta, *see* language: object  
object, 21, 25, 40  
stratified, 28, 37  
target, 25

non-parametric representation, 31, 50–52  
nursery, *see* cache: allocation

partial, 13

quasiquotation, 45, 46, 51, 58

rank-polymorphic, 24, 26, 29, 55  
rasterisation, 1, 6

sharing, 52, 85  
    recovery, 87–89  
shrinking, 108  
skeleton, *see* algorithmic skeleton  
SoA (struct-of-array), 50  
stable name, 87, 88  
stencil, 16–18  
surjective, 13

tuple, 31

type  
    equality, 34–36  
    representation, 31  
    surface, 31

weakening, 37, 40



