

Submitted in part fulfilment for the degree of BSc.

Functional GPU Programming: A Comparison between Obsidian and Accelerate

Alberto Sadde

28th April 2015

Supervisor: Colin Runciman

Number of words = 15727 , as counted by `texcount -inc -sum -1 report.tex`.
This includes the body of the report only.

Abstract

Graphics Processing Units (GPUs) can process large amounts of similar data much faster than current CPUs, but their different architecture makes them more difficult to be programmed. The languages currently used to program them (such as CUDA) do not provide any of the abstraction mechanisms offered by most modern languages. Functional languages like Haskell offer such mechanisms. In this report, we compare Accelerate and Obsidian, two domain-specific languages embedded in Haskell that produce CUDA code. We discuss how features unique to functional languages enable Accelerate and Obsidian to raise the level of abstraction and move away from the imperative programming model currently used. We compare the speed of both languages against hand-tuned CUDA by benchmarking four different programs: parallel reduce, matrix multiplication, Mandelbrot sets and sorting. We also compare the compilation and automation mechanisms of both Accelerate and Obsidian, their expressiveness, control over the GPU hardware and safety features. Although both languages solve the abstraction issues and enable us to describe GPU computations in a more simple and concise way, they are still far behind hand-optimised CUDA programs in terms of performance. Nevertheless, our experience with both languages, gives us hope that Accelerate and Obsidian will be able to become competitive languages in the GPU arena.

Contents

1	Introduction	6
1.1	Project Aims	6
1.2	Advice for the Reader	7
1.3	Statement of Ethics	7
2	Literature Review	8
2.1	Functional Programming: A Quick Review	8
2.1.1	Purity	8
2.1.2	Higher-Order Functions	8
2.1.3	Laziness	9
2.1.4	Types and Type-Classes	9
2.2	Parallelism	11
2.2.1	Data Parallelism	11
2.3	General Purpose GPU Programming	11
2.3.1	The Evolution of the GPU	11
2.3.2	The Modern GPU	12
2.3.3	CUDA Programming	13
2.4	Domain-Specific Languages	15
2.4.1	Types of DSLs	15
2.4.2	EDSLs	16
2.4.3	Types of Embedding	17
2.5	Related Work	18
2.5.1	Comparing Programming Languages for Parallel Processing	18
2.5.2	Similar Languages	18
2.6	Summary	19
3	Accelerate and Obsidian	20
3.1	Obsidian	20
3.1.1	Overview of the Obsidian Implementation	20
3.1.2	A Language to Manipulate Arrays	21
3.1.3	Programming the GPU Hierarchy	22
3.1.4	Obsidian Code Generation and Execution	23
3.2	Accelerate	24
3.2.1	The Front-End	25
3.2.2	The CUDA Back-End	26
3.3	Summary	27
4	Comparing Everything But Speed	28
4.1	Compilation and Automation	28
4.2	Control of CUDA Internals	29
4.3	Language Expressiveness	30
4.3.1	Manipulating Arrays	30
4.4	Safety	31
4.5	Summary	32

5	Benchmarking Accelerate and Obsidian	34
5.1	The Set-Up	34
5.1.1	A Note on Installation	34
5.1.2	Benchmarking Lazy Programs	35
5.2	Parallel Reduce	35
5.2.1	Reduce in Accelerate	36
5.2.2	Reduce in Obsidian	36
5.2.3	Reduce in Hand-coded CUDA	37
5.2.4	Reduce Benchmarks	37
5.3	Matrix Multiplication	40
5.3.1	Matrix Multiplication in Accelerate	40
5.3.2	Matrix Multiplication in Obsidian	41
5.3.3	Matrix Multiplication in Hand-coded CUDA	42
5.3.4	Matrix Multiplication Benchmarks	44
5.4	Mandelbrot Sets	45
5.4.1	Mandelbrot Sets in Accelerate	47
5.4.2	Mandelbrot Sets in Obsidian	47
5.4.3	Mandelbrot Sets Results	48
5.5	Sorting	48
5.5.1	Sorting in Accelerate: Radix Sort	49
5.5.2	Sorting in Obsidian: Sorting Network	49
5.5.3	Sorting in CUDA: The Thrust Library	50
5.5.4	Sorting Benchmarks	50
5.6	Language Tweaks and Bugs	51
5.6.1	Obsidian Tweaks	51
5.6.2	Bug in Accelerate	51
5.7	Summary	52
6	Conclusion and Further Work	53
6.1	What we Found	53
6.1.1	Performance	53
6.1.2	The Languages	54
6.2	Future Work	54
6.2.1	Improving our Work	54
6.2.2	Improving the Languages	55
	Appendices	55
A	Automatically Generated Code	56
A.1	VHDL Code Produced by Lava	56
A.2	CUDA Source Code Produced by Accelerate and Obsidian	57
A.2.1	Parallel Reduce in CUDA produced by Accelerate	57
A.2.2	Parallel Reduce in CUDA produced by Obsidian	59
A.2.3	Matrix Multiplication in CUDA produced by Accelerate	62
A.2.4	Matrix Multiplication in CUDA produced by Obsidian	66
B	CUDA Device Information	67
C	Email Correspondence	68

1 Introduction

Graphics processing units (GPUs) are computer devices that pack together thousands of cores suited for single-instruction multiple-data (SIMD) parallelism. In the last 20 years these devices changed from chips devoted to a very particular task – that of rendering graphics for computer screens – to powerful devices that every day find new applications. From medical imaging [1] to weather forecasting [2], GPUs are used everywhere. Nevertheless, this transformation was not smooth. Until recently, using these devices for general-purpose tasks was very complex: programmers needed to translate their programs into graphics computations, mapping their functions to graphics primitives, in order for them to run on the GPU.

With the introduction of programming tools such as Accelerator [3] and later CUDA [4], which provides a dialect of the C programming language to target Nvidia GPUs [5], general purpose GPU programming (GPGPU) has seen a surge in popularity. Nonetheless the process of writing such programs is still not ideal. Programmers need to take care of low-level operations such as memory allocations and data transfers.

An approach to simplify the writing of such programs comes from functional programming languages. These languages approach programming from a different perspective. Here the developer doesn't write an imperative program specifying how to perform a computation, instead she writes functions and declarative expressions that describe what the program needs to do. This different approach abstracts away many low-level details and lets the programmer focus on developing the algorithms she needs.

In this report we investigate Obsidian and Accelerate, two languages embedded in the functional language Haskell that attempt to abstract away from CUDA's low-level constructs by exploiting features unique to functional languages.

1.1 Project Aims

In this dissertation we analysed Accelerate and Obsidian from the programmer's point of view. We identified meaningful and measurable aspects of both languages (Table 4.1) such as their safety features, stability and expressiveness.

We also measured performance. We identified four tasks with different characteristics and compared their Accelerate and Obsidian implementations against hand-written and optimised CUDA versions. By taking this approach we aimed at comparing as fully as possible the speed of the three languages.

Other than our initial aims, we hope that the results and discussion shown in this report, will be valuable for developers considering writing powerful GPU programs without the difficulties of common GPU programming languages. Although in its early stages of development, we believe that functional programming has a bright future in the GPU realm.

1.2 Advice for the Reader

The report is divided into the following chapters:

- **Literature Review (Chapter 2):** This chapter surveys the existing literature on the subject. We introduce the main features of Haskell that are relevant to Obsidian and Accelerate. We introduce parallelism focusing on data-parallelism. Then we describe how GPUs evolved to become the powerful devices that we have today. We also introduce the basic concepts of CUDA. We then introduce domain-specific languages. In particular we describe embedded domain-specific languages. We end the chapter with a survey of related languages and projects similar to ours.
- **Accelerate and Obsidian (Chapter 3):** In this chapter we review the implementations of Accelerate and Obsidian. We introduce the main functions of both languages that we will use in our case studies in Chapter 5.
- **Comparing Everything But Speed (Chapter 4):** We compare the features of both Accelerate and Obsidian. The aims here are to show the powers and weaknesses of both languages without comparing their speed.
- **Benchmarking Accelerate and Obsidian (Chapter 5):** This chapter presents the implementation of a small selection of case studies. We benchmark four programs: parallel reduce (Section 5.2), matrix multiplication (Section 5.3), Mandelbrot sets (Section 5.4) and sorting (Section 5.5). We focus on measuring and comparing the performance of Accelerate, Obsidian and hand-written CUDA.
- **Conclusion and Further Work (Chapter 6):** In this chapter we review our findings and suggest improvements to both Accelerate and Obsidian in the light of the results obtained in Chapters 4 and 5.

The reader with experience in Haskell or GPUs can skip part or all of Chapter 2. In order to get a glimpse at what Accelerate and Obsidian offer, the reader should at least go through Chapters 4 and 5 using as reference Chapter 3.

1.3 Statement of Ethics

This project deals with experimental programming languages. No human participants other than the author were required for these experiments. Moreover all the software used is in the public domain. We encountered no ethical concerns.

2 Literature Review

In this chapter we survey and critique the existing literature on topics relevant to our project. We will describe the main concepts needed to support our discussion in Chapters 3, 4 and 5.

In Section 2.1 we briefly describe the aspects of Haskell that are relevant to our project. In Section 2.2 we introduce data-parallelism. In Section 2.3 we give a general view of GPUs and general purpose GPU programming. Section 2.4 introduces DSLs focusing on embedded domain-specific languages. Finally in Section 2.5 we survey projects similar to ours and languages related to Accelerate and Obsidian.

2.1 Functional Programming: A Quick Review

In what follows we use Haskell syntax unless otherwise stated. We assume that the reader has at least a general knowledge of functional programming and Haskell. Please see [6], [7] or [8] for more information about Haskell and functional languages.

2.1.1 Purity

In programming languages, computational results that change some state of the program in an observable way are called side-effects. Consider the function shown in Figure 2.1, written in C, with side effects. This function assigns the input value `in` to the output value `out` and alters the value of the variable `globalVar`, which alters the state of the program.

```
1 void functionWithSideEffects(*int in, *int out) {  
2     out = in;  
3     globalVar -= 1; //some global variable declared elsewhere  
4 }
```

Figure 2.1: A C function with side-effects.

Languages such as Haskell forbid the use of functions with side-effects. This freedom from side-effects enables us to reason about programs in a more modular way and reuse code more safely. In a pure language a function applied to a specific argument always returns the same result. This is known as *referential transparency*.

2.1.2 Higher-Order Functions

Functions that take other functions as arguments, or return functions as results, are known as *higher-order functions*. In functional languages, functions are first-class citizens enabling us to use them as arguments and results of other functions. Consider Figure 2.2, `map` takes a function as argument and applies the function to each element of its second argument, a list. Similarly, `filter` takes a function as its first argument and filters a list according to the function applied to each element of the list.


```

1 map :: (a → b) → [a] → [b]
2 map f []          = []
3 map f (x:xs)      = f x : map f xs
4
5 filter :: (a → Bool) → [a] → [a]
6 filter pred []     = []
7 filter pred (x:xs) =
8   | pred x         = x : filter pred xs
9   | otherwise      = filter pred xs

```

Figure 2.2: The higher-order functions `map` and `filter`.

This flexibility of functions increases the possibilities to reuse them and write more modular code. Both Obsidian and Accelerate rely on this property to compose together programs that generate reliable GPU functions, as we will see in Chapter 3.

2.1.3 Laziness

A language is called *lazy* if its expressions are only evaluated when their results are needed. This means that when an expression is bound to a variable, its result will not be computed until some other computation needs it. In Figure 2.3 we show how laziness lets us work with “infinite lists”.

```

1 -- Definition of take
2 take :: Int → [a] → [a]
3 take n _ | n <= 0 = []
4 take _ []         = []
5 take n (x:xs)     = x : take (n-1) xs
6
7 > take 10 (map (+2) [1..])
8 [3,4,...,12]

```

Figure 2.3: Working on infinite lists by exploiting laziness.

In a language without laziness, the entire result of the `map` application in line 7 would need to be evaluated first, so that `take` would never be applied. In Haskell, because of laziness, `map` creates only as many list-elements as needed by `take` so that the whole operation will consist in only adding two to the first ten elements of the list and returning those.

Even though with laziness we can operate over such “infinite lists”, laziness can lead to space leaks if used carelessly. Moreover, if not understood correctly, laziness can lead to strange results when benchmarking functional programs. As we will see in Section 5.1.2, we need to take care that expressions are fully evaluated when timing our case studies.

2.1.4 Types and Type-Classes

Types let us manipulate and group together similar kinds of data. Most modern languages provide predefined types such as `Int` or `Bool` and ways in which we can define our own types.

In Haskell there are three ways in which we can declare new types as shown in Figure 2.4. We define a new *algebraic type* using the `data` keyword. We introduce *type synonyms* with `type`. A type synonym lets us introduce a more meaningful name for an existing type: it is a purely syntactic way of making our code more readable [9]. The keyword `newtype` also enables us to give old types a new identity [8].

```

1 data Tree      = Leaf Int | Branch Tree Tree
2 type String    = [Char]
3 newtype Vector = Vector Int

```

Figure 2.4: Some type declarations in Haskell.

Every expression in Haskell has a static type. This enables the compiler to type-check the programs and catch many bugs before runtime. In Chapter 3 and Section 4.4, we will see how by exploiting the type-system and ensuring that all DSL expressions are also well-typed Haskell expressions, Obsidian and Accelerate are able to catch many errors before compile-time and forbid possibly unsafe operations that other GPU programming languages might permit.

Parametric polymorphism. Types may be parametrised by other types. We do this by introducing a type-variable. In Figure 2.5, the `Tree` declaration, introduced in Figure 2.4, is generalised to work on types other than `Int` by exploiting this type of polymorphism.

```

1 data Tree a = Leaf a | Branch (Tree a) (Tree a)

```

Figure 2.5: A polymorphic version of the `Tree` type.

Type-Classes are considered “*Haskell’s most distinctive characteristic*” [10]. They enable us to do *ad hoc polymorphism*, enabling us to implement the same operation over different types [11].

```

1 class Eq a where
2   (==), (/=) :: a -> a -> Bool
3   x /= y     = not (x == y)
4   x == y     = not (x /= y)
5
6 instance Eq a => Eq (Tree a) where
7   Leaf a      == Leaf b      = a == b
8   (Branch a b) == (Branch c d) = a == c && b == d
9   _           == _           = False

```

Figure 2.6: The automatically derived instance `Eq (Tree a)`.

Type-classes can be seen as Java-like *interfaces* or as a contract that types must respect. In Haskell, in order for types to overload operations, we need to make the type an instance of a type-class also known simply as class¹. Consider Figure 2.6. We show the *derived*² instance of the equality class `Eq` for the `Tree` type that we defined in Figure 2.5. A type-class is defined with `class` and it specifies all the functions that a type needs to implement in order to become an instance of the class. In lines 1-4, we show the definition of `Eq`. We make our type an instance of the class using the keyword `instance` after which we define the implementation of the required functions (lines 6-9). In Figure 2.6 we could have only defined one of the functions (e.g. `(==)`) as the compiler is smart enough to deduce the other from the class’ definition.

Types and type-classes will play an important role in the implementation of Accelerate and Obsidian. In Chapters 3 and 4 we will see how, by overloading many predefined operators, expressions in Obsidian and Accelerate look almost like idiomatic Haskell expressions.

¹Not to be confused with classes in object-oriented languages.

²A derived instance is one generated by the compiler on request. Several Haskell classes can be instantiated this way.

2.2 Parallelism

In this section we give a brief description of parallelism. We focus on data parallelism, a programming model that perfectly matches the GPU hardware and programming model.

Parallel computing consists of breaking down a large program into smaller parts that can be executed *simultaneously*, possibly, on different processors [12]. There exist different kinds of parallelism such as task-parallelism and data-parallelism. Writing parallel programs is not easy. We need to take care of potential race conditions and data-dependencies so we need more complex control structures to synchronise the execution of the different threads and avoid such races and dependencies.

2.2.1 Data Parallelism

Data parallelism is a form of parallelism where operations are carried out in parallel over a collection of data of similar kind [13]. We apply the *same* operation to different values of the same kind. This type of parallelism is *deterministic* [13]. Similar to referential transparency (see Section 2.1.1), given the same inputs, data parallel programs yield the same result. This makes functional languages a good match for this programming model as we will see in Chapter 5 where we implement and compare a small set of examples.

Data parallelism is interesting for the following reasons:

- Scalable model: This type of parallelism is independent of the platform used. Since large data sets are handled uniformly [13], it is easy to split the sets into the right number of chunks to occupy all the available processors.
- Simple control structure: Given the uniformity of operations over the sets of data, this type of parallelism requires little to no control structures. In Section 2.3 we will see that CUDA provides only one explicit control structure while the rest of the synchronisations are handled implicitly by the hardware.

The increasing need for computing power has prompted manufacturers to find new ways in which to increase the powers of their processing units. It is now common for CPUs to pack multiple cores on a single chip in an attempt to increase their power. Thanks to the characteristics of data parallelism, that we just described, GPUs have found applications away from graphic-intensive operations and are much faster than CPUs in many cases. In Section 2.3 we will see how this kind of parallelism matches the GPU hardware and its Single-Instruction Multiple-Data processing model.

2.3 General Purpose GPU Programming

In this section we describe the main concepts behind Graphics Processing Units as well as their evolution. We focus on CUDA a GPU programming framework developed by Nvidia to program their GPUs.

2.3.1 The Evolution of the GPU

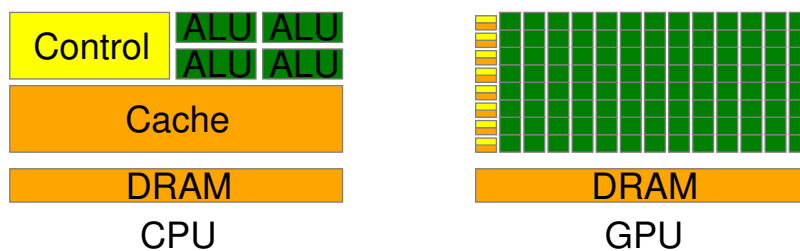
Current GPUs are massively parallel processors. GPUs were completely different devices when first introduced in the early 1980s. Early GPUs were microprocessors built around a “*graphics pipeline*” [14], dedicated to the translation of 3D shapes into 2D screen-pixels.

These early graphics processors were programmed using a set of graphics APIs such as DirectX [15] and OpenGL [16]. But while developers were demanding more and more features, the graphics pipelines remained fixed [17]. To perform non-graphics computations, programmers had to find clever ways to translate their programs into graphics operations such as vertex transformations. Programmers demanded more freedom. Graphics processors went through a series of transformations until they reached their current state:

- (1) Given that many pixel operations, such as pixel-colouring and transforming vertices positions, are data-independent, vendors started to pack together devices with more than one graphics pipeline, effectively transforming these devices into parallel processors.
- (2) In 2001, Nvidia took the first step towards a programmable GPU when it exposed certain parts of the internal instruction-set architecture of the GeForce 3 GPU to the programmer [17].
- (3) In 2006, a further step was taken by Nvidia with the introduction of the Nvidia GeForce 8800 GPU, which mapped the separate pipeline stages into an array of unified processors [17].
- (4) In 2007, the release of CUDA (Compute Unified Device Architecture) and the Tesla architecture introduced a more graphics-independent way of programming these devices [17]. The modern GPU was born.

2.3.2 The Modern GPU

The modern GPU is a many-core processor specialised for single-instruction multiple-data (SIMD) operations. It is specialised for highly parallel and compute intensive operations [4]. Compared to a CPU, the GPU emphasises memory bandwidth over latency [17]. Latency is minimised by having lots of threads running in parallel. This can be seen in Figure 2.7, the GPU devotes fewer transistors to memory than the CPU.



Orange: transistors devoted to memory. Green: transistors used for ALUs. Yellow: transistors devoted to Control units.

Figure 2.7: CPU vs. GPU: transistors devoted to memory. From [4].

A GPU consists of many multi-threaded Streaming Multiprocessors (SMs) which are composed of multiple SPs (Streaming processors) that feature fully pipelined integer and floating-point instruction sets [5]. The GPU features different types of memories such as global memory, local memory and shared memory [4]. We will explain how these types of memory affect the performance of a GPU program in Section 2.3.3.

2.3.3 CUDA Programming

CUDA is the parallel programming model developed by Nvidia to take advantage of the unified architecture of their GPUs. This programming model differentiates between the *host*, which consists of the CPU and its own memory, and the *device* that corresponds to the GPU and its separate memory [4].

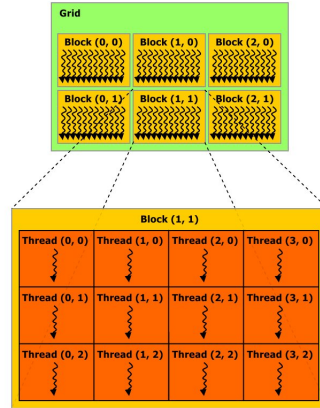


Figure 2.8: The CUDA grid of blocks of threads. From [4].

The main part of a CUDA program is the *kernel*, a function that is executed by multiple threads in parallel, on the device. Threads are grouped in blocks and blocks are divided into warps, which consist of 32 threads running in lock-step within a block. Multiple blocks form a grid. This structure is shown in Figure 2.8. From a hardware point of view, the grid corresponds to the whole GPU while a block of threads corresponds to a single SM. Each CUDA thread runs on a single SP.

This programming model works because blocks of threads are assumed to be independent and because a block is always guaranteed to run on a single SM. The scheduler can then run the blocks in any order [4]. Nevertheless, threads within a block are not guaranteed to execute the same instructions at the same time. CUDA provides a synchronization barrier, `__syncthreads()`, where threads within a block must stop and wait for the remaining threads in the block to reach the barrier before proceeding. Such synchronisations are usually necessary when performing atomic operations or when accessing shared memory.

Memory hierarchy. Given that memory plays such an important role during the execution of a GPU program, it is necessary to fully understand it. Usually a developer will only explicitly deal with the following types of memory:

- **Global Memory:** this memory corresponds to the DRAM shown in Figure 2.7. It is the most widely available and the slowest one. It is usually used to load, from CPU memory, all the necessary data for a GPU program. Current GPUs have around 4 GB of global memory³. Given that the memory bandwidth is limited, it is important to minimise and coalesce global memory accesses [4] following the “CUDA Best Practices” [18].
- **Shared Memory:** This type of memory is much faster than global memory. Only threads in the same block can access it. Current GPUs have around 48 KB of shared

³This is the amount of global memory of the GPU we used. The full details of this device are shown in Appendix B.

memory per block. Although much smaller than global memory, given its speed, *“any opportunity to replace global memory accesses by shared memory accesses should therefore be exploited”* [4]. In CUDA, shared memory is declared using the keyword `__shared__`.

In Chapter 5 we will see how these two types of memory are used to produce fast CUDA programs.

CUDA programs. Consider Figure 2.9 where we compute vector addition on the GPU. We define a kernel function using the keyword `__global__`. When the kernel is launched, CUDA assigns a unique `threadIdx` to each thread within a block. In order to uniquely identify all the threads in the kernel, we use the thread id `threadIdx` in combination with the block id `blockIdx` and the grid dimension `gridDim`. For convenience these ids are 3D vectors allowing us to model the grid of threads as a vector, matrix or volume [4].

```

1  __global__ vectorAddition(*int vec1, *int vec2,
2                             *int *result, int numElems) {
3
4      unsigned int id = threadIdx.x + blockIdx.x * gridDim.x;
5
6      if (tid < numElems) {
7          result[id] = vec1[id] + vec2[id];
8      }
9  }
```

Figure 2.9: Vector addition kernel in CUDA.

```

1  int main() {
2      ...
3      cudaMalloc((void**)&d_vec1, size);
4      cudaMalloc((void**)&d_vec2, size);
5      cudaMalloc((void**)&result, size);
6      cudaMemcpy(d_vec1, h_vec1, size, cudaMemcpyHostToDevice);
7      cudaMemcpy(d_vec2, h_vec2, size, cudaMemcpyHostToDevice);
8
9      //kernel invocation
10     vectorAddition<<<1, numElems>>>>(d_vec1, d_vec2, d_result, numElems);
11
12     //copy the result back to host
13     cudaMemcpy(h_result, d_result, size, cudaMemcpyDeviceToHost);
14
15     //free device memory
16     cudaFree(d_vec1);
17     cudaFree(d_vec2);
18     cudaFree(d_result);
19 }
```

Figure 2.10: CUDA host code for the vector addition kernel in Figure 2.9. The declarations of all the variables have been omitted.

Figure 2.9 shows only *device* code. In order to execute the kernel we also need *host* code, executed on the CPU as shown in Figure 2.10. The kernel is launched by calling the `vectorAddition` function with a pair `<<<numberOfBlocks, threadsPerBlock>>>` so that the kernel uses `numberOfBlocks × threadsPerBlock` threads. Notice how the CUDA programmer has to take care of all the low-level details of the program before running anything on the GPU. In lines 3-7 we explicitly allocate and transfer memory between the host and the device and in lines 15-18 we free all the memory used. All these statements are necessary. Forgetting one of them can lead to non-trivial errors and bugs. Both Obsidian and Accelerate abstract away from such details, not only reducing the lines of code written but also avoiding the programmer hours of bug-hunting, as we will see in Chapters 3 and 5.

2.4 Domain-Specific Languages

The term domain-specific language (DSL) is a vague one, with many different definitions and scopes [19]. For example, Backus-Naur Form constitutes, arguably, a DSL since it is built for a very specific purpose, that of defining the syntax of programming languages. Program application libraries could also be regarded as domain-specific languages, in the form of APIs. In order to avoid confusion we use the following simple definition of DSL:

“A domain-specific language is a computer programming language with limited expressiveness and focused in a particular domain.” [20]

This definition suits our current discussion. Both Obsidian and Accelerate are focused on a very specific domain (writing parallel programs for GPUs) and they are designed to express computations in this domain only.

2.4.1 Types of DSLs

Depending on the design choices and implementation techniques, we distinguish between different types of DSLs, as seen in Figure 2.11:

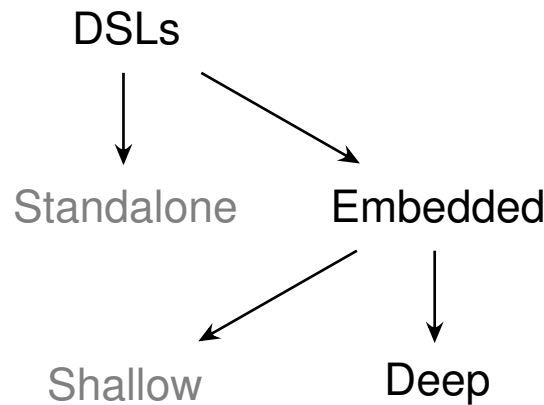


Figure 2.11: Types of DSLs. In this report we are interested in the DSLs highlighted in black.

- **Standalone DSLs.** These DSLs are similar to general-purpose languages. They have to be designed and implemented from scratch. Their designers need to provide the syntax and all the necessary tools (interpreters, compilers, debuggers) for the language to be usable. DSLs built using this approach include \LaTeX , VHDL and others [21].
- **Internal or embedded DSLs (EDSLs):** These DSLs, as the name suggests, are internal to another language. In this case the design and implementation process is much simpler. They take advantage of the host language and its associated tools. We only need an interpreter function to give the EDSL its semantics. In fact, these languages can be regarded as *“a library in a host language that has the look, feel, and semantics of its own language, customized to a specific problem domain”* [22].

```

1  main = fudlogue (shellF "Hello" (labelF "Hello,_world!" >+< quitButtonF))
2

```



Figure 2.12: Hello World! example using Fudgets.

Even though these approaches differ substantially in the design choices, they both share the objective of raising the level of abstraction and restricting the domain of the language. In fact a good DSL “*encourages programming at a higher level of abstraction*” [23]. This is by far the most important feature a DSL provides and, some even say that DSLs are “*the ultimate abstraction*” [24]. Because of the increased level of abstraction a DSL program should be easier to read, more concise, easier to maintain and reason about [21].

2.4.2 EDSLs

We concentrate on EDSLs since many properties of functional languages (discussed in Section 2.1) simplify the process of developing such languages. Both Obsidian and Accelerate are implemented as EDSLs using Haskell as their host. As stated in [10], Haskell has been very successful for embedded domain-specific languages.

An early example of using Haskell for EDSLs is the Fudgets [25] library, which provides a tool-kit to create GUIs for the X system. The main objective of this work was to show that functional languages are capable of implementing graphical user interfaces and that properties such as laziness enabled the system to deal with the large flow of data present in GUIs [25]. The Fudgets library uses as its main abstraction the *fudget*, the functional equivalent of the *widget*. Through this, many graphical interfaces components are implemented (menus, scroll bars, etc.) and are put together using combinators that compose multiple fudgets together.

In Figure 2.12 we present a “Hello World!” example using Fudgets. The functions that end with an *F* are functions that act over Fudgets. A Fudgets program consists of one or more fudgets put together. In this example we use the parallel composition function *>+<* to put together a fudget consisting of a “quit” button (*quitButtonF*) and a fudget consisting of a label displaying “Hello World!” (*labelF*). The function *shellF* creates a top-level window needed by any fudget program containing a GUI. Finally, the function *fudlogue* is used to connect the main fudget to Haskell’s I/O system [26].

This library is still in use and maintained today and it shows us that the abstraction mechanisms provided by functional languages (higher-order functions, polymorphism, lazy evaluation, etc.) are enough to cope with such systems.

Another example of particular interest is Lava [27], a language for hardware design. This language was created with the objective of simplifying hardware design. It enables developers to describe hardware at different levels of abstraction in order to analyse circuit descriptions from different viewpoints [27]. Like Fudgets, this tool is embedded in Haskell.


```

1 halfAdd :: (Signal Bool, Signal Bool) → (Signal Bool, Signal Bool)
2 halfAdd (a,b) = (sum,carry)
3   where
4     sum = xor2 (a,b)
5     carry = and2 (a,b)

```

Figure 2.13: A half-adder in Lava from [28].

```

1 -- Deep embedding
2 data Expr = Val Integer
3           | Add Expr Expr
4
5 eval :: Expr → Integer
6 eval (Val x) = x
7 eval (Add x y) = eval x + eval y
8
9 -- Shallow embedding
10 type ExprS = Integer
11
12 val :: Integer → ExprS
13 val x = x
14
15 add :: ExprS → ExprS → ExprS
16 add x y = x + y

```

Figure 2.14: Arithmetic expressions: Deep vs. Shallow embedding.

In this case the main abstraction is a circuit, which is treated as a first-class object. By using combinators, circuits can be composed into complex circuit descriptions. As we will see in Chapter 3, Obsidian is inspired by Lava. Earlier versions of Obsidian had syntax similar to Lava’s. This has now changed in favour of a more readable and GPU-oriented syntax.

In Figure 2.13 we show a half-adder written in Lava. A nice feature of Lava is the automatic generation of VHDL code. The interested reader can compare this small example with its corresponding VHDL code shown in Appendix A.1.

2.4.3 Types of Embedding

Depending on the objective and scope of the new language, there are two ways in which we can do the embedding:

- **Shallow embedding:** This method is called “shallow” because the DSL expressions are also valid in the host language without the need for any kind of interpretation. Languages implemented using this technique are easy to implement given that statements in the DSL are also valid sentences in the host language. This is probably the most common way of building EDSLs [22].
- **Deep embedding:** the evaluation of an EDSL computation results in an abstract syntax tree (AST) containing a representation of the EDSL expressions. This AST may be then interpreted by one or more functions, which will translate the AST expressions into expressions valid in the host language.

Consider Figure 2.14, where we define two alternative languages to deal with arithmetic expressions. In the deeply embedded language (lines 1-7) we see that by creating a new data-type for the arithmetic expressions we need to add a more complex interpretation

function (i.e. `eval`) compared to the `add` function of the shallow embedding in lines 15 and 16. Here the differences between the two approaches become apparent. By using a shallow embedding we get the interpretation for free since it is captured directly in the type of the evaluation function. Extending the language is complicated since it involves adding new interpretation functions. With deep embeddings the scenario is just the opposite. While interpretation requires more effort since we have to evaluate an AST, extending the language involves only extending the algebraic type and adding the corresponding interpretation to the `eval` function.

As we will see in Chapter 3, both Obsidian and Accelerate are implemented using a deep embedding. Both languages build ASTs for the GPU computations and these are then interpreted by different functions.

2.5 Related Work

In this section we describe other works that have attempted at comparing and benchmarking programming languages for parallel processing. We also briefly describe other languages for GPU programming similar to Accelerate and Obsidian.

2.5.1 Comparing Programming Languages for Parallel Processing

In [29] Fang et al. compare the performance of CUDA and OpenCL. CUDA is the programming model used in this report (see Section 2.3) and OpenCL is a programming framework that targets CUDA-capable GPUs as well as other architectures such as ATI GPUs and multi-core CPUs. The paper focuses on comparing the performance of both languages. They run benchmarks on 16 selected case studies and conclude that, on average, CUDA is 30% faster than OpenCL.

Another interesting report is [30]. The comparison this time is between three parallel functional languages: Parallel ML (PML), GPH and Eden. PML is an extension of Standard ML for parallel execution. Similar to Accelerate, it uses algorithmic skeletons to build parallel programs (see Section 3.2). GPH is an implicitly parallel extension of Haskell. Eden is another Haskell extension used for distributed and parallel programs. The comparison is based on three problems: matrix multiplication, solving linear equations and ray-tracing. The report not only compares the speed of executables compiled from these languages but also the constructs that they offer to create parallel tasks [30]. It also analyses the usability and states of the languages. We will assess similar aspects of Obsidian and Accelerate in Chapter 4.

To our knowledge there is only one other work directly comparing Accelerate and Obsidian as we aim to do. In [31] the authors use a single test case, matrix multiplication to compare the languages. They also compare other features including the "learning curve" of the languages. We take a different approach. We have selected a bigger set of case studies and identified other points of comparisons that are measurable and that directly reflect the features and design decisions of both Accelerate and Obsidian as we will see in Chapter 4.

2.5.2 Similar Languages

Vertigo [32] is one of the first attempts at simplifying GPU programming in a functional style. It is a domain-specific language embedded in Haskell for rendering 3D graphics.

Vertigo provides a shader language that targets the DirectX 8.1 shader model and provides an optimising compiler that generates code for graphics processors.

Nikola [33] is another DSL embedded in Haskell. This language shares many similarities with both Accelerate and Obsidian. It automatically handles some of CUDA's low-level details. A nice feature of Nikola is that we can either use the provided high-level abstractions or directly embed CUDA computations in it.

The Thrust [34] library raises the level of abstraction of CUDA programs. It is a C++ template library that hides most of GPU internals and lets the programmer specify the algorithms in terms of parallel building blocks such as scan and reduce in a similar fashion to Accelerate. We will use this library when measuring the performance of sorting algorithms in Section 5.5.

2.6 Summary

This chapter has introduced the main concepts of functional languages that make them ideal candidates to raise the level of abstraction. We described data-parallelism. We introduced GPUs and gave a concise introduction to CUDA programming. Finally we discussed endeavours similar to ours. We will use all these concepts in Chapters 3, 4 and 5 when we discuss and compare Accelerate and Obsidian.

3 Accelerate and Obsidian

In this chapter we introduce Obsidian and Accelerate. In Section 3.1 we describe Obsidian and in Section 3.2 we describe Accelerate. We will introduce the main features and functions that we will use in Chapter 5 in the implementation of our case studies.

3.1 Obsidian

Obsidian is a language embedded in Haskell whose objective is to generate CUDA programs. This language was created by Joel Svensson (and colleagues) at Chalmers University while working first on his Masters [35] and then on his PhD [36].

The interest in data-parallel programming sparked from the similarities this type of programming has with the methods used for hardware description. The Lava programming language, discussed in Section 2.4.2, was also developed at Chalmers. Like Lava, Obsidian relies on combinators (higher-order functions) that enable us to compose kernels together.

The main objective of Obsidian is not only to liberate the programmer from CUDA’s low-level details but to give him the tools to control the GPU [37]. In the words of the developers, “*we want to have our cake and eat it too*” [38]. The objective is to hide as many details as possible but still leave the developer in charge of what happens inside the device.

3.1.1 Overview of the Obsidian Implementation

Obsidian is an experimental language. It has gone through many iterations since its original implementation [35]. In the current iteration, an Obsidian program looks more like plain Haskell and less like Lava as it used to look in earlier iterations. This can be seen in Figure 3.1: in lines 1-2 we define a function to increment all the elements of a list in plain Haskell. Lines 4-6 show the same function defined using Obsidian to run on the GPU.

As we will see in the next section, Obsidian provides two main sets of functions, one to manipulate arrays, such as the `push` and `splitUp` functions (lines 5 and 6 of Figure 3.1), and another to lay out the computation on the GPU such as `asGridMap` (line 5 of Figure 3.1) which runs the computation in parallel on the GPU.

```
1 increment :: [Int] → [Int]
2 increment ls = map (+1) ls
3
4 incrementKernel :: Num a ⇒ Pull Word32 a → Pull Word32 a
5 incrementKernel arr = asGridMap $ (push . (fmap (+1))) arr'
6   where arr' = splitUp 2048 arr
```

Figure 3.1: Incrementing an Array in Haskell and Obsidian.

In this section, we introduce the concepts and constructs needed to understand and write Obsidian programs. We will use the latest version (0.4.0.0) available on-line [39]. While most concepts described in the available literature remain the same, we note some

differences from earlier iterations. When no distinction is made, the reader can assume that what we present here is the most up-to-date.

Obsidian can be described as two sub-languages. One to manipulate arrays and one to describe the mapping of computations to the GPU [40].

3.1.2 A Language to Manipulate Arrays

Operations on a GPU are naturally described as operations on arrays since the GPU grid is an array of multiprocessors. This is the reason why CUDA provides different ways of indexing arrays as we saw in Section 2.3.3. Originally, Obsidian provided a single representation of arrays defined as an indexing function together with a length [40]:

```
data Arr a = Arr (IndexE → a) Int
```

This representation is not efficient when manipulating the structure of the array. For example, to concatenate two arrays we need to use conditionals to know from which array (indexing function) we need to take the elements. If n_1 is the size of the first array and f , g are the indexing functions of the two arrays, then, to express the indexing function for their concatenation, we would need to do something similar to:

```
if index < n1 then
  f(index)
else
  g(index)
```

Having these conditionals in a CUDA kernel results in divergent threads within a block, eliminating all the advantages of parallel processing. To overcome such problems, Obsidian developers devised a novel representation of arrays [41] to complement the existing one.

```
1 data Pull s a = Pull {pullLen :: s, pullFun :: EWord32 → a}
2
3 data Push t s a = Push s (PushFun t a)
```

Figure 3.2: Obsidian Pull and Push Arrays.

The current version of Obsidian distinguishes between two types of arrays as seen in Figure 3.2:

- **Pull Arrays:** Obsidian’s original representation. They describe how to “pull”, or compute the elements of the array by applying the given function at each index.
- **Push Arrays:** This array representation complements that of Pull arrays. They describe where the elements end up in an array rather than how to compute them from the indexing function [42]. They encapsulate their own iteration function, which makes them very efficient when dealing with concatenation of arrays and similar operations. They are normally used when Pull arrays produce inefficient kernels.

Unlike C, Obsidian arrays do not represent an area in memory. They only describe how to compute the elements of the arrays. This property enables fusion of array operations for free. For example two `map` operations over an array are converted into a single `map` without writing anything to memory: `map f . map g = map (f.g)`.

Other than the usual operations (`zip`, `map`, `zipWith`, etc) to work over arrays, Obsidian provides functions to create arrays, change between array representations, manipulate them and make arrays manifest in memory as listed in Table 3.1. The array representations

Function	Input	Output
<code>mkPush</code>	length and iteration scheme	a Push array
<code>mkPull</code>	indexing function and length	a Pull array
<code>compute</code>	Pull or Push array	Pull array manifest in memory
<code>splitUp</code>	size of chunk and Pull array	Pull array of Pull arrays

Table 3.1: Functions to manipulate arrays in Obsidian.

are not readily interchangeable. While it is easy to convert from a Pull array to a Push one, because we only need to “inject” the indexing function inside the Push array, going in the other direction is expensive and involves writing all elements to memory before creating the final Pull array [41]. The function `compute` is used both to write elements to memory and to convert from Push to Pull.

```

1 sum :: Data a => Pull Word32 a -> Program Block (Push Block Word32 a)
2 sum arr
3   | len arr == 1 = return $ push $ arr
4   | otherwise   =
5     do
6       let (a1,a2) = halve arr
7       arr' <- compute (zipWith (+) a1 a2)
8       sum (+) arr'

```

Figure 3.3: Sum function in Obsidian. Adapted from [39].

Consider Figure 3.3. The `sum` function takes as input an array of numbers and outputs a single value, which is the result of summing all of its elements. Here we see how the interplay between arrays starts to be helpful. In line 7, the intermediate result of zipping the two half-arrays together is written to shared memory using `compute`. The `compute` function is overloaded¹ at the hierarchy level (see Section 3.1.3) enabling the developer to specify to which type of GPU memory the array will be written.

While both types of array are useful, they add complexity to the language. The developer not only needs to be familiar with the architecture of the GPU but also needs to be able to manipulate both types of arrays in a creative way. There is hope that the Obsidian array model will be simplified in the future. An approach to do this simplification is described in [43] where both representations of arrays are merged into a single one.

3.1.3 Programming the GPU Hierarchy

The Obsidian compiler knows two different ASTs. One is for scalar operations defined by the `Exp` type that describes all the basic language constructs (arithmetic operations, conditionals, etc.). The other describes `Program` statements [38].

The first half of Table 3.2 lists some type synonyms that Obsidian provides to make the code more readable. These will become relevant to us in Chapter 5.

¹In the literature `compute` was not overloaded and there were two functions `force` and `forcePull` to lay out the arrays in memory

Type Alias	Type
DPull	Pull EWord32
DPush t a	Push t EWord32 a
EWord32	Exp Word32
SPull	Pull Word32
Warp	Step Thread
Block	Step Warp
Grid	Step Block
Program Block	BProgram

Table 3.2: Some type aliases provided by Obsidian.

Like CUDA, Obsidian distinguishes between threads, warps, blocks and grids. Two types are defined, `Thread` and `Step` on top of which the rest of the hierarchy is built as listed in the second half of Table 3.2. `Program` statements are parametrised by the GPU level. Unlike CUDA, the number of blocks or warps is not limited by the hardware. When an array is larger than the available number of blocks or warps, Obsidian applies compiler-enabled virtualisation which is achieved by generating an extra sequential loop at the corresponding `Program` level [38].

We can now explain the type signature of `sum` in Figure 3.3. `Exp` lets us distinguish between arrays with static and dynamic size, given that expressions inside `Exp` will only be computed when traversing the AST. In our case, `sum` expects as input an array with static length (`Word32` is not wrapped in `Exp`) and produces a `Program` that is executed at the `Block` level. So, `sum` describes the behaviour of a single block of threads. We could have used the type aliases in Table 3.2 to simplify the signature:

```
sum :: Data a => SPull a -> BProgram (SPush Block a).
```

3.1.4 Obsidian Code Generation and Execution

Obsidian provides two ways of interacting with the GPU. (1) We can run CUDA kernels directly from Obsidian. (2) Similar to how Lava produces VHDL code (see Section 2.4.2), we can use Obsidian to generate the corresponding CUDA kernel and use it later with other CUDA programs.

Figure 3.4 illustrates both ways of using Obsidian. Lines 1-4 define a function that spreads the computation of the `sum` function (defined in Figure 3.3) in parallel, over the resources of the GPU. The mapping is carried out by the `asGridMap` function which applies `sum` to each sub-array passed as input. In lines 6-8 we define a function that outputs the generated CUDA kernel as a string. We can save this kernel to a file and use it later in plain CUDA programs. The function `genKernel` takes as inputs the number of threads per block to use, the name we want for the kernel and the function for which we want to generate the CUDA program. Finally, in lines 10-17 we execute the kernel directly from our Obsidian program. First we compile the kernel using the function `capture` (line 12). Then we create the input array and allocate memory on the GPU for the output array (lines 13,14). In lines 16,17 we execute the kernel. The function `<>` applies the kernel to the input array and `<==` runs the kernel and generates the result which is then copied to host memory using `peekCUDAVector`.

```

1 mapSum :: (Data a, Num a) => DPull (SPull a) → DPush Grid a
2 mapSum arr = asGridMap body arr
3   where
4     body arr = execBlock (sum arr)
5
6 generateKernel = putStrLn $
7     genKernel 256 "SumKernel"
8     (mapSum . splitUp 4096 :: DPull EInt32 → DPush Grid EInt32)
9
10 executeKernel = withCUDA $
11   do
12     kernel ← capture 256 (mapSum . splitUp 4096)
13     useVector (V.fromList [1..4096 :: Word32]) $
14       λi → allocaVector 1 $ λo →
15         do
16           o <= (2,kernel) <> i
17           r ← peekCUDataVector o

```

Figure 3.4: Obsidian: running and generating standalone kernels.

```

1 incrementHaskell :: Num b => [b] → [b]
2 incrementHaskell xs = map (+1) xs
3
4 incrementAcc :: (Elt b, Shape ix, IsNum b) => Acc (Array ix b) → Acc (Array ix b)
5 incrementAcc xs = map (+1) xs

```

Figure 3.5: Incrementing an Array in Accelerate and Haskell. Other than the type-signatures, both functions are identical.

3.2 Accelerate

Accelerate is a domain-specific language deeply-embedded in Haskell. It is used to describe collective array operations that run on parallel devices such as GPUs and multi-core CPUs [44]. Accelerate is a stratified language [45]. It is made up of a front-end, that describes the collective array operations and generates computation ASTs, and a back-end that processes the ASTs to produce the program for the desired target architecture. In this report we focus on the CUDA back-end of Accelerate, developed by Trevor McDonell while working on his PhD at the University of New South Wales in Australia [44].

The aim of Accelerate is to use higher-order functions as primitives of the collective array computations and use algorithmic skeletons to generate code for these primitive operations. An Accelerate collective array computation consists then of one or more higher-order functions acting on every element of the array. The primitive operations include replicate, map, zipWith, fold, scan and others [44].

Accelerate programs look very similar to plain Haskell ones. In Figure 3.5 we show how to increment a list of numbers. The Haskell version is shown in lines 1-2 and the Accelerate one in lines 4-5. Only the type-signatures of the functions render them different. While the first one will operate on Haskell lists the second one will operate on Accelerate arrays and the function will be computed on the GPU or on another type of processor, depending on the back-end used to interpret the code.

We will now describe Accelerate’s front-end and in Section 3.2.2 the CUDA back-end.


```

1 type Z = Z
2 data tail :: head = tail :: head
3
4 type DIM0 = Z
5 type DIM1 = Z :: DIM0 -- one-dimensional array
6 type DIM2 = Z :: DIM1 -- two-dimensional array, a matrix
7
8 type Array DIM0 e = Scalar e
9 type Array DIM1 e = Vector e

```

Figure 3.6: Array shapes and type aliases in Accelerate.

```

1 array1 = fromList (Z :: 256) [1..256] :: Array DIM1 Int
2
3 array2 = fromList (Z :: 2 :: 256) [1..256] :: Array DIM2 Int

```

Figure 3.7: An example of how to define one and two-dimensional arrays in Accelerate.

3.2.1 The Front-End

The front-end describes the functions and types that can be used to manipulate arrays. Accelerate arrays are parametric, regular and rank-polymorphic. They are based on the array representation introduced in the Repa language [46], another EDSL for parallel processing.

An Accelerate array consists of a shape and a type for the array elements:

```
data Arr sh e
```

The shape `sh` is defined inductively with the operator `::` and the type constructor `Z` as seen in Figure 3.6, which also shows type aliases that Accelerate introduces to reason inductively about array dimensionality.

In Figure 3.7 we show how to define one and two-dimensional arrays in Accelerate. The only thing that changes between these array definitions is the shape. This is because Accelerate’s multidimensional arrays are stored as one-dimensional (Haskell) arrays in linear row-major order [44]. This means that multidimensional arrays are filled from the right-most dimension first. Accelerate provides a rich API of functions to manipulate arrays. We will introduce some of them as needed in this section and in Chapter 5.

Arrays in Accelerate can only contain elements and tuples of Haskell’s primitive types (Ints, Booleans, etc.). Accelerate ensures this by defining the type-class `Elt` to which all accepted types must belong [44]. In the aliases defined in Figure 3.7 the type variables `e` must be instances of `Elt`.

There are two types that Accelerate uses to build the computation ASTs. One is `Acc`, which encapsulates all collective array operations. The other is `Exp`, which is the analogous to `Acc` but for embedded scalar operations [45]. This means that `Acc` computations can contain many scalar operations executed in parallel, but `Exp` operations cannot contain any `Acc` ones.

Consider Figure 3.8. Here we use the `zipWith` primitive to multiply two arrays together. The input arrays are wrapped in `Acc` with the function `use`, in the **let** expression in lines

```

1 zipWith :: (Shape ix, Elt a, Elt b, Elt c) => (Exp a -> Exp b -> Exp c)
2         -> Acc (Array ix a) -> Acc (Array ix b) -> Acc (Array ix c)
3
4 multiply :: (Shape ix, Elt a, Elt b, Elt c) => (Exp a -> Exp b -> Exp c)
5         -> Array ix a -> Array ix b -> Acc (Array ix c)
6 multiply xs ys =
7   let xs' = use xs
8       ys' = use ys
9   in
10    run $ zipWith (*) xs' ys'

```

Figure 3.8: Multiplying two arrays in Accelerate.

```

1 unit :: Elt e => Exp e -> Acc (Scalar e)
2
3 use  :: Arrays arrays => arrays -> Acc arrays

```

Figure 3.9: The use and unit Accelerate functions. They lift computations into Acc.

7-8, so that they can be used in the operation. (Applications of the use function trigger data-transfers as we will discuss in Section 3.2.2). Other than use, Accelerate provides the function unit to wrap single elements in Acc. The signatures of both functions are shown in Figure 3.9.

Lastly, Accelerate’s API contains functions that let us operate with arrays directly in Acc. Two such functions are fill and generate, shown in Figure 3.10. The fill function generates an array by applying a function at each index and generate creates an array where all the elements have the same value. These will come in handy in the next chapter when we implement our case studies.

```

1 fill :: (Shape sh, Elt e) => Exp sh -> Exp e -> Acc (Array sh e)
2
3 generate :: (Shape ix, Elt a)
4           => Exp ix -> (Exp ix -> Exp a) -> Acc (Array ix a)

```

Figure 3.10: Two functions for creating arrays in Accelerate.

3.2.2 The CUDA Back-End

In Accelerate, the generation of CUDA code is based on algorithmic skeletons. These are predefined and hand-optimised pieces of code that are instantiated at compile-time [45]. Accelerate defines a set of primitive operations each of which has a matching algorithmic skeleton. The algorithmic skeletons are hand-optimised following the CUDA best practices [18] in order to guarantee that the instantiated operations will make the best use of the GPU resources.

The approach taken by Accelerate relies on the fact that not all programs can be efficiently mapped to the GPU. Only those that show good data-parallelism are a good match for the GPU. Accelerate developers decided to target and optimise only a reduced set of parallel primitives. This is why Accelerate provides a set of primitive higher-order functions each of which is mapped to a single CUDA algorithmic skeleton.

Accelerate generates CUDA kernels dynamically. In CUDA, programs are pre-compiled and then run on the GPU. Accelerate generates GPU binaries at Haskell run-time [45].

The whole compilation process, which happens at Haskell run-time, can be described as follows: (1) the front-end generates computation ASTs corresponding to the collective array operations. (2) The back-end uses the information in the ASTs and information about the available GPU resources to instantiate the algorithmic skeletons. (3) It transfers the required data to the device. (4) It finally runs the program on the GPU.

3.3 Summary

This chapter has described the two domain-specific languages at the core of our discussion. In Section 3.1 we described Obsidian, paying great attention to the array representations used by it. We also described how Obsidian provides functions and types to program at the different levels of the GPU hierarchy. In Section 3.2 we introduced Accelerate. We described both its front-end and the CUDA back-end.

We will use the concepts and features that we introduced when we compare the features of both languages in Chapter 4 and later again in Chapter 5 when we compare and benchmark some interesting examples.

4 Comparing Everything But Speed

In this chapter we compare the features of Accelerate and Obsidian. We give a comprehensive view of the pros and cons of each language in matters other than performance. This is as important as measuring speed, given that both Obsidian and Accelerate aim at simplifying the programmer’s task when writing CUDA kernels and not only at competing against hand-optimised CUDA code. The hope is that the two languages are powerful enough so that the features offered by them justify a probable decrease in performance.

Each section corresponds to an aspect of the languages that is worth discussing in order to assess their potential and usability.

In Section 4.1 we compare the ways in which both languages generate CUDA programs. In Section 4.2 we discuss the extent to which each language lets us manipulate the GPU internals and in Section 4.3 we discuss the languages’ expressiveness. In Section 4.4 we describe how the languages ensure correctness of the generated CUDA programs. In Section 4.5 we assess the current state of each language and conclude the chapter with a table (Table 4.1) summarising our discussion.

4.1 Compilation and Automation

We introduced the different ways in which both Accelerate and Obsidian can generate CUDA programs in Chapter 3. Here we discuss both methods in more depth. In Figure 4.1 we show the compilation processes of each language.

Accelerate is capable of on-line compilation of CUDA programs. Compilation of GPU code happens at Haskell run-time. To minimise this run-time overhead, the generated kernel-binaries are cached and reused if needed [44]. Accelerate solves CUDA’s portability issues. Not only does Accelerate query the machine to check the available devices, it also uses a Haskell version of the CUDA occupancy calculator [4] to ensure maximum usage of memory and threads. In this way it produces an optimal program regardless of the architecture used. This is particularly important if we need to run the program on different GPU architectures.

Obsidian also offers on-line compilation. Even though Obsidian also queries the machine to get device information, it does not do any occupancy calculation. It is up to the developer to decide how many threads and blocks to use and how to lay out the data on GPU memory.

A key difference between both compilation strategies is that in Accelerate everything happens implicitly. A call to `run` compiles the kernel (if it’s the first time that we encounter it) and executes the kernel on the GPU. In Obsidian we need to first use `capture` to generate the kernel binary and then use the `<=` function to apply the kernel to an input. Data transfers between the host and the device are also handled differently. In Accelerate transfers are triggered by the `use` function. When traversing the AST, Accelerate extracts the occurrences of `use` and starts memory transfers asynchronously while generating the CUDA binaries in order to minimise the run-time overhead [44]. In Obsidian we need

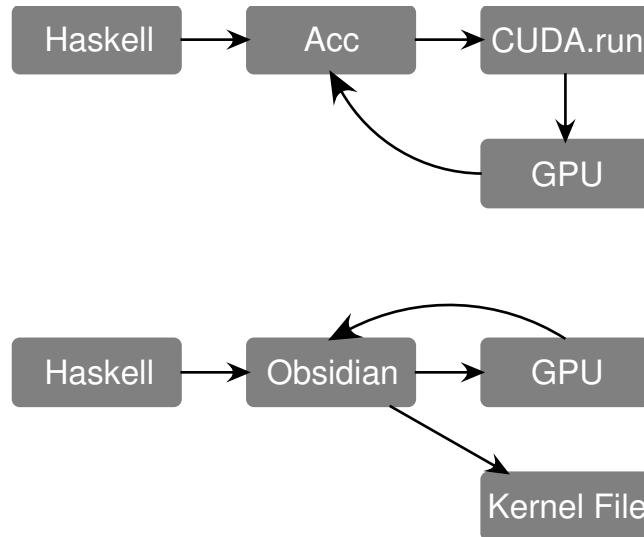


Figure 4.1: The compilation processes of Accelerate and Obsidian. Both languages are capable of running CUDA code from within Haskell. Obsidian can also generate standalone kernels.

to take care of the data transfers and memory allocation by hand using the functions `allocaVector` and `useVector`.

Another feature that Obsidian offers is the generation of stand-alone kernels. We can use our Obsidian program to output a file containing the generated CUDA kernel, which we can then use in a plain CUDA program. In Accelerate the only way to achieve a similar result is to use a debug flag to see the generated CUDA kernel when executing the program¹.

Neither compilation strategy is better than the other as they apply to different scenarios. Accelerate’s on-line compilation will make programs faster than Obsidian ones especially when the same kernel is used multiple times. But Obsidian’s kernel-generation option is a useful alternative when there is no need to run the programs from within Haskell, and we want to explore different possibilities for a kernel before embedding it in an existing CUDA project.

4.2 Control of CUDA Internals

Obsidian enables us to write functions that work at the different levels of the GPU hierarchy (as discussed in Section 3.1.3). We can even decide what data to store in shared memory. This makes Obsidian a good choice for design-exploration [47]. The language can be used to try different implementations of a program without swimming in a sea of bugs as usually happens when going on such quests in CUDA. We will see an example of this approach in Section 5.3 where we take an existing Obsidian program and try to speed it up by using shared memory. In contrast, by automating the whole compilation process, as we saw in the previous section, Accelerate completely hides the GPU from the developer.

Given that GPU performance is strongly related to architecture and how resources are used, we believe that the controls offered by Obsidian can be an advantage, especially

¹This is only possible if Accelerate is installed with debug capabilities.

when trying to parallelise complex tasks. Obsidian is the winner in the control battle.

4.3 Language Expressiveness

features offered by their common host. Through Haskell’s overloading facilities, both languages are able to overload most primitive operations to work over the EDSL expressions and they are even able to manipulate primitive types such as Ints and Floats. To manipulate Boolean expressions both languages define the functions `==*`, `/=*`, `<*`, `>*`, `<=*`, and `>=*`. (The normal Haskell Boolean operators cannot be overloaded, as they are not part of any type-class).

```

1  -- Accelerate
2  (?)      :: Elt t =>  Exp Bool → (Exp t, Exp t) → Exp t
3  cond     :: Elt t =>  Exp Bool → Exp t → Exp t → Exp t
4
5  while    :: Elt e =>  (Exp e → Exp Bool) → (Exp e → Exp e) → Exp e → Exp e
6  iterate  :: ∀ a. Elt a =>  Exp Int → (Exp a → Exp a) → Exp a → Exp a
7
8  (?!|)    :: Arrays a =>  Exp Bool → (Acc a, Acc a) → Acc a
9  acond    :: Arrays a =>  Exp Bool → Acc a → Acc a → Acc a
10 awhile  :: Arrays a =>  (Acc a → Acc (Scalar Bool)) → (Acc a → Acc a)
11           → Acc a → Acc a
12
13 -- Obsidian
14 ifThenElse :: Exp Bool → a → a → a

```

Figure 4.2: Accelerate and Obsidian control-flow functions.

Both languages offer functions to control the flow of the programs. The signatures of these functions are shown in Figure 4.2. Accelerate provides versions for both scalars (lines 1-6) and arrays (8-10). `(?)` and `(?!|)` are infix versions of `cond` and `acond` respectively. Obsidian only provides an if-then-else construct for Scalars as seen in line 14.

4.3.1 Manipulating Arrays

In Chapter 3 we introduced Accelerate and Obsidian arrays. Since arrays are the central abstraction in both languages we now explore and compare them in greater detail.

Obsidian provides a library of array functions most of which work over `Pull` arrays. (We can also use them over push arrays by first converting them to `Pull`). Two of the most interesting ones have their signatures shown in Figure 4.3. These are functions used to manipulate the way data is loaded onto the GPU: `coalesce` loads data in a coalesced fashion, while `evenOdds` splits the array into its even and odds indices. Obsidian also provides a wide range of common functions over arrays such as `zip`, `take`, `drop`, `tail`, `head` and many other standard array functions. What makes the Obsidian array model stand out is that the interplay between push and pull arrays (discussed in Section 3.1.2) enables us to write our own load functions which, depending on the problem setting, may result in more efficient memory use.

```

1 coalesce :: ASize 1 =>  Word32 → Pull 1 a → Pull 1 (Pull Word32 a)
2 evenOdds :: ASize 1 =>  Pull 1 a → (Pull 1 a, Pull 1 a)

```

Figure 4.3: Two of the Obsidian array-processing functions.

```

1 permute  :: (Shape ix, Shape ix', Elt a) => (Exp a → Exp a → Exp a)
2          → Acc (Array ix' a) → (Exp ix → Exp ix')
3          → Acc (Array ix a)   → Acc (Array ix' a)

```

Figure 4.4: One of the Accelerate array-processing functions.

Accelerate also offers an exhaustive set of array functions including operations to transpose two-dimensional arrays and different flavours of folds and scans. One of the most interesting functions is `permute` with the signature shown in Figure 4.4. This function is used to manipulate, with an indexing function, the way values are stored in arrays. This function and its complement (`backpermute`) are the core of other important operations such as `gather` and `scatter` which are used to manipulate array elements so that threads within a block can access them more efficiently. `Permute` needs to be used with care. The indexing function is not surjective. In order to avoid out of bounds errors, it initialises the output array with the given default values essentially doubling the amount of space used. This is bad because GPU memory is limited.

While in Obsidian arrays can be nested, enabling the possibility of limited nested-data parallelism [38], this is forbidden in Accelerate. Although we can work with higher-dimensional arrays, these are still stored as Haskell un-boxed one-dimensional arrays, as we saw in Section 3.2.1.

Obsidian’s array model is very complex. The benefit is that we can mix the array representations to manage explicitly how the GPU will handle data. However a more transparent and simple array representation, such as the one used by Accelerate, greatly simplifies the programmer’s tasks.

A language is not complete without an API, a list of predefined and common functions for programmers to use. Obsidian is a changing language, still being used by developers to test new research ideas. So it does not yet provide an extensive API, just a small set of common functions, many of which we have already encountered. Accelerate is a much more stable language and, although incomplete, provides an extensive API. For this reason and given that Accelerate’s array model is simpler and transparent, Accelerate is the winner in this section.

4.4 Safety

Both Accelerate and Obsidian depend on Haskell’s type-system and type-classes to ensure that programs are correct. Obsidian uses the `Program` and `Exp` types to ensure which programs can be executed on the GPU and which types can be used as array elements. `Program` statements are also parametrised by the GPU hierarchy levels. The type-system ensures that, for example, no statements at the `Thread` level can operate at the higher levels of the hierarchy. Accelerate works similarly. Only statements inside `Acc` can run on the GPU, and only types that are instances of `Elt` can be used as array elements.

Problems arise when we move to GPU territory. While both languages use the Haskell `cuda`² package which makes it possible to call CUDA functions from Haskell, there is no easy way of propagating CUDA exceptions back to Haskell-land. For example, in Accelerate, errors that arise when producing CUDA code will only be caught during run-time

²Available at <http://hackage.haskell.org/package/cuda>.

when the CUDA program is compiled [44]. Moreover, the types of CUDA exceptions we can get are architecture-dependent, so the same program running on different GPUs can give rise to different errors.

The most common errors are due to memory allocations. As in CUDA, Obsidian and Accelerate assume that the input data will fit into the device’s memory so we need to know in advance the maximum allowed global memory. Neither language does any check for this so the program will simply exit with an error if memory capacity is exhausted. Problems with shared memory are handled differently. Here Obsidian’s block and warp virtualisation comes to the rescue [38]. If the input array is too big then Obsidian ensures that different passes will be made over it. In Accelerate and in CUDA this will result in a program error not catchable before run-time.

When compiled with the `-fdebug` flag, Accelerate can print a more detailed view of the compilation and instantiation process. We didn’t find this particularly helpful. We used it only when we wanted to see the generated CUDA kernels and the number of threads and blocks used.

The safety point should go to Haskell given that most of the languages’ safety checks come from it. We shared the point equally between Accelerate and Obsidian.

4.5 Summary

	Accelerate	Obsidian
Compilation		
On-line compilation	●	●
Standalone kernels	◐	●
CUDA Internals manipulation	○	●
Expressiveness		
Haskell Primitive Types	●	●
Control flow constructs	●	◐
Boolean operators	●	●
Array functions	●	●
Nested data-parallelism	○	◐
API	●	◐
Safety		
Type-safety	●	●
Debug options	●	○
Catch CUDA errors	○	○

●Supported feature, ○Unsupported feature, ◐Partially supported feature.

Table 4.1: Accelerate vs. Obsidian (excluding performance).

In this chapter we compared Accelerate and Obsidian in matters other than performance. Both Accelerate and Obsidian attack the same problem, that of generating CUDA programs, but from a different point of view and following different principles. We do not prefer one language over the other but simply pointed out their advantages and weaknesses so that developers can make a more informed decision when choosing between the two.

Accelerate and Obsidian are both recent languages. They are experimental and as such they are still under active development.

Of the two, Accelerate is the more stable. The front-end, (discussed in Section 3.2.1) on which it relies, is well developed and it provides an extensive library as we saw in Section 4.3. An advantage of Accelerate is its versatility. While there are only two stable back-ends – the CUDA back-end, here discussed, and an interpreter – the prospect of running the same Accelerate program in different GPU and CPU architectures gives the language an advantage over its competitors.

Obsidian is mainly used for experiments. Its developers use it as a test-bed for new research ideas. While this is good for users who want to use the latest developments, it also means that the language is still changing rapidly. While writing this report Obsidian went through a major change. Most of the syntax was changed and more simpler and readable names were introduced. We adapted our code to the new syntax but this is something developers would not want to do regularly.

For the take-away summary of this chapter, in a concise form, see Table 4.1. It summarises the features supported by both languages. We hope this will be a valuable contribution to developers wanting to develop in either language.

5 Benchmarking Accelerate and Obsidian

In this chapter we implement and benchmark four programs. Our aim is to compare the speeds of similar computations in Accelerate, Obsidian and CUDA.

In Section 5.1 we review the set-up used to run the tests and briefly discuss how to benchmark lazy programs. We then first implement and benchmark parallel reduce in Section 5.2. In Section 5.3 we adapt the matrix multiplication used in [31] in order to check if their results are still valid. We also try to optimise the Obsidian version by using shared memory. In Section 5.4 we benchmark Mandelbrot-set generator programs to give a taste of what a more involved task looks like in Accelerate and Obsidian. In Section 5.5 we compare the sorting algorithms available in the literature. We end the chapter with a discussion of our findings.

All the code used for the benchmarking tests reported here is available at <https://github.com/aesadde/AccObsBenchmarks>.

5.1 The Set-Up

The experiments were all carried out on an Amazon EC-2 instance [48], although this was not our first choice (see Section 5.1.1). The instance features an Nvidia K520 GPU. This device is of compute capability 3.0. Nvidia classifies its GPUs according to their instruction-set architecture. For more information please refer to [49]. The complete details of the device are given in Appendix B.

We used the following software:

- The Glorious Glasgow Haskell Compilation System, version 7.8.3¹
- Accelerate and Accelerate-cuda 2.0.0.0, the latest available on Accelerate GitHub repository [50].
- Obsidian version 0.4.0.0. Latest version available on Obsidian GitHub repository [39].
- CUDA SDK release 6.5².

5.1.1 A Note on Installation

One of the main issues when programming with GPUs is that the choice of the device to use can limit the facilities and features available given the different architectures of these devices.

When this project started, the aim was to program with the GPU we had in hand, an Nvidia GeForce of compute capability 1.2. We had to hunt for a newer device when we

¹Available at <https://www.haskell.org/ghc/>

²Available at: <https://developer.nvidia.com/cuda-toolkit>

hit memory alignment errors with Obsidian. These errors could be solved by adding extra functions to the language but given time constraints, we decided to switch to a newer architecture. We got access to the University’s GPU compute node³. This time while the GPU featured a newer architecture, we failed to install all the necessary packages even with the help of the IT support team. After some frustration we settled for the Amazon Cloud instance, described above, which we had to set up from scratch.

5.1.2 Benchmarking Lazy Programs

In Section 2.1.3 we talked about laziness. In most cases laziness is an advantage, but when benchmarking lazy programs we need to be careful. Problems arise because in a lazy language the expressions are evaluated only when needed, as we saw earlier, and even in those cases, expressions are only evaluated as much as needed. This means that the compiler can run a computation only once and, also because of purity, save the results so that subsequent calls to a particular function do not incur in any extra costs.

In order to avoid bad timings due to laziness, we need to evaluate expressions as much as possible before running the benchmarks. To achieve this and gather the desired timings we used two benchmarking strategies:

1. Similar to how the Obsidian benchmarks are implemented in [39], we gathered the results measuring the elapsed time between calls to the `getCurrentTime` function, placed before and after the execution of the functions that we wanted to measure. To ensure that expressions were fully evaluated we used the `deepseq` function to evaluate expressions to normal form and used `print` where `deepseq` was not possible⁴.
2. The other strategy that we used was to use Criterion. Criterion is a Haskell benchmarking library that provides functions to evaluate expressions to normal form and can also deal with I/O operations. (In order to use this library with Obsidian we requested that some changes be made to the language, see Section 5.6 for more information.)

The hand-tuned CUDA programs were all benchmarked using the `cudaEvent` functions in the relevant places of the code. We ran each CUDA program five times to get an average.

In Appendix A.2 we list the CUDA code produced by Accelerate and Obsidian in the first two examples (parallel reduce and matrix multiplication) to give a taste of what the generated code looks like and how it compares to the hand-written CUDA programs that we will encounter in this chapter.

5.2 Parallel Reduce

In this section we discuss the implementation of parallel reduce. We will see how a simple program can help us bring to light many different aspects of both Accelerate and Obsidian.

Parallel reduce is a program that takes as input a set of data of similar kind and combines its elements to produce a single value as output. This process goes by many different

³York Advanced Research Computing Cluster (YARCC). See <https://wiki.york.ac.uk/display/RHPC/YARCC++>
York+Advanced+Research+Computing+Cluster

⁴`deepseq` can only be applied to elements that are instance of the `NFData` class, the class of types that can be fully evaluated.

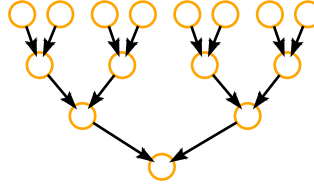


Figure 5.1: Reduce seen as an inverted tree.

names. In the functional languages community it is usually called `fold`. It is also the base of the MapReduce programming model [51]. This task is easily parallelised as it can be split into independent reductions of subsets of data. This can be seen as an inverted tree in Figure 5.1. This structure matches the GPU architecture enabling us to perform independent reductions in separate thread-blocks and then combine these results to get the final one.

Since CUDA does not have grid-wide synchronisation barriers, blocks of threads cannot communicate and combine their sub-reductions easily. Reduce is then usually implemented with a kernel executing one or more times until all the data fits in a single block of threads at which point it is either computed in sequence on the CPU or a last call to the kernel is made.

5.2.1 Reduce in Accelerate

In Accelerate we implement reduction as shown in Figure 5.2. The `fold` function used in line 2 is one of Accelerate’s primitive functions, it triggers the instantiation of an algorithmic skeleton.

```
1 accReduce :: (Elt a, IsNum a) => Vector a -> Vector a
2 accReduce arr = run $ fold (+) 0 (use arr)
```

Figure 5.2: Parallel Reduce in Accelerate. The produced CUDA code is shown in Appendix A.2.1.

Because Accelerate hides all the GPU details, this program is very simple. Only the `run` and `use` functions, introduced in Section 3.2, make the program look different from a sequential fold. The type-signature in line 1 contains type aliases that we defined in Section 3.2.1.

5.2.2 Reduce in Obsidian

In Obsidian we need to take care of what happens at every level of the GPU hierarchy. In Figure 5.3 we show an adapted version of the reduce kernel available in [38]. The program has three functions:

- `blockRed` (lines 1-10): Each block performs a final reduction of the reductions computed by `coalesceRed`. The results are stored in shared memory.
- `coalesceRed` (lines 12-18): Defines what each block of threads does. Each thread in a block performs a sequential reduce of 32 threads. In order to minimise memory latency, the elements are read in a coalesced fashion. The result of each thread-reduce is stored in shared memory (lines 16,17).

- `reduceKernel` (lines 20-25): This is the main function. It spreads the reduction over many blocks on the GPU.

The output of this program is an array containing the results of each block-reduce. The final reduction is executed sequentially on the CPU.

```

1 blockRed :: Data a
2   => Word32
3   → (a → a → a) → SPull a
4   → BProgram (SPush Block a)
5 blockRed cutoff f arr
6   | len arr == cutoff = return $ push $ fold1 f arr
7   | otherwise = do
8     let (a1,a2) = halve arr
9     arr' ← compute (zipWith f a1 a2)
10    blockRed cutoff f arr'
11
12 coalesceRed :: Data a
13   => (a → a → a) → SPull a → Word32
14   → BProgram (SPush Block a)
15 coalesceRed f arr et =
16   do arr' ← compute $ asBlockMap (execThread' . seqReduce f)
17     (coalesce et arr)
18     blockRed 2 f arr'
19
20 reduceKernel :: Data a
21   => (a → a → a) → Word32 → DPull (SPull a)
22   → DPush Grid a
23 reduceKernel f et arr = asGridMap body arr
24   where
25     body arr = execBlock (coalesceRed f arr et)

```

Figure 5.3: Obsidian reduce, adapted from [38]. The CUDA code produced is shown in Appendix A.2.2.

5.2.3 Reduce in Hand-coded CUDA

The hand-optimised CUDA version of the reduce kernel, shown in Figure 5.4, was adapted from [52]. The program reduces multiple elements per thread sequentially and, like Obsidian, it reduces the last sub-array on the CPU. The algorithm is parametrised by `blockSize`, the size of a block of threads. In our tests the best performance was obtained with a blocks of 512 elements.

It is interesting to compare this hand-tuned CUDA program with the CUDA programs produced by Accelerate and Obsidian, they are shown in Appendix A.2. Both Accelerate and Obsidian produce kernels that are almost impossible to read and understand compared to the hand-tuned CUDA version described in this section.

5.2.4 Reduce Benchmarks

We measured different aspects of the program to give a detailed account of the main differences between Accelerate, Obsidian and CUDA. We first measured the run-times of Accelerate and Obsidian against a sequential reduce⁵ as seen in Figure 5.5. Then we compared the run-times of the reduce kernels in Figure 5.6. Finally, in Figure 5.7 we compared the performance of the kernels taking into account data-transfers between the host and the device.

```

1 template <unsigned long blockSize>
2 __global__ void reduce( long *g_idata, long *g_odata, unsigned long n) {
3     extern __shared__ long sdata[];
4     unsigned long tid = threadIdx.x;
5     unsigned long i = blockIdx.x*(blockSize*2) + tid ;
6     unsigned long gridSize = blockSize*2*gridDim.x;
7     sdata[tid] = 0;
8
9     while (i < n) {
10         sdata[tid] += g_idata[i] + g_idata[i + blockSize]; i += gridSize;
11     }
12     __syncthreads();
13     if ( blockSize >= 512) { if ( tid < 256) { sdata[tid] += sdata[tid + 256]; }
14         __syncthreads();
15     }
16
17     if ( blockSize >= 256) { if ( tid < 128) { sdata[tid] += sdata[tid + 128]; }
18         __syncthreads();
19     }
20
21     if ( blockSize >= 128) { if ( tid < 64) { sdata[tid] += sdata[tid + 64]; }
22         __syncthreads();
23     }
24
25     if (tid < 32) warpReduce<blockSize>(sdata,tid);
26     if (tid == 0) g_odata[blockIdx.x] = sdata[0];
27 }

```

Figure 5.4: Reduce in hand-coded CUDA.

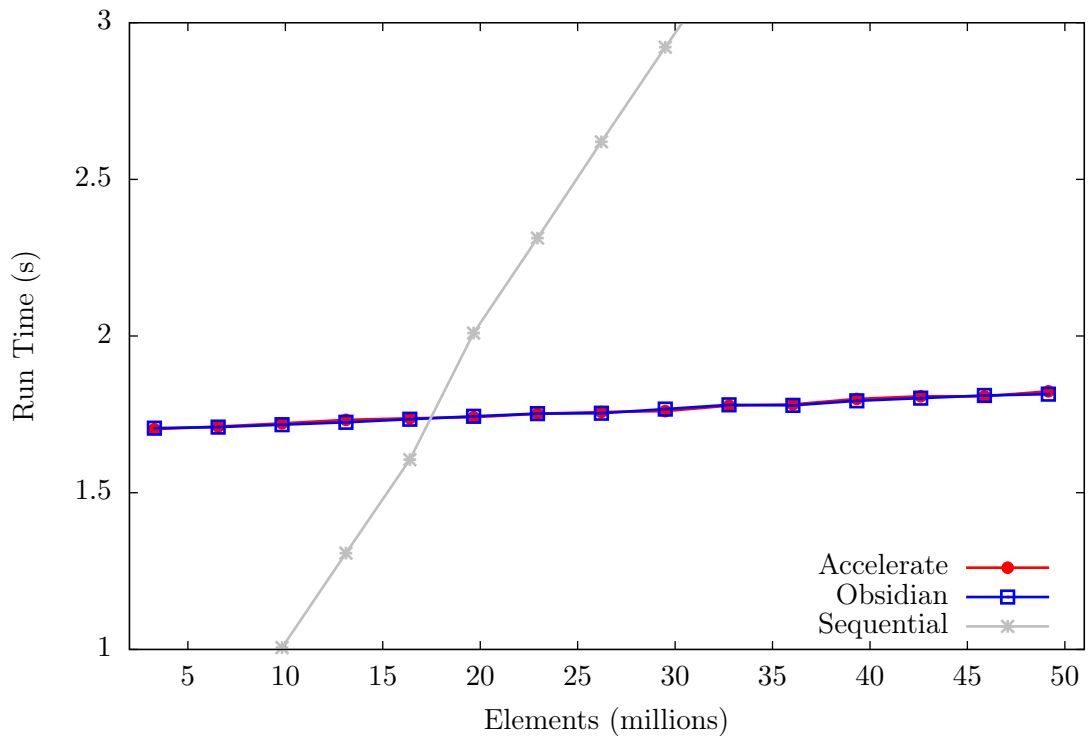


Figure 5.5: Performance of a sequential Reduce against parallel Reduce in Obsidian and Accelerate.

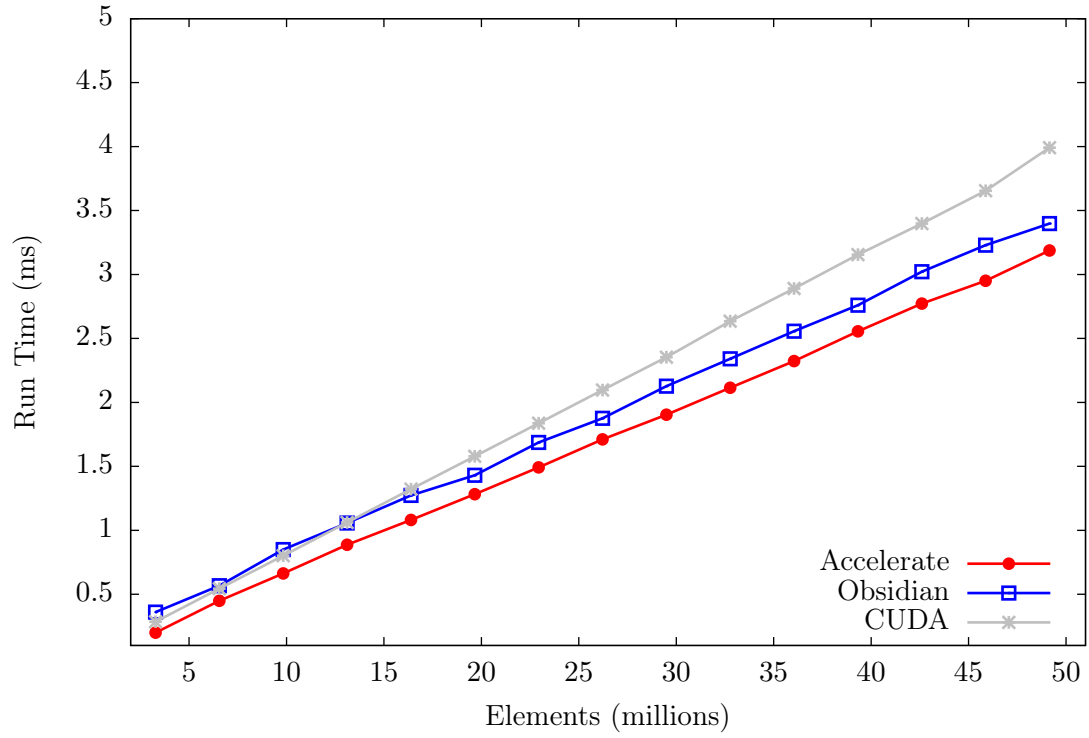


Figure 5.6: Performance of the Reduce kernels produced by Accelerate and Obsidian and the hand-written kernel in CUDA.

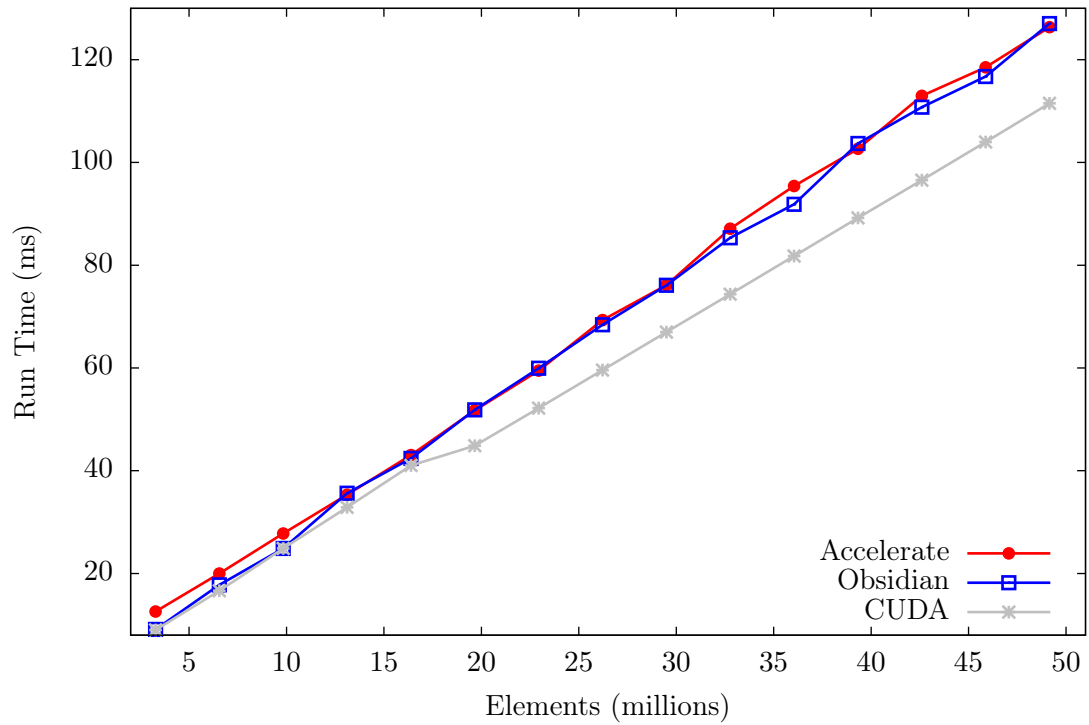


Figure 5.7: Performance of the Reduce kernels taking into account data-transfers.

In Figure 5.5 we see that a sequential reduce is only the best choice when dealing with arrays with less than 16 million elements. This happens because of the compilation overhead of both Obsidian and Accelerate. Between Obsidian and Accelerate there is no real winner given that the overall performance is dominated by the time they spend compiling the CUDA binaries. In the case where the kernel is used multiple times, Accelerate might be faster given that it caches all the compiled binaries.

Some differences arise when we look at the kernels in detail. In Figure 5.6 we see that the Accelerate kernel is the fastest one. This is due to the fact that Accelerate's kernels are optimised and also because Accelerate invokes the kernel two times instead of reducing the last sub-array on the CPU as done by our Obsidian and hand-written CUDA implementations.

When we include the time spent doing data-transfers, Figure 5.7, we discover some surprising facts. CUDA is the fastest program as expected. The surprises come when we compare Obsidian and Accelerate. In Figure 5.6 we saw that Accelerate is the fastest kernel, and given that Accelerate performs asynchronous data-transfers (see Section 4.1), we were expecting Accelerate to be faster than Obsidian, but this is not the case. They perform almost identically. We can only conjecture that the way Accelerate performs the asynchronous transfers is not optimal. Further work would be needed to assess this, see Section 6.2.1 for more information.

Even though Obsidian and Accelerate are good choices and their speed is comparable to hand-optimised CUDA, the hand-written CUDA kernel that we used is not the fastest one available. There are faster versions which use shuffling tricks as seen in [53].

Although Accelerate and Obsidian try to minimise the run-time overheads (see Section 4.1), they will need to improve further if they want to compete against optimised CUDA code.

5.3 Matrix Multiplication

Matrix multiplication is a common algebraic operation that can be easily parallelised as each entry of the resulting matrix can be computed in parallel by computing a dot-product.

In this section we describe and measure the speed of matrix multiplication implemented in Accelerate, CUDA and Obsidian. We focus only on the overall speed of Accelerate and Obsidian and see how they perform against CUDA. For Accelerate and Obsidian we used the matrix multiplication implementations available in [31]. For CUDA we used the implementation available in [4].

5.3.1 Matrix Multiplication in Accelerate

The Accelerate implementation is shown in Figure 5.8. This implementation is best described in [46]. The program uses multi-dimensional arrays to represent the input and output matrices. It then builds two cubes by replicating both matrices along a new dimension (lines 10-13). We end up with two cubes with the same dimensions. The program then computes the resulting matrix by multiplying each corresponding cube-cell and reducing one dimension with the `sum'` function (line 3).

⁵In Haskell this amounts to apply `sum` to a list of numbers.


```

1 accMatMul a b = A.run $ sum' (prod aCube bCube)
2   where
3     sum' = A.fold (+) 0
4     prod = A.zipWith (*)
5     t    = A.transpose b
6     getRow = A.indexHead . A.indexTail
7     getCol = A.indexHead
8     rowsA = getRow (A.shape a)
9     colsB = getCol (A.shape b)
10    sliceA = lift (Z :. All :. colsB :. All)
11    sliceB = lift (Z :. rowsA :. All :. All)
12    aCube = A.replicate sliceA a
13    bCube = A.replicate sliceB t

```

Figure 5.8: Matrix Multiplication in Accelerate. The produced CUDA code is shown in Appendix A.2.3.

```

1 matMul :: (Num a, Data a) => SPull (SPull a)
2   → SPull (SPull a) → SPush Grid a
3 matMul a b = asGridMap body a
4   where
5     body x = matMulRow x (transpose b)
6
7 matMulRow :: (Num a, Data a) => SPull a
8   → SPull (SPull a) → SPush Block a
9 matMulRow row mat = asBlockMap (dotProd row) mat
10
11 dotProd :: (Num a, Data a) => SPull a
12   → SPull a → SPush Thread a
13 dotProd a b = execThread' $ seqReduce (+) (zipWith (*) a b)
14
15 transpose :: SPull (SPull a) → SPull (SPull a)
16 transpose arr = mkPull n1 (λi → mkPull n2 (λj → (arr ! j) ! i))
17   where
18     n2 = len arr
19     n1 = len (arr ! 0)

```

Figure 5.9: Matrix Multiplication in Obsidian. The produced CUDA code is shown in Appendix A.2.4.

Again, the only clue that this program will run on the GPU is given by the `run` function in line 1. This high level of abstraction is what let us write such a concise and clear program but, as we will see in Section 5.3.4, beauty and simplicity do not always guarantee a good performance.

5.3.2 Matrix Multiplication in Obsidian

We adapted the matrix multiplication program available in [39] and [31]. This is a basic implementation. The program is shown in Figure 5.9. Each entry of the result matrix is computed by performing a dot-product of each row of the first matrix with each row of the second matrix transposed. As with the Obsidian reduce program (Figure 5.3), the computation is described at the different levels of the GPU hierarchy:

- `matMul` (lines 1-5): The main function that spreads the computation in parallel over the GPU.
- `matMulRow` (lines 7-9): Each block of threads computes the dot-product of a single row of the first matrix with all the columns of the second matrix. This results in a complete row of the output matrix.

- `dotProd`(lines 11-13): This function describes how the dot-product of two vectors is computed in sequence by a single thread using the Obsidian `zipWith` function.
- `transpose` (lines 15-19): This function transposes a matrix. The Obsidian array representation (described in Section 3.1.2) ensures that this computation is cheap since the arrays are not manifest in memory at this point and the transposition consists only on changing the indexing function.

```

1 matMulRow :: (Num a, Data a)
2   => SPull a
3   => SPull (SPull a) -> SPush Block a
4 matMulRow row mat = exec $ do
5   row' <- compute $ push row
6   return $ asBlockMap (dotProd row') mat

```

Figure 5.10: Block-wide matrix multiplication with shared memory. Each block of threads loads one row into shared memory before performing the dot-products.

We wanted to see if we could improve the results available in [31]. One of the simplest ways of improving the performance of CUDA applications is by using shared memory to load and store initial and intermediate results. We used shared memory by modifying the `matMulRow` function. The modified version is shown in Figure 5.10. We attempted to first load into shared memory the row used by all the threads in the same block (line 5) and then computing the dot-product of this row with each column of the second matrix (line 6). The rest of the program is the same as the one shown in Figure 5.9.

In Section 5.3.4 we will see that many times such an “optimisation” is not enough to speed-up our programs. We will see that, in this case, loading items into shared memory before performing the computation does not give us any tangible advantages.

5.3.3 Matrix Multiplication in Hand-coded CUDA

The hand-tuned CUDA implementation of the matrix multiplication program is shown in Figure 5.11 and Figure 5.12. We adapted the code available in [4].

The program works by computing the multiplication of sub-matrices in parallel in multiple blocks of threads. Figure 5.11 defines the following functions:

- `GetElement` and `SetElement` (lines 10-16): The matrices are stored in row-major order. These functions are used to retrieve and set the values of the matrix. For clarity we use a `struct` to represent a `Matrix` (lines 1-6). Each `Matrix` contains information about the width, height and stride of its elements.
- `GetSubMatrix`(lines 18-26): This function generates a sub-matrix from the given matrix. Each value of the sub-matrix is loaded by a single thread. Each sub-matrix has size `BLOCK_SIZE × BLOCK_SIZE`.

Figure 5.12 defines the `MatMulKernel` kernel. It describes how each sub-matrix is loaded into shared memory and how the multiplication of two sub-matrices is carried out and written back to shared memory. This program is optimal. It makes good use of all the available GPU resources. In Section 5.3.4 we will see that this program performs much faster than both Obsidian and Accelerate but this is not for free. This program consists of almost 100 lines of code while both Accelerate and Obsidian programs are described in less than 25 lines.

```

1 typedef struct {
2     int width;
3     int height;
4     int stride;
5     float* elements;
6 } Matrix;
7
8 #define BLOCK_SIZE 16
9
10 __device__ float GetElement(const Matrix A, int row, int col) {
11     return A.elements[row * A.stride + col];
12 }
13
14 __device__ void SetElement(Matrix A, int row, int col, float value) {
15     A.elements[row * A.stride + col] = value;
16 }
17
18 __device__ Matrix GetSubMatrix(Matrix A, int row, int col) {
19     Matrix Asub;
20     Asub.width = BLOCK_SIZE;
21     Asub.height = BLOCK_SIZE;
22     Asub.stride = A.stride;
23     Asub.elements = &A.elements[A.stride * BLOCK_SIZE * row
24                                     + BLOCK_SIZE * col];
25     return Asub;
26 }

```

Figure 5.11: Device functions for the hand-written CUDA Matrix Multiplication kernel.

```

1
2 __global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
3     int blockRow = blockIdx.y;
4     int blockCol = blockIdx.x;
5
6     Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
7     float Cvalue = 0;
8     int row = threadIdx.y;
9     int col = threadIdx.x;
10
11     for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
12
13         Matrix Asub = GetSubMatrix(A, blockRow, m);
14         Matrix Bsub = GetSubMatrix(B, m, blockCol);
15
16         __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
17         __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
18
19         As[row][col] = GetElement(Asub, row, col);
20         Bs[row][col] = GetElement(Bsub, row, col);
21         __syncthreads();
22
23         for (int e = 0; e < BLOCK_SIZE; ++e)
24             Cvalue += As[row][e] * Bs[e][col];
25         __syncthreads();
26     }
27     SetElement(Csub, row, col, Cvalue);
28 }

```

Figure 5.12: Matrix Multiplication in hand-written CUDA.

5.3.4 Matrix Multiplication Benchmarks

In this section we present the results of the matrix multiplication benchmarks. This time we are only interested in the overall run-time of the programs, excluding the compilation time of the Accelerate and Obsidian CUDA binaries, in order to assess the power of our two subject languages.

We used the Criterion library (see Section 5.1.2) to benchmark the Obsidian and Accelerate implementations.

Size	Accelerate	Obsidian	Obsidian Sh-mem	CUDA
64	4.5	0.851	0.864	0.23
128	8.72	3.19	2.89	0.32
256	31.2	10.9	10.9	1.1
512	199	42.7	45.6	5.4
1024	1240	176	184	33.7
2048	10700	705	733	237.8
8192	error	27438	24619	13883.4

Table 5.1: Matrix Multiplication: overall run-time of Accelerate, CUDA and Obsidian. Times are in milliseconds (ms). The leftmost column shows the size n of the side of the matrices.

In Table 5.1 we show the results. We ran the benchmarks on $n \times n$ matrices. The hand-tuned CUDA implementation is the fastest one by a big margin while the Accelerate implementation is the slowest one. This scenario, while not surprising, is strange. While we expected the CUDA implementation to be the winner, we did not expect the Accelerate program to perform so poorly. The run-times of the Accelerate program increase proportional to the input-size. With inputs bigger than 1024×1024 the run-times seem to explode. At first, this seemed to suggest that the instantiated algorithmic-skeletons used for this program were not optimal and also that the cube structure used in the algorithm (see Section 5.3.1) was spending too much time copying the input elements along the new dimension. But, after testing the Accelerate program with bigger input sizes we found a bug. We will discuss this in more detail in Section 5.6.

Another interesting point shows up when we compare the speeds of the two Obsidian implementations. Figure 5.13 offers a more detailed view of this comparison. By loading the corresponding row into shared memory before performing any computation we expected the program to be faster than the non-shared memory version. Instead, what we see is that, at first, there is no real benefit from pre-loading the data into shared memory. The “optimised” version performs better only in the last case of our test. This suggests that the benefits of using shared memory become apparent only with matrices with more than 16 million elements.

Another approach at optimising the Obsidian program would be that of computing multiplication of sub-matrices in a way similar to what is done in the hand-written CUDA version (Section 5.3.3). We will discuss this in Section 6.2 when we talk about future work.

In this section we started to see the shortcomings of Accelerate and Obsidian. We saw that even though it is much simpler to write Accelerate and Obsidian programs than it is to write the same programs, by hand, in CUDA, the speed gap between the CUDA

code produced by our subject languages and the hand-optimised code written directly in CUDA is very big.

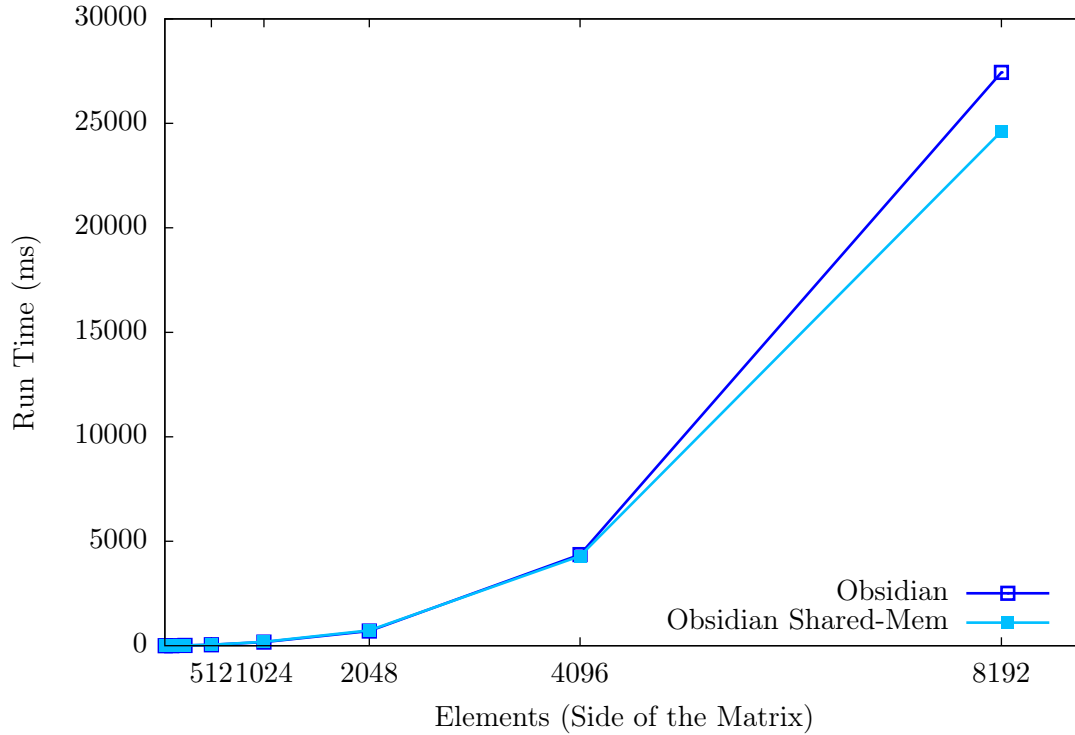


Figure 5.13: Matrix Multiplication in Obsidian with and without shared memory.

5.4 Mandelbrot Sets

The Mandelbrot set is a self-similar image (see Figure 5.14) generated by complex numbers plotted in a 2D plane. The numbers in the set come from the recurrence formula

$$z_{n+1} = z_n^2 + c$$

A number c is said to be in the set if, starting with $z_0 = 0$ and iterating the formula, the magnitude $|z_n|$ does not diverge.

Other than being interesting because of the strange and often pretty patterns that the set exhibits, Mandelbrot sets are interesting to us because they present two interesting properties that make them ideal for our benchmarks: (1) Computing each pixel (complex number) of the image can be done, independently, in parallel and (2), given that in practice we can't iterate the formula forever, the program will include a condition to test for divergence which might introduce divergent threads in our code.

In this section we present implementations of programs that generate Mandelbrot sets. Other than comparing performance, the purpose of this example is to show how more realistic and involved programs look like in Accelerate and Obsidian.

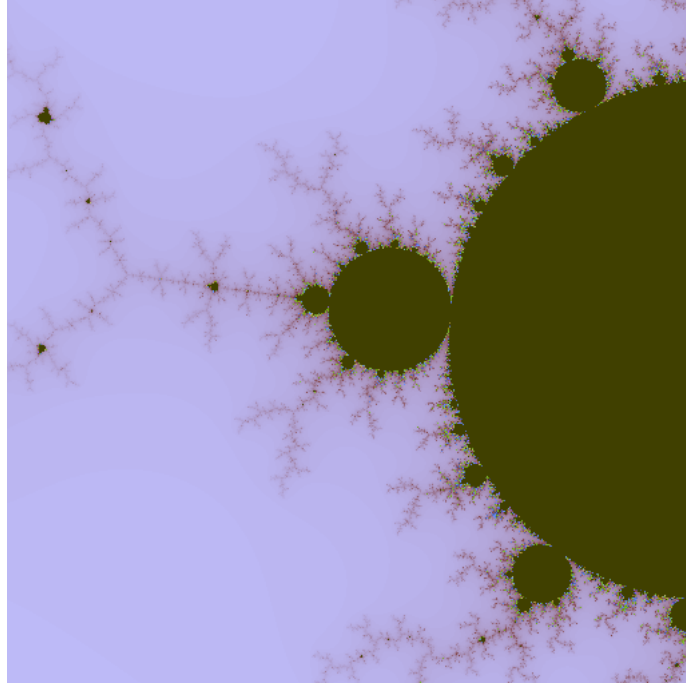


Figure 5.14: A 512×512 Mandelbrot set image. (This image was generated with the Accelerate implementation described in [54]).

```

1 mandelbrot :: ∀ a. (Elt a, IsFloating a)
2   ⇒ Int → Int → Int
3   → Acc (Scalar (View a)) → Acc Bitmap
4 mandelbrot screenX screenY depth view =
5   generate (constant (Z:.screenY:.screenX))
6   (λix → let c = initial ix
7           in prettyRGBA LIMIT
8             $ A.snd
9             $ A.while (λzi → A.snd zi <= LIMIT
10                        &&* dot (A.fst zi) <= 4)
11                        (λzi → lift1 (next c) zi)
12                        (lift (c, constant 0)))
13 where
14   (xmin,ymin,xmax,ymax) = unlift (the view)
15   sizex                  = xmax - xmin
16   sizey                  = ymax - ymin
17   viewx                  = constant (P.fromIntegral screenX)
18   viewy                  = constant (P.fromIntegral screenY)
19
20 initial :: Exp DIM2 → Exp (Complex a)
21 initial ix = lift ( (xmin + (x * sizex) / viewx)
22                   :+ (ymin + (y * sizey) / viewy) )
23 where
24   pr = unindex2 ix
25   x = A.fromIntegral (A.snd pr :: Exp Int)
26   y = A.fromIntegral (A.fst pr :: Exp Int)
27
28 next :: Exp (Complex a)
29   → (Exp (Complex a), Exp Int32) → (Exp (Complex a), Exp Int32)
30 next c (z, i) = (c + (z * z), i+1)
31 dot c = let r := i = unlift c in r*r + i*i
32
33 LIMIT = constant (P.fromIntegral depth)

```

Figure 5.15: Mandelbrot set generator in Accelerate. From [44].

```

1 next :: EFloat → EFloat
2   → (EFloat, EFloat, EWord32) → (EFloat, EFloat, EWord32)
3 next b t (x,y,iter) =
4   ((x*x) - (y*y) + (xmin + t * deltaP), 2*x*y + (ymax - b * deltaQ), iter+1)
5   where
6     deltaP, deltaQ :: EFloat
7     deltaP = (xmax - xmin) / 512.0
8     deltaQ = (ymax - ymin) / 512.0
9
10 cond :: (EFloat, EFloat, EWord32) → EBool
11 cond (x,y,iter) = (((x*x) + (y*y)) <= 4) && iter <= 512
12
13 iters :: EWord32 → EWord32 → Program Thread EW8
14 iters bid tid =
15   do (_,_,c) ← seqUntil (next bid' tid') cond (0,0,1)
16   return (color c)
17   where
18     color c = (w32ToW8 (c `mod` 16)) * 16
19     tid'    = w32ToF tid
20     bid'    = w32ToF bid
21
22 genRect :: EWord32 → Word32
23   → (EWord32 → EWord32 → SPush Thread b)
24   → DPush Grid b
25 genRect bs ts p = asGrid
26   $ mkPull bs
27   $ λbid → asBlock $ mkPull ts (p bid)
28
29 mandel :: EWord32 → Word32 → DPush Grid EW8
30 mandel width height = genRect width height body
31   where
32     body i j = execThread' (iters i j)

```

Figure 5.16: Mandelbrot set generator in Obsidian. From [39].

5.4.1 Mandelbrot Sets in Accelerate

Figure 5.15 shows the Mandelbrot set generator available in the literature. In this program we can appreciate some interesting uses of the functions we introduced in Section 3.2. The `while` in line 9 computes the next value in the iteration as long as the condition evaluates to `True`. In this case the condition is that $|z| < 4$ and that the divergence test stops at the `LIMIT` (line 33). As discussed in the introduction to this section, this test introduces threads that could diverge. Depending on the result of the test different threads can take different execution paths (we discussed this when we introduced Obsidian arrays in Section 3.1.2). In this case however, threads exit the `while`-loop as soon as they fail the test and then simply wait for all the other threads to finish. So the conditional test does not reduce the benefits of parallel execution.

5.4.2 Mandelbrot Sets in Obsidian

Figure 5.16 shows the Obsidian implementation of the Mandelbrot set generator. Similar to the Accelerate implementation, the program features functions `iters` (lines 13-20) and `cond` (lines 10-11) to express the iteration of the recurrence formula and test for divergence.

This program computes one pixel per thread (`execThread'` in line 32). Each pixel is computed by iterating until the limit, which in this case is hard-coded to 512 (line 11). The function `iters` expresses the iteration at the Thread level. Again, the possibility of divergent threads is removed by the way the loop is expressed. Each thread ends the loop as soon as it fails the test and it then simply waits for the rest of the threads to finish.

Size	Accelerate	Obsidian
512x512	2.56	1.19
1024x1024	15.1	4.24

Table 5.2: Performance of Mandelbrot set generators in Accelerate and Obsidian. Times are in milliseconds.

5.4.3 Mandelbrot Sets Results

Table 5.2 shows the performance of both programs generating Mandelbrot set images. We start to see an interesting trend: Obsidian performs, once again, better than Accelerate.

In this case, the difference in performance can also be explained by the following factors:

- **Kernels:** The Obsidian program generates a single kernel and we have complete control over what is happening inside the GPU. The Accelerate version produces kernels that do not fully occupy the GPU hardware and the instantiated algorithmic skeletons might not be sufficient to generate the best possible code.
- **Colouring:** both programs transform the computed pixels into RGBA values. The Accelerate function `prettyRGBA` has a conditional that could introduce divergent threads and slow down the program. In Obsidian, the colour mapping is done on the fly, in a less elegant but faster way.

5.5 Sorting

Sorting is usually done on the CPU given the inherent data-dependencies. Even though this limits the possibilities of sorting on GPUs, many attempts at doing so exist. In this section we compare the sorting programs available in the literature.

Sorting algorithms are usually used as building blocks in many applications so we can expect a good library to include some pre-defined sorting functions. The Thrust library (introduced in Section 2.5.2) includes such functions, but Accelerate and Obsidian, probably because they are still in development, do not include any of these. Nevertheless, different sorting experiments exist in both languages.

```

1 sort :: Radix a => Acc (Vector a) -> Acc (Vector a)
2 sort = sortBy id
3
4 sortBy :: ∀ a r. (Elt a, Radix r)
5         => (Exp a -> Exp r) -> Acc (Vector a) -> Acc (Vector a)
6 sortBy rdx arr = foldr1 (>->) (P.map radixPass [0..p-1]) arr
7   where
8     p = passes (undefined :: r)
9     --
10    deal f x      = let (a,b) = unlift x in (f ==* 0) ? (a,b)
11    radixPass k v = let k'    = unit (constant k)
12                   flags    = A.map (radix (the k') . rdx) v
13                   idown     = prescanl (+) 0 . A.map (xor 1) $ flags
14                   iup       = A.map (size v - 1 -) . prescanr (+) 0 $ flags
15                   index     = A.zipWith deal flags (A.zip idown iup)
16                   in

```

Figure 5.17: Radix Sort in Accelerate.

5.5.1 Sorting in Accelerate: Radix Sort

In Figure 5.17 we show the implementation of radix sort available in [44]. The interesting functions are `radixPass` (lines 11-15) and `>->` (line 6): `radixPass` sorts the vector `v` based on the value of bit `k`, `>->` is a library function that pipes two array computations together. This enables multiple passes of `radixPass` over the array until it is sorted.

5.5.2 Sorting in Obsidian: Sorting Network

```
1 -- Sorter
2 sortObs :: (Scalar a, Ord a) => DPull (Exp a) -> DPush Grid (Exp a)
3 sortObs arr = asGridMap sort (splitUp 1024 arr)
4
5 sort :: ∀ a . (Scalar a, Ord a) => SPull (Exp a) -> SPush Block (Exp a)
6 sort = divideAndConquer $ shexRev' compareSwap
7
8 divideAndConquer :: ∀ a . Data a
9                 => (∀ t . (Array (Push t), Compute t)
10                  => SPull a -> SPush t a)
11                 => SPull a -> SPush Block a
12 divideAndConquer f arr = execBlock $ doIt (logLen - 1) arr
13   where logLen = logBaseI 2 (len arr)
14         doIt 0 a =
15             do
16                 return $ (f :: SPull a -> SPush Block a) a
17
18         doIt n a | currLen > 1024 = blockBody
19                 | currLen > 32    = warpBody
20                 | otherwise      = threadBody
21   where
22     currLen = 2^(logLen - n)
23     arrs = splitUp currLen a
24     blockBody =
25         do
26             arr' ← compute
27                   $ asBlockMap (f :: SPull a -> SPush Block a)
28                   $ arrs
29             doIt (n - 1) arr'
30     warpBody =
31         do
32             arr' ← compute
33                   $ asBlockMap (f :: SPull a -> SPush Warp a)
34                   $ arrs
35             doIt (n - 1) arr'
36     threadBody =
37         do
38             arr' ← compute
39                   $ asBlockMap (f :: SPull a -> SPush Thread a)
40                   $ arrs
41             doIt (n - 1) arr'
```

Figure 5.18: Sorting Network in Obsidian. The definitions of `compareSwap` and `shexRev'` have been omitted.

There are some interesting sorting experiments implemented in Obsidian. In [42] the authors describe counting and occurrence sort. In Figure 5.18 we show an attempt at implementing a sorting network in Obsidian adapted from [39].

The interesting bit of this example lies in lines 8-41, where the computation is described at the different levels of the GPU hierarchy. Note that the computation is split across blocks and warps depending on the size of the input which has to be a power of 2 given that CUDA warps consist of 32 threads (see Section 2.3.3). Moreover, it is interesting to compare the Obsidian implementation of the algorithm to the CUDA code automatically

```

1 void thrustSort (int numElements) {
2
3     //declare and allocate memory
4     thrust::host_vector<float> h_keys(numElements);
5     thrust::host_vector<float> h_keysSorted(numElements);
6
7     // Fill up with some random data
8     thrust::default_random_engine rng(clock());
9     thrust::uniform_real_distribution<float> u01(0, 1);
10
11     for (int i = 0; i < (int)numElements; i++)
12         h_keys[i] = u01(rng);
13
14     // Copy data onto the GPU
15     thrust::device_vector<float> d_keys = h_keys;
16     thrust::sort(d_keys.begin(), d_keys.end());
17
18     // Get results back to host
19     thrust::copy(d_keys.begin(), d_keys.end(), h_keysSorted.begin());
20 }

```

Figure 5.19: Sorting an array on the GPU using Thrust.

generated by Obsidian. Although we do not include the code in the appendix because we need to respect a page-limit, it suffices to say that the generated CUDA code is more than 2000 lines long!

Given the difficulties of mapping such algorithms to GPUs, this experiment relies heavily on shared memory and is hard-coded to only work on arrays of 1024 elements to avoid any errors. We included it here only to show how we can approach sorting using Obsidian. We did not use this program in the benchmarks. See Section 6.2.2 for an idea of how to further improve this program.

5.5.3 Sorting in CUDA: The Thrust Library

Thrust provides a function to sort elements of a vector. In Figure 5.19 we show how to sort a vector using this library. This time the code is much simpler than a plain CUDA program given that Thrust raises the level of abstraction. It is even simpler than the sorting programs implemented in Accelerate and Obsidian shown in this section.

Instead of using the `cudaMalloc` and `cudaMemcpy` functions, memory allocation happens implicitly when declaring the vectors (lines 4,5). Transfers are also hidden: in line 15, the assignment triggers data transfers between the host and the device.

5.5.4 Sorting Benchmarks

In Figure 5.20 we show the speeds of Accelerate and Thrust. The Thrust sorting function outperforms the Accelerate implementation by a big margin. This was the expected result. Both Accelerate and Obsidian need to improve and provide reliable library functions if they want to compete against Thrust and similar CUDA libraries.

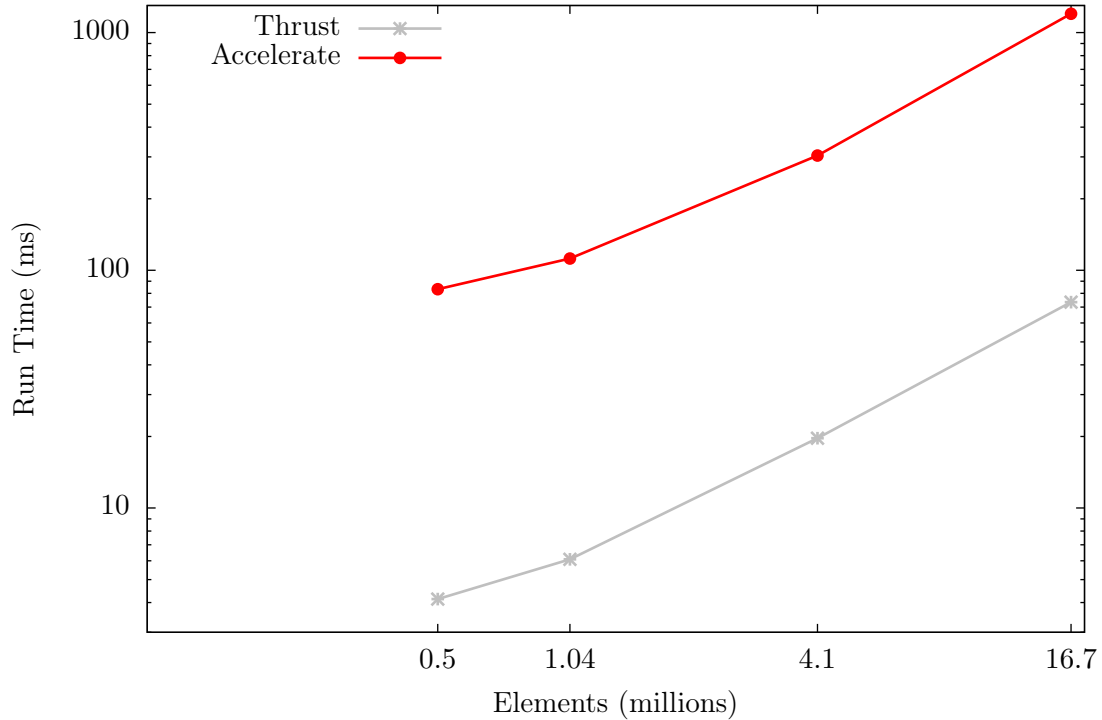


Figure 5.20: Sorting: Thrust vs Accelerate. Note the log scale.

5.6 Language Tweaks and Bugs

During implementation and set-up of the benchmarks we faced challenges that we describe in this section.

5.6.1 Obsidian Tweaks

While setting up the Obsidian benchmarks to work with the Criterion library, we realised that the results were not accurate given that Criterion was also measuring the time that Obsidian took compiling the CUDA binaries. In order to take the compilation step out of the benchmarks and be able to use Criterion, we needed some Obsidian data-types to be strict. After email correspondence with Obsidian’s lead developer, strictness was added to the corresponding data-types. Because of this, we ended up using an Obsidian version that has not yet been released. For clarity, the correspondence with the Obsidian developer is given in Appendix C.

5.6.2 Bug in Accelerate

We found a bug while testing the Accelerate matrix multiplication program. The error appears with matrices of size 8192×8192 or bigger. Even though we submitted the bug to the Accelerate team⁶, at the time of this writing the exact cause and solution have not been found. The bug is most likely related with a problem with memory allocation on the GPU. The device used for our tests has 4 GB of memory (see Section 5.1) while the

⁶The bug submission can be found at: <https://github.com/AccelerateHS/accelerate/issues/250>

program tries to allocate at most 512 MB of data on the GPU.

Encountering such issues is not surprising given that both Accelerate and Obsidian are still experimental languages under active development.

5.7 Summary

In this chapter we measured the performance of both our subject languages, Obsidian and Accelerate, through four test cases and found some interesting results:

1. Parallel Reduce (Section 5.2): We measured the performance of parallel reduce implemented in Accelerate, hand-written CUDA and Obsidian. We found that the kernels produced by Accelerate are, in this case optimal. They beat both the CUDA and Obsidian versions. We also found that when the compilation and data-transfers times are added up, the hand-written CUDA program becomes the fastest one while the Obsidian and Accelerate programs perform almost identically.
2. Matrix Multiplication (Section 5.3): The hand-written CUDA version of this program hugely outperforms the Accelerate and Obsidian versions. We showed that the Obsidian version can be optimised by using shared memory but that the benefits of using this type of memory are apparent only with matrices of more than 16 million elements. We found a bug in Accelerate (see Section 5.6) related to memory allocation on the GPU. This bug has not yet been resolved.
3. Mandelbrot Sets (Section 5.4): The purpose of this example was to show how a realistic program looks like in Accelerate and Obsidian. We described the use of iteration functions provided by both languages. We saw how, in cases where conditionals are unavoidable, thread-divergence can still be avoided. We saw that that programming in Accelerate is closer to programming in idiomatic Haskell while programming in Obsidian feels closer to programming in CUDA given that we need to describe the computation at the different levels of the GPU hierarchy.
4. Sorting (Section 5.5): Sorting is an important task that is not easy to parallelise. We discussed the importance of providing standard sorting functions and saw that neither Obsidian nor Accelerate provide such functions yet. We used Thrust to show how real-world CUDA libraries look like and to give a different view of the speed gap between our subject languages and libraries that are currently the top of the league.

6 Conclusion and Further Work

The main objective of our work was to compare programming languages that attempt to simplify the process of writing GPU programs against the current languages used.

We found these alternatives in the land of functional languages. We used Accelerate and Obsidian, two DSLs embedded in Haskell, as the subjects of our investigation. We aimed at comparing both languages as fully as possible, choosing measurable and specific points of comparison other than performance.

Other than [31], to the best of our knowledge, this is the only other report that compares Accelerate and Obsidian also comparing matters other than speed.

In this chapter we present our findings and results, and suggest avenues of future explorations.

6.1 What we Found

Our findings can be classified into two sections: Section 6.1.2 discusses what we found with respect to the languages' features and design. Section 6.1.1 describes what we found in terms of speed.

6.1.1 Performance

While the set of benchmarks in Chapter 5 is not at all exhaustive we found that, overall, hand-written CUDA is still the language to beat and that Obsidian is faster than Accelerate.

We found that with simple highly data-parallel tasks (such as parallel reduce described in Section 5.2), Accelerate produces optimal kernels that run faster than Obsidian ones. But Accelerate is penalised by its handling of data: Obsidian, even without producing optimal kernels, performs as fast as Accelerate when we take into account data-transfer times. In Section 5.3 and Section 5.4 where we compared the overall speed of more complex tasks, we found that Obsidian always outperformed Accelerate.

These results show that the benefits of hand-optimised algorithmic skeletons are apparent only when dealing with simple tasks that can be perfectly matched to one of these skeletons. But when dealing with more involved tasks, where the mapping of computations to the Accelerate primitives is not so easy, the greater control over the GPU, offered by Obsidian, enables us to make a better use of the available hardware and create faster programs.

Although Obsidian is the fastest of our two subject languages, it is still not able to even come close to the performance of hand-tuned CUDA programs. In all our test cases we found that, when considering the overall run-times of the programs, hand-optimised CUDA always performed better than Accelerate and Obsidian.

Developers needing very fast programs that are starting development from scratch are recommended to stick to CUDA at least for the time being. While our hope is that one

day functional languages for GPU programming will be the option to beat, that day is not here yet.

6.1.2 The Languages

Other than comparing speed our aim was to explore the design decisions and features of both languages (see Table 4.1).

Through our work in Chapter 4 we saw that both languages solve the abstraction issue to a great extent: both Accelerate and Obsidian raise the level of abstraction and hide many hardware details, but they do so in different styles and levels. On the one hand Accelerate hides any notion of GPU and, by separating the language into a front-end and back-end, opens the possibility to target different architectures and devices. Obsidian, on the other hand, raises the level of abstraction without limiting the developer’s control over the GPU internals.

We also found out that both languages owe much of their features to Haskell. At the beginning using Haskell as their host might have been seen as an arbitrary choice but after distilling the expressiveness (Section 4.3), and safety properties (Section 4.4) we understood that only a pure language, with a very rigid type-system, such as Haskell, could be used to achieve what Obsidian and Accelerate have achieved to this day: developers can now produce GPU programs without leaving the comfort of Haskell.

Accelerate and Obsidian are two strong candidates to attack general-purpose GPU programming from a functional perspective. While they are similar languages and try to solve the same issue they do so, as we have seen, in different styles and are therefore suited for different kinds of applications. Although the ultimate use decision will be in the hands of each developer, we would recommend the use of Accelerate when the aim is that of speeding-up portions of existing Haskell programs that could be easily parallelised, while we would recommend the use of Obsidian when the nature of the computation can benefit both from a functional style of programming and greater control over the GPU internals.

6.2 Future Work

The work and languages presented in this report are by no means concluded. In this section we suggest different ways in which our work can be extended (Section 6.2.1) and suggest improvements to both Accelerate and Obsidian (Section 6.2.2).

6.2.1 Improving our Work

Our work could be extended in a number of ways:

- More and improved test cases: This project surveyed and compared existing implementations of the selected programs (parallel reduce, matrix multiplication, Mandelbrot sets and sorting). In order to give a more complete comparison of Accelerate and Obsidian it might be interesting to develop new and improved test cases. In particular we could extend two of our test cases as follows:
 1. Matrix Multiplication: The Accelerate matrix multiplication could be improved by using a different matrix representation since the cube structure used is slow and causes errors with big inputs (see Section 5.6.2). We tried to improve the Obsidian matrix multiplication by using shared memory. It could be further

improved by mimicking the hand-written CUDA program as we suggested in Section 5.3.2.

2. Sorting: The sorting algorithms that we used in Section 5.5 are by no means optimal. The Obsidian version is not even general enough to handle different input sizes. An interesting endeavour would be to write sorting kernels that use the sorting network shown in Section 5.5.2 as a building block to sort sub-arrays of 1024 elements. Also useful would be to develop suitable sorting functions both in Accelerate and Obsidian and include them in the APIs.
- Compare memory use and data-transfers: In Section 5.2 we saw that the way Accelerate and Obsidian transfer data between the host and the device can have a big impact on performance. We could expand the comparisons presented in this report to include benchmarks of memory usage and data management features. This, we think, would be a valuable source of additional information available to developers.

6.2.2 Improving the Languages

The experience we gained by working with Accelerate and Obsidian has enabled us to make the following recommendations:

- Accelerate: (1) a valuable addition would be that of giving at least a limited amount of control over the GPU internals. We think that letting the developer choose the number of threads and blocks to use as well as the type of memory would be a valuable addition. We know that this might be very hard and might even involve re-writing most of the CUDA back-end. (2) The addition of functions to produce stand-alone kernels following Obsidian's style would be valuable to developers. The current way of seeing the generated CUDA code (using the debug flags) is cumbersome.
- Obsidian: (1) Change the array representation to only use one type of arrays (we mentioned this already in Section 3.1.2). (2) It is of utmost importance to provide developers with a more complete library of standard functions in the form of an API. (3) Provide better documentation: Accelerate provides both and it was much easier to learn and start to program with. Learning Obsidian was done by trial and error reading past publications and trying to understand the available under-documented examples.

Finally, as we saw in Section 4.4, even though both languages exploit Haskell's type-system to avoid many errors, it would be beneficial if both languages provided means to propagate CUDA errors into Haskell.

A Automatically Generated Code

This appendix contains VHDL code produced by Lava and CUDA code produced by Obsidian and Accelerate. We show them in the same order as they appear in the body of the report.

A.1 VHDL Code Produced by Lava

This code is generated by the example in Figure 2.13.

```
1 -- Generated by Lava 2000
2
3 use work.all;
4
5 entity
6   halfAdd
7 is
8 port
9   -- clock
10  ( clk : in bit
11
12   -- inputs
13   ; inp_1 : in bit
14   ; inp_2 : in bit
15
16   -- outputs
17   ; outp_0 : out bit
18   ; outp_1 : out bit
19 );
20 end entity halfAdd;
21
22 architecture
23   structural
24 of
25   halfAdd
26 is
27   signal w1 : bit;
28   signal w2 : bit;
29   signal w3 : bit;
30   signal w4 : bit;
31 begin
32   c_w2      : entity id      port map (clk, inp_1, w2);
33   c_w3      : entity id      port map (clk, inp_2, w3);
34   c_w1      : entity xor2    port map (clk, w2, w3, w1);
35   c_w4      : entity and2    port map (clk, w2, w3, w4);
36
37   -- naming outputs
38   c_outp_0   : entity id      port map (clk, w1, outp_0);
39   c_outp_1   : entity id      port map (clk, w4, outp_1);
40 end structural;
```


A.2 CUDA Source Code Produced by Accelerate and Obsidian

A.2.1 Parallel Reduce in CUDA produced by Accelerate

The code was generated by the parallel reduce program shown in Figure 5.2. The kernel `foldAll` corresponds to an Accelerate algorithmic skeleton.

```
1 #include <accelerate_cuda.h>
2 extern "C" __global__ void
3 foldAll(const Int64 shIn0_0, const double* __restrict__ arrIn0_0,
4         const Int64 shOut_0, double* __restrict__ arrOut_0)
5 {
6     extern volatile __shared__ double sdata0[];
7     double x0;
8     double y0;
9     double z0;
10    const Int64 sh0 = shIn0_0;
11    const int shapeSize = sh0;
12    const int gridSize = blockDim.x * gridDim.x;
13    int ix = blockDim.x * blockIdx.x + threadIdx.x;
14
15    if (ix < shapeSize) {
16        y0 = arrIn0_0[ix];
17        for (ix += gridSize; ix < shapeSize; ix += gridSize) {
18            x0 = arrIn0_0[ix];
19            z0 = y0 + x0;
20            y0 = z0;
21        }
22    }
23    sdata0[threadIdx.x] = y0;
24    ix = min(shapeSize - blockDim.x * blockIdx.x, blockDim.x);
25    __syncthreads();
26    if (threadIdx.x + 512 < ix) {
27        x0 = sdata0[threadIdx.x + 512];
28        z0 = y0 + x0;
29        y0 = z0;
30    }
31    __syncthreads();
32    sdata0[threadIdx.x] = y0;
33    __syncthreads();
34    if (threadIdx.x + 256 < ix) {
35        x0 = sdata0[threadIdx.x + 256];
36        z0 = y0 + x0;
37        y0 = z0;
38    }
39    __syncthreads();
40    sdata0[threadIdx.x] = y0;
41    __syncthreads();
42    if (threadIdx.x + 128 < ix) {
43        x0 = sdata0[threadIdx.x + 128];
44        z0 = y0 + x0;
45        y0 = z0;
46    }
47    __syncthreads();
48    sdata0[threadIdx.x] = y0;
49    __syncthreads();
50    if (threadIdx.x + 64 < ix) {
51        x0 = sdata0[threadIdx.x + 64];
52        z0 = y0 + x0;
53        y0 = z0;
54    }
55    __syncthreads();
56    sdata0[threadIdx.x] = y0;
```

```

57  __syncthreads();
58  if (threadIdx.x < 32) {
59      if (threadIdx.x + 32 < ix) {
60          x0 = sdata0[threadIdx.x + 32];
61          z0 = y0 + x0;
62          y0 = z0;
63          sdata0[threadIdx.x] = y0;
64      }
65      if (threadIdx.x + 16 < ix) {
66          x0 = sdata0[threadIdx.x + 16];
67          z0 = y0 + x0;
68          y0 = z0;
69          sdata0[threadIdx.x] = y0;
70      }
71      if (threadIdx.x + 8 < ix) {
72          x0 = sdata0[threadIdx.x + 8];
73          z0 = y0 + x0;
74          y0 = z0;
75          sdata0[threadIdx.x] = y0;
76      }
77      if (threadIdx.x + 4 < ix) {
78          x0 = sdata0[threadIdx.x + 4];
79          z0 = y0 + x0;
80          y0 = z0;
81          sdata0[threadIdx.x] = y0;
82      }
83      if (threadIdx.x + 2 < ix) {
84          x0 = sdata0[threadIdx.x + 2];
85          z0 = y0 + x0;
86          y0 = z0;
87          sdata0[threadIdx.x] = y0;
88      }
89      if (threadIdx.x + 1 < ix) {
90          x0 = sdata0[threadIdx.x + 1];
91          z0 = y0 + x0;
92          y0 = z0;
93          sdata0[threadIdx.x] = y0;
94      }
95  }
96  if (threadIdx.x == 0) {
97      if (shapeSize > 0) {
98          if (gridDim.x == 1) {
99              x0 = 0.0;
100             z0 = y0 + x0;

```

A.2.2 Parallel Reduce in CUDA produced by Obsidian

The code was generated by the parallel reduce program shown in Figure 5.3.

```
1 #include <stdint.h>
2 extern "C" __global__ void reduce_kernel(double* input0, uint32_t n0,
3                                           double* output1)
4 {
5     __shared__ uint8_t sbase[12288U];
6     uint32_t bid = blockIdx.x;
7     uint32_t tid = threadIdx.x;
8     volatile double* arr0 = (volatile double*) (sbase + 128);
9     volatile double* arr1 = (volatile double*) (sbase + 0);
10    double* arr2 = (double*) (sbase + 8192);
11    double* arr3 = (double*) (sbase + 0);
12    double* arr4 = (double*) (sbase + 0);
13    double* arr5 = (double*) (sbase + 2048);
14    double* arr6 = (double*) (sbase + 0);
15    volatile double* arr7 = (volatile double*) (sbase + 512);
16    volatile double* arr8 = (volatile double*) (sbase + 0);
17    volatile double* arr9 = (volatile double*) (sbase + 128);
18    double v10;
19
20    for (int b = 0; b < n0 / 32768U / gridDim.x; ++b) {
21        bid = blockIdx.x * (n0 / 32768U / gridDim.x) + b;
22        v10 = input0[bid * 32768U + tid];
23        for (int i11 = 0; i11 < 31U; ++i11) {
24            v10 = v10 + input0[bid * 32768U + (tid + 1024U * (i11 + 1U))];
25        }
26        for (int i12 = 0; i12 < 1U; ++i12) {
27            arr3[tid] = v10;
28        }
29        __syncthreads();
30        if (threadIdx.x < 512) {
31            tid = 0 + threadIdx.x;
32            arr2[tid] = arr3[tid] + arr3[tid + 512U];
33        }
34        tid = threadIdx.x;
35        __syncthreads();
36        if (threadIdx.x < 256) {
37            tid = 0 + threadIdx.x;
38            arr4[tid] = arr2[tid] + arr2[tid + 256U];
39        }
40        tid = threadIdx.x;
41        __syncthreads();
42        if (threadIdx.x < 128) {
43            tid = 0 + threadIdx.x;
44            arr5[tid] = arr4[tid] + arr4[tid + 128U];
45        }
46        tid = threadIdx.x;
47        __syncthreads();
48        if (threadIdx.x < 64) {
49            tid = 0 + threadIdx.x;
50            arr6[tid] = arr5[tid] + arr5[tid + 64U];
51        }
52        tid = threadIdx.x;
53        __syncthreads();
54        if (threadIdx.x < 32) {
55            tid = 0 + threadIdx.x;
56            arr7[tid] = arr6[tid] + arr6[tid + 32U];
57        }
58        tid = threadIdx.x;
59        if (threadIdx.x < 16) {
```

```

60         tid = 0 + threadIdx.x;
61         arr8[tid] = arr7[tid] + arr7[tid + 16U];
62     }
63     tid = threadIdx.x;
64     if (threadIdx.x < 8) {
65         tid = 0 + threadIdx.x;
66         arr9[tid] = arr8[tid] + arr8[tid + 8U];
67     }
68     tid = threadIdx.x;
69     if (threadIdx.x < 4) {
70         tid = 0 + threadIdx.x;
71         arr1[tid] = arr9[tid] + arr9[tid + 4U];
72     }
73     tid = threadIdx.x;
74     if (threadIdx.x < 2) {
75         tid = 0 + threadIdx.x;
76         arr0[tid] = arr1[tid] + arr1[tid + 2U];
77     }
78     tid = threadIdx.x;
79     if (threadIdx.x < 1) {
80         tid = 0 + threadIdx.x;
81         output1[bid + tid] = arr0[0U] + arr0[1U];
82     }
83     tid = threadIdx.x;
84     bid = blockIdx.x;
85     __syncthreads();
86 }
87 bid = gridDim.x * (n0 / 32768U / gridDim.x) + blockIdx.x;
88 if (blockIdx.x < n0 / 32768U % gridDim.x) {
89     v10 = input0[bid * 32768U + tid];
90     for (int i11 = 0; i11 < 31U; ++i11) {
91         v10 = v10 + input0[bid * 32768U + (tid + 1024U * (i11 + 1U))];
92     }
93     for (int i12 = 0; i12 < 1U; ++i12) {
94         arr3[tid] = v10;
95     }
96     __syncthreads();
97     if (threadIdx.x < 512) {
98         tid = 0 + threadIdx.x;
99         arr2[tid] = arr3[tid] + arr3[tid + 512U];
100    }
101    tid = threadIdx.x;
102    __syncthreads();
103    if (threadIdx.x < 256) {
104        tid = 0 + threadIdx.x;
105        arr4[tid] = arr2[tid] + arr2[tid + 256U];
106    }
107    tid = threadIdx.x;
108    __syncthreads();
109    if (threadIdx.x < 128) {
110        tid = 0 + threadIdx.x;
111        arr5[tid] = arr4[tid] + arr4[tid + 128U];
112    }
113    tid = threadIdx.x;
114    __syncthreads();
115    if (threadIdx.x < 64) {
116        tid = 0 + threadIdx.x;
117        arr6[tid] = arr5[tid] + arr5[tid + 64U];
118    }
119    tid = threadIdx.x;
120    __syncthreads();
121    if (threadIdx.x < 32) {
122        tid = 0 + threadIdx.x;

```

```

123         arr7[tid] = arr6[tid] + arr6[tid + 32U];
124     }
125     tid = threadIdx.x;
126     if (threadIdx.x < 16) {
127         tid = 0 + threadIdx.x;
128         arr8[tid] = arr7[tid] + arr7[tid + 16U];
129     }
130     tid = threadIdx.x;
131     if (threadIdx.x < 8) {
132         tid = 0 + threadIdx.x;
133         arr9[tid] = arr8[tid] + arr8[tid + 8U];
134     }
135     tid = threadIdx.x;
136     if (threadIdx.x < 4) {
137         tid = 0 + threadIdx.x;
138         arr1[tid] = arr9[tid] + arr9[tid + 4U];
139     }
140     tid = threadIdx.x;
141     if (threadIdx.x < 2) {
142         tid = 0 + threadIdx.x;
143         arr0[tid] = arr1[tid] + arr1[tid + 2U];
144     }
145     tid = threadIdx.x;
146     if (threadIdx.x < 1) {
147         tid = 0 + threadIdx.x;
148         output1[bid + tid] = arr0[0U] + arr0[1U];
149     }
150     tid = threadIdx.x;
151 }
152 bid = blockIdx.x;
153 __syncthreads();
154 }

```

A.2.3 Matrix Multiplication in CUDA produced by Accelerate

The code shown here corresponds the matrix multiplication shown in Figure 5.8.

```
1 #include <accelerate_cuda.h>
2 extern "C" __global__
3 void fold(const Int64 shIn0_1, const Int64 shIn0_0,
4           const float* __restrict__ arrIn0_0, const Int64 shIn1_1,
5           const Int64 shIn1_0, const float* __restrict__ arrIn1_0,
6           const Int64 shOut_1, const Int64 shOut_0,
7           float* __restrict__ arrOut_0)
8 {
9     extern volatile __shared__ float sdata0[];
10    float x0;
11    float y0;
12    float z0;
13    const Int64 sh2 = min(shIn0_1, shIn0_1);
14    const Int64 sh1 = min(shIn1_0, shIn1_0);
15    const Int64 sh0 = min(shIn0_0, shIn1_1);
16    const int numIntervals = sh2 * sh1;
17    const int intervalSize = sh0;
18    int ix;
19    int seg;
20
21    if (intervalSize == 0 || numIntervals == 0) {
22        const int gridSize = blockDim.x * gridDim.x;
23
24        for (ix = blockDim.x * blockIdx.x + threadIdx.x;
25             ix < shOut_1 * shOut_0; ix += gridSize) {
26            arrOut_0[ix] = 0.0f;
27        }
28        return;
29    }
30    for (seg = blockIdx.x; seg < numIntervals; seg += gridDim.x) {
31        const int start = seg * intervalSize;
32        const int end = start + intervalSize;
33        const int n = min(end - start, blockDim.x);
34
35        if (threadIdx.x >= n)
36            return;
37
38        ix = start - (start & warpSize - 1);
39        if (ix == start || intervalSize > blockDim.x) {
40            ix += threadIdx.x;
41            if (ix >= start) {
42                const Int64 v2_0 = ix;
43                const Int64 v2_1 = v2_0 / min(shIn0_0, shIn1_1);
44                const Int64 v2_2 = v2_1 / min(shIn1_0, shIn1_0);
45                const Int64 v3 = v2_2 % min(shIn0_1, shIn0_1);
46                const Int64 v4 = v2_1 % min(shIn1_0, shIn1_0);
47                const Int64 v5 = v2_0 % min(shIn0_0, shIn1_1);
48
49                y0 = ({
50                    const Int64 v6 = v3 * shIn0_0 + v5;
51
52                    ;
53                    arrIn0_0[v6];
54                }) * ({
55                    const Int64 v7 = v5 * shIn1_0 + v4;
56
57                    ;
58                    arrIn1_0[v7];
59                });
```

```

60     }
61     if (ix + blockDim.x < end) {
62         const Int64 v2_0 = ix + blockDim.x;
63         const Int64 v2_1 = v2_0 / min(shIn0_0, shIn1_1);
64         const Int64 v2_2 = v2_1 / min(shIn1_0, shIn1_0);
65         const Int64 v3 = v2_2 % min(shIn0_1, shIn0_1);
66         const Int64 v4 = v2_1 % min(shIn1_0, shIn1_0);
67         const Int64 v5 = v2_0 % min(shIn0_0, shIn1_1);
68
69         x0 = ({
70             const Int64 v6 = v3 * shIn0_0 + v5;
71
72             ;
73             arrIn0_0[v6];
74         }) * ({
75             const Int64 v7 = v5 * shIn1_0 + v4;
76
77             ;
78             arrIn1_0[v7];
79         });
80         if (ix >= start) {
81             z0 = y0 + x0;
82             y0 = z0;
83         } else {
84             y0 = x0;
85         }
86     }
87     for (ix += 2 * blockDim.x; ix < end; ix += blockDim.x) {
88         const Int64 v2_0 = ix;
89         const Int64 v2_1 = v2_0 / min(shIn0_0, shIn1_1);
90         const Int64 v2_2 = v2_1 / min(shIn1_0, shIn1_0);
91         const Int64 v3 = v2_2 % min(shIn0_1, shIn0_1);
92         const Int64 v4 = v2_1 % min(shIn1_0, shIn1_0);
93         const Int64 v5 = v2_0 % min(shIn0_0, shIn1_1);
94
95         x0 = ({
96             const Int64 v6 = v3 * shIn0_0 + v5;
97
98             ;
99             arrIn0_0[v6];
100         }) * ({
101             const Int64 v7 = v5 * shIn1_0 + v4;
102
103             ;
104             arrIn1_0[v7];
105         });
106         z0 = y0 + x0;
107         y0 = z0;
108     }
109 } else {
110     const Int64 v2_0 = start + threadIdx.x;
111     const Int64 v2_1 = v2_0 / min(shIn0_0, shIn1_1);
112     const Int64 v2_2 = v2_1 / min(shIn1_0, shIn1_0);
113     const Int64 v3 = v2_2 % min(shIn0_1, shIn0_1);
114     const Int64 v4 = v2_1 % min(shIn1_0, shIn1_0);
115     const Int64 v5 = v2_0 % min(shIn0_0, shIn1_1);
116
117     y0 = ({
118         const Int64 v6 = v3 * shIn0_0 + v5;
119
120         ;
121         arrIn0_0[v6];
122     }) * ({

```

```

123         const Int64 v7 = v5 * shIn1_0 + v4;
124
125         ;
126         arrIn1_0[v7];
127     });
128 }
129 sdata0[threadIdx.x] = y0;
130 __syncthreads();
131 if (threadIdx.x + 512 < n) {
132     x0 = sdata0[threadIdx.x + 512];
133     z0 = y0 + x0;
134     y0 = z0;
135 }
136 __syncthreads();
137 sdata0[threadIdx.x] = y0;
138 __syncthreads();
139 if (threadIdx.x + 256 < n) {
140     x0 = sdata0[threadIdx.x + 256];
141     z0 = y0 + x0;
142     y0 = z0;
143 }
144 __syncthreads();
145 sdata0[threadIdx.x] = y0;
146 __syncthreads();
147 if (threadIdx.x + 128 < n) {
148     x0 = sdata0[threadIdx.x + 128];
149     z0 = y0 + x0;
150     y0 = z0;
151 }
152 __syncthreads();
153 sdata0[threadIdx.x] = y0;
154 __syncthreads();
155 if (threadIdx.x + 64 < n) {
156     x0 = sdata0[threadIdx.x + 64];
157     z0 = y0 + x0;
158     y0 = z0;
159 }
160 __syncthreads();
161 sdata0[threadIdx.x] = y0;
162 __syncthreads();
163 if (threadIdx.x < 32) {
164     if (threadIdx.x + 32 < n) {
165         x0 = sdata0[threadIdx.x + 32];
166         z0 = y0 + x0;
167         y0 = z0;
168         sdata0[threadIdx.x] = y0;
169     }
170     if (threadIdx.x + 16 < n) {
171         x0 = sdata0[threadIdx.x + 16];
172         z0 = y0 + x0;
173         y0 = z0;
174         sdata0[threadIdx.x] = y0;
175     }
176     if (threadIdx.x + 8 < n) {
177         x0 = sdata0[threadIdx.x + 8];
178         z0 = y0 + x0;
179         y0 = z0;
180         sdata0[threadIdx.x] = y0;
181     }
182     if (threadIdx.x + 4 < n) {
183         x0 = sdata0[threadIdx.x + 4];
184         z0 = y0 + x0;
185         y0 = z0;

```



```

186     sdata0[threadIdx.x] = y0;
187 }
188 if (threadIdx.x + 2 < n) {
189     x0 = sdata0[threadIdx.x + 2];
190     z0 = y0 + x0;
191     y0 = z0;
192     sdata0[threadIdx.x] = y0;
193 }
194 if (threadIdx.x + 1 < n) {
195     x0 = sdata0[threadIdx.x + 1];
196     z0 = y0 + x0;
197     y0 = z0;
198     sdata0[threadIdx.x] = y0;
199 }
200 }
201 if (threadIdx.x == 0) {
202     x0 = 0.0f;
203     z0 = y0 + x0;
204     y0 = z0;
205     arrOut_0[seg] = y0;
206 }
207 }
208 }

```

A.2.4 Matrix Multiplication in CUDA produced by Obsidian

The code shown here corresponds the matrix multiplication shown in Figure 5.9.

```
1 #include <stdint.h>
2 extern "C" __global__ void matMul_kernel(float* input0, uint32_t n0,
3                                           float* input1, uint32_t n1,
4                                           float* output2)
5 {
6     uint32_t bid = blockIdx.x;
7     uint32_t tid = threadIdx.x;
8     float v4;
9
10    for (int b = 0; b < 8192U / gridDim.x; ++b) {
11        bid = blockIdx.x * (8192U / gridDim.x) + b;
12        for (int i = 0; i < 8; ++i) {
13            tid = i * 1024 + threadIdx.x;
14            v4 = input0[bid * 8192U] * input1[tid];
15            for (int i5 = 0; i5 < 8191U; ++i5) {
16                v4 = v4 + input0[bid * 8192U + (i5 + 1U)]
17                    * input1[(i5 + 1U) * 8192U + tid];
18            }
19            for (int i6 = 0; i6 < 1U; ++i6) {
20                output2[bid * 8192U + tid] = v4;
21            }
22        }
23        tid = threadIdx.x;
24        bid = blockIdx.x;
25        __syncthreads();
26    }
27    bid = gridDim.x * (8192U / gridDim.x) + blockIdx.x;
28    if (blockIdx.x < 8192U % gridDim.x) {
29        for (int i = 0; i < 8; ++i) {
30            tid = i * 1024 + threadIdx.x;
31            v4 = input0[bid * 8192U] * input1[tid];
32            for (int i5 = 0; i5 < 8191U; ++i5) {
33                v4 = v4 + input0[bid * 8192U + (i5 + 1U)]
34                    * input1[(i5 + 1U) * 8192U + tid]; }
35            for (int i6 = 0; i6 < 1U; ++i6) {
36                output2[bid * 8192U + tid] = v4;
37            }
38        }
39        tid = threadIdx.x;
40    }
41    bid = blockIdx.x;
42    __syncthreads();
43 }
```

B CUDA Device Information

This appendix shows the complete information of the Nvidia card we used in our benchmarks.

```
Device 0: GRID K520
CUDA capability:          3.0
CUDA cores:              1536 cores in 8 multiprocessors (192 cores/MP)
Global memory:           4 GB
Constant memory:         64 kB
Shared memory per block: 48 kB
Registers per block:     65536
Warp size:               32
Maximum threads per multiprocessor: 2048
Maximum threads per block: 1024
Maximum grid dimensions: 2147483647 x 65535 x 65535
Maximum block dimensions: 1024 x 1024 x 64
GPU clock rate:          797.0 MHz
Memory clock rate:       2.5 GHz
Memory bus width:        256-bit
L2 cache size:           512 kB
Maximum texture dimensions
  1D:                     65536
  2D:                     65536 x 65536
  3D:                     4096 x 4096 x 4096
Texture alignment:       512 B
Maximum memory pitch:    2 GB
Concurrent kernel execution: Yes
Concurrent copy and execution: Yes, with 2 copy engines
Runtime limit on kernel execution: No
Integrated GPU sharing host memory: No
Host page-locked memory mapping: Yes
ECC memory support:      No
Unified addressing (UVA): Yes
PCI bus/location:        0/3
Compute mode:            Default
  Multiple host threads can use the device simultaneously
```

C Email Correspondence

In this appendix we show the relevant email correspondence with Obsidian developers. Through these conversations we requested the addition of strictness properties to the language as discussed in Section 5.6.1.

Re: Obsidian version

Alberto Sadde O. <aes530@york.ac.uk>
To: Joel Svensson <bo.joel.svensson@gmail.com>

Tue, Mar 17, 2015 at 1:36 PM

Hi Joel,

I write to you again because I've run out of options!
I have been running many benchmarks without problems but haven't managed to use Criterion and my supervisor thinks it would be a good idea to use it!


I tried to using the captureIO function and forcing it before passing it to the criterion benchmarks but it still seems that it recompiles everytime it is used.


I am sending you the example I'm trying to get working. I don't know if its something strictly related to Criterion or if I'm doing something wrong with Obsidian. I added to NFDdata instances in order to use Criterion.

Hope you can help me!
Thanks,

Alberto
[Quoted text hidden]

2 attachments

 ReduceCriterion.hs
2K

 ReduceObsidian.hs
3K

Re: Obsidian version

Joel Svensson <bo.joel.svensson@gmail.com>
To: "Alberto Sadde O." <aes530@york.ac.uk>

Wed, Mar 18, 2015 at 5:00 PM

I Added some strictness annotations to the data types defined in Run.CUDA.Exec.
This is available on the master-dev branch (along with possibly some other small changes compared to master)

Don't know yet if this does it... Since I haven't had the time to try out your code. (Busy with a deadline again).

/Joel
[Quoted text hidden]

Re: Obsidian version

Joel Svensson <bo.joel.svensson@gmail.com>
To: "Alberto Sadde O." <aes530@york.ac.uk>

Tue, Mar 17, 2015 at 3:58 PM

I will look at this some tomorrow.
It does seem strange that you would get recompilations with the setup you are using.
An nvcc recompilation can take many seconds! so its not good if that happens in benchmark loop.

One idea is to use:
!kern <- captureIO ...

To use a bang (!) there you need to turn on a GHC language pragma. Think it is BangPatterns? {-# LANGUAGE BangPatterns 3-}

It also seems a little bit strange that you do a ker <- liftIO kern in the perform function.

One thing that maybe should be done is that the various data types in in run/cuda should be strict.
I can make them strict and do a push... but all that will wait for tomorrow.

/Joel
[Quoted text hidden]

Re: Obsidian version

Alberto Sadde O. <titicosadde@gmail.com>
To: Joel Svensson <bo.joel.svensson@gmail.com>

Thu, Mar 19, 2015 at 10:11 AM

Hi,

I tried your suggestions.
If I only use force the evaluation with ! and don't use ker <- liftIO kern, I get a CUDA error:
CUDA Exception: invalid device context

Using it as I had it originally works but with the unwanted compilation appearing in the benchmarks.

Thanks a lot.

Alberto
[Quoted text hidden]

68

Re: Obsidian version

Joel Svensson <bo.joel.svensson@gmail.com> Thu, Mar 19, 2015 at 10:18 AM
To: "Alberto Sadde O." <titicosadde@gmail.com>

I am thinking that in the new setup you would not need to use evaluate or force... both those should be pointless just do

Ikern <- captureIO ...

The error seems to indicate that there is an issue with initialization (or something)

I will look into the details.
This is not fresh in my memory at this point but I think that withCUDA initializes things.
I think I also saw and initialization in your code. We may need to split this up differently.

Please send me your latest code and I will try to hack something together around it.

Thank you
/Joel

[Quoted text hidden]

Re: Obsidian version

Joel Svensson <bo.joel.svensson@gmail.com> Thu, Mar 19, 2015 at 3:55 PM
To: "Alberto Sadde O." <titicosadde@gmail.com>

Ok fixed it.

If you pull the latest changes on the master-dev branch now it seems to work.

56ms for Obsidian
756ms for sequential

This definitely does not include compilation time.

I made some changes but you will see what those are when you look at the attached files to this mail ..

Thank you for your help.
/Joel

[Quoted text hidden]
Ok good. Thanks
/Joel

[Quoted text hidden]
[Quoted text hidden]

Alberto

[Quoted text hidden]
[Quoted text hidden]

Re: Obsidian version

Alberto Sadde O. <titicosadde@gmail.com> Thu, Mar 19, 2015 at 10:31 AM
To: Joel Svensson <bo.joel.svensson@gmail.com>


Thanks for the help.


The files are basically the same, I just remove the force, evaluate and liftIO and used !.

Alberto

[Quoted text hidden]

2 attachments

 ReduceCriterion.hs
2K

 ReduceObsidian.hs
2K

Bibliography

- [1] F. Xu, A. Khamene, and O. Fluck, “High Performance Tomosynthesis enabled via a GPU-based Iterative Reconstruction Framework,” in *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, ser. Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series, vol. 7258, Feb. 2009, p. 5.
- [2] J. Michalakes and M. Vachharajani, “GPU Acceleration of Numerical Weather Prediction,” *Parallel Processing Letters*, vol. 18, no. 04, pp. 531–548, 2008.
- [3] D. Tarditi, S. Puri, and J. Oglesby, “Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses,” in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII. ACM Press, 2006, pp. 325–335.
- [4] NVIDIA, *CUDA C Programming Guide*. NVIDIA, 2014, [Online] Available: <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#axzz3XHyzp3w> [Accessed 15 November 2014].
- [5] NVIDIA, “NVIDIA’s Next Generation CUDA Compute Architecture,” NVIDIA Corporation, Tech. Rep., 2014, [Online] Available: <http://international.download.nvidia.com/pdf/kepler/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf> [Accessed 15 April 2015].
- [6] G. Hutton, *Programming In Haskell*. Cambridge University Press, 2007.
- [7] M. Lipovaca, *Learn You a Haskell for Great Good!: A Beginner’s Guide*. No Starch Press, 2011, [Online] Available: <http://learnyouahaskell.com/chapters> [Accessed 20 November 2014].
- [8] B. O’Sullivan, J. Goerzen, and D. B. Stewart, *Real World Haskell*. O’Reilly Media, 2008, [Online] Available: <http://book.realworldhaskell.org/read/> [Accessed 10 November 2014].
- [9] S. Marlow, “Haskell 2010 Language Report,” 2010, [Online] Available: <http://haskell.org/onlinereport/haskell2010> [Accessed 13 November 2014].
- [10] P. Hudak, J. Hughes, S. P. Jones, and P. Wadler, “A History of Haskell: Being Lazy With Class,” in *Third ACM SIGPLAN History of Programming Languages Conference*. ACM Press, 2007, pp. 1–55.
- [11] S. Thompson, *Haskell: The Craft of Functional Programming*, 3rd ed. Addison-Wesley, 1999.
- [12] B. Barney, *Introduction to Parallel Computing*. Lawrence Livermore National Laboratory, 2014, [Online] Available: http://computing.llnl.gov/tutorials/parallel_comp/ [Accessed 21 April 2015].
- [13] B. Lisper, “Data Parallelism and Functional Programming,” in *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*. Springer Berling Heidelberg, 1996, vol. 1132, pp. 220–251.

- [14] C. McClanahan, “History and Evolution of GPU Architecture,” 2010, [Online] Available: <http://mcclanahoochie.com/blog/wp-content/uploads/2011/03/gpu-hist-paper.pdf> [Accessed 5 January 2015].
- [15] K. Gray and M. Corporation, *Microsoft DirectX 9 Programmable Graphics Pipeline*, ser. Developer Reference Series. Microsoft Press, 2003.
- [16] M. Segal and K. Akeley, “The OpenGL Graphics System: A Specification,” 2010, [Online] Available: <https://www.opengl.org/registry/doc/glspec13.pdf> [Accessed 22 April 2015].
- [17] D. B. Kirk and W. mei W Hwu, *Programming Massively Parallel Processors*. Elsevier Inc., 2010.
- [18] NVIDIA, *CUDA C Best Practices Guide*. NVIDIA, 2014, [Online] Available: <http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#axzz3XHxyzp3w> [Accessed 10 January 2015].
- [19] M. Mernik, J. Heering, and A. M. Sloane, “When and How to Develop Domain-Specific Languages,” *ACM Computing Surveys*, vol. 37, no. 4, pp. 316–344, 2005.
- [20] M. Fowler, *Domain-Specific Languages*, 1st ed. Addison-Wesley Professional, 2010.
- [21] P. Hudak, “Domain Specific Languages,” in *Handbook of Programming Languages*, P. H. Salas, Ed. MacMillan, 1997, pp. 39–60.
- [22] A. Gill, “Domain-specific Languages and Code Synthesis using Haskell,” *Communications of the ACM*, vol. 57, no. 6, pp. 42–49, Jun. 2014.
- [23] D. Ghosh, “DSL for the Uninitiated,” *ACM Queue*, vol. 9, no. 6, pp. 1–11, 2011.
- [24] P. Hudak, “Building Domain-specific Embedded Languages,” *ACM Computing Surveys*, vol. 28, p. 6, 1996.
- [25] M. Carlsson and T. Hallgren, “FUDGETS: A Graphical User Interface in a Lazy Functional Language,” in *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. Copenhagen, Denmark: ACM Press, 1993, pp. 321–330.
- [26] T. Hallgren and M. Carlsson. Fudgets. [Online]. Available: <http://www.altocumulus.org/Fudgets/>
- [27] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, “Lava: Hardware design in Haskell,” in *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 1998, pp. 174–184.
- [28] K. Claessen and M. Sheeran, “A Slightly Revised Tutorial on Lava: A Hardware Description and Verification System,” pp. 1–99, Apr. 2010, [Online] Available: <http://projects.haskell.org/chalmers-lava2000/Doc/tutorial.pdf> [Accessed 10 April 2015].
- [29] J. Fang, A. L. Varbanescu, and H. Sips, “A Comprehensive Performance Comparison of CUDA and OpenCL,” in *2011 International Conference on Parallel Processing (ICPP)*. Delft, The Netherlands: IEEE, 2011, pp. 216–225.

- [30] H. W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Peña, S. Priebe, Á. J. Rebón, and P. W. Trinder, “Comparing Parallel Functional Languages: Programming and Performance,” *Higher-Order and Symbolic Computation*, vol. 16, no. 3, pp. 203–251, 2003.
- [31] C. Ouwehand, P. Hijma, and W. J. Fokkink, “GPU Programming in Functional Languages,” 2013, [Online] Available: <http://hgpu.org/?p=10242> [Accessed 14 April 2015].
- [32] C. Elliott, “Programming Graphics Processors Functionally,” in *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*. Snowbird, Utah: ACM Press, 2004, pp. 45–56.
- [33] G. Mainland and G. Morrisett, “Nikola: Embedding Compiled GPU Functions in Haskell,” in *Proceedings of the third ACM Haskell symposium on Haskell*. ACM Press, 2010, pp. 67–78.
- [34] N. Bell and J. Hoberock, “Thrust: A Productivity-Oriented Library for CUDA,” in *GPU Computing Gems: Jade Edition*, W. mei W. Hwu, Ed. NVIDIA, 2011.
- [35] J. Svensson, “An Embedded Language for Data-Parallel Programming,” 2008, [Online] Available: <http://www.cse.chalmers.se/~joels/writing/joelmscthesis.pdf> [Accessed 10 Febraury 2015].
- [36] B. J. Svensson, “Embedded Languages for Data-parallel Programming,” Ph.D. dissertation, Chalmers University of Technology and Göteborg University, Göteborg, 2013.
- [37] J. Svensson, M. Sheeran, and K. Claessen, “Obsidian: A Domain Specific Embedded Language for Parallel Programming of Graphics Processors,” in *Implementation and Application of Functional Languages*. Springer, 2011, vol. 5836, pp. 156–173.
- [38] R. N. B. J. Svensson and M. Sheeran, “A Language for Nested Data Parallel Design-space Exploration on GPUs,” Indiana University, Tech. Rep., 2014, [Online] Available: <http://www.cs.indiana.edu/pub/techreports/TR712.pdf> [Accessed 19 November 2014].
- [39] B. J. Svensson, “Obsidian: Source code,” [Online] Available: <http://github.com/svenssonjoel/Obsidian> [Accessed 4 March 2015].
- [40] J. Svensson, M. Sheeran, and K. Claessen, “GPGPU Kernel Implementation and Refinement using Obsidian,” *Procedia Computer Science*, vol. 1, no. 1, pp. 2065–2074, 2010, iCCS 2010.
- [41] K. Claessen, M. Sheeran, and B. J. Svensson, “Expressive Array Constructs in an Embedded GPU Kernel Programming Language,” in *DAMP ’12: Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming*. ACM Press, Jan. 2012, pp. 21–30.
- [42] J. D. Svenningsson, B. J. Svensson, and M. Sheeran, “Counting and Occurrence Sort for GPUs Using an Embedded Language,” in *FHPC ’13: Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-Performance Computing*. ACM Press, Sep. 2013, pp. 37–46.

- [43] B. J. Svensson and J. Svenningsson, “Defunctionalizing Push Arrays,” in *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing*, ser. FHPC ’14. ACM Press, 2014, pp. 43–52.
- [44] T. L. McDonell, “Optimising Purely Functional GPU Programs,” Ph.D. dissertation, University Of New South Wales, Sydney, 2014.
- [45] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover, “Accelerating Haskell Array Codes with Multicore GPUs,” in *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, ser. DAMP ’11. ACM Press, 2011, pp. 3–14.
- [46] G. Keller, M. M. T. Chakravarty, R. Leshchinskiy, S. P. Jones, and B. Lippmeier, “Regular, Shape-Polymorphic, Parallel Arrays in Haskell,” in *ICFP ’10: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ACM Press, Sep. 2010, p. 261.
- [47] B. J. Svensson, M. Sheeran, and R. Newton, “Design Exploration through Code-generating DSLs,” *Queue*, vol. 12, no. 4, Apr. 2014.
- [48] “Amazon Web Services,” [Online] Available: <http://aws.amazon.com/>.
- [49] NVIDIA. NVIDIA GPUs. [Online]. Available: <http://developer.nvidia.com/cuda-gpus>
- [50] M. Chakravarty, T. McDonell, R. Everest, and R. Newton, “Accelerate: Source Code,” [Online] Available: <https://github.com/AccelerateHS> [Accessed 4 March 2015].
- [51] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” in *OSDI’04: Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation*. USENIX Association, 2004.
- [52] M. Harris, “Optimizing Parallel Reduction in CUDA,” 2007, [Online] Available: <http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf> [Accessed 13 February 2015].
- [53] J. Demouth, “SHUFFLE (SHFL): Tips and Tricks,” Mar. 2013, [Online] Available: <http://on-demand.gputechconf.com/gtc/2013/presentations/S3174-Kepler-Shuffle-Tips-Tricks.pdf> [Accessed 10 February 2015].
- [54] S. Marlow, *Parallel and Concurrent Programming in Haskell*. O’Reilly Media, 2013.