# Warp Speed Haskell

William Jones
⟨wlj05@doc.ic.ac.uk⟩
Department of Computing
Imperial College London

Supervisor: Dr. Tony Field ⟨ajf@doc.ic.ac.uk⟩
Second Marker: Prof. Susan Eisenbach ⟨sue@doc.ic.ac.uk⟩

December 31, 2009

## Abstract

We present an approach to parallel programming that expresses parallelism through *types*. We use operator overloading to recognise uses of these types and to build a representation of the parallelisable components of a program dynamically at runtime. We implement these ideas within Haskell – a pure functional language with a rich type system. We explore a simple stream-based model of parallelism that is expressed in a platform-independent manner through a suitably defined stream data type and document the development of a code generator for a specific target architecture – the CUDA platform for Nvidia's family of General Purpose Graphical Processing Units (GPGPUs). Experiments with small benchmarks show that extremely efficient CUDA code can be generated from very high-level Haskell source code. Haskell's rewrite system is also shown to be an excellent vehicle for optimising the performance of parallel programs through source-level program transformation. We show that the application of optimisation rules targeted specifically at stream types can lead to substantial performance improvements in some cases.

# Acknowledgements

Firstly, I would like to thank Dr. Tony Field, for dedicating countless hours of his time to listening to me whine about programming. Without his support and guidance this project would not have been realised. I would also like to thank a number of Haskell experts – to Simon Peyton-Jones, Simon Marlow, Andy Cheadle and Tristan Allwood I extend my utmost gratitude for giving me time in their busy schedules to discuss the intricacies of compiling functional programming languages. Finally, I'd like to thank my friends – whom I daren't list explicitly in case I miss someone, and my parents, for putting me through university and giving me the best four years of my life to date.

# Contents

# Introduction 1

Writing parallel software is hard. Non-deterministic execution, race conditions and synchronisation are just a few of the common pitfalls encountered when writing parallel programs. Attempts to rectify these issues in the form of annotations [1] or libraries [2] have been proposed, but these still entail a degree of program *markup*: code that exposes parallelism but that is extraneous to the functionality of the application. Furthermore, heterogeneous platforms such as *graphical processing units* (GPUs) and specialised coprocessors impose further problems such as non-uniform memory access times and the lack of a single global address space.

Consequently, writing general purpose parallel code remains a non-trivial process. Commitments to a paradigm and platform often end up being deeply woven into an application, and portability and robustness suffer as a result. To illustrate this, consider the following very simple C program.

```
float f(float);

...

void main(int argc, char **argv) {
  int i;
  int N = ...;
  float *xs = ...;
  float *ys = ...;

  ...

  for (i = 0; i < N; i++) {
    ys[i] = f(xs[i]);
  }
}
```

Here, the function `f` is applied to each element of an `N` element array of single-precision floating point numbers. If the loop is parallelisable, an equivalent OpenMP (Section 2.6.2) program could be written as follows.

```
float f(float);

...

void main(int argc, char **argv) {
  int i;
  int N = ...;
  float *xs = ...;
  float *ys = ...;

  ...

  #pragma omp parallel shared(xs, ys, N) private(i)
  {
    #pragma omp for schedule(dynamic, chunk)
    {
      for (i = 0; i < N; ++i) {
        ys[i] = f(xs[i]);
      }
    }
  }
}
```

With a good compiler, this should perform well on any commodity multi-core CPU: a team of threads will be spawned to tackle different parts of the loop in parallel. However, it is also well suited to execution on a vector processor, such as a *general purpose graphical processing unit* (GPGPU). We could, for example, rewrite the program for Nvidia's CUDA platform (Section 2.4.1):

```
__global__ void kernel(float *xs, float *ys, int N) {
  int thread_id = blockIdx.x * blockDim.x + threadIdx.x;

  if (thread_id < N) {
    // f is inlined into the kernel and uses
    // xs[thread_id].
    float f_x = ...;

    ys[thread_id] = f_x;
  }
}

void main(int argc, char **argv) {
  ...

  kernel<<<..., ...>>>(...);

  ...
}
```

Both these programs require introducing parallelism at quite a low level. In the OpenMP code, we must deal with scheduling the threads that execute the loop. In CUDA, we must rewrite `f` as a *kernel*. In both cases, we have no portability. Ideally, we'd like to forget about these low level details and not write any explicit parallel code at all.

## 1.1 The Idea

In this project we explore parallelism at a much higher level of abstraction. Specifically we investigate how *pure functions* can allow us to concentrate on the data and not potential *side-effects* that affect how it is handled. We look at how the use of pure functions allows us to make guarantees about the code we parallelise, letting us generate efficient code without making compromises with regards to expressiveness. The functional programming language *Haskell* provides an excellent testbed for such an undertaking – in Haskell, all functions are pure, and powerful abstractions can be created using *higher-order functions* – functions that take other functions as arguments.

Higher-order functions have been used to study parallelism before; in [3], *skeletons* are introduced as a functional method of capturing *patterns* of parallel computation such as farming and pipelining. In this report we look at exposing parallelism using the *type system*, specifically by providing data types which are known to be parallelisable. We explore a *stream-based model* of parallel computation, which is simple yet powerful. To illustrate the idea, consider the program below, which is a Haskell version of our earlier C program that uses streams instead of arrays.

```
f :: H Float → H Float
f x = ...

main :: IO ()
main =
  do
    let
      xs = streamFromList ...
      ys = mapS f xs

    ...
```

The only difference between this and a normal Haskell program is the `H` that annotates the type signature. `H` changes functions so that they return *descriptions* of the values they compute rather than the values themselves, i.e., `H` is an *abstract syntax tree* (AST). While this seems like a substantial change to impose on a function or program, we later show that this is all that is needed to parallelise this Haskell program.

## 1.2 Contributions

In the remainder of this report we document our efforts to parallelise programs automatically using domain-specific knowledge embedded in the type system. We make the following contributions:

- We present a Haskell (Section 2.2) framework for runtime code generation (Section 3.3), and apply it to a stream-based model of computation (Section 2.5).
- We use our framework to exploit stream-based parallelism (Section 4.2) on an example platform, specifically Nvidia's CUDA platform for their family of GPGPUs (Section 2.4.1).
- We evaluate our implementation against a serial counterpart, and find that extremely efficient CUDA code can be generated from pure Haskell source.

- We discuss how we can exploit rewriting in GHC, the Glasgow Haskell Compiler (Section 2.3), and see that it can dramatically improve performance in some cases.

- We discuss an alternative method for modifying GHC in order to achieve similar results at compile-time, and compare it with the runtime approach that forms the core of the project.

# Background | 2

In this chapter we present the background for our work, both with respect to Haskell and its compilers and parallel programming in general. We first define automatic parallelisation, and look at prior work that has been conducted within the field. We then detail the programming language Haskell and its flagship compiler, GHC. Finally we explore some of the platforms and models that are available for parallel programming in order to evaluate effectively the criteria for successfully creating high level abstractions thereof.

## 2.1 Automatic Parallelisation

Automatic parallelisation is the act of removing all responsibility for parallelisation from the programmer. Parallelisation of this nature has been studied now for many years [4, 5, 6], but advances in the field have largely come as compromises. Commonly a language's expressibility is the first sacrifice, limiting the knowledge required by the compiler to generate correct parallel code; contrast the progress that has been made with Fortran 77 with that made with C/C++. Recent years have seen more activity within this scope – developments such as *SPIRAL* [7, 8] (for *digital signal processors* (DSPs)) and *pMatlab* [9] [1] (for MATLAB) provide further evidence that *domain-specific languages* (DSLs) are substantially easier to parallelise than their general purpose siblings.

But what does it mean for parallelisation to be "automatic" in this context? Definitions vary in strictness with respect to how much the programmer can do to "help" the compiler, ranging from nothing at all to dense annotations; this project takes a somewhat intermediate standpoint based upon the following criteria:

- Parallelisation is *not* considered automatic if the user consciously writes code to tell the compiler to parallelise part of a program, as in *OpenMP* (see Section 2.6.2) or *CUDA* (see Section 2.4.1).
- The autonomy of parallelisation is *unaffected* if the user chooses to use specific data structures or libraries which he or she knows to have a parallel evaluation semantics. Examples of this approach include *parallel arrays* in *Data Parallel Haskell* (DPH, see Section 2.3.4) and *distributed arrays* in *pMatlab* [9]. The distinction remains that the user is not telling the machine *how* or *where* to parallelise the program, just that parallelism should be exploited.

It is important to note that our definition is not a denouncement of the work that falls outside its scope; far from it – discussions within the scope of this project will likely encompass *generating* this "help" for the compiler as an intermediate stage of program transformation. Such parallelisation would then fall under our definition, since the annotations and hints are no longer decisive additions made by the user.[2]

[1] Which bears resemblances to the approach used in DPH, discussed in Section 2.3.4.

[2] Or rather, the intermediate stage *autonomously* functions as a user of these technologies.

## 2.2 Haskell

Haskell, named after the mathematician and logician Haskell B. Curry, is a *purely functional, lazy* programming language. Its inception in Autumn of 1987 was designed to address the lack of a common functional programming language; the most widely adopted at the time being *Miranda*, a proprietary product. Over the last twenty years it has matured as *Haskell 98* [10], a diverse platform for users, developers and researchers alike with its empowering features and open specification [11]. 2006 marked the beginning of work on Haskell 98's successor, *Haskell'*,[3] but at the time of writing the specification is incomplete and thus has yet to be implemented or adopted. There are currently numerous implementations of Haskell 98 in circulation; GHC is arguably the most widely known and is discussed in Section 2.3, but there are several such as Hugs [12], NHC [13, 14], YHC [15, 16] and JHC [17]. One of Haskell's most powerful features is its support for *monads*, which we provide a brief introduction of here.

### 2.2.1 Monads

Monads are a construction that allow a pure functional language such as Haskell to handle *side-effects* such as failure, state and I/O. As an example of how monads can be used, consider Figure 2.1, where we have two functions `f` and `g` (all examples are adapted from [18]).

```
f :: b → c
f x = ...

g :: a → b
g x = ...
```

Figure 2.1: The functions `f` and `g`.

Like all functions in Haskell, `f` and `g` are *pure* – for any argument, they always return the same result when given that argument. But suppose we wish for `f` and `g` to print out a message (represented by a `String`) when they are called. In Haskell, the only way that a function can affect the environment is through the value that it returns. We therefore need to change the types of `f` and `g`—let us call these newly typed functions `f’` and `g’` respectively—to return *two* values: the first being the result of applying `f` or `g` to the input, and the second being the message. Figure 2.2 gives the types of `f’` and `g’`.

```
f’ :: b → (c, String)
f’ = ...

g’ :: a → (b, String)
g’ = ...
```

Figure 2.2: The functions `f’` and `g’`.

`f’` and `g’` are now message producing variants of `f` and `g`, as required, but in solving this problem, we have created another. While we can compose `f` and `g` to combine their computations, this is not the case with `f’` and `g’` – their types no longer match. We can alleviate this issue by writing code such as that in Figure 2.3.

[3] Pronounced "Haskell Prime."

```
let (y, s) = g' x
    (z, t) = f' y in (z, s ++ t)
```

Figure 2.3: Code for combining f' and g'.

This kind of "glue code" is likely to be common, so we can refactor it into a *higher order function* that allows us to *bind* f' and g' together conveniently. Figure 2.4 gives the definition of the binding function.

```
bind :: (b → (c, String)) → ((b, String) → (c, String))
bind f' (y, s) =
  (z, s ++ t)
  where
    (z, t) = f' y
```

Figure 2.4: The bind function.

With this we can now combine f' and g' by writing bind f' . g'. For convenience, we will combine bind and composition into the infix operator ≫= . Thus we can rewrite bind f' . g' as f'≫= g'. You might now ask if there is an *identity* message producing function for bind—let us name it unit—such that unit behaves in an analogous manner to Haskell's id function (as in Figure 2.5).

```
f . id = id . f = f

unit ≫= f = f ≫= unit = f
```

Figure 2.5: The unit function's required behaviour.

The identity on strings with respect to concatenation is the empty string, so we can define unit as in Figure 2.6.

```
unit :: a → (a, String)
unit x =
  (x, "")
```

Figure 2.6: The unit function.

We have in fact just defined Haskell's `Writer` monad (apart from the fact that messages can be of a more general type).[4] This is the purpose of monads: they allow programmers to compose sequences of actions which exhibit side-effects into larger actions. As mentioned earlier, side-effects encompass a wide range of features such as state and I/O. Consequently monads are defined as a type class, as shown in Figure 2.7.

```
class Monad m where
  (≫=)    :: m a → (a → m b) → m b
  (≫)     :: m a → m a → m b
  return  :: a → m a
  ...
```

Figure 2.7: The `Monad` type class.

If we define:

```
type MessageProducing a = (a, String)
```

and substitute `MessageProducing` into `m`, we see that `return` is the same as the `unit` function we defined earlier: it *returns* a value that has been lifted into the monad. ≫= is exactly the same (albeit its generalised type signature) as its earlier definition, and ≫ is like ≫= except that it discards the result produced by its first argument. ≫'s definition is not strictly necessary since it can be defined in terms of ≫= .

We can illustrate the type class using another example. Consider the type `Maybe`, presented in Figure 2.8.

```
data Maybe a
  = Just a
  | Nothing
```

Figure 2.8: The `Maybe` type.

`Maybe` represents the possibility of failure, using the `Nothing` constructor. Its monadic instance declaration is given in Figure 2.9.

```
instance Monad Maybe where
  return         = Just
  Just x  ≫= f = f x
  Nothing ≫= f = Nothing
```

Figure 2.9: `Maybe`'s monadic instance declaration.

We see that lifting a value `x` into the `Maybe` monad is the same as writing `Just x`, and that ≫= 's job is to propagate failure by moving `Nothing`s through a chain of computations that may fail. This ability to sequentially compose actions is what makes monads so useful in a language such as Haskell.

[4] Defined in `Control.Monad.Writer`.

## 2.3 The Glasgow Haskell Compiler

Commonly referred to as *The Glasgow Haskell Compiler* or simply *GHC*, the Glorious Glasgow Haskell Compilation System is an open source optimising compiler for Haskell that is itself written mostly in Haskell.[5] GHC is compliant with the Haskell 98 standard and provides a large number of optional extensions. It is an implementation of the *Spineless Tagless G-Machine*, an abstract machine that allows for the interpretation of functional programming languages on commodity hardware [19, 11].

The GHC project started soon after the specification for Haskell 1.0 had been finalised in January 1989 when Kevin Hammond at the University of Glasgow began working on a prototype compiler in Lazy-ML. Whilst this first version was resource hungry and exposed limitations and drawbacks in both the use of Lazy-ML and Haskell's design, it had almost completely implemented the Haskell 1.0 specification by June of the same year. In the Autumn, a team consisting of Cordelia Hall, Will Partain and Simon Peyton-Jones completely redesigned GHC with the idea that it would be written in Haskell – and thus *bootstrapped* by the prototype compiler [11]. It is this incarnation that has evolved into the current version of GHC.

### 2.3.1 Compiler Architecture

GHC's compilation process is structured as a pipeline in which a Haskell program is successively refined before delivery of a *binary* (or whichever output the user has requested); this is illustrated in Figure 2.10. At a first approximation, a program is transformed through four intermediate representations:

#### HsSyn

`HsSyn` is a *collection of data types* that describe the *full abstract syntax* of Haskell. It is capable of representing all of the "sugar"[6] that Haskell offers and is therefore the largest and most complex of GHC's intermediate representations. All error checking procedures operate on the `HsSyn` type (with a few exceptions, discussed later), giving GHC the ability to produce error messages that relate to what the user wrote, not some stripped down internal manifestation [20].

`HsSyn` is parameterised over the type of terms which it contains, which is also indicative of which phase of the compilation it is in:

- The *parser* generates terms parameterised by the `RdrName` type. To a degree a `RdrName` is what the programmer originally wrote. It can be thought of as containing either one or two strings – these contents represent names and qualified names respectively.

- The *renamer* generates terms parameterised by the `Name` type, which can be conceptualised as a `RdrName` paired with a unique value. This prevents scoping conflicts and allows GHC to check that all names referencing a single binding occurrence share the same unique value.

- The *typechecker* generates terms parameterised by the `Id` type, which is yet another pairing; this time of a `Name` and a type.

These three parts of the pipeline comprise what is known as the *front end* of GHC [20].

[5] Strictly speaking the code is mostly *Literate Haskell*, a variant of the language where human readability is the primary goal.

[6] "(Syntactic) Sugar" is a collective term for features in a language that do not affect its functionality but merely make it easier or more natural for a programmer to write code.

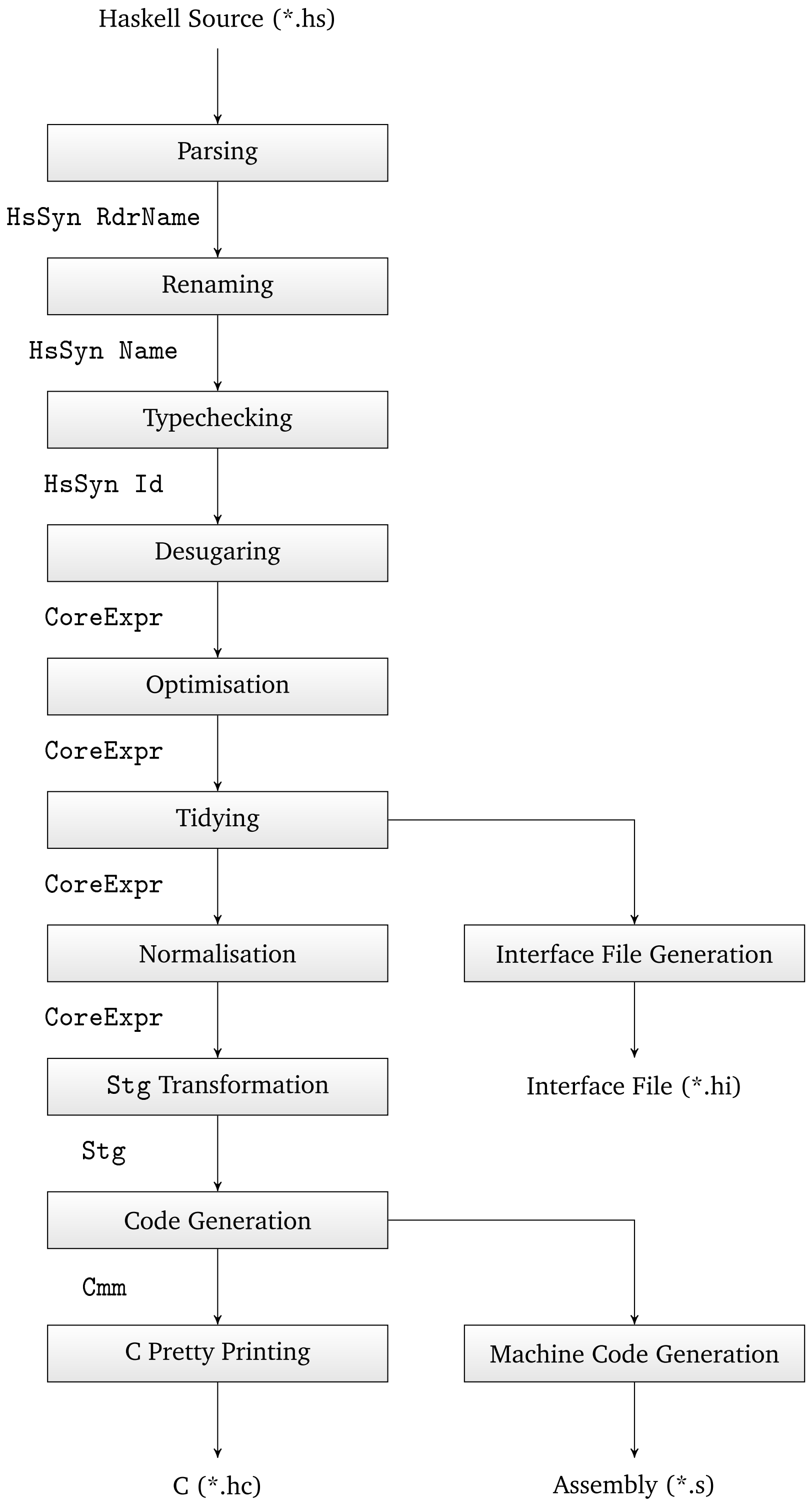

Figure 2.10: The GHC Compilation Pipeline, adapted from [20].

### Core

Core is GHC's implementation of *System $F_C$*, an extension of *System F*[7] which is expressive enough to support features such as newtypes and *generalised abstract data types* (GADTs) [21, 20]. It pervades the next five phases of the compilation:

- The *desugarer* transforms HsSyn into Core. Core is a dramatically smaller data type than HsSyn and, as such, a lot of information deemed unnecessary is discarded. Errors can be produced at this point, but these often pertain to more generic problems than those noted in the front end (pattern matching overlaps, for example).

- The *optimiser* is composed of several passes—known collectively as the *SimplCore* pass—which apply a series of Core-to-Core transformations; currently these include the following [20]:

  1. *Simplification* and *inlining* [22, 23]
  2. *Float-out* and *float-in* transformations [24]
  3. *Strictness analysis*
  4. *Liberate-case* transformations
  5. *Constructor-specialisation*
  6. *Common sub-expression elimination*

- The *tidying* phase (denoted *CoreTidy* internally) cleans up the generated Core so that it is in a form suitable for processing later in the pipeline.

At this point two independent operations are applied:

- The Core is transformed and output into an *interface file*. An interface file contains information accumulated by the compiler that allows it to make additional safety checks and assertions when using and linking the corresponding code. In an unoptimising compilation it is little more than a collection of type signatures, but when the optimiser is active a lot more data about the applied transformations is logged, resulting in a larger file.

- A final Core-to-Core pass named *CorePrep* translates the Core into *Administrative Normal Form* (ANF), a *canonical form* of the program in which all arguments are either variables or literals.

The last part, *CorePrep*, is the first part of what is known as the *back end* of GHC, comprising Stg and Cmm.

### Stg

Stg (an abbreviation for *Spineless Tagless G-Machine*, the abstract machine on which GHC is built [19, 11]) is the last form a program takes before code generation occurs. It is produced by the *CoreToStg* translation, although this transformation performs little work due to the similarities between Core in ANF and Stg.

The differences between Stg and its predecessor are mainly a result of further normalisation and expansion – constructor and primitive operator applications are saturated, lambda forms can only appear on the right hand side of a let binding and type information is largely discarded [20]. Stg retains some decorations [20] that are not present in the Core, particularly:

[7] Also known as the *polymorphic lambda calculus* or the *second-order lambda calculus*.

```
forkIO :: IO () → IO ThreadId
forkOS :: IO () → IO ThreadId
```

Figure 2.11: Functions for forking threads in Concurrent Haskell.

- All lambda forms list their *free variables*. These are the variables in the body of the expression that are bound by the `let`.
- Case expressions list their *live variables*, that is, the variables that will be reachable from their continuation.

*Static Reference Tables* (SRTs) are also added to the program's lambda forms. SRTs contain references that enable the garbage collector to locate *constant applicative forms* (CAFs)[8] relating to the object.

**Cmm**

`Cmm` is produced from `Stg` by GHC's code generator, and is an implementation of *C−−* [25].[9] The decision to use a high level target language like `Cmm` gives GHC the ability to work with a variety of back ends, and indeed it currently supports native (assembly) and C code generation in this manner [20]. This part of the compilation process is where code generation actually occurs.

### 2.3.2 Concurrent Haskell

Concurrent Haskell [26] refers to GHC's `Control.Concurrent` library, which provides mechanisms for writing threaded (and therefore non-deterministic) programs in Haskell. It offers two functions for forking threads, shown in Figure 2.11.

- `forkIO` spawns a thread that is scheduled by the GHC runtime system, known as a *lightweight* thread.[10] Such threads are usually one or two times more efficient (in terms of both time and space) than *native* threads (those managed by the operating system) [27].
- `forkOS` creates a native thread, referred to as a *bound* thread. Bound threads may be cross foreign interfaces where other parts of the program expect operating system threads, and may carry thread-local state.

In order to alleviate the potential for unnecessary blocking, GHC schedules lightweight threads onto underlying operating system threads.[11] A lightweight thread may be scheduled onto many different native threads in its lifetime [27].

Synchronisation can be accomplished using *mutable shared variables*[12] and locks, or through GHC's support for *Software Transactional Memory* (STM). Software Transactional Memory is an *optimistic* alternative to lock-based synchronisation that views all operations on memory as *transactions*. Transactions execute in the context of a thread, and are *atomic*; between a transaction's initialization and completion there is no way for another thread to observe an intermediate state [28]. Concurrent Haskell's interface to STM is the `Control.Concurrent.STM` library, and is particularly comprehensive, supporting *composable transactions* and sophisticated failure handling operations [29, 27].

[8] A CAF is an expression that is only evaluated once within the execution of a program.

[9] Pronounced "C Minus Minus."

[10] Also known as a *green* thread.

[11] When the Haskell program in question is linked against the multi-threaded version of the GHC runtime (through use of the `-threaded` option when building).

[12] `MVar`s in the `Control.Concurrent` library.

```
infixr 0 ‘par‘
infixr 1 ‘seq‘

par :: a → b → b
seq :: a → b → b
```

Figure 2.12: Annotation functions provided by the `Control.Parallel` library.

```
import Control.Parallel

nFib :: Int → Int
nFib n  | n <= 1    = 1
        | otherwise = par x (seq y (x + y + 1))
        where
          x = nFib (n - 1)
          y = nFib (n - 2)
```

Figure 2.13: Parallelising the computation of the $n^{\text{th}}$ Fibonacci number with annotation functions.

### 2.3.3 Parallel Haskell

Parallel Haskell encompasses GHC's support for running Haskell on multiple processors. Primarily this is achieved either through the direct mapping of threads (provided by Concurrent Haskell) onto processors or by the use of so called *annotation functions* [27], as in Figure 2.12.

- The expression `par x y` hints that `x` could be executed in parallel with `y`. Specifically, the call to `par` *sparks* the evaluation of `x` to *weak head normal form* (WHNF). It is important to realise that sparks are *not* threads; sparks are queued in *first in, first out* (FIFO) order and then progressively *converted* into real threads whenever the runtime detects a free (idle) core.

- A call `seq x y` ensures that `x` is evaluated *before* `y` and returns `y` as its result. `seq` is strict with respect to both of its arguments, which means that the compiler may occasionally rewrite its applications unfavourably. For these scenarios there is an alternative, `pseq`, that is strict only in its first argument.[13]

We can illustrate the use of `par` and `seq` using a simple example (adapted from [27]) that parallelises the retrieval of the $n^{\text{th}}$ Fibonacci number; this is shown in Figure 2.13.

In this program we see that, for cases other than the zeroth and first Fibonacci numbers, a spark is created to evaluate `nFib (n - 1)`. `seq`'s importance is now apparent: for the parallel evaluation to have any benefit we must force the parent thread to evaluate `nFib (n - 2)` before returning the inclusive sum.

In their basic forms `par` and `seq` are examples of *control parallelism* – the evaluation of arbitrary subexpressions in parallel. More elaborate *parallelisation strategies* have been built around these functions[14] that express higher level constructs such as *data parallelism* (Section 2.3.4), *divide and conquer* and *pipelining* [30, 27]. Such abstractions demonstrate the power of annotation functions but do not remove the need for the programmer to understand their purpose altogether.

[13] Both `par` and `pseq` are defined in the `Control.Parallel` library.

[14] Available in the `Control.Parallel.Strategies` library.

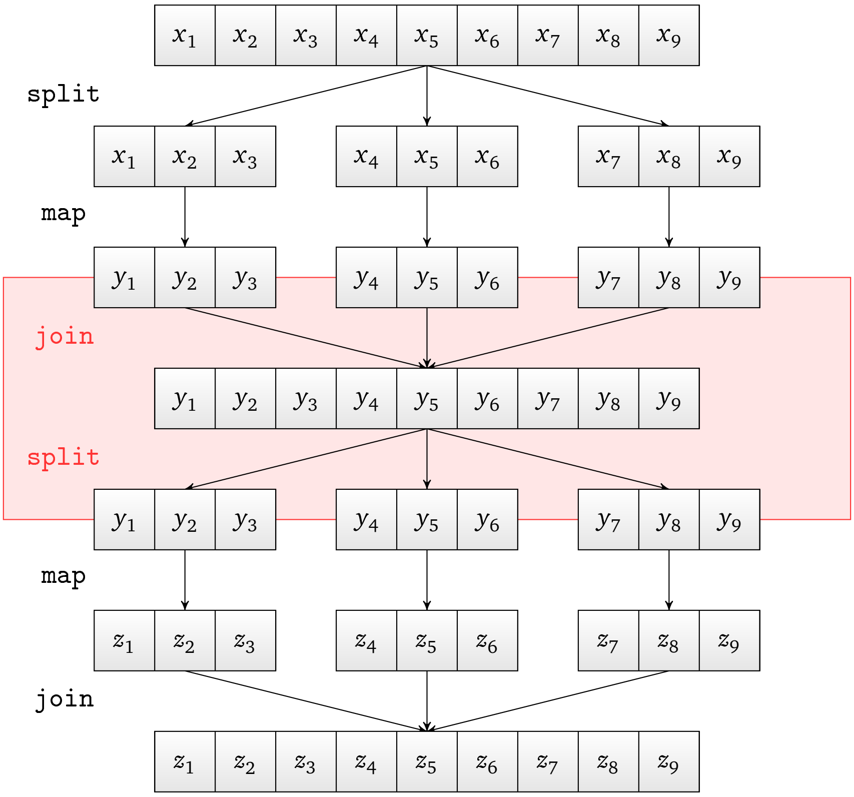
split
map
join
split
map
join
$x_1$ $x_2$ $x_3$ $x_4$ $x_5$ $x_6$ $x_7$ $x_8$ $x_9$
$y_1$ $y_2$ $y_3$ $y_4$ $y_5$ $y_6$ $y_7$ $y_8$ $y_9$
$z_1$ $z_2$ $z_3$ $z_4$ $z_5$ $z_6$ $z_7$ $z_8$ $z_9$

Figure 2.14: Removing unnecessary synchronisation points between data parallel operations.

### 2.3.4 Data Parallel Haskell

*Data Parallelism* (also *flat data parallelism*) is the act of performing the *same task on different pieces of data* in parallel. Focus is placed on the distribution of data as opposed to the processing, as in control parallelism. *Nested Data Parallelism* (NDP) describes what happens when flat data parallelism occurs at many levels, over an array of arrays, for example.

Data Parallel Haskell (DPH) is a relatively recent development in the GHC codebase working towards an implementation of nested data parallelism in Haskell [31]. The key abstraction is the *parallel array*, a *strictly evaluated* data structure. The strictness property means that in evaluating one element of a parallel array, the remaining elements will be demanded also and therefore might be evaluated in parallel. Syntactically they bear a resemblance to lists, but there are several differences:

- Parallel arrays are denoted with `[:` and `:]`. Analogous to lists, `[:a:]` represents the parallel array with elements of type `a`.

- Pattern matching is limited – the pattern `[:x, y, z:]` matches a parallel array with *exactly* three elements which can be matched to `x`, `y` and `z` respectively. Parallel arrays are *not inductively defined*; there is no notion of “cons” (:) as there is with lists.

Such semantics mean that parallel arrays have excellent optimisation potential [32], indeed DPH offers several possibilities for performance gain:

- *Flattening* (or *vectorisation*) transforms all operations on nested data structures into their equivalents on flat data structures, which allows for efficient implementation on commodity hardware.

- *Redundant synchronisation points* between computation are exposed and can be removed without affecting the overall result (see Figure 2.14). These can be implemented using *rewrite rules* at the Haskell source level (supported by GHC).

- *Array contraction* reduces both space and time overhead, and is implemented using a technique discussed in [32] which the authors call *stream fusion*.

All these features leave DPH in a favourable light, but unfortunately much of that described in [31] and [32] is still work in progress. This leaves a current version which is an actively developed compromise between elegance and efficiency, with major changes happening without warning.

## 2.4 Parallel Architectures

In this section some existing parallel architectures and their respective programming models are discussed. Later on, a brief evaluation of their benefits and shortcomings with respect to this project is presented.

### 2.4.1 CUDA

CUDA (*Compute Unified Device Architecture*) is a generalised parallel computing platform developed by Nvidia atop of their range of *Graphical Processing Units* (GPUs). It is a massively multi-threaded architecture (described by Nvidia as *single instruction, multiple thread* (SIMT)) which uses busy thread scheduling to hide memory latency [33]. The motivation for exposing the capabilities of GPUs to general purpose computing is apparent - a GPU dedicates many more transistors to data processing—potentially floating point ALUs—than a traditional CPU, allowing many programs utilising CUDA to run an order of magnitude faster than their CPU-only counterparts [34, 35, 36].

#### Concepts and Nomenclature

Applications harness CUDA by instructing the CPU (the *host*) to delegate the processing of various *kernels* to the GPU (the *device*). Each kernel is run by $N$ threads in parallel on the device. Threads are grouped into *blocks*, with multiple blocks forming a *grid* upon which the kernel is executed. Both block and grid dimensions can be specified on a per-kernel basis, described in the next section.

At a physical level, a CUDA-capable GPU consists of copious multi-processors (see Figure 2.15). Each multi-processor is equipped with several *scalar processors* (SPs) (the exact number is platform dependent), two *special functional units* (SFUs) (used for transcendentals) and four types of on-board memory [33]:

- One set of 32 *registers* per SP.

- A *shared memory* that can be accessed by all the SPs.

- A *read-only constant cache*—again accessible from all SPs—that speeds up reads from the constant space held in device memory.

- A *read-only texture cache* that provides the same functionally as the constant cache, only for the texture space in device memory. Access to the texture cache is via a *texture unit* that implements the necessary addressing modes and filtering operations to work correctly with texture memory.

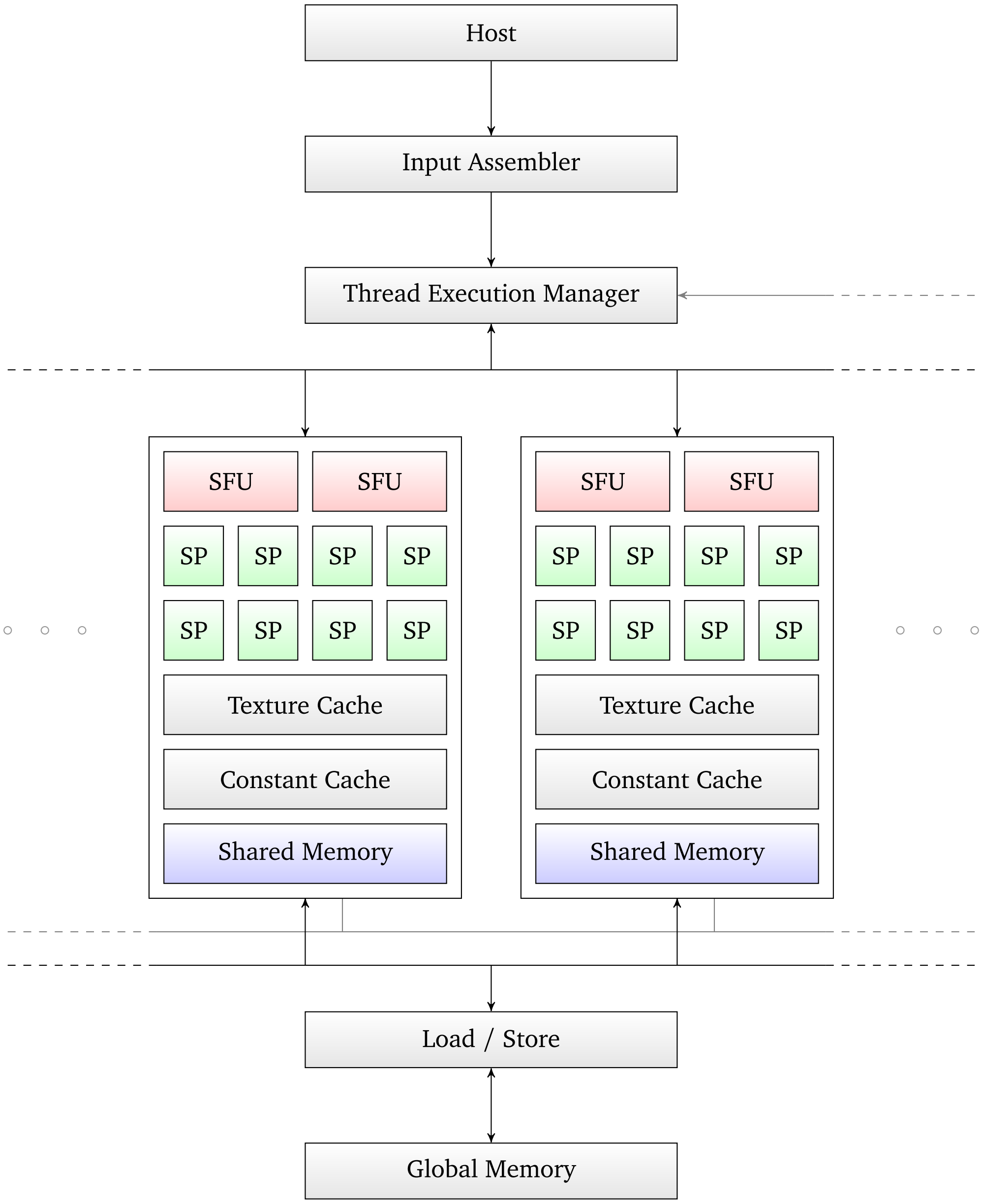

Figure 2.15: A CUDA-capable GPU's architecture, as described in Section 3.1 of [33]. Repeated multiprocessors (indicated by ellipses) have been omitted for clarity.

```
for (i = 0; i < N; i++) {
  xs[i] = xs[i] * xs[i];
}
```

Figure 2.16: Squaring an array with serial C code.

At runtime, the threads are scheduled in groups of 32, known as *warps*,[‡] with each thread being mapped onto one SP.[15] Only one kernel may be present on the device at a time, but the CPU is free to perform other operations once the device has been allocated work.

### Programming Interface

CUDA kernels and their host programs are written in a *subset* of C with a small base of extensions; a proprietary *software development kit* (SDK) available from Nvidia[16] provides the necessary toolchain and libraries. The most notable differences with ANSI C are the prohibitions of *recursive functions* and *function pointers*, which would cause problems for CUDA's evaluation model.[17] One might imagine that the use of C means a significantly reduced learning curve for developers than that imposed by say, the use of a DSL, and to an extent this is the case. However, CUDA is not without its idiosyncrasies, the understanding of which will often enable the creation of more elegant, more performant solutions [33, 37]:

- The *mindset* associated with CUDA programming is very different to that one must adapt when writing conventional serial code. Threads are much more lightweight entities than their CPU-world counterparts, and can often be used at a much finer granularity than one might expect; one per pixel, for example, is not uncommon.

- Developers should ensure that they use *the whole device*, spawning *at least as many blocks as multiprocessors* and multiples of 32 threads per block in order to utilise the card as best as possible.

- *Memory access* comes at a high price, but if each thread in a warp uses a different address then the accesses will be *coalesced*. Equally important is the consideration of whether a thread needs to load data at all, since other threads may load data which exhibits *spatial locality*.

- *Thread synchronisation* is limited to threads within a block (via a call to `__syncthreads()`) – there is no "across the card" barrier facility.

The first point is best illustrated by an example. Figure 2.16 contains a trivial serial for-loop that squares an array of single-precision floating point numbers, `xs`, which we assume has already been initialised. In contrast, Figure 2.17 constructs the equivalent CUDA kernel, which is a C function annotated with the `__global__` keyword. Since the resulting compiled code will be designed to run on the device, any data it processes must also be on the card. Consequently CUDA programs typically entail a certain amount of "boilerplate" code involving memory copying and initialisation; this code has been omitted to preserve clarity.

We observe a couple of key differences, which also serve to emphasize some of the other points made above:

[‡] Hence the title of this project.

[15] The term *half-warp* is also used as one might expect, referring to either the first or second half of a warp.

[16] `http://www.nvidia.com/object/cuda_get.html`

[17] That is to say, each of hundreds of threads stepping through the *same instruction* at any one point.

```
__global__ void squareArray(float *xs, int N) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  if (i < N) {
    xs[i] = xs[i] * xs[i];
  }
}
```

Figure 2.17: Squaring an array in parallel using CUDA.

```
dim3 dimBlock(256);
dim3 dimGrid(N / dimBlock.x + 1);

squareArray<<<dimGrid, dimBlock>>>(xsDevice, N);
```

Figure 2.18: Calling the `squareArray` kernel from C.

1. There is no for-loop: rather than squaring *N* items sequentially there are now *N* threads each squaring one item in parallel. This is a pattern commonly seen in CUDA that can be additionally extended to nested loops using the various levels of block and thread indexing available.

2. Indices are calculated from the thread's location in the grid; since each thread has a *unique* location the memory addresses requested by each warp will not collide and the accesses will be coalesced.

As mentioned earlier the grid and block sizes can be chosen on a per-kernel basis. Specifically they are determined at the kernel's *point of call* through CUDA's `<<<...>>>` syntax. Figure 2.18 depicts an example in which the `squareArray` kernel is called with a block size of 256 and a grid size that ensures there is at least one thread per array element. The ability to use runtime variables in deciding grid and block size contributes to the portability that CUDA enjoys somewhat more than other parallel platforms, but peak performance will always depend on the exact generation and model of the Nvidia GPU the code is running on.

### 2.4.2 The Cell Broadband Engine Architecture

The Cell Broadband Engine Architecture (usually referred to as just *Cell* or *Cell BE*) is based on a multi-core chip consisting of a *Power Processing Element* (PPE) and eight *Synergistic Processing Elements* (SPEs), linked together by an *Element Interconnect Bus* (EIB) [38]. The bus is composed of four uni-directional channels that connect the processing cores and the other important on-board components, namely the *Memory Interface Controller* (MIC) and two I/O interfaces. This makes for a configuration of twelve interlinked units as shown in Figure 2.19.

**The Power Processing Element**

The PPE is a 64-bit two-way multi-threaded core based on IBM's Power Architecture. It supports a *reduced instruction set computer* (RISC) architecture and acts as the controller for the other elements, as such it has supplemental instructions for starting, stopping and scheduling tasks on the SPEs [38]. Unlike the SPEs it can also read from and write to main memory, as well as the local memories of the SPEs themselves. Consequently it is the PPE that runs the operating system, and typically the top-level thread of an application [39].

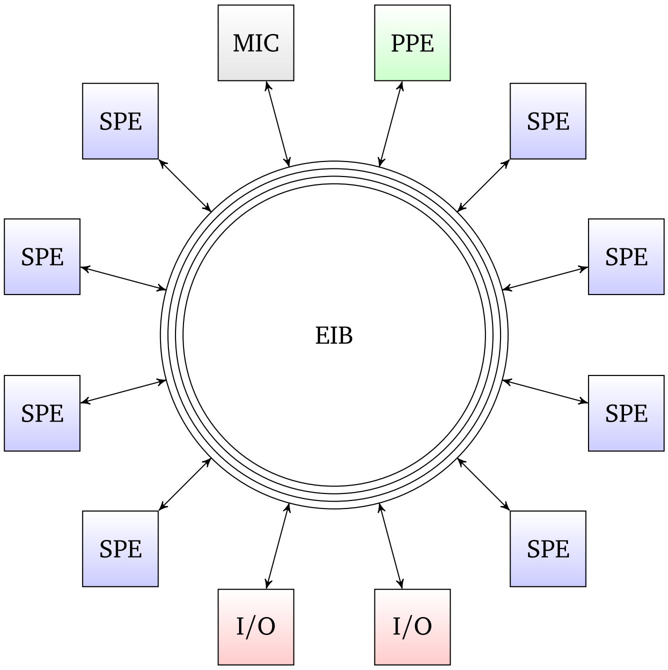

Figure 2.19: The Cell Processor.

### The Synergistic Processing Elements

Each SPE is also a RISC processor, specialised for floating point calculations. As mentioned above only the PPE can communicate with main memory; to this end all of the SPEs are equipped with an asynchronous *direct memory access* (DMA) controller. Both the PPE and the SPEs are *single instruction, multiple data* (SIMD) processors,[18] with the SPEs handling 128-bit instructions for both single and double precision operations [40].

### Programming Interface

Applications for the Cell are written in C with a set of hardware-specific extensions:

- *SIMD computation* is facilitated by intrinsics that support vectorised data types, as well as functions that wrap the underlying VMX instruction set that operates on them.

- *Threading and SPE control* is achieved using library calls,[19] which form a set of APIs for working with the synergistic units.

Separate applications must be written for the PPE and the SPEs – the SPEs may all run identical kernels but this does not always yield optimal results as discussed below. Strictly speaking only the PPE needs to be programmed, but if one wants to realise the Cell's potential then the SPEs must be used effectively, which in this context means considering factors such as the following:

- *SPE assignment* – Dividing a problem into a number of independent subproblems and solving those simultaneously is often a natural way to parallelise the task at hand, but the Cell's ring structure offers the potential for

[18] Although they do not share the same SIMD capabilities; a fact that contributes to their heterogeneity.

[19] Provided by `libspe2` at the time of writing.

*pipelining* between SPEs running different kernels, with varying performance returns depending on the application.

- *Memory management* – The forced use of DMA on the SPEs means that efficient memory usage is critical to performance. The local memories of the SPEs can be used as caches but they do not function as such automatically in that their use is *not* transparent to the programmer [39].

The Cell has seen adoption in the image processing and digital media markets, with Sony's Playstation 3 being perhaps the most widespread example of its use. But with a challenging programming model that emphasizes peak computational throughput over code simplicity [41], it has yet to gain a strong position in general purpose computing.

## 2.5 Stream Programming

Stream programming is a paradigm not dissimilar to SIMD that is concerned with the application of concurrent kernels to a sequence of data – a *stream*. It was originally developed for media and imaging [42] (see Section 2.4.1 for a discussion of its use in CUDA), but is applicable to a wide range of problems and fields. The stream programming model has been developed over many years [43], but they remain an active area of research; we examine a few of the contributions.

### 2.5.1 StreaMIT

Developed at MIT, StreaMIT is a language and compiler toolchain that enables the development of stream-based applications at a high level of abstraction [44]. StreaMIT was implemented initially as an extension to the Kopi Java compiler, and has evolved into a successful platform on which much research has been conducted [45, 46].

Programming in StreaMIT focuses around the construction of a *stream graph*, which is optimised by the compiler for the target architecture. A stream graph is composed of a number of *blocks* which define operations upon the streams passing through them. The smallest block is the *filter*, whose body consists of code that operates on a *single input* to produce a *single output*. More complex kernels can be built up using *composite blocks* [47]:

- *Pipelines* connect the inputs and outputs of a number of child streams.

- *Split-joins* allow computation to take place in parallel, possibly on different parts of the input stream.

- *Feedback loops* have a *body stream*, the output of which is split into two streams. The first leaves the loop, whilst the second is joined with the next input and passed back into the body stream.

Figure 2.20 gives examples of each of these composite blocks in operation. All blocks work on the same basis of single input, single output, and so may be recursively composed to form applications of arbitrary complexity. Each block is also obliged to declare its *data rate*, that is how much input and output it consumes and produces per use; such rates may be static or dynamic [48, 47].

Syntactically StreaMIT bears some resemblances to Java and other imperative languages; semantically it is a rich system that also supports messaging and *parameterised stream types*.[20]

[20] Analogous to C++'s templates; see [48] for more information.

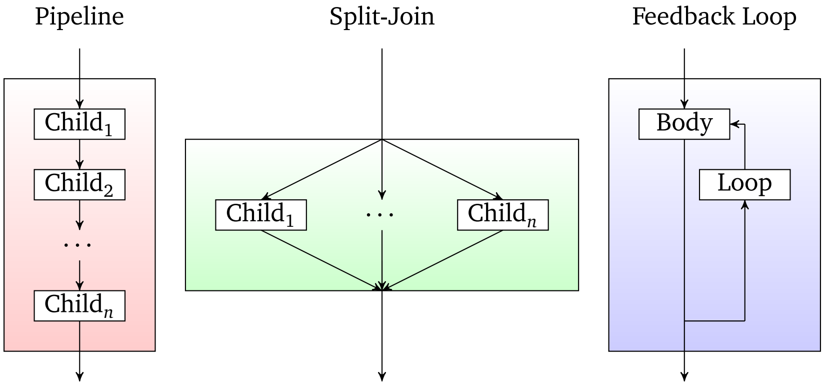

Figure 2.20: Composite blocks in StreaMIT, adapted from [48].

### 2.5.2 Streamware

Streamware is an effort to move away from dedicated stream processors and implement the stream programming model on general purpose multi-core CPUs. Specifically it exposes a runtime system that can be targeted by higher-level compilers [42]; essentially a *stream virtual machine* (SVM). From the compiler's perspective, Streamware provides a set of *local memories* (LMs), used for storing stream data, and *kernel processors*, for executing kernels.

What distinguishes Streamware from previous SVMs is that the runtime accepts a model of the physical machine as input. Compilation then generates a *parameterised model*, into which machine dependent values are substituted upon execution. There are three such parameters of importance in Streamware:

- `NUM_DMAS` represents the number of LMs available in the underlying machine.

- `NPROC[ndma]` is the number of kernel processors sharing the LM with identifier `ndma`.

- `LM_SIZE[ndma]` defines the capacity (in bytes) of the LM with identifier `ndma`.

From these values the runtime's control thread can calculate derived parameters, such as those relating to *strip size* when optimising bulk memory operations. In many cases these performance gains justify the overheads of having an intermediate runtime system, with some applications approaching linear scaling factors [42].

Unfortunately the youth of the research means that little has been constructed atop its foundations, though it offers a promising insight into general purpose stream-based computing.

### 2.5.3 Streams and OpenMP

In [49], Gaster presents an implementation of streams that builds upon OpenMP (Section 2.6.2). It consists of a C++ API and a "modest set of extensions," and aims to address OpenMP's lack of consideration for non-shared memories, exploiting accelerator cores such as those found in GPUs (Section 2.4.1) and heterogeneous processors like the Cell (Section 2.4.2).

The C++ API is not much more than a declarative wrapper around the language's arrays, and can therefore be used in programs that don't utilise OpenMP.

```
double sumSquares(double xsArray[], int size) {
  double sum = 0.0;
  int i;

  Stream *xs = Stream.create(
    xsArray,        // The stream’s backing array.
    size,           // The stream’s size.
    CHUNK_SIZE,     // The chunk size.
    LINEAR_FORWARD // The access pattern to use.
  );

  #pragma omp parallel reduction(+:sum) connect(xs)
  {
    while (!xs->endOfStream(xs)) {
      double x = xs->getNextElement();

      if (xs->streamActive()) {
        sum += x;
      }
    }
  }

  xs->destroy();

  return sum;
}
```

Figure 2.21: Using the parameterised `Stream<T>` type and the OpenMP `connect` clause in Gaster’s model of streams. Adapted from [49].

The real power of the application comes from the addition of the `connect` clause to OpenMP's `parallel` construct. Conceptually, `connect` *joins* a stream defined outside a `parallel` region to the gang of threads executing the contained block. On the block's termination, data is flushed to the stream defined in the outer scope. An example of this behaviour is shown in Figure 2.21.

Gaster also considers formalisation: an *operational semantics* is defined for the provided API that reasons about the evaluation of programs utilising streams.

## 2.6 Other Programming Models

Streams are a very simple and powerful model for parallel programming. They are by no means the only model, however. Here we look at some other models which are used for writing parallel applications.

### 2.6.1 MPI

MPI (*Message-Passing Interface*) is a *message-passing library interface specification* [2] that allows many computers to communicate with one another. It is maintained and developed by the *MPI Forum*, a broad group of vendors, developers and specialists, and is currently available in two versions:

- *MPI 1.2* (*MPI-1* in common usage, where the minor number is omitted) is the most current revision of MPI-1, and modifies MPI 1.0 only through the addition of clarifications and corrections.

- *MPI 2.1—MPI-2* as indicated above—adds completely new functionality to its predecessor, such as support for parallel I/O and bindings for Fortran 90 and C/C++.

MPI provides a plethora of capabilities, all of which are underpinned by a set of core concepts [2].

#### Processes

Processes are the building blocks of an MPI application. They execute autonomously in a *multiple instruction, multiple data* (MIMD) style, communicating using the MPI primitives discussed later in this section. Consequently there is no requirement for any pair of processes to share the same program code (or even architecture if they are running on different platforms).

#### Point-to-Point Communication

Two processes may communicate directly with each through use of *send* and *receive* primitives, exposed as `MPI_Send` and `MPI_Recv` respectively. Such sending and receiving forms the basis of MPI's communication mechanism, with support for both *blocking* (*synchronous*) and *non-blocking* (*asynchronous*) sending available. A few esoteric mechanisms, such as the *ready send*, where a send attempt will fail if a matching receive has not been posted prior, are also specified.

#### Collective Communication

Collective communication is defined as "communication that involves a group or group of processes," [2] and forms a large portion of MPI. Numerous methods for such communication exist in MPI, some of which are shown in Figure 2.22.

Collective operations do not operate on groups of processes, rather *communicators*, a layer of abstraction inserted by MPI that pairs groups with the notion of

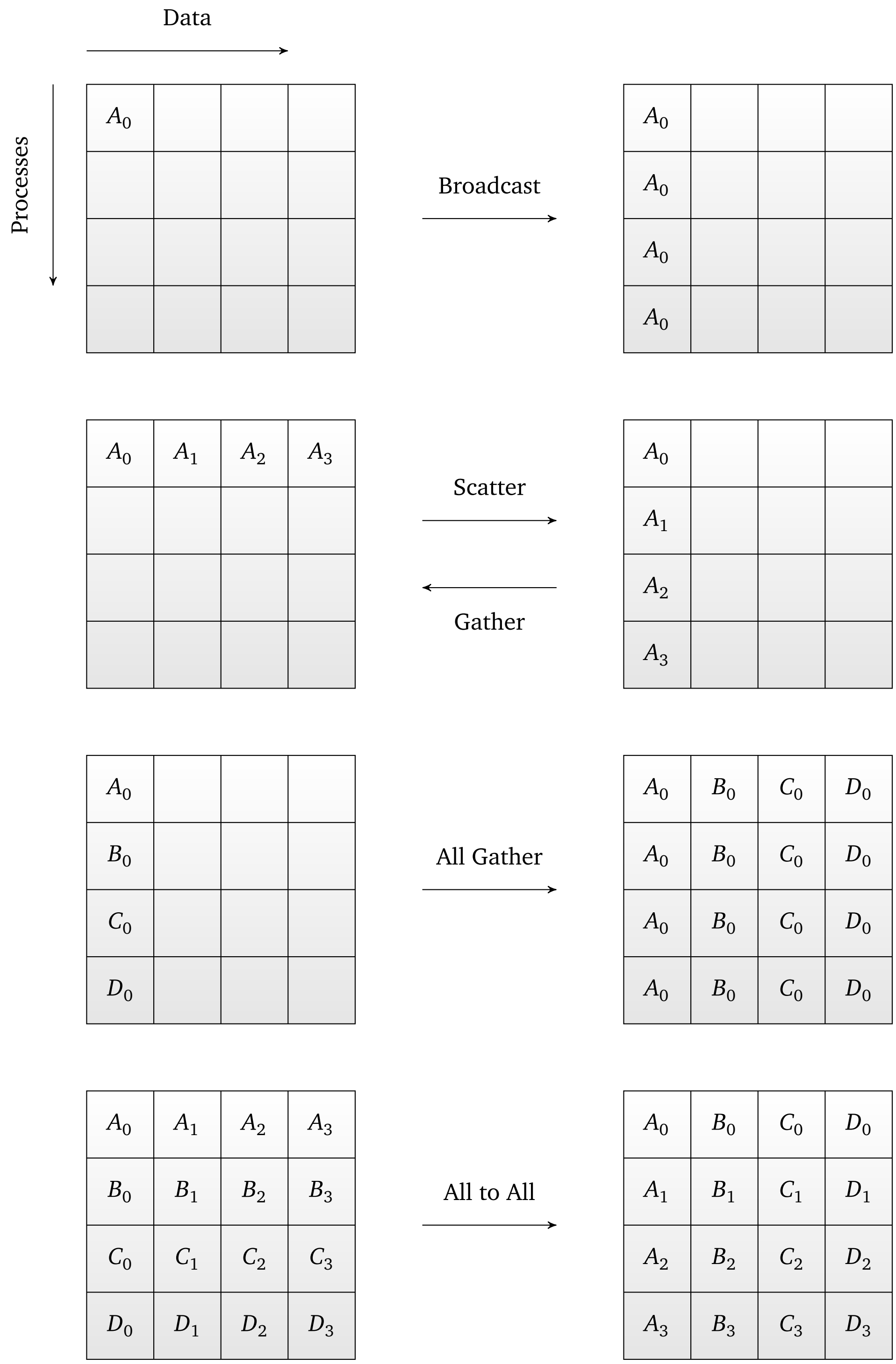

Figure 2.22: MPI collective move operations for a set of four processes, adapted from [2]. Each row represents data locations within the same process.

*context*. Contexts partition the communication space such that messages received in one context cannot be received in another; to this end there are two types of communicator:

- *Intra-communicators* are concerned with processes in the same group. Operations such as those depicted in Figure 2.22 fall within the domain of intra-communication.
- *Inter-communicators* allow the exchange of data between different groups, an action which is initiated by a point-to-point communication occurring between two processes in separate groups.

Initially there is just one communicator, known as `MPI_COMM_WORLD`, from which other communicators can be constructed through progressive "slicing" of the communication space. The decision to prohibit creating communicators from scratch provides additional guarantees of safety, though the specification does recognise it as a "chicken and egg" scenario [2].

### Implementations

The initial implementation of MPI was *MPICH* [50], a project which has demonstrated the application and scalability of MPI-1 on supercomputers and cluster platforms [51]. *MPICH2* has since superseded it, bringing a new implementation and support for MPI-2.[21]

Bindings are also available for a variety of other languages, including Python (*mpi4py*, *PyMPI* and *MYMPI* to name but a few), and OCaml (the `OCamlMPI` module). Success with Java has been limited due to the language's lack of explicit pointers and memory management.

## 2.6.2 OpenMP

*Open Multi-Processing* (OpenMP) is an API presented as a set of *compiler directives* that allows *multi-platform, shared memory multiprogramming* in C, C++ and Fortran. In each case the base language remains unchanged, leaving the implementation design and detail to the compiler writer [1]. Designed to be platform independent, OpenMP's structure can be partitioned into five distinct parts [52]:

### Thread Creation

The key to thread creation in OpenMP is the `parallel` construct. When a thread reaches an instance of `parallel`, a *team* of threads is spawned to execute the region contained within it. Consider Figure 2.23, a (contrived) example that uses multiple threads to print a string in C/C++.

The thread that encounters the `parallel` construct becomes the *master* of the team that is created as a result. Since `parallel` regions can be nested, a thread may master one group and merely participate in another.[22] There is an implied barrier at the end of a `parallel` instance's execution, at which point the team is disbanded and only the master thread continues processing [1, 52].

### Work Sharing

OpenMP's work sharing directives must appear within a `parallel` region, and control how work is distributed amongst a team of threads. There are currently three constructs that fall into this category (excluding the Fortran-specific `workshare`

[21] `http://www.mcs.anl.gov/research/projects/mpich2/index.php`

[22] Though this depends on the compiler's support for nested `parallel` constructs.

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char **argv) {
  #pragma omp parallel
  {
    printf("Hello␣world!\n");
  }

  return 0;
}
```

Figure 2.23: Thread creation using the C language bindings for OpenMP.

```
#pragma omp sections
{
  #pragma omp section
    ...

  #pragma omp section
    ...
}
```

Figure 2.24: Use of the OpenMP `sections` construct in the C language bindings.

construct); in each case the team of threads referenced is that created by the innermost `parallel` instance:

- The *loop* construct (designated by the `for` construct in C/C++) segregates the team of threads so that the iterations of one or more associated loops are divided amongst the group.[23] As the keyword hints, in C/C++ the loops in question must be *for-loops*. They must also obey the rules set out in Section 2.5.1—"Loop Construct"—of [52].

- The `sections` construct provides a model akin to that with which most parallel programmers will likely be familiar, that is the declaration of structured tasks which are to be executed in parallel. Individual blocks are delimited with the `section` construct, vis Figure 2.24. Each `section` is executed *once*, with the scheduling order being implementation specific.

- The `single` construct specifies that the child region should be processed by *one thread only*, which is *not necessarily the master of the team*.

- The `master` construct specifies that the enclosed region should be executed by *the master thread only*. For this reason it is also sometimes considered a synchronisation primitive, the rest of which are discussed later.

Both the loop and `sections` constructs may be "chained" onto the declaration of a `parallel` region to create a *combined parallel worksharing* construct. Such a construct is semantically equivalent to a `parallel` instance enclosing *only* the chosen work sharing construct; Figure 2.25 gives an example. Whether work sharing constructs are included plainly or combinatorially, certain restrictions govern their usage, outlined in [52]:

[23] The programmer may use the `schedule` clause to control exactly how work is assigned.

```
#pragma omp parallel for
{
  for (...) {
    ...
  }
}

...

#pragma omp parallel sections
{
  #pragma omp section
    ...

  #pragma omp section
    ...
}
```

Figure 2.25: Combined parallel constructs in the C language bindings for OpenMP.

1. Each work sharing region must be encountered either by all threads in a team or by none at all.
2. The sequence in which worksharing regions are encountered must be identical for all threads in a team.

By default a work sharing region mirrors the `parallel` construct in that there is an implicit barrier upon its completion. This can however be removed (using the `nowait` clause), allowing threads which complete early to continue executing code within their enclosing region.

**Data Environment**

The data environment directives provided by OpenMP allow the control of data visibility and access within a `parallel` region. There are three options when choosing the locality of data:

- The `shared` clause accepts a list of variables that are in scope at the point at which it is defined. Any reference to a variable in this list will refer to the location of the original variable, regardless of the thread in which the reference appears. Synchronisation remains the responsibility of the programmer, who must ensure that the variable is active for the duration of the threaded region.
- The `private` clause accepts a list of variables in the same manner as `shared`. References to list items refer to new, thread local copies of the original item. The item is not initialised[24] and its value is not kept for usage outside the `parallel` region. By default loop counters in work sharing regions are `private`.
- The `threadprivate` clause operates identically to the `private` clause with one exception: `threadprivate` variables maintain their value across `parallel` regions.

[24] Though the `firstprivate` and `lastprivate` directives can be used to annotate instances where this is not the case.

### Synchronisation

Synchronisation clauses can be used to demarcate areas in the program where thread interleaving would otherwise cause problems:

- `critical` and `atomic` specify areas that must be entered by only one thread at a time. In particular, `atomic` hints to the compiler that special machine instructions should be used for better performance, but the compiler is permitted to ignore this and act as if `critical` had been used.
- A `barrier` creates an explicit barrier at the point it is declared, causing all threads in the innermost team to wait for each other before continuing execution.
- The `ordered` clause can be used to define a region that is executed in the order that the innermost loop construct would have executed had it been processed sequentially.

The `nowait` clause mentioned earlier is also categorised as a synchronisation directive; as stated its purpose is to remove the barrier that is implicitly present at the end of all work sharing regions.

### Runtime Library and Environment Variables

OpenMP's runtime library provides definitions for the important data types and functions that allow developers to work with its capabilities during program execution:

- *Execution environment* routines relate to threads and their parallel scope. Functions for managing the number of running threads (`omp_get,set_num_threads()`) and working with team sizes (`omp_get_team_size()`) are examples of this type of procedure.
- *Lock* functions form the basis of general purpose synchronisation within OpenMP, and work with the lock data types also defined in OpenMP's runtime library. Both *simple locks*—which may only be locked once before being unlocked—and *nested locks*—which may be locked multiple times (by the same owning thread) before being unlocked—are supported.
- Two *timing* procedures support a "portable wall clock timer" [52].

### Implementations

OpenMP has seen considerable adoption in commercial products, with vendors such as IBM, Intel, Sun and HP implementing the specification for C, C++ and Fortran.[25] As of version 4.2 the *GNU Compiler Collection* (GCC) has supported version 2.5 of OpenMP, an effort coordinated under the *GOMP* project; GCC 4.4 is expected to support OpenMP 3.[26]

[25] `http://www.openmp.org/wp/openmp-compilers`

[26] `http://gcc.gnu.org/projects/gomp`

# Streams in Haskell | 3

We now discuss the use of streams in Haskell to write parallel programs. Streams offer a powerful yet simple model of parallelism which frees the user from the burden of explicitly parallelising code; the use of streams is the “hint” that a compiler or runtime needs in order to parallelise code automatically. In this chapter we discuss the methods that may be used to parallelise stream code written in Haskell using CUDA, both at compile-time and runtime. We focus on a runtime approach and present Haskgraph (Section 3.3), a pure Haskell library that uses metaprogramming to generate code for parallelising stream code. Chapter 4 continues with details of how the library is built with respect to an implementation on CUDA.

## 3.1 Aims and Motivations

Any high level abstraction must satisfy the two criteria of simplicity and predictability. The stream programming model fulfills both of these:

- Conceptually a stream is just a set of data – analogous to a *list* in Haskell. Anyone that can understand lists should therefore be able to understand streams.
- By providing a similar family of stream processing functions to those for lists, code and results are predictable.

In principle we would like to be able to write a program as in Figure 3.1. Even with no knowledge of streams or their combinators, it is clear what this program does.[1]

```
f :: a → b
g :: a → b → c

...

main :: IO ()
main =
  print zs
  where
    xs = streamFromList [1..100]
    ys = mapS f xs
    zs = zipWithS g xs ys
```

Figure 3.1: A stream program in Haskell.

But how does it end up being parallelised? The answer depends on the target platform.

[1] Not including the definitions of `f` and `g`, of course.

### 3.1.1 Underlying Architecture

Stream computers have been implemented on a variety of targets using various methods of abstraction. Section 2.4 presented two heterogeneous platforms on which parallel software has been developed: Nvidia's GPU-based CUDA and STI's Cell Broadband Engine. In this project we explore a solution that utilises CUDA, for the following reasons:

- Performance gains have been widespread with CUDA [34, 35, 36]; the literature on its advantages, disadvantages and methodologies is consequently substantial.
- CUDA's programming interface is relatively uncluttered and intuitive, largely due to the use of a domain-specific language (DSL).
- Good software engineering practices should ensure that architecture-specific back ends can be swapped in and out or combined with little effort.

So how might the program in Figure 3.1 be realised in CUDA? Taking the application of `mapS f` in line 11, for example, and assuming that `f` has type `Int` → `Int`, we would perhaps write its equivalent CUDA kernel as in Figure 3.3.

```
__global__ void mapSCuda(int *xs, int *ys, int N) {
  int thread_id = blockIdx.x * blockDim.x + threadIdx.x;

  if (thread_id < N) {
    int x = xs[thread_id];
    int y;

    ...
    /* The inlined body of f, which assigns to y. */
    ...

    ys[thread_id] = y;
  }
}
```

Figure 3.2: A CUDA kernel for the function `mapS f`.

Conceptually this would be launched with `N` threads (where the stream `xs` has `N` elements), each of which applies `f` to a unique element in the stream. This is illustrated in Figure 3.3.

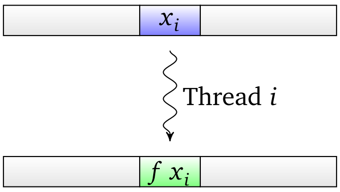

Figure 3.3: Applying the function `mapS f` using CUDA.

Generating this kernel from the stream code is non-trivial; there are several points that must be addressed:

#### Type Correspondence

Haskell's type system is richer than that of C/CUDA. In our example, Haskell `Ints` are mapped to C `ints`. This merging of type systems presents several potential pitfalls:

- There may be no suitable CUDA representation for a given Haskell type. Consider, for example, an infinite data structure, or the `Integer` type, which can handle arbitrarily precise numbers.

- Only a *partial representation* may exist: this is the case in our example – the upper bound of the `Int` type is actually implementation specific: there is consequently no guarantee that it will fit in an `int`.

- C's lack of bounds checking means that the *sizes* of variable length structures must be supplied to the kernel as extra parameters (here, the value `N`).

#### Function Code Generation

We need to convert `f` to some CUDA equivalent. In order to do this we must be certain that `f` is computable on CUDA. Data movement is also an issue: instead of returning values, CUDA kernels make use of *output parameters*. In our example this is the purpose of the `ys` parameter, and its relationship with the return value is clear. In general however, we may be required to encode arbitrary return types. The separation of inward and outward parameters also raises questions about how much data movement should occur – do ingoing parameters have to be copied back from the GPU to the host, for example?

## 3.2 Implementation Options

There are two points within a program's life cycle at which we can parallelise code: *compile-time* and *runtime*. Instrumentation from inside the compiler brings some attractive performance benefits – a one-off cost is paid in the form of program compilation time in exchange for a self-contained, parallelised executable file. A compile-time implementation using GHC's `Cmm` code generator was explored in the initial stages of the project, but was later abandoned. It turns out that exploiting parallelism well at compile-time is very hard to do, since a lot of necessary information isn't available until runtime. An outline of an alternative approach using GHC is given in Section 6.1.1; for the remainder of this and the next chapter we present Haskgraph, a runtime approach for handling stream code.

## 3.3 Haskgraph

At runtime, compilation and linking are expensive, but the availability of extra information may facilitate optimisations; for example, if the kernel in Figure 3.2 was generated at runtime, the parameter `N` could be omitted and hard wired into the source code. Haskgraph is a pure Haskell library that uses runtime code generation to parallelise stream applications without any modifications to the compiler. In this section we discuss the interface it presents to a user, and provide a first look at how it generates parallel code. In Chapter 4 we focus on implementation details, covering a CUDA specific back end in Section 4.2.

### 3.3.1 Restricting Streamable Types

Consider once again the application of the function `mapS f`. What is `f`'s type? Our experience with `map` and its applications to lists suggests that it should be `a → b`. This is a problem: `f` is *polymorphic*, but we noted earlier that not all types may have a representation on a GPU, or indeed any other multi-core back end. We must restrict the types on which `f` may operate. Typically this is achieved using *type classes*, and it is in this manner that we introduce the notion of a *describable type*, a type which any back end architecture can represent. The class' definition is given in Figure 3.4.

```
data TypeDesc
  = Int
  | Float
  ...

class DescribableType a where
  getTypeDescriptor :: a → TypeDesc
```

Figure 3.4: `DescribableType`s have a representation in CUDA (or any back end).

`f`'s type becomes `(DescribableType a, DescribableType b) ⇒ a → b`; applying it to invalid objects is now a *type error*, and will be raised as such at compile-time.

### 3.3.2 Describing Computation

In a similar vein let us think about `f`'s body. With the proposed approach we want to make `f` *responsible for generating its own code*, i.e., `f` doesn't so much compute an answer as *describe* how it should be computed. This can be modeled with an *abstract syntax tree* (AST). `Haskgraph.Core.Expr` is a data type for building such an AST; its definition is shown in Figure 3.5.

```
data Expr a where
  Lit    :: DescribableType a ⇒ Const a → Expr a
  Val    :: DescribableType a ⇒ Var a → Expr a
  App    :: DescribableType a ⇒ Op → [Expr a] → Expr a
  Cast   :: Castable b a ⇒ Expr b → Expr a
  If     :: DescribableType a ⇒
            BoolExpr → Expr a → Expr a → Expr a
```

Figure 3.5: The `Haskgraph.Core.Expr` type.

`Expr` is a small DSL for working with basic arithmetic, additionally supporting *conditional expressions* (`if...then...else` in Haskell) and *type casting*. As a first attempt we could use it to build an arbitrary expression representing `f`, giving `f` a type of `Expr a → Expr b`. Note that `f`'s restriction over describable types is preserved through the contexts of `Expr`'s constructors.

Unfortunately, this representation of `f` suffers a few limitations. In an imperative environment such as that of CUDA, functions are composed of *statements*, which implement *actions* such as assignment. We therefore need to be able to accumulate these statements in order to construct kernels – we need to maintain *state*. As mentioned in Section 2.2.1, this can be achieved using a suitably defined monad. In Haskgraph, this monad is called `H`. It is important to note that `Expr` is still used to construct `f`: the monadic type `H a` *returns* values of type `Expr`, whilst allowing statements to be accumulated as a side-effect. The internals of `H` are discussed further in Chapter 4

### 3.3.3 Overloading for Expression Construction

Constructing expressions explicitly in terms of the constructors of `Expr` is inelegant. The expression `2 + x`, for example, would have to be written as `App Add [Lit Const 2, x]`. Since operators are just *infix functions* in Haskell, we can overload their behaviour so that they build expressions directly. This is done through instance definitions, since most operators belong to type classes. Figure 3.6 illustrates this point using the Prelude's `Num` class, all instances of which must provide arithmetic operators for working with integers.

```
class (Eq a, Show a) ⇒ Num a where
  (+)         :: a → a → a
  (-)         :: a → a → a
  (*)         :: a → a → a
  negate      :: a → a
  abs         :: a → a
  signum      :: a → a
  fromInteger :: Integer → a

instance Num a ⇒ Num (H a) where
  (+)        = liftedBinaryApplication Add
  ...
```

Figure 3.6: Overloading arithmetic operators using the `Num` class.

We can now write the above expression as just `2 + x` – our instance definition means that the function actually called is `liftedBinaryApplication Add 2 x`, a function which builds the expression explicitly for us.

### 3.3.4 Streams as a Type Class

We now turn to the representations of streams themselves. In our example, `mapS f` is being applied to the stream `xs`. It is possible that `xs` is nothing more than an array in memory, but it could also be a closure (cf. Haskell lists). Ultimately, it doesn't matter – we want to provide a consistent interface irrespective of issues such as this. This leads us to define the streams as a type class, shown in Figure 3.7.

```
class DescribableType a ⇒ Stream s a where
  newStream        :: Int → a → s a
  newEmptyStream   :: Int → s a
  streamFromList   :: [a] → s a

  mapS             :: Stream s b ⇒ (H a → H b) → s a → s b
  zipWithS         :: (Stream s b, Stream s c) ⇒
                      (H a → H b → H c) → s a → s b → s c
  foldS            :: (H a → H a → H a) → s a → a
```

Figure 3.7: Haskgraph's `Stream` type class.

```
let
  xs = newStream 5 1
  -- xs = [1, 1, 1, 1, 1]

  ys = newEmptyStream 5
  -- ys = [⊥, ⊥, ⊥, ⊥, ⊥]

  zs = streamFromList [1..5]
  -- zs = [1, 2, 3, 4, 5]
```

Figure 3.8: Stream construction using the `Stream` class. Streams have been written out as lists for clarity.

Lines 2 to 4 are functions for *constructing* streams – `newStream` and `newEmptyStream` both take a size and, in the case of `newStream`, an initial value for all elements of the list. `streamFromList` creates a stream from a list. Figure 3.8 gives an example of each.

A stream is parameterised by two types:

- `s` is the type of the *container*. This can be seen as analogous to a list of type `[a]`. Here, the container *is the list*, i.e., `[ ]`. An example of this is given in Section 4.2, where we introduce the `CudaStream` container, a type well suited for moving between host and GPU devices.

- `a` is the type of the *elements*, which must be of a describable type. In the case of the list with type `[a]`, this is the type `a`.

By offering an interface parameterised in this fashion, implementing back ends can choose to represent streams in a manner that can be easily manipulated on the target platform. Issues such as contiguity and alignment, for example, can be dealt with on a per-module basis.

### 3.3.5 A Working Stream Program

Recall the program we wished to be able to write at the beginning of this chapter (reproduced in Figure 3.9), along with possible definitions for floating point versions of the functions f and g.

```
f :: Float → Float
f x =
  sin x / cos x

g :: Float → Float → Float
g x y =
  2 * x + y

main :: IO ()
main =
  print zs
  where
    xs = streamFromList [1..100]
    ys = mapS f xs
    zs = zipWithS g xs ys
```

Figure 3.9: A stream program in Haskell.

The program we can actually write using Haskgraph is given in Figure 3.10.

```
f :: H Float → H Float
f x =
  sin x / cos x

g :: H Float → H Float → H Float
g x y =
  2 * x + y

main :: IO ()
main =
  print zs
  where
    xs = streamFromList [1..100]
    ys = mapS f xs
    zs = zipWithS g xs ys
```

Figure 3.10: A working stream program using Haskgraph.

We observe that only the type signatures of f and g are different, having had their parameters wrapped by H. Thanks to overloading their bodies actually remain the same, despite now doing completely different things. This will be the case in general, assuming f and g are *pure* functions (i.e., produce no *side-effects*).

# Implementing Haskgraph 4

In this chapter we turn to the internals of the Haskgraph library: how computational kernels are built and how code is generated. We examine these features with respect to a CUDA back end, but the library's design should facilitate their implementation on any multi-core platform.

## 4.1 Haskgraph.Core

`Haskgraph.Core` is at the top of Haskgraph's module hierarchy (depicted in Figure 4.1). Together with the `Stream` type class (defined in `Haskgraph.Stream` and discussed in Section 3.3.4), it comprises the interface presented to user programs. It defines describable types (Section 3.3.1) and provides the `Expr` data type (Section 3.3.2), along with its associated instance definitions.

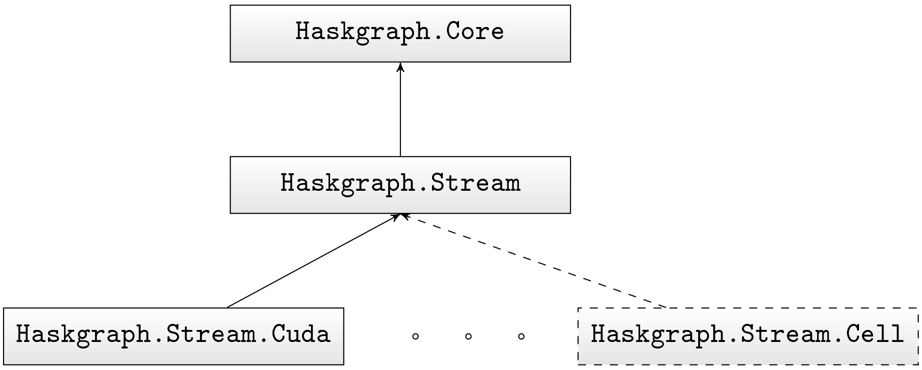

Figure 4.1: The Haskgraph module hierarchy. Dashed lines indicate modules that do not exist, but which could be added to extend the library's capabilities.

### 4.1.1 Why State is Necessary

In Section 3.3.2 the `H` monad was introduced to capture state when generating code for Haskgraph expressions. You might ask *why* – we could just build an expression tree representing a kernel and generate all the code in one go, when needed. The problem with such an approach becomes apparent when compounding expressions, as in Figure 4.2.

```
1 let
2   x = 3
3   y = 2 + x
4   z = 5 * y
```

Figure 4.2: Compounding expressions in stream computation.

On line 4, `z` makes use of `y`. If we had just used pure `Expr` values, then generating code for `z` would involve generating code to compute `y` *twice*: once when computing `y` itself and once when computing `z`. Clearly this is inefficient. By building kernels incrementally, we can avoid this; now line 3 can return an expression holding the *name of a variable* that will be assigned the value `2 + x` at runtime. However, as a *side-effect*, it also adds a statement to the current kernel body that makes such an assignment. In this manner, `z` now references `y` *symbolically*, and duplicate computation is avoided. This is illustrated in Figure 4.3, where we see the two possible expression trees for `z` (`x` has been inlined for simplicity).

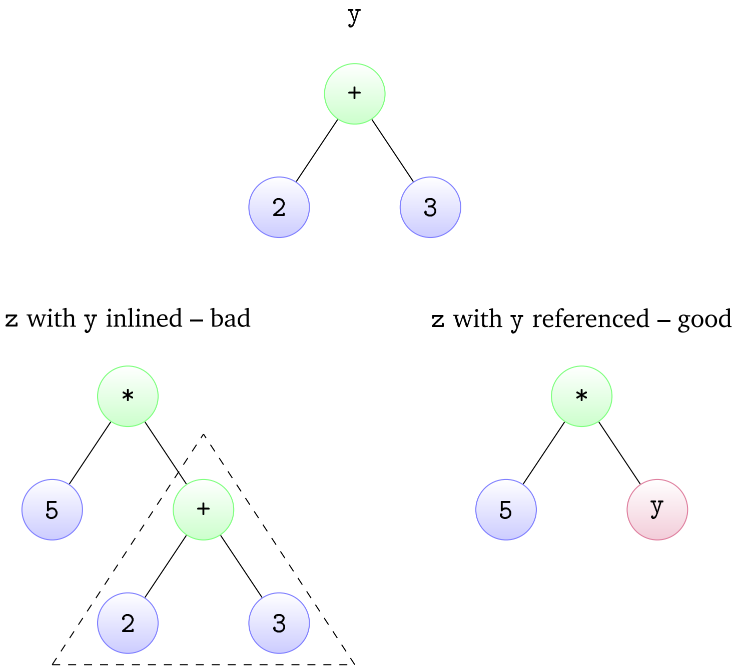

Figure 4.3: Expression trees created through expression compounding.

We note that the same point is not valid with respect to `y`'s use of `x` – since `x` is a constant there is no performance lost if it is used in multiple places.

### 4.1.2 Accumulating Kernel Bodies

Since state is (perhaps surprisingly) a common requirement when writing Haskell programs, the standard libraries define `MonadState` and `State`[1], a type class and a monad instance for threading state through a program. Figure 4.4 contains their definitions; the syntax `m → s` in a class definition's head expresses that `s` is *functionally determined* by `m`, that is, there can only be one value for `s` associated with each value for `m`.

```
class Monad m ⇒ MonadState s m | m → s where
  get :: m s
  put :: s → m ()

newtype State s a = State {
  runState :: s → (s, a)
}

instance MonadState s (State s) where
  ...
```

Figure 4.4: The `MonadState` type class and the accompanying `State` monad. Here, `s` is the type of the state and `a` is the type of the return value.

We can use the `State` monad in a first attempt at defining `H`. An example of how this might be applied to our earlier example (Figure 4.2) is given in Figure 4.5.

```
type H a = State [Stmt] a

evaluate :: ... → H (Expr a)
evaluate ... =
  do
    -- Generate a variable to assign the result to.
    result ← generateVariable ...

    -- Add an assignment to the generated variable.
    assign result ...

    -- Return the name of the variable.
    return result

assign :: ... → H ()
assign ... =
  do
    body ← get

    let assignment = ...

    put $ body ++ [assignment]
```

Figure 4.5: Using the `State` monad to build kernels.

Here, `evaluate` is an arbitrary function that might handle addition, for example. It creates a statement that assigns the result of some expression into a

[1] Part of the `Control.Monad.State` module.

freshly generated variable, and returns that variable as its result. We see that this definition of H is still not quite right according to the definitions given in Sections 3.3.3 and 3.3.4, but that it implements the accumulation we discussed in this and the previous section.

### 4.1.3 Name Generation

When kernels are produced, the code generator creates a fresh name for each identifier it encounters. For example, an application of the function mapS (2+) to a stream of floating point numbers might produce a kernel as shown in Figure 4.6.

```
__global__ void name_1(float *name_4, float *name_5) {
  int name_6 = ((blockDim.x * blockIdx.x) + threadIdx.x);

  if (name_6 < 65536) {
    float name_7 = name_4[name_6];
    float name_8 = (2.0 + name_7);
    name_5[name_6] = name_8;
  }
}
```

Figure 4.6: Using generated identifiers to construct a kernel.

Clearly each name much be unique! Unique values are also a common requirement, and to this end the Haskell libraries provide the Unique type, as in Figure 4.7

```
data Unique = ...

newUnique    :: IO Unique
hashUnique   :: Unique → Int
```

Figure 4.7: The Unique type, provided by the Data.Unique module.

We notice that Unique values exist only within the IO monad, but that we have already committed to working inside the environment provided by State. To overcome this issue we can *transform* the IO monad so that it supports stateful computation using the StateT type, a so-called *state transformer*. StateT's definition is given in Figure 4.8. Here, s is (as before) the type of the state and m is the underlying monad that StateT is transforming.

```
newtype StateT s m a = StateT {
  runStateT :: s → m (a, s)
}

instance Monad m ⇒ MonadState s (StateT s m) where
  ...
```

Figure 4.8: The StateT state transformer monad.

Applying a transformer in this manner creates a *monad stack*. This is the Haskgraph monad, defined in Figure 4.9.

```
data HaskgraphState = HaskgraphState {
  graphBody :: [Stmt]
}

newtype Haskgraph a = Haskgraph {
  run :: StateT HaskgraphState IO a
}
```

Figure 4.9: The Haskgraph monad.

We have generalised the state to a record type so that extra information may be passed around. We see that lifting expressions into this monad encapsulates the necessary side-effects discussed up to this point whilst facilitating the construction of arbitrary ASTs; finally, this yields the H monad introduced in Section 3.3.2. Figure 4.10 gives H's definition.

```
type H a = Haskgraph (Expr a)
```

Figure 4.10: Defining H.

Note that H is actually a *type synonym* that lets us talk explicitly about expressions that have been lifted into the Haskgraph monad.

## 4.2 Hs$_{\text{CUDA}}$

Hs$_{\text{CUDA}}$ is an example of a code generating back end that could be implemented using Haskgraph. It provides the CudaStream type, an instance of the Stream type class that generates CUDA kernels at runtime to exploit parallelism exposed by the usage of streams. Figure 4.11 shows the type's definition.

```
newtype CudaStream a = CudaStream (StorableArray Int a)

instance DescribableType a ⇒ Stream (CudaStream a) where
  ...
```

Figure 4.11: The CudaStream type.

StorableArray is a data type provided by Haskell's array libraries that offers the following guarantees:

- *Contiguous storage* – the array is laid out in a C compatible manner, and all elements are *unboxed* i.e. in-place as opposed to referenced by pointers.

- Access to an underlying *foreign pointer*, so that the data structure may be passed across language barriers (C to Haskell, for instance).

- Use of *pinned memory*, ensuring that any use of a foreign pointer will not be interrupted by the garbage collector.

These points mean `StorableArray`s are ideal for using with CUDA; the type's support for foreign pointers means that we can use Haskell's *foreign function interface* (FFI) to move streams from Haskell to C and back, while marshalling is simplified by the fact that stream data on the heap is unboxed and contiguous.

### 4.2.1 Generating CUDA Code

Like `Haskgraph.Core`, Hs$_{\mathrm{CUDA}}$ pairs a domain-specific language (`Cuda.Expr`) with a monad (`Cuda`) for code generation. Since `Cuda.Expr` is used exclusively by Hs$_{\mathrm{CUDA}}$, it does not prevent the construction of many potentially dangerous statements. Despite this it could be modified with little effort to produce a general-purpose DSL for working with CUDA in Haskell. Definitions are given in Figure 4.12.

```
data CudaType
  = Void
  | Pointer CudaType
  ...

data Direction
  = HostToDevice
  | DeviceToHost

data Expr where
  Constant      :: DescribableType a ⇒ a → Expr
  Variable      :: String → Expr
  Apply         :: Op → [Expr] → Expr
  Cast          :: CudaType → Expr → Expr
  AddressOf     :: Expr → Expr
  Dereference   :: Expr → Expr
  Index         :: Expr → Expr → Expr
  CopyDirective :: Direction → Expr

data Stmt = ...

data CudaState = CudaState {
  kernelBody :: [Stmt]
}

newtype Cuda a = Cuda {
  run :: StateT CudaState IO a
}
```

Figure 4.12: `Cuda.Expr` and the `Cuda` monad.

`Cuda.Expr` is a closer realisation of a CUDA expression's AST, supporting *pointers*, *void types* and directives for moving data between the host and the GPU. Kernel bodies are again realised as sequential lists of statements, supporting operations like control flow, memory management and block-local synchronisation (`__syncthreads()`). As with `Haskgraph.Expr`, `Cuda.Expr` is also an instance of common type classes such as `Num` and `Floating`, making it easy to construct arbitrary expressions.

### 4.2.2 Wrappers

Wrappers are pure C functions callable from Haskell that wrap calls to CUDA kernels. They are also used for compartmentalising several "boilerplate" actions.

#### Global Synchronisation

There is no facility in CUDA for synchronising threads "across the card" (see Section 2.4.1). This restriction is typically overcome by using kernel launch (a relatively cheap operation) as a global synchronisation primitive. An illustration of this technique is given in Figure 4.13, whereby a stream of elements is being summed. After the first invocation of the summing kernel, a set of partial sums remain. By invoking the same kernel again, we obtain the single result we need.

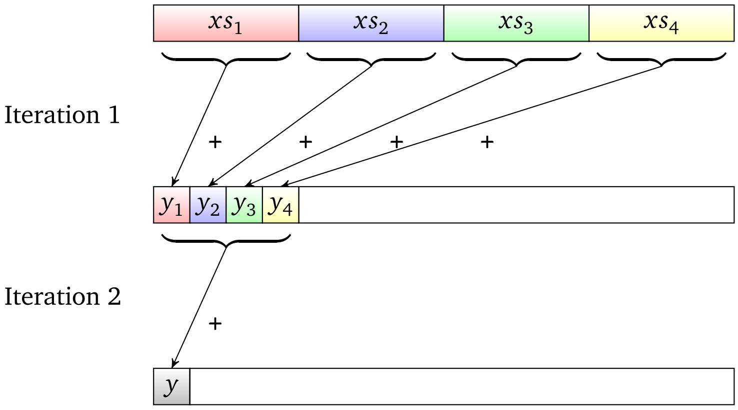

Figure 4.13: Using kernel invocation to synchronise data in a sum.

Hs$_{\text{CUDA}}$ supports this type of synchronisation through *iterative kernels*, i.e., kernels whose domain and codomain are the same. Kernels of this nature are repeatedly launched with their input and output parameters switched (an $O(1)$ operation since all parameters are pointers). This is cheaper than moving data between iterations and, in the event that there are an even number of invocations, an extra kernel for copying the data to its correct location (in parallel) is generated and called after the last iteration has completed execution.

#### Marshalling and Blocking

Marshalling is the act of gathering data and copying it between the host and the GPU, prior to and following computation. As stated earlier, this is a relatively simple process as a result of choosing the `StorableArray` type as a stream representation. However, it requires that *blocking* be taken into account. Blocking is the act of dividing up streams of data into pieces—blocks—that will fit into GPU memory and arranging for the pieces to be processed separately. This is not a straightforward process: it is heavily dependent on the data and algorithm being used. We overcome this in Hs$_{\text{CUDA}}$ by introducing *parameter strategies*, which describe how partial computations on separate blocks should be composed into an end product. Strategies are defined as an ADT, the definition of which is given in Figure 4.14.

```
data ParameterStrategy
  = Reduce (Expr → Expr → Cuda Expr)
```

Figure 4.14: Parameter strategies as defined in $\text{Hs}_{\text{CUDA}}$.

Currently, only one scheme is supported: `Reduce f` specifies that the function `f` should be used to reduce pairs of results. Since blocks could could be processed in any order, potentially using multiple GPUs,[2] `f` should be associative and commutative. For example, addition could be implemented as in Figure 4.15.

```
reduceAddition :: Expr → Expr → Cuda Expr
reduceAddition x y = return $ x + y

...

let
  strategy = Reduce reduceAddition
```

Figure 4.15: Using addition as a reduction function with the `Reduce` parameter strategy.

No checks are made with regard to whether `f` is associative and commutative – it is assumed that the user knows what they are doing.[3]

Kernel parameters are also associated with a *high level type*, another ADT defined as in Figure 4.16.

```
data ParameterType
  = Array Int32
  | Scalar ParameterStrategy
```

Figure 4.16: High-level parameter types for generating boilerplate code.

In the case of an array we pair it with its size; currently only scalar parameters can be assigned strategies. This does not limit the library at present – arrays are implicitly assigned a strategy whereby their blocks are processed separately and copied back into the correct part of the array.

[2] Neither of these features is implemented currently.

[3] The “users” in this context are the implementations of `mapS` and the like; users of the stream type class do not need to concern themselves with this.

### 4.2.3 Code Generation and Compilation

The features we have discussed up to now are aggregated by the `generateKernel` and `generateIterativeKernel` functions, whose type signatures are shown in Figure 4.17.

```
generateKernel            :: [ParameterSpec]
                          → (Expr → [(Expr, Expr)] → Cuda ())
                          → IO (String, String)

generateIterativeKernel   :: Int32
                          → [ParameterSpec]
                          → (Expr → [(Expr, Expr)] → Cuda ())
                          → IO (String, String)
```

Figure 4.17: Type signatures for the kernel generation functions in $\text{Hs}_{\text{CUDA}}$.

Aside from the fact that `generateIterativeKernel` takes an additional integer argument – the number of iterations, they both take the same parameters:[4]

- A kernel *signature*, given as a list of *formal parameter specifications*. The `ParameterSpec` type is a synonym for a triple containing the following:
  - A parameter's *direction* (either `In` or `Out`). Only outward parameters are copied back from the GPU to the host; this level of control can allow for simple optimisations or just tactful variable reuse.
  - A *primitive type*. This is a `CudaType` as defined in Figure 4.12.
  - The parameter's high level type, as given in in Figure 4.16.
- A *body generation function*. The parameters that the function must accept are the current thread's unique identifier and a list of *actual parameters*, each of which is a pair composed of:
  - An expression representing the corresponding formal parameter; this can be used inside the generator's closure to reference the parameter's value.
  - The parameter's size, with blocking taken into account.

  The generation function can also make use of CUDA-specific variables (`blockDim` and `threadIdx`, to name a couple) through similarly named functions; `gridDim.x`, for example, becomes `gridDim X`.

With all this information, generation and compilation proceeds as follows:

1. *Kernel generation* is almost entirely directed by the body generation function – the statement list is evaluated and code is produced.
2. *Wrapper generation* is hidden from the caller; it is performed using the formal parameter list:
   - Boilerplate code is generated through the analysis of parameter directions and strategies. For example, an steam may contain more elements than there are threads, and will be blocked as a result. All code is divided into pre- and post- kernel invocation blocks.

[4] `generateKernel` is, as you might expect, implemented in terms of `generateIterativeKernel`.

- Kernel calls are instrumented with appropriate sizings – iterative kernels are handled as described earlier. The blocks of boilerplate code are then assembled around the invocations.

3. The CUDA compiler[5] is called to compile the kernel and wrapper into a shared library. In the case of the wrapper, which is pure C, the host compiler will be invoked automatically by the CUDA compiler.

The names of the wrapper and shared library are then returned as a tuple, so that the caller has enough information to utilise the generated code.

**A Kernel Generation Example**

Figure 4.18 shows the call to `generateKernel` used in `zipWithS`.

```
1  generateKernel
2    [ (In, x_type, arrayP size),
3      (In, y_type, arrayP size),
4      (Out,z_type, arrayP size) ] $
5
6    \thread_id [(xs, xs_size), (ys, ys_size), (zs, zs_size)] →
7      ifThen (thread_id *< xs_size) $ do
8        x ← declareAssignLocal x_type (xs *!! thread_id)
9        y ← declareAssignLocal y_type (ys *!! thread_id)
10
11       z ← insertHaskgraph (f (parameter x) (parameter y))
12
13       zs *!! thread_id *= z
```

Figure 4.18: The kernel generation code used in `zipWithS`. Operators beginning with `*` are Hs$_{\text{CUDA}}$ versions of their vanilla Haskell equivalents.

Given an application of `zipWithS (+) xs ys` for two streams of one million floating point numbers, the CUDA code generated by this implementation is shown in Figure 4.19 (snipped and formatted for clarity). Let us quickly correlate the two pieces of code:

- Lines 2 to 4 of the Haskell code define the formal parameter list of the CUDA kernel and its wrapper. This corresponds to lines 1 and 13 in the generated code.

- Lines 8 to 9 declare local variables for holding one element of each of the two streams. These declarations appear in lines 5 and 6 of the generated code.

- Line 11 inserts the code that represents the function, `f`, that was passed to `zipWithS`. In this case `f` is the function `(+)`, and this is reflected in line 7 of the CUDA kernel code.

- Line 13 assigns the computed result back into the output parameter – this occurs in line 8 of the generated code.

We see also that the wrapper takes care of blocking and marshalling (lines 22 to 42), keeping the kernel and the Haskell code required to build everything relatively small.

[5] `nvcc` at the time of writing.

```
__global__ void name_1(float* name_4, float* name_5, float* name_6) {
  int name_7 = ((blockDim.x * blockIdx.x) + threadIdx.x);

  if (name_7 < 65536) {
    float name_8 = name_4[name_7];
    float name_9 = name_5[name_7];
    float name_10 = (name_8 + name_9);
    name_6[name_7] = name_10;
  }
}

extern "C" {
  void name_2(float* name_14, float* name_17, float* name_20) {
    float* name_15;
    cudaMalloc((void**)&name_15, (sizeof(float) * 65536));
    float* name_18;
    cudaMalloc((void**)&name_18, (sizeof(float) * 65536));
    float* name_21;
    cudaMalloc((void**)&name_21, (sizeof(float) * 65536));
    dim3 name_11(256);
    dim3 name_12(256);
    cudaMemcpy(name_15, (name_14 + 0),
      (sizeof(float) * 65536), cudaMemcpyHostToDevice);
    cudaMemcpy(name_18, (name_17 + 0),
      (sizeof(float) * 65536), cudaMemcpyHostToDevice);
    cudaMemcpy(name_21, (name_20 + 0),
      (sizeof(float) * 65536), cudaMemcpyHostToDevice);
    name_1<<<name_11, name_12, 2048>>>(name_15, name_18, name_21);
    cudaMemcpy((name_20 + 0), name_21,
      (sizeof(float) * 65536), cudaMemcpyDeviceToHost);

    ... ✂ 70 lines snipped

    cudaMemcpy(name_15, (name_14 + 983040),
      (sizeof(float) * 16960), cudaMemcpyHostToDevice);
    cudaMemcpy(name_18, (name_17 + 983040),
      (sizeof(float) * 16960), cudaMemcpyHostToDevice);
    cudaMemcpy(name_21, (name_20 + 983040),
      (sizeof(float) * 16960), cudaMemcpyHostToDevice);
    name_1<<<name_11, name_12, 2048>>>(name_15, name_18, name_21);
    cudaMemcpy((name_20 + 983040), name_21,
      (sizeof(float) * 16960), cudaMemcpyDeviceToHost);
    cudaFree(name_15);
    cudaFree(name_18);
    cudaFree(name_21);
  }
}
```

Figure 4.19: The CUDA code generated by the Haskell code shown in Figure 4.18.

### 4.2.4 Other Considerations

The use of output parameters means that all kernels constructed by Hs$_{\text{CUDA}}$ are *destructive*, that is, they return results by overwriting their output parameters in the Haskell heap. Clearly this is against the functional programming ethos, whereby change is reflected in returning *new* values. Rather than make the link between Haskell and C bidirectional, this behaviour is emulated by creating the required number of empty streams in Haskell code and passing them to wrappers. This is done with the expectation that they will be filled as a result of expectation. Consequently the stream functions retain their purity, leaving their parameters unmodified.

# Evaluation | 5

At this point we turn to the evaluation of the Haskgraph library. We first evaluate the compromises between speed and ease of programming by comparing several serial benchmarks with their equivalent Haskgraph implementations. We then examine how program rewriting can be exploited to minimise unnecessary use of the library, and the impact this has upon performance.

## 5.1 Benchmarking

All benchmarks were conducted on a machine with a 3.00Ghz Intel Core 2 Duo CPU with 6MB of shared L2 cache. The CUDA-compatible GPU used was an Nvidia GeForce 8600 GTS/PCIe, targeted by version 2.2 of the `nvcc` compiler. A maximum GPU thread limit of 65536 (a grid of 256 blocks of 256 threads each) was imposed so that the effects of blocking could be analysed, and each test was run five times with an average result taken. Unless stated otherwise, all tests were compiled using GHC 6.10.1 with no optimisations or other parameters.

### 5.1.1 Non-Iterative Benchmarks

The first set of benchmarks were *non-iterative*, that is, no looped computation occurred on the GPU.

`mapS`

Figure 5.1 shows the how runtime varies with stream size as the cosine of every element in a stream of single-precision floating point numbers is computed using `mapS`. We see that the serial implementation easily outperforms CUDA. This is to be expected – the non-iterative nature of the application means that the work done per byte copied is very low. Consequently, data movement dominates the CUDA application's runtime, resulting in poor performance. The extreme decrease in speed between 100,000 and 1,000,000 elements can be attributed to blocking – with only 65536 threads available on the GPU, the application suddenly moves from not needing blocking at all to requiring around 16 separately blocked computations. Since the current version of Haskgraph does not asynchronously stream new data to the GPU whilst existing data is being processed, this imposes a heavy additional cost. Figure 5.2 shows the increasing overhead that synchronous blocking adds to `mapS`'s runtime.

`foldS`

Figure 5.3 graphs runtime against stream size for computing the sum of a list of single-precision floating point numbers. Again we see that the overheads are dominant even for large streams, moreso than with `mapS`. We attribute this to the use of multiple kernel launches per block of computation, as covered in Section 4.2.2.

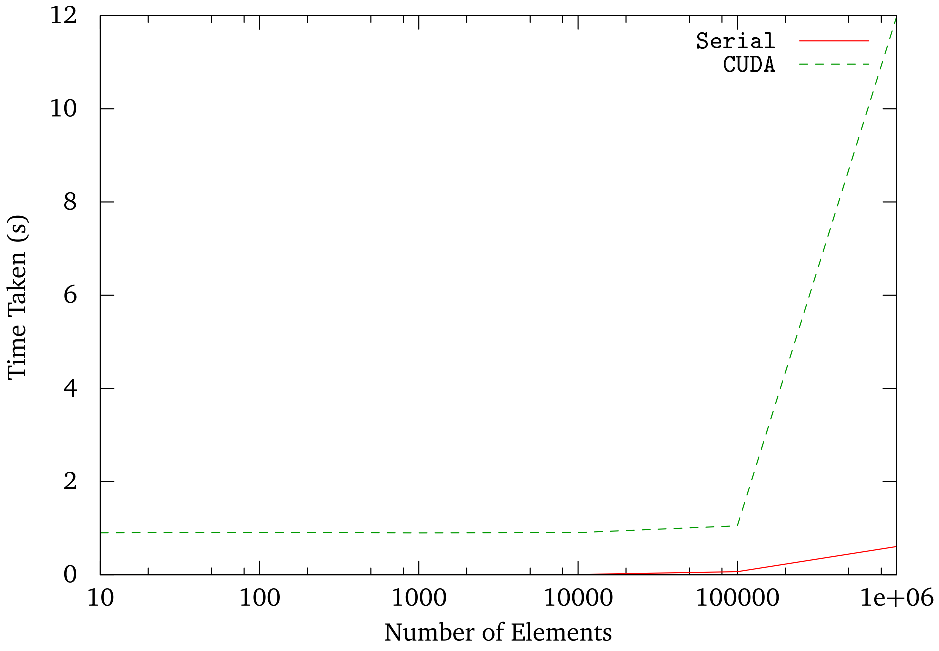

Figure 5.1: Performance of `mapS cos` vs. `map cos`.

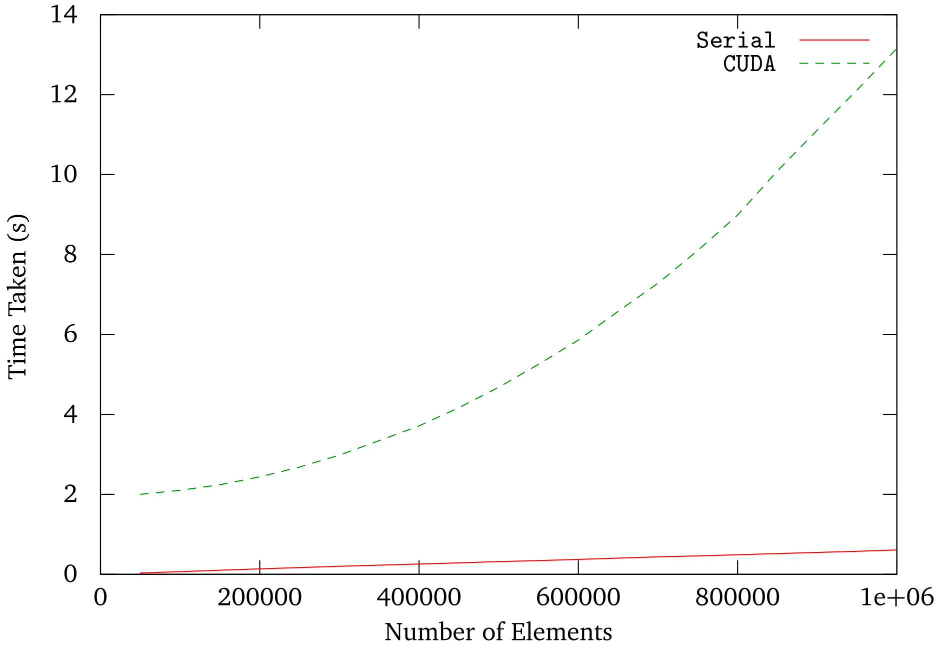

Figure 5.2: The effect of blocking on `mapS`'s performance.

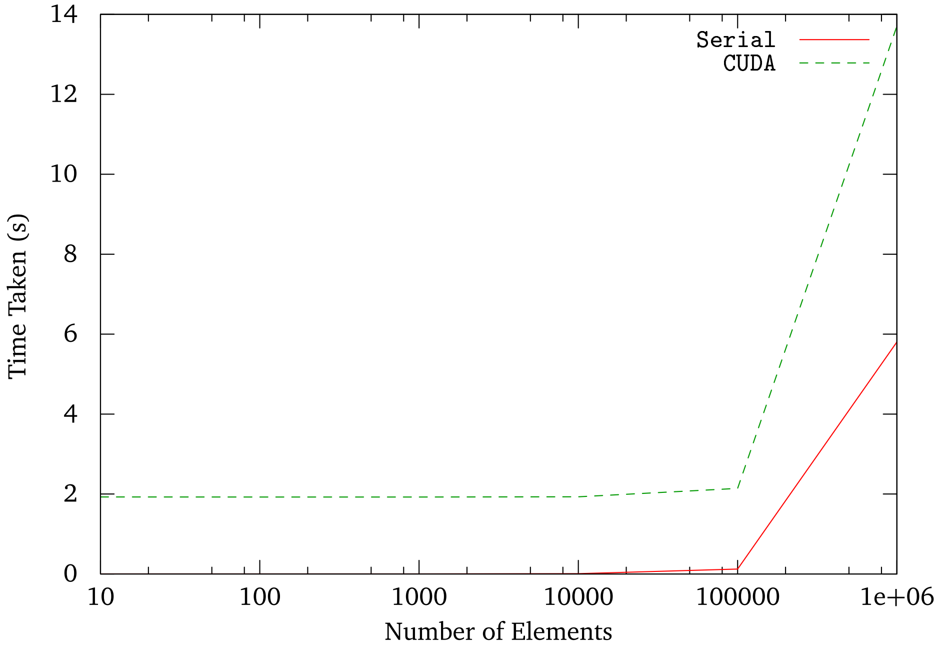

Figure 5.3: Performance of `foldS (+)` vs. `foldr (+) 0`.

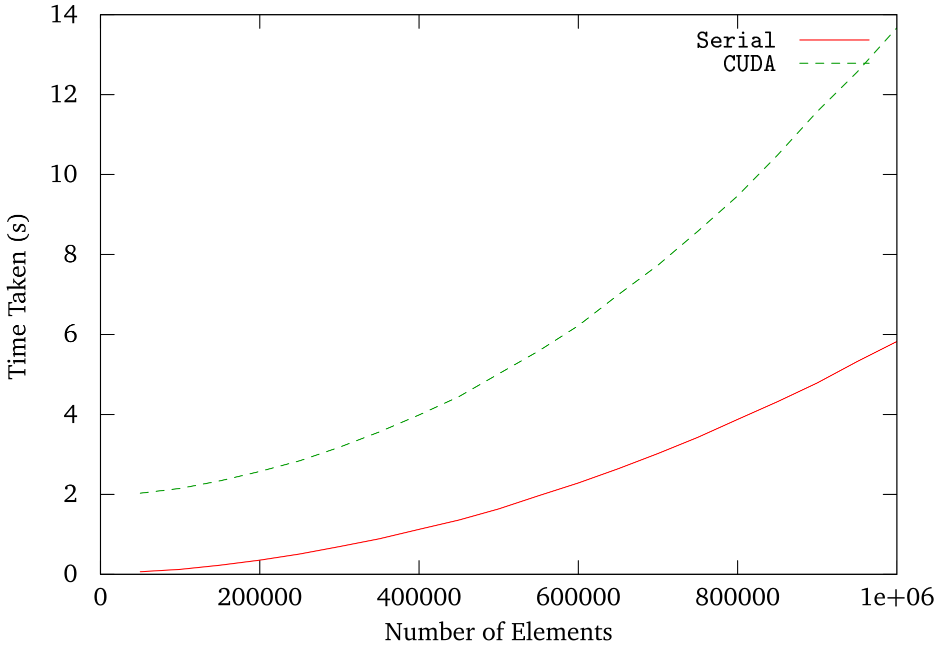

Figure 5.4: The effect of blocking on `foldS`'s performance.

### 5.1.2 Iterative Benchmarks

The second set of benchmarks were *iterative*, that is, looped computation occurred on the GPU. Iteration in Haskgraph is provided by the `iterateH` function, defined in Figure 5.5.

```
iterateH :: Int32 → (H a → H a) → H a → H a
```

Figure 5.5: Haskgraph's `iterateH` function.

`iterateH` takes an iteration count, a function to iterate and an initial value. It constructs an AST representing an iterated computation and returns an expression holding a reference to the result. In $\text{Hs}_{\text{CUDA}}$, iterated expressions are then transformed into `for`-loops. This often presents good opportunities for the compiler to unroll iterated computation, since the bounds are hard wired into the kernel's source code.

In Haskell, we emulate iterative behaviour using `iterateN`, a recursive function defined in Figure 5.6. Though similar to the Prelude's `iterate`, `iterateN` takes an iteration count, and only returns the final result of its computation.

```
iterateN :: Int32 → (a → a) → a → a
iterateN n f x =
  iterateN' n f x x
  where
    iterateN' :: Int32 → (a → a) → a → a → a
    iterateN' 0 f x a = a
    iterateN' n f x a =
      iterateN' (n - 1) f x (f a)
```

Figure 5.6: The `iterateN` function, used for emulating iterative behaviour in Haskell.

`iterateN` is written in a *tail recursive* manner, so that GHC is able to compile it to iterative code if it can do so.

#### `mapS`

Figure 5.7 plots runtime against iteration count for repeatedly doubling every element in a stream of single-precision floating point numbers. We see that the CUDA applications are relatively unaffected by rising iteration count; this is further demonstrated in Figure 5.8. Note that the scale in Figure 5.8 is much finer, and that the fluctuation in Figure 5.8 are therefore marginal and arguably unavoidable.

These results suggest that iteration on CUDA is very cheap, which is as we would expect: as a platform CUDA is perfectly suited for looped floating point computation. The serial implementations suffer from being tail recursive – GHC actually compiles tail recursive to a series of jumps to entry code, and not as loops. Consequently they grow linearly with respect to the number of iterations. In terms of space their growth will also be linear due to the use of the stack; contrast this with the constant space requirements of the CUDA versions.

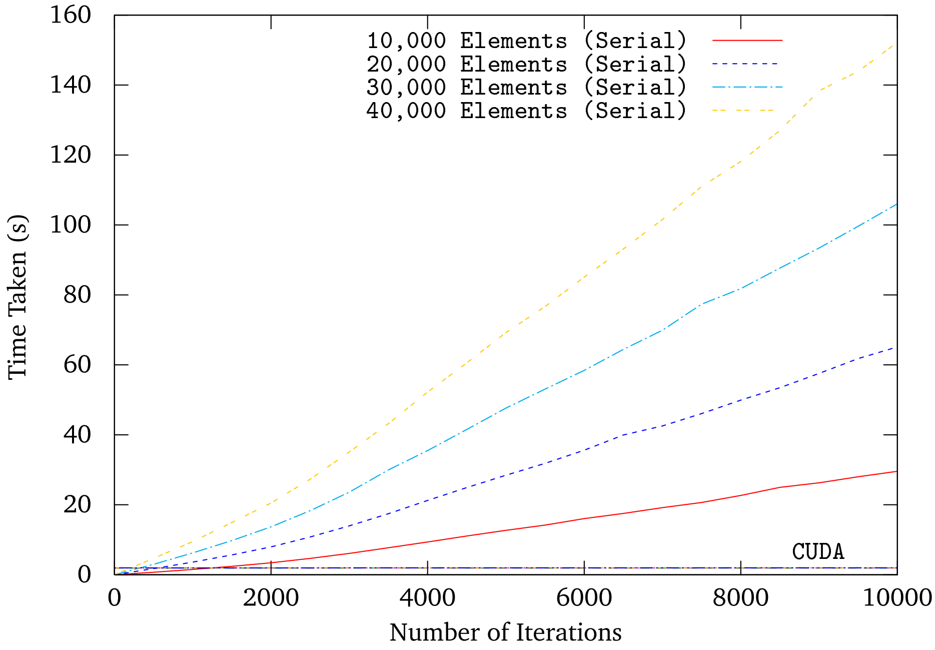

Figure 5.7: `mapS (iterateH (2*))` vs. `map (iterateN (2*))`.

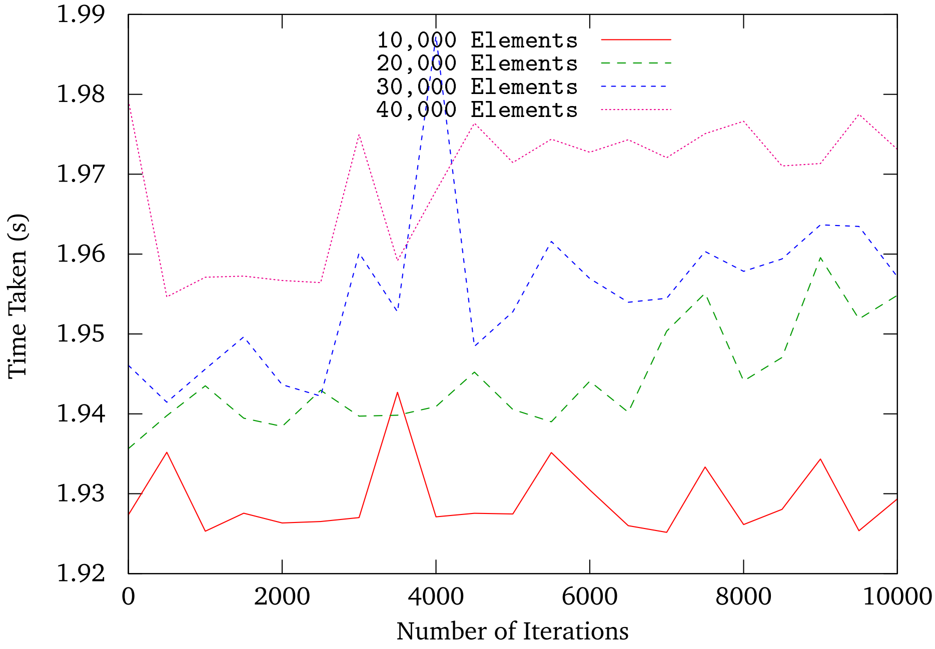

Figure 5.8: The performance of `mapS (iterateH (2*))`.

zipWithS

Figures 5.10 and 5.11 graph analogous runtimes for the zipWithS combinator. The functions zipperH and zipper are defined in Figure 5.9.

```
zipperH :: H Float → H Float → H Float
zipperH x y =
  cos (x + y)

zipper :: Float → Float → Float
zipper =
  cos (x + y)
```

Figure 5.9: The functions zipperH and zipper.

Similar results are again produced, although the CUDA runtimes are less variadic; this is a consequence of the extra stream that must be moved between the host and the device in a zipWithS. It is clear from all of the results we present that CUDA excels when computational load is intensive – when iterating 10,000 times over each of 50,000 elements we see that CUDA performs *over 200 times faster* than vanilla Haskell. This is further illustrated by the fact that CUDA continues to perform well for much larger iteration counts, as shown in Figure 5.12. For these iteration counts, serial benchmarking became intractable as stack operations dominated the runtime.

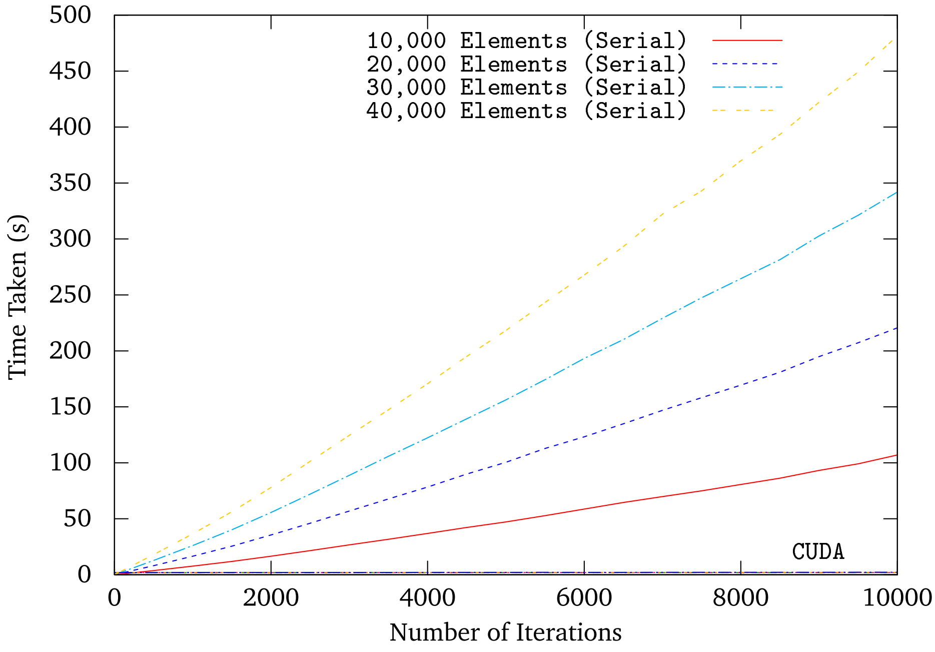

Figure 5.10: zipWithS (iterateH zipperH) vs. zipWith (iterateN zipper).

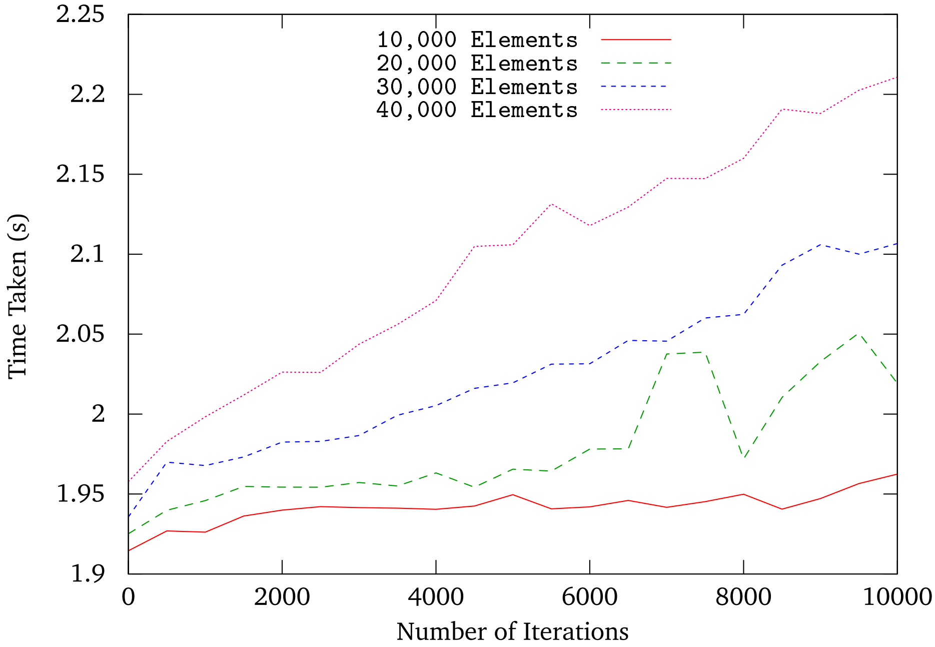

Figure 5.11: The performance of `zipWithS (iterateH zipperH)`.

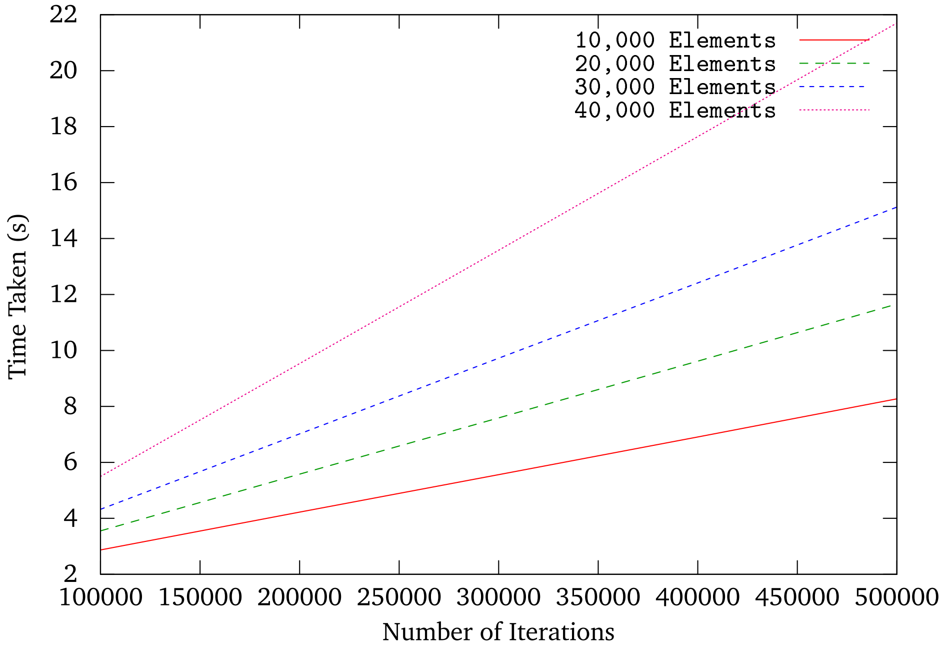

Figure 5.12: `zipWithS`'s performance for large iteration counts.

## 5.2 Rewriting for Optimisation

Rewriting is the process of applying a series of *source-to-source transformations* at compile-time. This is typically done in order to produce a better program, for some given definition of "better" – faster or more space efficient, for example. Figure 5.13 shows an application of this functionality as provided by GHC (Section 2.3).

```
 1 {-# RULES
 2   "map/map"
 3     forall f g xs. map f (map g xs) = map (f . g) xs
 4   #-}
 5
 6 main :: IO ()
 7 main =
 8   do
 9     let
10       xs = [1..10]
11       ys = map (2*) (map (2+) xs)
12
13     print ys
```

Figure 5.13: Rewrite rules as supported by GHC.

Rewriting actions are specified as a sequence of *rewrite rules*, denoted by GHC's `RULES` pragma. Each rule is assigned a name—used only for debugging purposes—and a corresponding body. During compilation, GHC will rewrite instances of the body's left hand side with the body's right hand side.

In Figure 5.13, line 11 will generate an unnecessary intermediate list through the double use of `map`. With the application of the `map/map` rule, this is no longer the case. The functions `(2*)` and `(2+)` will be composed, and the result will be an improvement in both space and time.

### 5.2.1 Rewriting Stream Code

The same principle can be applied to the stream functions provided by Haskgraph. Consider `mapS`, for example. Figure 5.14 shows how its type is compatible with the same compositions—and consequently rewritings—as `map`.

```
f :: b → c
g :: a → b

map f . map g :: [a] → [c]
map (f . g)   :: [a] → [c]

f' :: H b → H c
g' :: H a → H b

mapS f' . mapS g' :: s a → s c
mapS (f' . g')    :: s a → s c
```

Figure 5.14: Composing `mapS` with other stream functions.

Since `mapS` has significant startup overheads compared to `map`, rewriting to reduce its applications may offer considerable increases in performance. In this section we quantify these increases by comparing the execution times of programs which have been compiled several times, with rewriting enabled and with rewriting disabled. As before, the benchmarks were conducted on a machine with a 3.00Ghz Intel Core 2 Duo CPU (6Mb L2 cache) and an Nvidia GeForce 8600 GTS/PCIe, and the average time was taken over five executions.

### mapS and mapS

First let us consider the stream analogue of the program given in Figure 5.13, that is, the program given in Figure 5.15.

```
 1 {-# RULES
 2   "mapS/mapS"
 3     forall f g xs. mapS f (mapS g xs) = mapS (f . g) xs
 4   #-}
 5
 6 main :: IO ()
 7 main =
 8   do
 9     let
10       xs = streamFromList [1..10]
11       ys = mapS (2*) (mapS (2+) xs)
12
13     print ys
```

Figure 5.15: Applying the mapS/mapS rule to a stream program.

Only one rewrite will occur – the application on line 11. Figure 5.16 shows the runtimes of the original and rewritten programs over a scale of ten to one million elements.

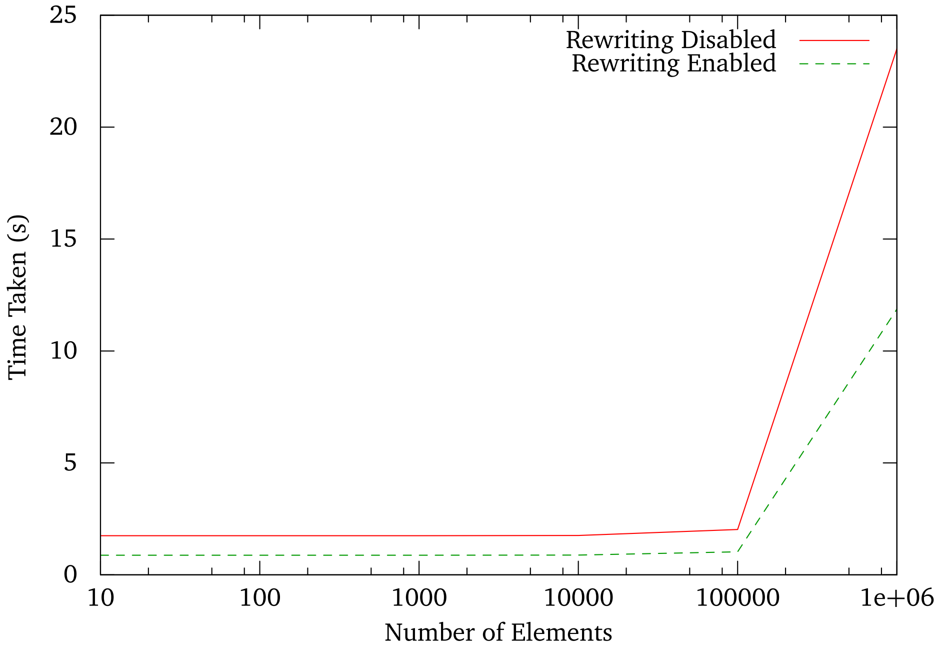

Figure 5.16: The performance gains of rewriting applications of mapS.

Overall we see about a 50% reduction in execution time if rewriting is enabled. Arguably this is as you might expect. One kernel is generated and compiled as opposed to two, and the copying of data between the host and the GPU only happens once, and not twice. The severe drop in performance between 10,000 elements and 1,000,000 is again a result of blocking. Figures 5.17 and 5.18 illustrate this point at coarse and fine levels, where we observe a steady increase in runtimes for both programs as more blocking is needed.

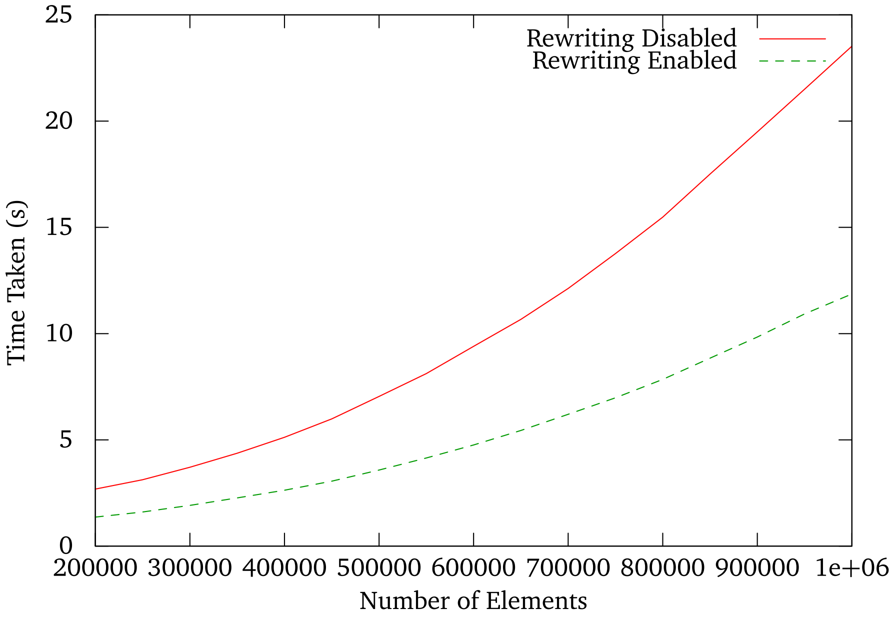

Figure 5.17: The effects of blocking on `mapS` (coarse grained).

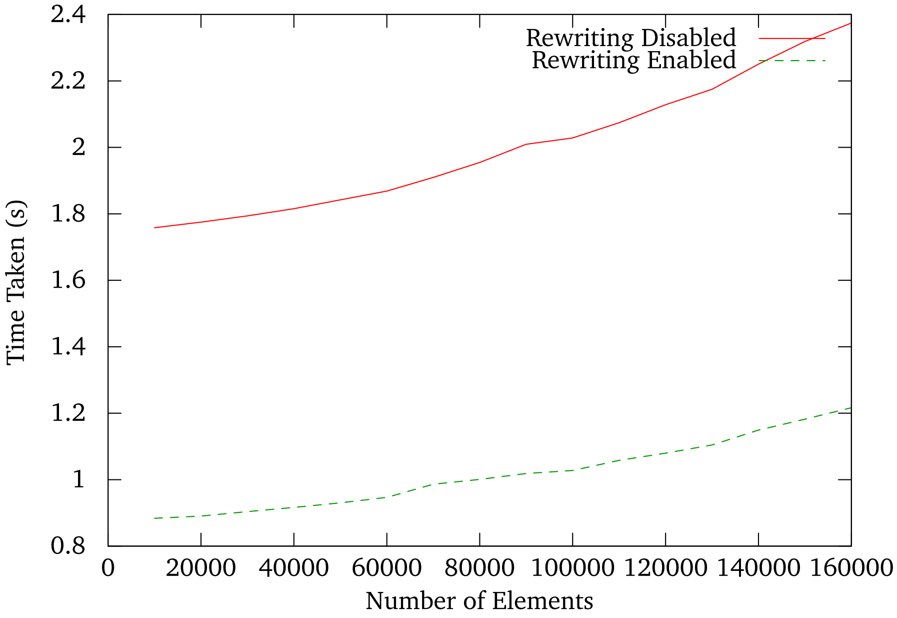

Figure 5.18: The effects of blocking on `mapS` (fine grained).

### mapS and zipWithS

We now turn to the composition of mapS and zipWithS. There are three possible compositions here – the application of mapS g to either zipWithS's first stream parameter, its second stream parameter, or both. Here we only consider the first and third cases: the second is analogous to the first and can be built using the flip function. For benchmarking purposes, we test the program shown in Figure 5.19 in three configurations:

1. With rewriting disabled.
2. With rewriting enabled and the inclusion of the zipWithS/mapS rule only. This will result in one rewriting, and two kernel generation / invocation processes.
3. With rewriting enabled and the inclusion of the zipWithS/mapS3 rule only. This will result in one complete rewriting of the zipWithS call, leaving only one kernel generation and invocation process.

```haskell
{-# RULES
  "zipWithS/mapS"
    forall f g xs ys.
      zipWithS f (mapS g xs) ys = zipWithS (f . g) xs ys;

  "zipWithS/mapS3"
    forall f g h xs ys.
      zipWithS f (mapS g xs) (mapS h ys) =
        zipWithS (flip (f . h) . g) xs ys
  #-}

main :: IO ()
main =
  do
    let
      xs = streamFromList [1..10]
      ys = streamFromList [1..10]
      zs = zipWithS (+) (mapS (2*) xs) (mapS (3*) ys)

    print zs
```

Figure 5.19: Applying the zipWithS/mapS rule to a stream program.

The results for varying stream sizes are plotted in Figure 5.20. We observe that we remove approximately 33% of the total work for each call to mapS we rewrite away. Avoiding this data movement is especially important as blocking increases, as indicated by Figures 5.21 and 5.22 by the divergent nature of the results.

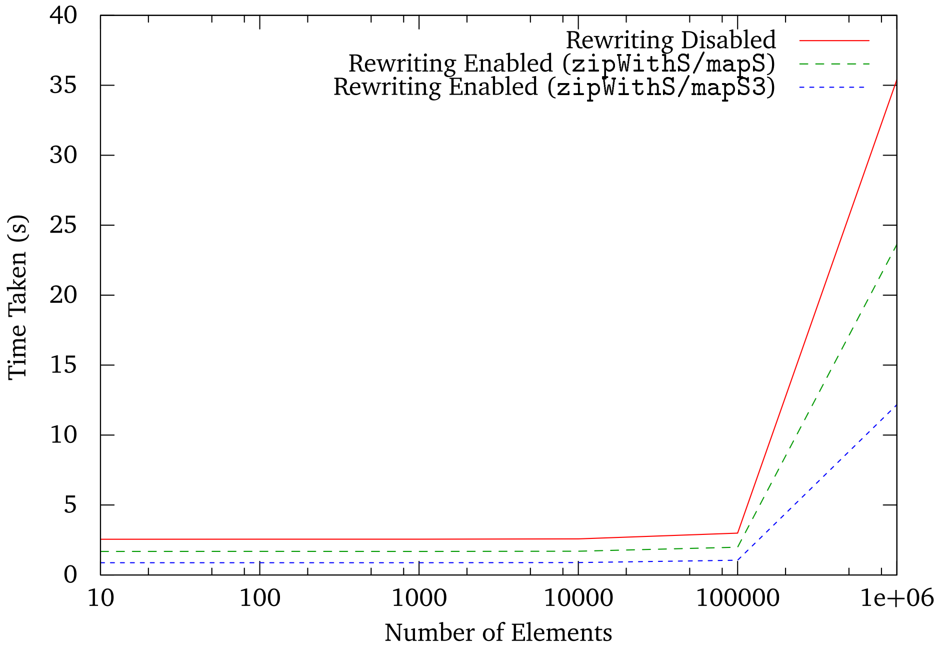

Figure 5.20: The performance gains of rewriting applications of `zipWithS`.

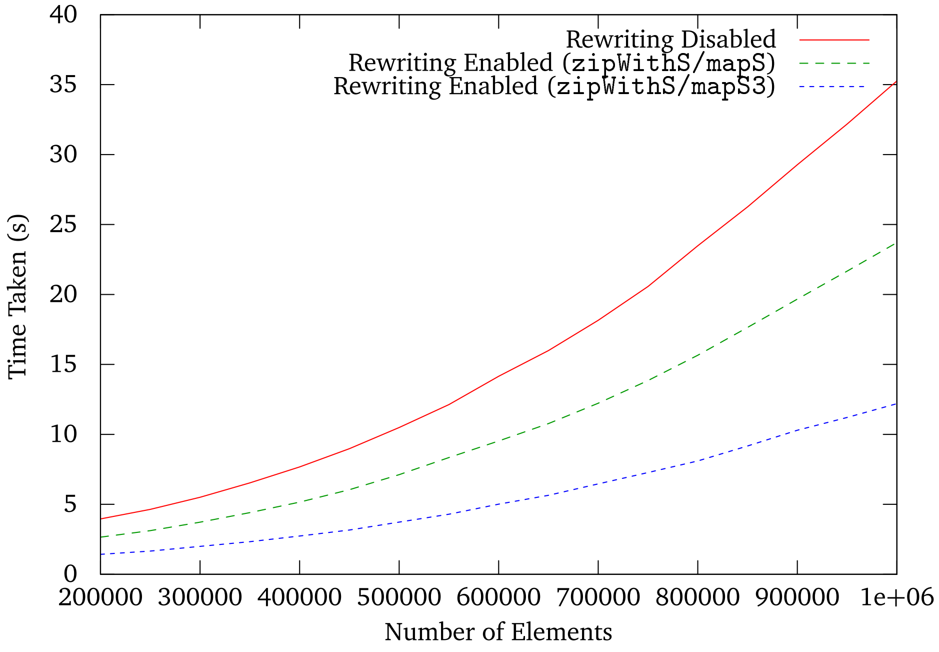

Figure 5.21: The effects of blocking on `zipWithS` (coarse grained).

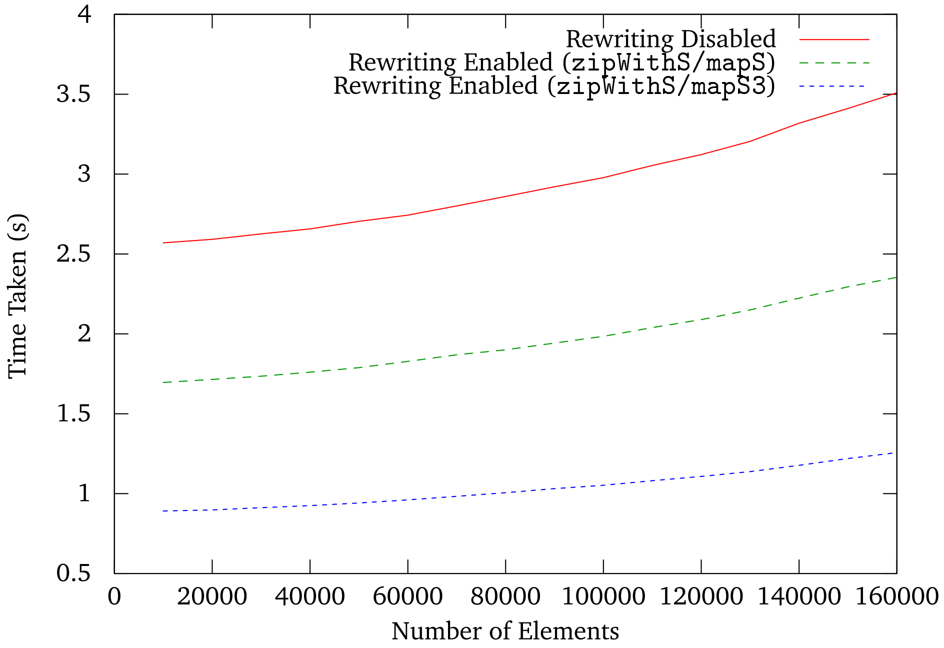

Figure 5.22: The effects of blocking on `zipWithS` (fine grained).

### `mapS`, `iterateH` and `iterateN`

We already saw in Section 5.1.2 that iterated computation is one of CUDA's strengths. Consequently, our last experiment concerns rewriting iterated computations onto the GPU in order to transform tail recursion into iteration. Consider the two methods of repeatedly applying a function to every element of a stream, as presented in Figure 5.23.

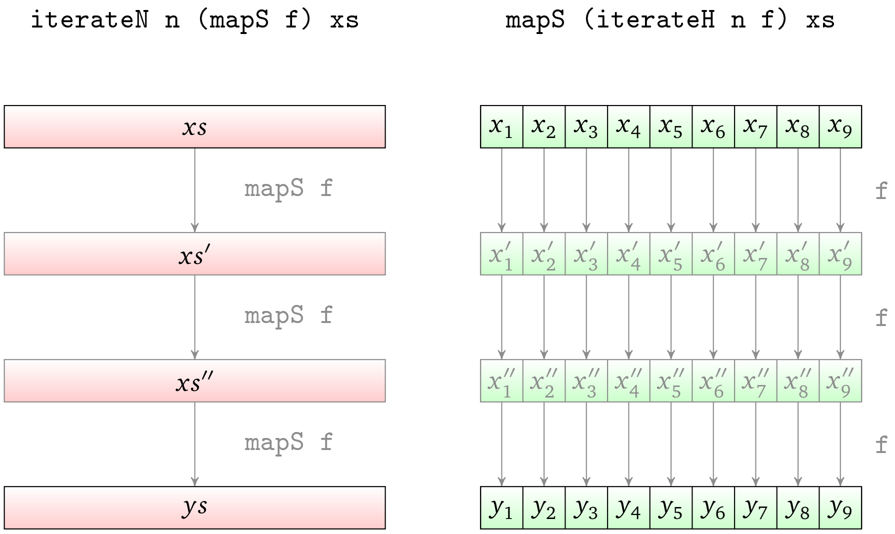
iterateN n (mapS f) xs
mapS (iterateH n f) xs
xs
mapS f
xs′
mapS f
xs″
mapS f
ys
$x_1$ $x_2$ $x_3$ $x_4$ $x_5$ $x_6$ $x_7$ $x_8$ $x_9$
f
$x'_1$ $x'_2$ $x'_3$ $x'_4$ $x'_5$ $x'_6$ $x'_7$ $x'_8$ $x'_9$
f
$x''_1$ $x''_2$ $x''_3$ $x''_4$ $x''_5$ $x''_6$ $x''_7$ $x''_8$ $x''_9$
f
$y_1$ $y_2$ $y_3$ $y_4$ $y_5$ $y_6$ $y_7$ $y_8$ $y_9$

Figure 5.23: Rewriting the composition of iterated functions for better performance.

We expect that the rightmost application will be faster – the other will likely incur repeated data movement costs. The leftmost method is also tail recursive, whilst the rightmost will be compiled to purely iterative code. Figure 5.24 shows how we can rewrite the slow version to the fast version.

```
{-# RULES
  "iterateN/mapS"
    forall n f xs.
      iterateN n (mapS f) xs = mapS (iterateH n f) xs
  #-}
```

Figure 5.24: Rewriting `iterateN` as `iterateH` to eliminate tail recursion.

Figure 5.25 shows the results of this rewriting for streams of 10,000 and 100,000 elements, with a varying number of iterations.

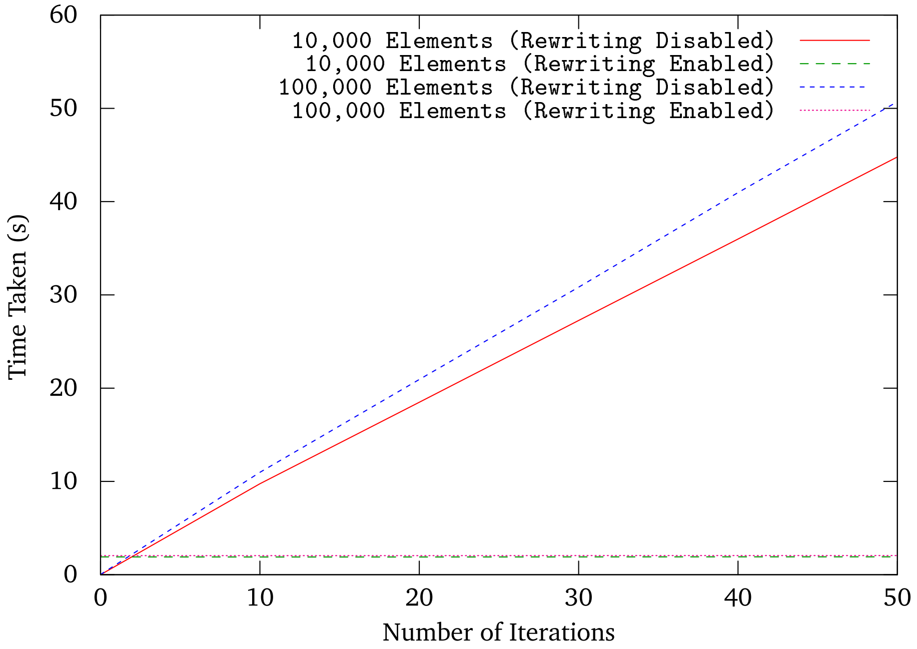

Figure 5.25: The performance gains of rewriting `iterateN` as `iterateH`.

The results are as expected: iteration is much cheaper on the GPU. Since Haskgraph does not currently implement any *caching*, each call to `mapS` in the slow version creates and compiles a new kernel. The cost of doing so, combined with the cost of moving data to and from the card for each invocation, is vastly greater than performing all of the iteration on the GPU.

# Conclusions | 6

In this report we have demonstrated that it is possible to provide high levels of abstraction for parallel programming. We have shown that the type system can be used to expose opportunities for parallelism, by introducing stream data types to Haskell. In evaluating our implementation, we showed that speed need not be sacrificed for these conveniences. We also exploited existing features such as GHC's rewrite engine to make some programs run even faster. We conclude the following:

- Using the type system to expose parallelism is an effective method that reduces programming complexity. Overloading means that markup code is minimised and program clarity is maintained (Section 3.3.3)

- Runtime code generation is a viable technique for producing parallel software (Section 5.1). In many cases the costs of generation and linking can be offset against the performance gained by parallelisation.

- Program transformations such as rewriting can provide further performance improvements by reducing unnecessary parallelisation (Section 5.2)

## 6.1 Future Work

In the remainder of this chapter we discuss some extensions and improvements that could be investigated, and their implications for what we have presented.

### 6.1.1 Constructing a `Core`-to-`Core` Pass

An alternative approach to parallelising code is at compile-time. In GHC, this could be achieved with a `Core`-*to*-`Core` *pass*, such as the *SimplCore* pass described in Section 2.3.1. A `Core`-to-`Core` pass receives a list of bindings from the program being compiled and returns a new list, representing the results of that pass' execution. In this manner stream functions could be recognised and *replaced* with parallelised equivalents. While parallelisation of this nature is typically more conservative since less information is available, it is likely that a statically linked executable would perform better than one that links in parallel components at runtime.

### 6.1.2 Complex Data Types

The current version of Haskgraph does not support the streaming of complex data types such as tuples or records. To make the library more useful, this would need to be addressed. Also of interest is an investigation into whether *infinite* data structures (lists, for example) could be handled by progressively blocking successive parts of the computation.

### 6.1.3 Using Performance Models

In the evaluation we presented many parallel applications that actually ran slower than their serial counterparts. These results were attributed to the overheads of data movement, and a low ratio of work done per byte copied. A possible solution to this is to use *performance models* to *speculatively* evaluate stream functions on CUDA. This is feasible since a function's structure can be determined by traversing its AST. For example, a simple model could only send iterative computations to the card, but a more complex one could potentially analyse the complexity of a function on a per-application basis.

### 6.1.4 Back End Specific Optimisations

The introduction of complex data types as described in Section 6.1.2 could present problems for a back end such as CUDA, where memory coalescing is needed to maximise performance returns. For example, it is often much more efficient to use *structures of arrays (streams)* (SoA) rather than *arrays (streams) of structures* (AoS). In general, transformations such as these can be considered back end specific optimisations, and adding support for them would greatly add to the value of the Haskgraph library.

### 6.1.5 Pluggable Back Ends

In this report we documented a CUDA-based implementation, but the layers of abstraction provided by Haskgraph mean that writing a back end for another platform (the Cell, for example) should be possible. It is expected that multiple platforms could be combined: since each back end provides a different stream data type, stream function calls could be delegated to different platforms, potentially in parallel.

# Bibliography

[1] Various Contributors, OpenMP: A Proposed Industry Standard API for Shared Memory Programming, available online at http://www.openmp.org/mp-documents/paper/paper.ps (PostScript) and http://www.openmp.org/mp-documents/paper/paper.html (HTML) (October 1997).

[2] Various Contributors, MPI: A Message-Passing Interface Standard, Version 2.1, available online at http://www.mpi-forum.org/docs/mpi21-report.pdf (September 2008).

[3] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, R. L. While, Parallel Programming using Skeleton Functions, in: Parallel Architectures and Languages Europe, Vol. 694, Springer-Verlag, 1993, pp. 146–160.

[4] H. P. Zima, H.-J. Bast, M. Gerndt, P. J. Hoppen, Semi-Automatic Parallelisation of Fortran Programs, in: CONPAR '86: Conference on Algorithms and Hardware for Parallel Processing, Springer-Verlag, 1986, pp. 287–294.

[5] Z. Li, P.-C. Yew, Efficient Interprocedural Analysis for Program Parallelisation and Restructuring, in: PPEALS '88: Proceedings of the ACM SIGPLAN Conference on Parallel Programming: Experience with Applications, Languages and Systems, ACM, 1988, pp. 85–99.

[6] B. D. Martino, C. W. Keβler, Two Program Comprehension Tools for Automatic Parallelisation, IEEE Concurrency 8 (1) (2000) 37–47.

[7] J. M. F. Moura, J. Johnson, R. W. Johnson, D. Padua, V. K. Prasanna, M. Püschel, M. Veloso, SPIRAL: Automatic Implementation of Signal Processing Algorithms, in: HPEC: High Performance Embedded Computing, 2000.

[8] M. Püschel, B. Singer, J. Xiong, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, R. W. Johnson, SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms, Journal of High Performance Computing and Applications 18 (2004) 21–45.

[9] N. T. Bliss, J. Kepner, pMatlab Parallel Matlab Library, The International Journal of High Performance Computing Applications 21 (3) (2007) 336–359.

[10] S. L. Peyton-Jones, Haskell 98 Language and Libraries: The Revised Report, Cambridge University Press, 2003.

[11] P. Hudak, J. Hughes, S. L. Peyton-Jones, P. Wadler, A History of Haskell: Being Lazy with Class, in: HOPL III: Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages, ACM, 2007.

[12] M. P. Jones, The Implementation of the Gofer Functional Programming System, Tech. Rep. YALEU/DCS/RR-1030 (May 1994).

[13] N. Röjemo, NHC: Nearly a Haskell Compiler, Chalmers Tech Report, available online at `http://www.cs.chalmers.se/pub/haskell/nhc/paper.ps.Z`.

[14] N. Röjemo, Highlights from NHC—A Space-Efficient Haskell Compiler, in: FPCA '95: Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture, ACM, 1995, pp. 282–292.

[15] Various Contributors, The Official YHC Manual, available online at `http://www.haskell.org/haskellwiki/Yhc` (2008).

[16] N. Mitchell, D. Golubovsky, M. Naylor, `Yhc.Core` - From Haskell to `Core`, The Monad.Reader (7) (2007) 45–61, , available online at `http://www-users.cs.york.ac.uk/~ndm/downloads/paper-yhc_core-30_apr_2007.pdf`.

[17] J. Meacham, The JHC Haskell Compiler, hosted and maintained at `http://repetae.net/computer/jhc` (2008).

[18] D. P. (sigfpe), You Could Have Invented Monads!, available online at `http://blog.sigfpe.com/2006/08/you-could-have-invented-monads-and.html` (August 2006).

[19] S. L. Peyton-Jones, Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine, Journal of Functional Programming 2 (1992) 127–202.

[20] S. L. Peyton-Jones, S. Marlow, Various Contributors, The GHC Commentary, available online at `http://hackage.haskell.org/trac/ghc/wiki/Commentary` (2008).

[21] M. Sulzmann, M. M. T. Chakravarty, S. L. Peyton-Jones, K. Donelly, System F with Type Equality Coercions, in: ACM SIGPLAN–SIGACT Symposium on Types in Language Design and Implementation, ACM, 2007.

[22] S. L. Peyton-Jones, A. Tolmach, T. Hoare, Playing by the Rules: Rewriting as a Practical Optimisation Technique in GHC, in: Haskell Workshop, ACM SIGPLAN, 2001, pp. 203–233.

[23] S. L. Peyton-Jones, S. Marlow, Secrets of the Glasgow Haskell Compiler Inliner (Revised Version), in: Journal of Functional Programming, Vol. 12, Cambridge University Press, 2002, pp. 393–434.

[24] S. L. Peyton-Jones, W. Partain, Let-floating: Moving Bindings to give Faster Programs, in: Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, ACM SIGPLAN, 1996, pp. 1–12.

[25] S. L. Peyton-Jones, N. Ramsey, F. Reig, C−−: A Portable Assembly Language that Supports Garbage Collection, in: Proceedings of the International Conference on Principles and Practice of Declarative Programming, Springer-Verlag, 1999, pp. 1–28.

[26] S. L. Peyton-Jones, A. Gordon, S. Finne, Concurrent Haskell, in: POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, 1996, pp. 295–308.

[27] Various Contributors, The GHC Online Documentation, available online at http://www.haskell.org/ghc/docs/latest/html (2008).

[28] N. Shavit, D. Touitou, Software Transactional Memory, in: Symposium on Principles of Distributed Computing, 1995, pp. 204–213.

[29] T. Harris, S. Marlow, S. L. Peyton-Jones, M. Herlihy, Composable Memory Transactions, Communications of the ACM 51 (8) (2008) 91–100.

[30] P. W. Trinder, K. Hammond, H.-W. Loidl, S. L. Peyton-Jones, Algorithm + Strategy = Parallelism, Journal of Functional Programming 8 (1998) 23–60.

[31] M. M. T. Chakravarty, G. Keller, R. Lechtchinsky, W. Pfannenstiel, Nepal — Nested Data Parallelism in Haskell, in: Euro-Par '01: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Programming, Springer-Verlag, 2001, pp. 524–534.

[32] M. M. T. Chakravarty, R. Leshchinskiy, S. L. Peyton-Jones, S. Marlow, G. Keller, Data Parallel Haskell: A Status Report, in: DAMP '07: Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming, ACM, 2007, pp. 10–18.

[33] Nvidia, Nvidia CUDA: Compute Unified Device Architecture Programming Guide, Version 2.0, available online at http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf (July 2008).

[34] Various Contributors, The CUDA Zone, hosted and maintained at http://www.nvidia.com/object/cuda_home.html (2008).

[35] P. Harish, P. J. Narayanan, Accelerating Large Graph Algorithms on the GPU Using CUDA, in: HiPC '07: Proceedings of the 14th International Conference on High Performance Computing, 2007, pp. 197–208.

[36] L. Buatois, G. Caumon, B. Lévy, Concurrent Number Cruncher: An Efficient Sparse Linear Solver on the GPU, Lecture Notes in Computer Science 4782 (2007) 358.

[37] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, W.-M. W. Hwu, Optimisation Principles and Application Performance Evaluation of a Multithreaded GPU using CUDA, in: PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, 2008, pp. 73–82.

[38] M. Gschwind, Chip Multiprocessing and the Cell Broadband Engine, in: CF '06: Proceedings of the 3rd Conference on Computing Frontiers, ACM, 2006, pp. 1–8.

[39] IBM, The Cell BE Programming Tutorial, Version 3.0, available online at http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/FC857AE550F7EB83872571A80061F788 (October 2007).

[40] T. Chen, R. Raghavan, J. Dale, E. Iwata, The Cell Broadband Engine Architecture and its First Implementation, IBM developerWorks, available online at http://www-128.ibm.com/developerworks/power/library/pa-cellperf.

[41] A. E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. Gschwind, An Optimising Compiler for the Cell Processor, in: PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, IEEE Computer Society, 2005, pp. 161–172.

[42] J. Gummaraju, J. Coburn, Y. Turner, M. Rosenblum, Streamware: Programming General-Purpose Multi-core Processors using Streams, in: ASPLOS XIII: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, 2008, pp. 297–307.

[43] H. Abelson, G. J. Sussman, J. Sussman, Structure and Interpretation of Computer Programs, 2nd Edition, MIT Press, 1985.

[44] B. Thies, M. Karczmarek, S. Amarasinghe, StreaMIT: A Language for Streaming Applications, in: International Conference on Compiler Construction, 2002, pp. 179–196.

[45] J. Sermulins, W. Thies, R. Rabbah, S. Amarasinghe, Cache Aware Optimisation of Stream Programs, in: Languages, Compilers and Tools for Embedded Systems, 2005, , available online at `http://cag.lcs.mit.edu/commit/papers/05/sermulins-lctes05.pdf`.

[46] S. Agrawal, W. Thies, S. Amarasinghe, Optimising Stream Programs Using Linear State Space Analysis, in: The International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, 2005, , available online at `http://cag.lcs.mit.edu/commit/papers/05/agrawal-cases05.pdf`.

[47] Various Contributors, The StreaMIT Cookbook, available online at `http://cag.csail.mit.edu/streamit/papers/streamit-cookbook.pdf` (September 2006).

[48] Various Contributors, The StreaMIT Language Specification, Version 2.1, available online at `http://cag.csail.mit.edu/streamit/papers/streamit-lang-spec.pdf` (September 2006).

[49] B. R. Gaster, Streams: Emerging from a Shared Memory Model, in: IWOMP: The International Workshop on OpenMP, Springer-Verlag, 2008, pp. 134–145.

[50] W. Gropp, E. Lusk, N. Doss, A. Skjellum, A High-Performance, Portable Implementation of the MPI (Message Passing Interface) Standard, Parallel Computing 22 (6) (1996) 789–828.

[51] W. Gropp, E. Lusk, A High-Performance MPI Implementation on a Shared-Memory Vector Supercomputer, Parallel Computing 22 (11) (1997) 1513–1526.

[52] Various Contributors, The OpenMP Application Program Interface, Version 3.0, available online at `http://www.openmp.org/mp-documents/spec30.pdf` (May 2008).