Part II Project

GPU Accelerating the Ypnos Programming Language

Samuel Pattuzzi

Robinson College

April 20, 2013

Proforma

Name:	**Samuel Pattuzzi**
College:	**Robinson College**
Project Title:	**GPU Accelerating the Ypnos Programming Language**
Examination:	**Part II Project, June 2013**
Word Count:	**12714** [1]
Project Originator:	D.A. Orchard
Supervisor:	D.A. Orchard

Original Aims of the Project

Creating a backend to the Ypnos programming language to support GPU accelerated computation. This will require the implementation of translation from the Ypnos stencil syntax to GPU code. Furthermore, it will require, as a minimum, two language primitives to be implemented: *run* and *reduce*. Both the translation and primitives must be correct and, furthermore, outperform the CPU implementation.

Work Completed

All that has been completed appears in this dissertation.

Special Difficulties

No special difficulties were encountered.

[1] This word count was computed by `texcount`.

Declaration

I, Samuel Pattuzzi of Robinson College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date April 20, 2013

Contents

1. Introduction

In this project I created a compiler for the Ypnos programming language that targets modern GPUs (Graphical Processing Units) allowing for massive speed-ups of programs in this language. The language allows programmers to use a very concise syntax to describe certain types of parallel grid operations. Using this syntax it is now possible to target machines both with and without compatible GPUs.

1.1. Motivation

GPUs have always been excellent exploiters of SIMD parallelism for graphics applications. In recent years, however, the GPU pipeline has become more general than ever. Beyond just providing programmable shaders[1] the platform has been opened up, allowing general programming of GPUs, such as video codec acceleration and even scientific computing. The ubiquity and low cost of this hardware opens up an opportunity for researchers, professionals and hobbyists alike.

The programming model enforced by the APIs of a GPU can be a significant hurdle to programming. The APIs of such general purpose GPU (GPGPU) tends to be low-level but at the same time restricted in ways that may be unfamiliar to the user. Particularly, memory access within a thread is limited to allow effective parallelism. This requires API users to undergo a steep learning curve to achieve such speed-ups.

An alternative to using these lowest level API is higher level computational paradigms which expose the SIMD parallelism of a program. Often these paradigms are not as powerful as the underlying APIs, but allow for very concise programs within a certain field of interest. For example, in video editing we are interested in fast decoding and so a GPU accelerated decoding library would enable us to easily take advantage of GPU speed-ups without writing the low-level code. In the field of scientific computing many operations can be described in terms of matrix operations, so a matrix manipulation library could expose the SIMD parallelism needed.

The approach taken in the Ypnos programming language is to target a type of computation common to graphics algorithms and certain scientific simulation. By taking a simple and easy to learn paradigm and combining it with a declarative API, Ypnos is able to exploit as much SIMD parallelism as needed under the surface. Furthermore, Ypnos is back-end agnostic, meaning its programs can easily be transported from a GPU to a multi-core CPU.

[1]A shader is a small program used in graphics applications for simulating lighting effects on scenes and objects.

Prior to this project, Ypnos supported only CPU execution; this project implements a GPU backend.

1.2. Related work

Ypnos was originally proposed by Orchard et al.[6], as a language embedded within Haskell with a CPU prototype. Haskell supports GPU computation via the Accelerate library [1]. I will briefly introduce the reader to both, giving a grounding for the work to come.

1.2.1. Ypnos

Stencil computations are an idiom used in parallel programming. They comprise a *kernel* (or *stencil*) which is applied to each element of an array[2] of data. The kernel computes a new value for an array location using its old value and the old values of its neighbouring cells. Convolutions are a well-known example of stencil computation. The Gaussian blur is an example of a convolution operation which can be implemented with stencils.

The idiom is particularly useful in the fields of graphical processing and scientific computing, where some typical applications include Gaussian blur, Laplacian of Gaussian (an example of differential equation approximation), Canny edge detection and many other filter based methods. In the scientific domain, they are used in the simulation of physical systems via fluid, stress or heat dynamics.

Ypnos is an *embedded domain specific language* (EDSL) for stencil computations embedded within the Haskell programming language [6, 7]. This allows Ypnos to share much of the syntax and implementation of its host language. Haskell is a particularly good fit for stencil computations as its purity allows the programmer to write parallel programs without worrying about the interaction and sharing of state.

1.2.2. Accelerate

Accelerate [1] is also an EDSL for the Haskell language. It implements parallel array computations on the GPU. Modern GPUs provide vast amounts of SIMD parallelism via general purpose interfaces (GPGPU). Accelerate uses matrix operations, which are easier for a programmer to understand, to expose SIMD parallelism to the GPU.

The primary target GPUs of Accelerate are those which support NVIDIA's CUDA extension for GPGPU programming. The library uses algorithmic skeletons for online CUDA code generation [3]. It provides operations such as `map`, `zip` and `fold` and implements its own stencil convolution function.

[2]In Ypnos, arrays are known as *grids* to abstract from the implementation details

2. Preparation

This chapter will introduce the subject detail which will be required to understand the subsequent chapters of the dissertation. This includes a brief introduction to some of Haskell's more advanced features, the Ypnos programming language and the Accelerate library, all of which were core technologies of my project.

Furthermore, this chapter discusses some of the planning and design choices which laid the foundation for the rest of the project. This includes the analysis of the initial system requirements, choice of tools, libraries, programming languages, as well as software engineering methodology.

2.1. Requirements Analysis

Requirements analysis undertaken in the early stages of this project allowed me to proceed smoothly and identify potential points of failure early. Each major goal of the project was categorized according to priority, difficulty and risk (see Table 2.1).

The *priority* signifies the importance to the completion of the project: essential requirements have been marked as high and optional extensions as low. Other important factors not mentioned in the proposal have also been included and marked as medium priority. The *difficulty* gave an estimate of how hard certain requirements would be to achieve and so help provide a rough estimate of how much time and resource should be dedicated. The *risk* embodies the uncertainty about the implementation details that was present at the start of the project. A high risk requirement is one that could easily take more time than initially foreseen.

The goals were further divided into *functional* and *non-functional* requirements, i.e, things that the system "must do" and things that it "must be". Functional requirements specify what had to be implemented and built during the course of the project. Non-functional requirements specify how the system should perform and be tested.

High risk and high priority requirements needed special attention to prevent the project from falling behind schedule. In scheduling the tasks, I took a risk driven approach: trying to implement the highest risk functional requirements first and test the highest risk non-functional requirements early. At the same time, to ensure compilation correctness I took a test-driven approach to development. I will talk about this in more detail in Section 2.3.2.

From the dependency analysis of Figure 2.1 a clear ordering to the tasks is visible. The following order was adopted for the project: stencil compilation, correctness verification, essential primitives, evaluation, extensions.

Table 2.1.: Categorization of the main project requirements. The *Ext* column marks requirements which are considered extensions.

Requirements	**Priority**	**Difficulty**	**Risk**	**Ext**
Functional				
Stencil compilation – compilation of Ypnos stencil syntax to functions runnable on the GPU.	●●●	●●●	●●	
Primitive support – implementation of the primitive functions on the GPU:				
Run	●●●	●●	●●	
Reduce	●●●	●●	●●	
Iterate	●	●	●	✔
Zip	●	●●●	●	✔
Non-Functional				
Correct translation – ensure that the stencil translation is correct.	●●●	●●	●●●	
Better scaling than CPU – verify that the GPU implementation outperforms the CPU implementation.	●●●	●●	●●●	
Usable API – ensure that the API presented to the programmer is usable.	●●	●●	●●	✔

2.2. Choice of Tools

As with any good software engineering project I made use of many existing tools: both development tools, such as programming languages and source control, as well as libraries for software reuse. In this section I will highlight the choices of programming language, development tools and libraries. For each I will describe the benefits and drawbacks of the tool as well as the reason for which it was chosen.

To familiarise myself with Ypnos, Accelerate and other tools, I started with implementing sample functions in Ypnos and Accelerate. The main sample function was an average stencil (which calculates the mean of its neighbour cell). We will see this function more in the coming chapters.

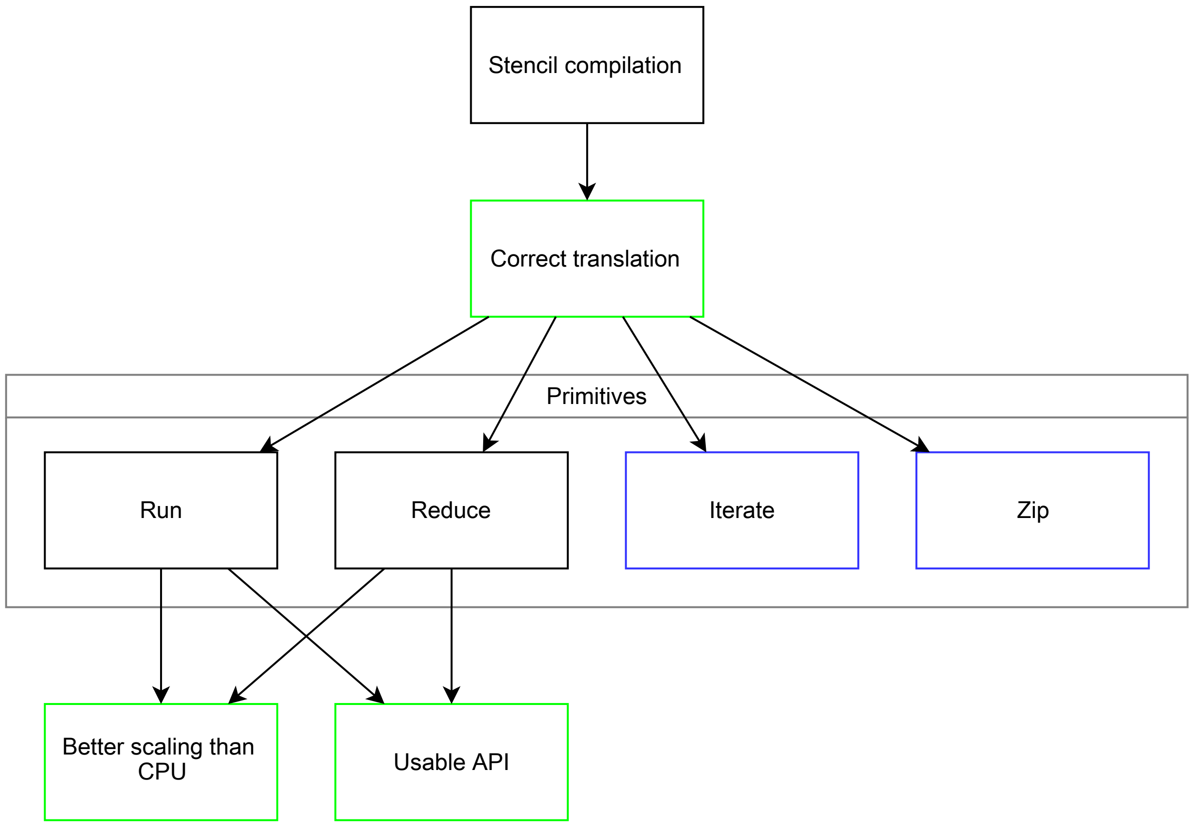

Figure 2.1.: This figure depicts the major task dependencies. The black boxes represent functional requirements, whereas the green boxes depict non-functional ones. Blue boxes are extensions.

2.2.1. Programming Languages

Haskell was the obvious choice of programming language, given that the Ypnos is already developed in it. Having not programmed in Haskell before, I had to become familiar with its more advanced features: *type classes*, *type families* and *data families*.

Haskell has excellent tools for developing compilers: strong typing, pattern matching and strong parsing libraries (Parsec[1]). Furthermore, using the same language as the original implementation allowed for much code reuse.

2.2.2. Development Tools

To aid the fetching of dependencies and the building of various targets I used the *Cabal* build system for Haskell[2]. Cabal features automatic dependencies resolution and fetching as well as project building tools. By writing some toy functions to test my knowledge of the Ypnos

[1]http://hackage.haskell.org/package/parsec
[2]http://www.haskell.org/cabal/

language I was also able to set up a test build system in Cabal that I would later use in the rest of my project. Cabal was chosen as it is the de facto standard for building projects in Haskell. It allowed me to automatically fetch and install all the dependencies for my project as well as manage their versions and compatibility.

Git version control was used extensively throughout this project for logging and backup. Although I was already quite familiar with this system, the project allowed me to make use of some of Git's more advanced features such as *stashing*, *sub-projects* and *branching*. It was chosen primarily for these advanced features as well as tight integration with free hosting services such as Github. This allowed my project to be frequently backed up to the cloud.

2.2.3. Libraries

Accelerate

Accelerate is a Haskell library which provides GPU accelerated array computations. Because the API focuses on generic array operations it is able to support multiple back-ends (though at the moment only one is implemented). It is the only GPU programming library in Haskell which is sufficiently powerful for my needs. Furthermore, it already includes some functions for performing stencil computations over grids.

I chose Accelerate because of the native Haskell support and stencil operations. It allowed me to abstract away from compiling to low-level C code and, instead, concentrate on translating to a more abstract and general API.

CUDA

CUDA is a General Purpose GPU platform for NVidia devices[5]. It is the oldest framework of its kind but has recently been joined by the more cross-platform OpenCL. The reason I chose CUDA over OpenCL was the library support in Haskell. The Accelerate library, on which I was relying, had the most stable support for CUDA (though some experimental support for OpenCL also exists). Furthermore, the GPU made available to me for testing and development supported both CUDA and OpenCL, making it easy for me to use either.

As I do not own a machine with a CUDA enabled graphics card, I was using a remote machine located in the Computer Laboratory. The sample functions allowed me to set up the machine with the drivers and configuration required in order to run the Accelerate library.

2.3. Software Engineering Techniques

This section highlights the software engineering approach taken in the development of this project.

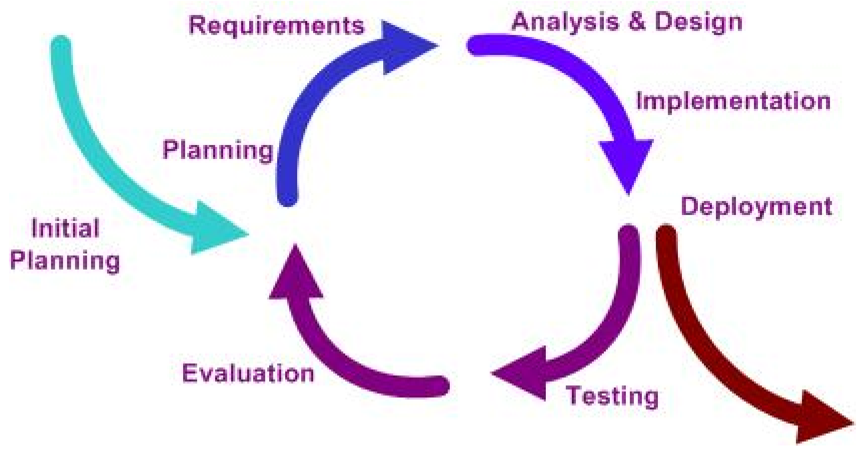

Figure 2.2.: One possible iterative development cycle. In the case of my project the deployment stage happened at the deadline and did not include a roll-out to actual customers.
Image courtesy of Wikipedia

2.3.1. Iterative Development

Given my unfamiliarity with Haskell and GPU programming I followed the *interative* model [2] to integrate exploration with development. The end goal is to satisfy the project requirements. This and the time constraints are the limiting factors of the iterative cycle.

An iteratively developed project starts with an inception phase in which initial requirements and goals are layed out. The project then proceeds through cycles which consist of the following stages (see figure 2.2):

- Gathering requirements
- Design
- Coding
- Testing
- Evaluation

The first and last stages in the cycle merge together as requirements for the next iteration feed off the examination of the last. After each cycle there is an optional deployment phase in which the product is put into the user's environment. This phase is omitted from the project.

Iterative development may be run *ad infinitum* or it may be allowed to finish once certain criteria have been met or resources depleted. For me the limiting resource was the time allocated for this project and the finishing criteria were the goals laid out in the initial proposal, namely the success criteria.

Throughout the project various design approaches were tried and re-evaluated based on issues found in the last iteration. A risk-driven approach was taken where the most difficult parts of the system were attempted first in order to reduce the amount of uncertainty in the project as it progressed.

2.3.2. Test-Driven Development

The correctness of my implementation was a central goal from the beginning of the project. In order to achieve this I took a test-driven approach to development. This meant that while writing the implementation I was simultaneously writing unit tests for the code. The approach allowed me to quickly and effectively find bugs which had not already been found by the Haskell type system.

QuickCheck is Haskell's de facto standard unit testing library. In most unit testing libraries for other platforms the programmer has to provide sets of test data for the library to check against the program. The code for generating this data is left to the programmer. QuickCheck takes a different approach: instead of specifying testing functions, which include the test generation, the programmer specify properties which take the data to be tested as an argument. The generation of this data is done by the library.

QuickCheck is able to generate random testing data for most built-in Haskell data types. For user defined types, the programmer must provide an instance of the class `Arbitrary` which allows QuickCheck to generate random samples for testing.

2.4. Introduction to Ypnos

Ypnos is an existing language with a fully formed syntax and a partial reference implementation. Before I could start coding the translation from Ypnos to GPU, I first had to understand and appreciate the reasoning behind the current choices in the language and implementation.

TODO: describe grids

2.4.1. Syntax

The Ypnos language provides a custom syntax for defining stencil functions as well as a collection of primitive operations for their manipulation and use.

The syntax for a simple two dimensional averaging stencil is shown in Listing 2.1. Compare this with the much more verbose imperative version of the same stencil in Listing 2.2. The `fun` macro is used to provide a special *grid pattern* syntax. The basic syntax of Ypnos can be summarised as follows:

```
avg2D :: Grid (Dim X :* Dim Y) a → a
avg2D = [fun| X*Y:|_  a _|
                  |b @c d|
                  |_  e _| → (a + b + c + d + e )/5|]
out = run avg2D in -- Running the stencil on 'in' to produce 'out'
```

Listing 2.1: A simple mean function. Computes the mean of the neighbourhood of cells.

```
double in [M][N]; // Input array
double out [M][N]; // Output array
for (i = 1; i < M-1; i++){
  for (j = 1; j < N-1; j++){
    out[i][j] =  (in[i-1][j] +
     in[i][j-1] + in[i][j] + in[i][j+1]
               + in[i+1][j]) / 5;
  }
}
```

Listing 2.2: An imperative implementation of the average function.

`X*Y:`	The syntax defines the dimensionality of the stencil and helps to parse the arguments. `X` and `Y` are both dimension variables (as is `Z`). They are combined using the `*` operator.
`|`	The arguments are enclosed within pipe characters. Their arrangement in code is typically indented to reflect their grid shape.
`a,b,_`	Arguments can either be named or "don't care" denoted respectively with either a variable name or an underscore.
`@`	This annotation denotes the variable is the cursor, the central cell whose position is used for the result of the stencil.
`->`	Delimiter that separates the grid pattern and the stencil body. It can be almost any Haskell syntax apart from recursion and function definition.

A generic stencil has a type of `Grid D a -> a` where `D` is the dimensionality and `a` – the element type. The dimensionality would take the form `Dim X :* Dim Y` for a 2D grid.

2.4.2. Primitives

As well as the syntax for stencil functions, Ypnos provides a library of primitive operations. The primitives can be combined to create complex accelerated computations over grids. The main primitive in Ypnos is *run* (see Listing 2.3), which applies the stencil computation to a grid.

```
run :: (Grid D a → b) → Grid D a → Grid D b
```

Listing 2.3: The basic run primitive as defined in the original Ypnos paper[6].

```
reduce :: (a → a → a) → Grid D a → a
```

Listing 2.4: The basic reduction primitive as defined in the original Ypnos paper.

The application is done by moving the stencil cursor over each location in the grid. The arguments of the stencil are taken from positions in the grid relative to the cursor. The value is then computed using the specified computation and put into the same cursor location in a *new* grid.

In some locations near the edge of the grid there may not be enough neighbours to satisfy a stencil. In this case Ypnos provides a special syntax for dealing with these *boundaries*. The implementation of boundaries beyond the scope of this project. However, a brief description of their behaviour will aid the reader's understanding.

For each boundary of the grid, outside of which the stencil may access, a value is computed by a user defined function. The function may use the current location index and values from the grid (accessed via a specially bound variable). A common boundary – the *mirror* boundary – works by providing the closest value inside the grid when an outside access is made. This is the boundary that I have tacitly assumed in my implementation.

Another vital primitive of the Ypnos language is the *reduce* primitive whose purpose is to summarise the contents of a grid in one value (see Listing 2.4). It may be used to compute functions such as the mean, sum or minimum/maximum.

The primitive uses an associative operator (of type `a -> a -> a`) to combine all the elements of the grid to one value. A more general version of this operator also exists (see Listing 2.5), which supports an intermediary type (or partial value).

The *Reducer* data type takes the following parameters:

- a founction reducing an element and a partial value to a partial value,
- a function reducing two partial values,
- a default partial value

```
reduceR :: Reducer a b → Grid D a → a
mkReducer :: exists b. (a → b → b)
                     → (b → b → b)
                     →  b
                     → (b → c)
                     → Reducer a
```

Listing 2.5: The more general version of the reducer allowing for intermediary values.

- and a conversion from a partial value to the final value.

This Reducer is passed into the *reduceR* primitive taking the place of the associative operator in the reduce primitive. Clearly, reduce can be implemented in terms of reduceR and so the latter is the more general.

2.5. Introduction to Accelerate

We have already mentioned Accelerate as one of the implementors of stencil convolution. In fact, Accelerate is an excellent target for intermediary code compilation. While the stencil semantics of Accelerate and Ypnos differ in some respects, the former is powerful enough to represent the latter. Therefore, the project will target the Accelerate language instead of CUDA.

Accelerate uses the Haskell type system to differentiate between arrays on the CPU and GPU. It does this by introducing a type encapsulating GPU operations. There is a further *stratification* of this type into scalar and array values. Scalar computations may be composed into array computations.

2.5.1. GPU Computation

TODO: make show in right place

In order for a process on the CPU to execute a CUDA program it must first send the program and the data to the GPU. When the result is ready it must be copied back into the main memory of the process concerned. I will call these two procedure *copy-on* and *copy-off* respectively (see Figure 2.3).

Accelerate chose to represent this difference in the type system. The `Acc` type denotes an operation on the GPU. For the purposes of Accelerate, the only operations allowed on the GPU are those over arrays. As such, `Array sh e` denotes an array of shape `sh` and element type `e`. `Acc (Array sh e)` denotes the same but in GPU memory and encapsulate an operation. This means that when such an array is evaluated a CUDA program must be executed on the GPU.

Arrays are signalled for use on the GPU via the `use` primitive. They are copied-on, executed and copied-off via the `run`. This primitive is responsible for the run-time compilation and actual data transfer. All other operations build an abstract syntax tree (AST) to be compiled by the run primitive. Together use and run form the constructors and destructors of the `Acc` data type (see Listing 2.6).

2.5.2. Stratified Language

```
use :: Array sh e → Acc (Array sh e)
run :: Acc (Array sh e) → Array sh e
```

Listing 2.6: The basic constructors and destructors for moving arrays too and from the GPU in Accelerate.

```
map :: (Exp a → Exp b) → Acc (Array sh a)
       → Acc (Array sh b)
```

Listing 2.7: The type of the `map` operation as defined by Accelerate.

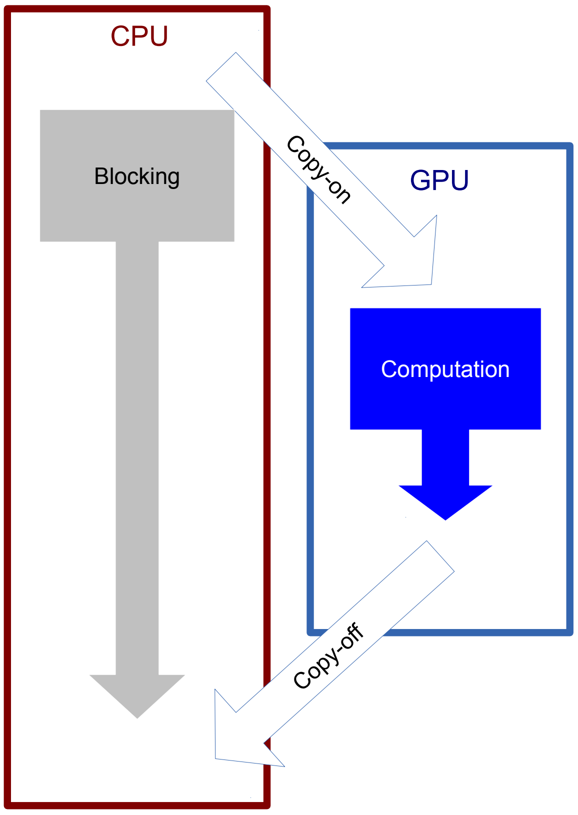

Figure 2.3.: An illustration of copy-on and copy-off times.

The main type of operation in Accelerate is over arrays. However, it is often desirable to build arrays out of multiple scalar values or functions over scalars. A classic example of this is the map function which transforms an entire array by a function over the individual values[3]. For this reason, in addition to the `Acc` type, Accelerate also provides the `Exp` type where the former represents collective operations and the latter represents scalar computations. The two types correspond to the two different types of AST built: one for scalar and the other for array operations.

The map function is depicted in Listing 2.7.

Scalar operations do not support any type of iteration or recursion in order to prevent divergent operation at run-time. However, most other Haskell code is allowed. This is achieved by the Haskell class mechanism – Accelerate provides instances of `Exp a` for most common classes.

For example, to support addition, subtraction and other numerical operations, Accelerate provides an instance of the type class `Num`. This means that operations can be typed as shown in Listing 2.8.

[3]In fact, the map function is conceptually similar to stencil application. The difference being that stencils also take into account the neighbourhood of a cell to compute the next value.

```
(+) :: Exp a → Exp a → Exp a
1 + 2 + 3 :: Exp Integer
```

Listing 2.8: The type of addition overloaded by Accelerate.

```
stencil :: Stencil sh a sten ⇒
           (sten → Exp b) →
           Boundary a →
           Acc (Array sh a) →
           Acc (Array sh b)

instance Stencil DIM2 a ((Exp a, Exp a, Exp a)
                        ,(Exp a, Exp a, Exp a)
                        ,(Exp a, Exp a, Exp a))
```

Listing 2.9: The type of the stencil application function in Accelerate. I have also included an example instance of the `Stencil` type class. Many others are also possible.

2.5.3. Stencil Support

Whilst map for an array applies a scalar function to every element in an array, Accelerate provides support for stencil computations via the `stencil` function (see Listing 2.9).

TODO: compare run and map? Comonads?

The first parameter is a function which represent the stencil. We see that `sten`, the type of the stencil, takes the form of a tuple grid of `Exp a` element type. This allows Accelerate to use Haskell's function syntax to define stencils.

The second parameter is the type of boundary. In Accelerate, the types of boundary allowed are fixed as opposed to Ypnos boundaries which can be fully specified. One of the types allowed is `Mirror` which deals with an out of bounds access by picking the nearest coordinate from within the array.

With these two parameters we have defined an operation which performs the stencil convolution.

2.6. Summary

In this section we have seen an analysis of the project's requirements which allowed me to prioritise the work for the project. Based on the requirements a choice of tools and libraries was made. Throughout the project an iterative approach to development was chosen to meet as many of the requirements as possible in the time given.

At the beginning of the project, time was spent on familiarisation with the tools and libraries as well as the Ypnos language itself. Complex parts of the Haskell language were investigated and understood (type classes and families). The Accelerate library, central to the project, was investigated and “toy” programs were implemented in both Ypnos and Accelerate.

3. Implementation

The work of the implementation can be roughly split into two large chunks: the compilation of stencils and the implementation of the primitives. In both cases it was not immediately obvious which approach would be best and so I took the iterative approach (described in Section 2.3.1) and prototyped a number of different possible solutions. In this chapter I will highlight the major implementation approaches taken in both cases as well as an example usage of the system.

3.1. Stencil Compilation

Compilation of stencils was a central task in this project. The abstract Ypnos syntax allows much flexibility in the underlying implementation. Ypnos achieves this via Haskell's quasiquoting mechanism[] for compiling custom syntax to Haskell AST. Accelerate's implementation has overridden much of the Haskell operators required for this translation stage, so the bulk of the effort went into producing the functions that contained the computation. These functions take the form of Listing 3.1

The arguments are formed as tuples of tuples. The rest of the stencil is normal Haskell code. However, the return type, Exp a, insures that all the operations actually use Accelerate's overridden methods to build an AST. The AST is then translated at run-time into CUDA code.

Haskell's quasiquoting mechanism is a language feature which allows the library author to provide custom syntax for domain specific languages. As such, it is a perfect fit for Ypnos, which would like to hide the underlying implementation from the user. The programmer

```
avg :: Exp a ⇒ Stencil3x3 a → Exp a
avg (( _, a, _ )
    ,( b, c, d )
    ,( _, e, _ )) = (a + b + c + d + e) / 5

type Stencil3x3 = ((Exp a, Exp a, Exp a)
                  ,(Exp a, Exp a, Exp a)
                  ,(Exp a, Exp a, Exp a))
```

Listing 3.1: The “average” stencil defined using Accelerate's syntax.

```
data Expr  =  IntExpr Integer
           |  BinopExpr (Integer → Integer → Integer) Expr Expr

e1 = BinopExpr (+) (IntExpr 1) (IntExpr 3)
e2 = [expr| 1 + 3 |]
```

Listing 3.2: A simple calculator defined using an AST (Expr) and using a quasiquoter for abbreviated syntax. The definition of expr is omitted.

must provide a parser object (refered to as a quasiquoter) and a syntax for applying it within normal Haskell code. The essential function of a quasiquoter is to provide an abbreviation for entering the AST manually.

Take, for example, the situation in which we want to write an embedded language to act as a calculator. We have the AST for our simple calculator in Listing 3.2.

We see that the quasiquoter `expr` allows us to abbreviate the expression `e1` to the more obvious form of `e2`.

Clearly, if we were to swap out the quasiquoter this would be an effective way of producing multiple programs from the same syntax. This is what Ypnos achieves in its stencil syntax. The aim is to be able to change the quasiquoter and fully change the underlying implementation without any other modifications.

We could sensibly do the translation from Ypnos to Accelerate stencils in one of two ways: we (a) use Haskell's type system to mask the difference between the two types of stencil computation or (b) we use run-time conversion to mask the difference between the implementations and maintain the resemblance of the types. I explored each of these approaches in the course of the project and the benefits and drawbacks of both are presented in the next section.

3.1.1. Type System Approach

As we saw in the previous section, the types of the Ypnos CPU stencil and the Accelerate library's stencil differ wildly. Let's take a closer look at the precise differences between them in the types of our stencil `avg`[1] given in Listing 3.3

In the GPU case we see that the type (once expanded) is tuples of tuples of `Exp a`. This allows Accelerate to make use of the built-in Haskell syntax for functions. This is of little use as Ypnos defines its own syntax. On the other hand, in the CPU case we see that arguments take the form of a grid, which is exactly the same type as the grids it operates on.

This is no accident as Ypnos grid type is a comonad (see Appendix ??), the theoretic dual of the of the monad. This restrains the type of the run operation to be of the form given in

[1]For the sake of simplicity I have excluded the type constraints relating to boundaries as these are very long and complicated.

```
avgCPU :: Array a ⇒
          Grid (Dim X :∗ Dim Y) a → a
avgGPU :: Floating (Exp a) ⇒
          Stencil3x3 a → Exp a
```

Listing 3.3: The average function implemented on both the CPU and GPU. Notice how the types differ. The expression `Dim X :* Dim Y` denotes a 2D grid.

```
cobind :: (D a → b) → D a → D b
```

Listing 3.4: The definition of cobind. Let `D` be a grid of a certain dimension and `a` and `b` be the types of that grid.

Listing 3.4

The type system approach (or compile-time approach) means that we translate the Ypnos stencil syntax directly to a function with Accelerate type (e.g. `avgGPU`). We mask the differences in types using either data families or type families. The details, advantages and disadvantages of the two approaches will be discussed further in Section 3.2.

Unfortunately, by translating directly to the Accelerate stencil type we lose the comonadic nature of the type. This is a shame, because this type is both informative to the programmer, yet flexible enough that by changing the instance of `D` we change the implementation.

The advantage of this method (as we will see more in detail when we discuss the alternative) is that all the translation effort is done at compile-time allowing the running of the stencil to be more efficient.

3.1.2. Centring

Another way in which Accelerate and Ypnos stencils differ is that the former assumes that the cursor is centred whereas the later allows the user to specify the centre. This can be translated by padding the stencil given to Accelerate such that the cursor is centred.

This is, perhaps, best illustrated by example. Say that we have the a one dimensional stencil with the cursor at an off-centre location (denoted by `c`) – Figure 3.1.

Now we must determine the padding such that the cursor is centred. This is given by the following two equations TODO: check equations, can a ever be bigger than b?:

$$pad_{start} = max\{a, b-a-1\} - a$$

$$pad_{end} = max\{a, b-a-1\} - b + a + 1$$

This means that after centring we get Figure 3.2

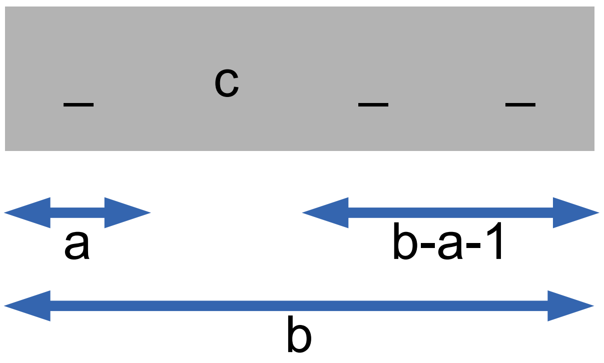

Figure 3.1.: We define the variables a to be the position of the cursor and b to be the length of the stencil.

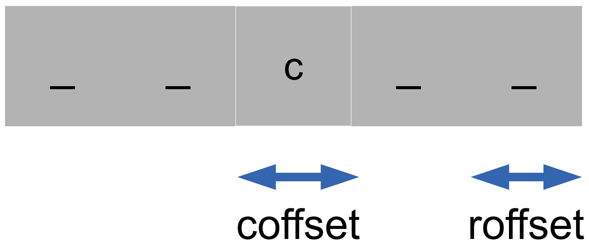

Figure 3.2.: In this diagram *coffset* represents the offset which needs to be applied to the centre, i.e the amount to pad the start of the grid. By contrast, *roffset* represents the amount to pad the end of the grid.

In order to implement the centring I had to consider both the one and two dimensional cases. It would be quite easy to deal with this in two separate cases, except it must be possible to extend the approach to higher dimension eventually. I considered three principle approaches to doing this: using lists as intermediaries; using arrays as intermediaries; or operating on the grid patterns directly via type classes (Haskell's ad-hoc polymorphism mechanism). Before addressing the approaches I will mention the types I was converting (see Listing 3.5).

`GridPattern` is the type in the Ypnos AST corresponding to the parsed pattern of arguments. We see that it takes both a 1D and 2D form where the variables (`VarP`) are a list and a list of lists respectively. We may also note that the dimensionality is expressed directly in the constructor and as such is not present in the type.

```
data GridPattern =
    GridPattern1D DimTag [VarP] |
    GridPattern2D DimTag DimTag [[VarP]]
```

Listing 3.5: The data type which stores the grid patterns in Ypnos. Notice that the dimensionality is not exposed in the type.

```
class Eq a where
  (==) :: a → a → Bool

instance Eq Integer where
  x == y =  x ‘integerEq‘ y
```

Listing 3.6: An example type class for equality. Showing the declaration and the instance for integers. Where `integerEq` is the implementation of integer equality on the target machine.

The pattern of arguments in Accelerate is expressed as a tuple in the 1D case and a tuple of tuples in the 2D case. This representation contains no information about which variables are cursors as we discussed in the previous section.

Type Classes

Haskell provides both *parametric* and *ad-hoc* polymorphism. The former is provided by default in function definitions: each function is made to work over the most general type possible. The latter is provided via the mechanism of *type classes.* I required ad-hoc polymorphism in order to operate differently on patterns of different dimensionalities.

A type class is declared in two parts: interface and instance. An interface has a number of type parameters (also called indexes) and function type declarations. It is then possible to declare which types are instances of which class. This is done by providing concrete types for the indexes and concrete definitions for the functions. Listing 3.6 gives an example of a type class declaration and instance.

Type Families

Type families (also known as indexed type families) allow us to apply the same kind of parameterisation as in type class to the types. Formally, type families are type functions from one or more types to a single type. As with type classes we have both a head declaration and instances which define the family. The interface describes the *"kind"*[2] of the family and defines how many type arguments are taken.

Type families come in two flavours: data families and type synonym families. The former allows the data type to be declared differently for different indexes, whereas the latter allows different types to be synonymous. Listing 3.7 gives an example a data family being used to expose the dimensionality of a pattern. This allows for the use of ad-hoc polymorphism later on.

[2]A kind is the type theoretic name for a type for types. $*$ denotes the kind of base types in Haskell.

```
data family GridPatt :: * → * → *
data instance GridPatt (Int) a =
    GridPatt1D Int [a]
data instance GridPatt (Int,Int) a =
    GridPatt2D (Int, Int) [[a]]
```

Listing 3.7: The data family declares two different constructors for 1D and 2D lists. The dimensionality of the list is exposed in the type.

```
class (Ix i, Num i, ElMax i) ⇒ GridIx i where
    data GridPatt :: * → * → *
    size :: GridPatt i a → i

instance GridIx (Int) where
    data GridPatt (Int) a = GridPatt1D Int [a]
    size (GridPatt1D s _) = s
```

Listing 3.8: The data family from listing 3.7 has now been associated with the class `GridIx` to provide the function `size` for various dimensionalities.

Both flavours can be associated with a type class. In this case the index of the type class must form part of the index of the type family. The interface and instance declarations of the type family are bound to the corresponding declarations of the type class. Listing 3.8 gives an example of an associated data family for patterns. The usage of type synonym families will be discussed in detail later.

Intermediate Approaches

The first approach taken involved first, converting from grid patterns into lists, then balancing these lists, and finally converting them into the centred tuples needed for the Accelerate functional representation. In order to do this I would have to define functions for measuring the location of the cursor, and padding the lists before and after. This approach proved difficult as lists did not explicitly incorporate their dimensionality in the type. This made it hard to treat the 1D and 2D cases differently.

The second approach attempted to use existing array code in order avoid writing such functions. The hope was that by converting to arrays, rather than lists, functions for appending and prepending rows and columns would already exist. However, this was not the case and I would have had to write these myself. As such, the intermediary array stage was not the best choice.

```
class (Ix i, Num i, ElMax i) ⇒ GridIx i where
    data GridPatt i :: ∗ → ∗
    addBefore :: i → a → GridPatt i a → GridPatt i a
    addAfter :: i → a → GridPatt i a → GridPatt i a
    find :: (a → Bool) → GridPatt i a → i
    size :: GridPatt i a → i
```

Listing 3.9: The class declaration of `GridIx` showing the main functions defined for the grid manipulation.

```
run :: Comonad g ⇒ (g a → b) → g a → g b
```

Listing 3.10: The comonadic run type. Changing the type of `g` could change the backend used.

Direct Approach

The third and final approach was to operate directly on the lists extracted from the `GridPattern` types. As previously mentioned, to retain dimensionality information in the type system a type class was required. I designed a class `GridIx` (see Listing 3.9) to perform the basic operations – `addBefore`, `addAfter`, `find` and `size` – in a dimension-sensitive way while still being polymorphic.

The associated data type `GridPatt` would take the type of the particular dimensionality of list that is appropriate for a given instance. In the case of the index type `Int` we would get `GridPatt Int a = [a]` and in the case of `(Int, Int)` we get `[[a]]`. This approach allows the algorithms for centring to be described more generally regardless of the number of dimensions actually involved.

All points considered this is the best and most efficient approach in terms of code reuse. This is why I chose to adopt this approach in the centring used for compile-time stencil translation. Next we will look at a different approach to stencil translation all together: the run-time approach.

3.1.3. Run-time Approach

The second approach to the translation of stencils was to keep the types the same (or similar, as we will see) to Ypnos' original implementation. This is alluring as it allows us to both expose more information to the user through the program's type and maintain the theoretic underpinnings of Ypnos – the comonadic structure. In order to achieve this, some run-time type translations had to be done. These have an overhead for the performance of the stencil application but this can be mitigated, as I will discuss at the end of this section.

As already seen, we would like the `run` primitive to take the form given in Listing 3.10

```
run :: Comonad g ⇒ (g a ‘arrow‘ b) → g a → g b
```

Listing 3.11: The type run is generalised to using the `arrow` type.

```
stencil :: Comonad g ⇒ g (Exp a) → Exp b
stencil’ :: Comonad g ⇒ g a ‘arrow‘ b
```

Listing 3.12: Here we see the type the stencil must have in Accelerate (`stencil`) and the type we can generalise to using the `arrow` type (`stencil’`).

We have also seen that Accelerate does not accept stencils of this form (see Section 3.1.1). To solve this we previously broke the comonadicity of the operation but we could attempt to preserve it by introducing an *arrow* data constructor to absorb the differences in type between the notion of a function in Accelerate and Ypnos. This changes the run function to that seen in Listing 3.11.

The data constructor is parametrized on both `g a` and `b`. To build up an instance of `arrow` we must pass in the stencil function to a special constructor. The constructor chosen decides the implementation used.

While previously we had to use different versions of the quasiquoter to produce different stencils at compile-time, we now use the same quasiquoter but convert the function at run-time. We achieve this by taking advantage of Haskell’s polymorphism which allows a function over type `a` to generalise to a function of type `Exp a`. This generalisation in concert with the arrow data constructor allows our stencil functions to have the type in Listing 3.12

Because of the arrow type, `stencil` and `stencil’` can actually have the same type.

However, the type is still not of the desired form. The type of stencil accepted by Accelerate is still not of the form `g (Exp a) -> Exp b`. I achieve this stencil of this form by a conversion function which builds an Accelerate stencil (call it *stencil A*) at run-time using the stencil encapsulated in the arrow data type (call it *stencil B*). Stencil A’s arguments are used to build up a grid of type `g (Exp a)`, then stencil B is used on this grid to produce the result of type `Exp b`.

While this run-time conversion creates an overhead, it also, as we have seen, simplifies the types significantly. However, a technique called deforestation may be used to mitigate this[3]. I decided that such optimisation would be beyond the scope of this project due to time constraints.

[3]Deforestation is also known as “short cut fusion”. It is essentially an optimiser method which merges some function calls into one.

```
run :: (Stencil sh x sten) ⇒
       (sten → Exp y) → Grid d Nil Static x → Grid d Nil Static y
```

Listing 3.13: The type of run required by Accelerate.

3.2. Primitives

The primitives are the second central component of the translation. Without them we could not run our translated stencils on the GPU. Like the stencil translation, the implementation of the primitives took two approaches: the first was to re-implement the primitives in a separate module (a non-unifying approach). In this case, the user would import whichever implementation they required. This approach had some fatal drawbacks – for example, it required the user to change too much of their code between implementations.

This led to the second approach of extracting the functionality of the primitive into a type class (a type class approach). This approach required the use of some complicated type features in order to make the types unify. This lead to a further three possibilities: (a) using a type class parameter for unification, (b) associating a type family and (c) associating a data type. More detail on how these features work can be found in Section ??.

The resultant approach was a hybrid of these. In this section I will detail all the approaches taken and at the end of this section I will discuss the trade-offs which lead to the final approach.

3.2.1. Non Unifying Approach

The initial implementation approach of the run primitive used the compile-time implementation of the stencil function. At the highest level this meant that the function `run` had type given in Listing 3.13

However, we see that the type of `sh` (required by Accelerate) and `d` (required by Ypnos) do not unify directly requiring another constraint to reconcile the two. Furthermore, constraints need to then be added for the types of `x` and `y` to satisfy Accelerates `stencil` function. In the end this type becomes unwieldy – it is not straight forward for the user to replace it in their code.

Similar problems would have plagued the implementation of the `reduce` primitive. However, having seen the first implementation of the `run` primitive I decided that a different approach was necessary so this incarnation of the `reduce` primitive was never implemented.

3.2.2. Introducing Type Classes

In this project I am aiming both to make an accurate and fast translation as well as one which is easy for the programmer to use. Practically, this means that converting between CPU and

GPU implementations of the same program should require minimal code changes. With the previous approach we saw how this did not work for two reasons: (a) the run primitive I implemented was not related (as far as Haskell was concerned) to the original CPU primitive, and (b) the types of the two primitives differed, which could cause compilation to fail if they were swapped.

It would be nice to have one function which behaves differently under certain program conditions. The perfect tool for this job is adhoc polymorphism which is provided in Haskell via type classes (see Section 3.1.2). The result is an implementation of the primitive which changes dependent on a particular type parameter. The obvious parameter in our case is the grid type as this is common to all Ypnos primitives and so can universally (across all primitives) define whether to use a CPU, GPU or another backend.

We have seen this before: in some of the code examples I have used the notation "Comonad g" to refer to a grid which implements the primitives of Ypnos. This was a type class approach. However, we run into the same problems as with stencil translation (see Section 3.1.3), namely the types of stencil required by Accelerate and Ypnos differ.

Type Class Parameter

The first approach to reconciling the stencil types makes use of the fact that Haskell type classes can be parametrized on more than one type. This allows us to extract parts of the type that change to give a unified type. As the reduce primitive was the first to bring about such issues, let's examine how this approach can be applied to it (see Listing 3.14).

In this approach we are able to have instances for `Reducer` for the CPU and GPU based on the grid type yet we also change the types of values accepted by the functions of the Reducer. These values correspond to different types of functions which tells Haskell to use Accelerate's overloaded versions of operators.

We also see that the `RunGrid` type class is treated in a similar manner: the type of grid uniquely determines the type of stencil function required. This is achieved in Haskell using a *functional dependency*. The notation that denotes this in Haskell is `grid -> sten`. We see this a couple of times in the given example.

Unlike the `reduceG` example, Haskell cannot, without help from the programmer, choose a different quasiquoter (as is required with the static approach as seen in Section 3.1.1).

In theory, this approach should work, however, it brings some usability problems. Let's further examine the type of the `reduceG` primitive when applied to `GPUGrids` (see Listing 3.15)

Notice that both the return value and default value have type `Exp`, which is problematic, as *lifting* and *unlifting*[4] is not easy for the user to do and the wrapped value is not particularly useful or meaningful once returned to the user. One approach to changing this would be to introduce dependent type parameters for the functions rather than the values. However, the approach taken next offers greater flexibility.

[4]Lifting is the process of promoting something of type `a` to type `Exp a`. Unlifting is the inverse process.

```
class ReduceGrid grid a b c | grid → a,
                              grid → b,
                              grid → c where
    reduceG :: Reducer a b c→ grid → c

data Reducer a b c where
    Reducer ::   (a → b → b)
              → (b → b → b)
              → b
              → (b → c)
              → Reducer a b c

instance ReduceGrid CPUGrid a b c
instance ReduceGrid GPUGrid (Exp a) (Exp b) (Exp c)

class RunGrid grid sten | grid → sten where
    runG :: sten → grid → grid

instance RunGrid CPUGrid CPUStencil
instance RunGrid GPUGrid GPUStencil
```

Listing 3.14: The `ReduceGrid` type class defined with type parameters for each variable: `a`, `b` and `c`.

Associated Type Families

We already encountered associated type families in Section 3.1.2. Here we will be using type synonym families as opposed to data families. These allow the unification of the different function types required (see Listing 3.16 for an example of type synonyms families unifying different function types).

The ideal type for the `Reducer` in the GPU implementation is given in Listing 3.17

Clearly, this is an improvement to the user as they get a simple return value which they know how to use. By examining this we can deduce that there are actually two types of abstract function involved: 1- and 2-argument functions of Exps. If we implement these as two associated type families we get the behaviour required (see Listing 3.18)

Next I wanted to extend this approach to the run primitive. However, with run we do not simply have a conversion of types, but also conditions on those types (called contexts in Haskell). It is possible to encode contexts in a type family method using a Haskell language extension called *ConstraintKinds*. This allows us to define a type family which has the *kind* of `Constraint` instead of the usual `*` (denoting type as seen in Section 3.1.2). An example of the `RunGrid` class modified in this way is given in Listing 3.19.

As we see, using associated type families is not very nice or general because we are exposing `sten` – a type variable which has no relevance to the CPU implementation. Though it can be

```
Reducer :: (Exp a → Exp b → Exp b)
        → (Exp b → Exp b → Exp b)
        → (Exp b) -- Default value
        → (Exp b → Exp c)
        → Reducer (Exp a) (Exp b) (Exp c)
reduceG :: Reducer (Exp a) (Exp b) (Exp c)
        → GPUGrid
        → Exp c -- Return value
```

Listing 3.15: The type of the reducer once the Accelerate types are applied.

```
type family Fun :: * → * → * → *
type instance Fun CPUGrid a b = (a → b)
type instance Fun GPUGrid a b = (Exp a → Exp b)
```

Listing 3.16: The type synonym family is used as a type function. It is used to work out the element type of a collection. Here the Fun family (representing a one-argument function) can take two forms depending on the compilation target.

safely ignored, it exposes too much of the underlying type difference which we are actually coding for and so does not decouple the two implementations. As we will see in Section 3.2.3, this problem can be mitigated by taking a hybrid approach.

Associated data families

As opposed to specifying the stencil type as an associated type family we may wish to be explicit about the type of stencil function being creating – much like the using a type class parameter. To achieve this we can make use of another Haskell type system extension called associated data families. These work in much the same way as type families, except that rather than binding a particular synonym to a class we bind a type definition – a data type in Haskell parlance.

Listing 3.20 shows the `RunGrid` type class defined using data families. We can see that the data family has replaced both the type and constraint families from Section 3.2.2.

```
Reducer :: (Exp a → Exp b → Exp b)
        → (Exp b → Exp b → Exp b)
        → b
        → (Exp b → Exp c)
reduceG :: Reducer a b c → GPUGrid → c
```

Listing 3.17: The optimal type for the reduce primitive under Accelerate.

```
Reducer :: Fun2 g a b b
        → Fun2 g b b b
        → b
        → Fun1 g b c

class ReduceGrid g where
    type Fun1 g a b
    type Fun2 g a b c
    reduceG :: Reducer g a b c → g → c
```

Listing 3.18: The application of type families to the reduce primitive.

```
class RunGrid g where
    type ConStencil g a b sten :: Constraint
    type Stencil g a b sten :: *
    run :: ConStencil g a b ⇒ (Stencil g a b) → g x → g y

instance RunGrid g where
    type ConStencil g a b sten = (Stencil sh a ~ sten, ShapeOf g ~ sh)
    type Stencil g a b sten = sten → Exp b
```

Listing 3.19: The application of type families to the run primitive.

We are able to do this due to another Haskell type extension called *generalized algebraic data types* (GADTs) which allow us to place arbitrary type constraints on constructors (see Listing 3.21) This allows for much cleaner implementation on our part but requires the programmer to use different data constructors for the different implementations (CPU versus GPU stencil functions). This is manageable when we are only dealing with the different stencil types, however, when we add in the different types of reduction function too, the programmer must make many code changes.

3.2.3. Final implementation

The final implementation made a trade off between the two approaches we have seen already. It combines the type parameter for the `arrow` type[5] in the `RunGrid` class with associated type

[5]This is effectively the same as having used an associated data type, except it requires the data type to be defined outside of class and does not require type system extensions.

```
class RunGrid g where
    data Sten g a b :: *
    runG :: Sten g a b → g a → g b
```

Listing 3.20: RunGrid with associated data family.

```
data Sten (Array sh) a b where
        Sten :: (Shape sh, Stencil sh a sten,
                 Elt a, Elt b) ⇒
                (sten → Exp b)
                → Sten (Array sh) a b
```

Listing 3.21: An example of a stencil data type for the GPU

```
class RunGrid g arrow | arr → g where
    type RunCon g arrow x y :: Constraint
    runG :: RunCon g arrow x y ⇒
            (x ‘arr‘ y)
            → g x → g y

class ReduceGrid g where
    type ConstFun1 g a b :: Constraint
    type ConstFun2 g a b c :: Constraint
    type Fun1 g a b
    type Fun2 g a b c
    reduceG :: Reducer g a c → g a → c
```

Listing 3.22: The final signatures of the `RunGrid` and `ReduceGrid` classes.

families for constraints and generalized functions in the `ReduceGrid` class (see Listing 3.22).

The `RunGrid` class makes use of functional dependencies to ensure that by using an `arrow` constructor the programmer has specified also the grid implmentation to be used. As a knock-on effect this ensures that any subsequent uses of `reduceG` are fixed to the same grid implementation. Using generalized constructors and destructors (Section ??) means that if the programmer is building their grids from lists then the correct implementation's grid will be decided based on the `arrow` type used.

By using a type and not type synonyms for the stencil function I have eliminated the need to expose a type variable in the declaration for type synonyms (as seen in Section 3.2.2). Now this can be neatly encapsulated within a GADT.

3.3. Usage

A description of the system would not be complete without usage examples. The examples in Listing 3.23 are taken directly from the unit tests for the application. They show the usage of the generalized constructors as well as the `run` and `reduce` primitives.

```
-- Take a list of integers and their dimensions and return the sum.
sum :: [Int] → (Int, Int) → Int
sum xs (x,y) =  reduceG (mkReducer (+) (+) 0 id) arr
    where arr = fromList (Z :. x :. y) (cycle xs)

-- Run a floating point stencil of any type
runF sten xs (x, y) = gridData (runG sten xs')
    where xs' = listGrid (Dim X :* Dim Y)
                         (0, 0) (x, y)
                         (cycle xs)
                         mirror

-- The average stencil
avgY = [funGPU| X*Y:|a  b c|
                    |d @e f|
                    |g  h i| →
        (a + b + c + d + e + f + g + h + i)/9|]

-- Run the average function on the CPU
runAvgGPU = runF (GPUArr avgY)

-- Run the average function on the GPU
runAvgCPU = runF (CPUArr avgY)
```

Listing 3.23: Usage of the final system taken from the unit tests.

3.4. Summary

In this chapter we have seen how stencil compilation and the primitives (reduce and run) were implemented. We saw how stencil compilation was attempted in a compile-time and run-time fashion and how centring was implemented. We also saw the primitives implemented in a non-unifying way and then an attempt to unify them through various methods: type class parameters, associated types and data families.

The pros and cons of each approach were discussed and at the end of the chapter I described the final approach chosen. The chapter is rounded off with a brief usage example for the translation, primitives, constructors and destructors.

4. Evaluation

The main aims of this project were to produce a correct translation and speed up over the CPU implementation. In order to test these two goals I have implemented unit tests throughout the course of this project and implemented an evaluation suite of programs. The GPU is a type of co-processor and, as a result, incurs an overhead for copying results to and from its local memory. In evaluating the speed up of using the GPU I have accounted for this.

Further to the performance evaluations, in this section I will also discuss the measures taken to ensure a correct translation. At the end of the chapter I will highlight the usability evaluation conducted using the method of *cognitive dimensions*.

4.1. Performance

Before embarking on the evaluation I postulated that the GPU should provide a speed up over the CPU due to its capacity for parallel computation. Seeing as the stencil computation is highly data parallel, it is a perfect fit for the SIMD parallelism of the GPU. More specifically, I expected that as grid sizes increased the run time of the computation would increase less quickly in the GPU case compared with the CPU case.

4.1.1. Methodology

To measure the run-time I made use of a library called *Criterion* which provides functions for:

- Estimating the cost of a single call to the `clock` function. The function that times the CPU and GPU.
- Estimating the clock resolution.
- Running Haskell functions and timing them discounting the above variations in order to get a sample of data.
- Analysing the sample using *bootstrapping*[] to calculate the mean and confidence interval.

In my experimental setup I am using a confidence interval of 95% and a sample size of 100 and a resample size of 100,000. The result from Criterion is a mean with a confidence interval of 95%. I will use these results to compare the performance of the various functions implemented.

The machine being used for benchmarking was provided by the labs and remotely hosted. The machine specifications are as follows:

- Ubuntu Linux 12.04 64-bit edition
- Quad core Intel Core i5-2400S CPU clocked at 2.50GHz with a 6M cache
- 16GB of core memory
- Nvidia GeForce 9600 GT graphics card featuring the G94 GPU with a 256M framebuffer.

4.1.2. Overhead

In order to show the speed-up, I must first discount the effects of copying to and from the GPU. This was done via an `identity` function implemented in Accelerate. The identity function works by copying the data from the CPU to the GPU, performing no operations on the GPU, then copying the data back. This will allow us to have a ceiling measure of how fast our computations could be without this overhead.

4.1.3. Benchmark suite

The benchmark suite must test the speed-up of both primitives: `run` and `reduce`. I have implemented a set of representative functions for each primitive to test speed across a representative set of calculations. These functions include:

- **The Average Stencil** [ref] that we have seen in the previous sections. This function is representative of convolution style operations which we may wish to perform on the data. It operates over floating point numbers which is a common use case for scientific computing.
- **The Game of Life stencil** makes use of various boolean functions as well as externally declared functions used to count the number of *true* values in a list.
- **The Sum** which constitute one of the most common reduction operations over grids.

4.1.4. Results

Run

The results of the benchmarking showed that the GPU implementation outperforms the CPU for grids of width greater than 30. This is in accordance with the expected outcome, that the GPU is able to perform better modulo copy on/off times.

On a scale of 0-100 (see Figure 4.1), the slowdown of the GPU is barely visible above random variation. However, on a greater scale (Figure 4.2) it is clear that the performance is actually degrading, albeit at a much slower rate than the CPU performance.

The ceiling measurement is the copy on/off time. My implementation, by its very nature, incurs such a cost due to copying the data to and from the GPU. We see that on the smaller

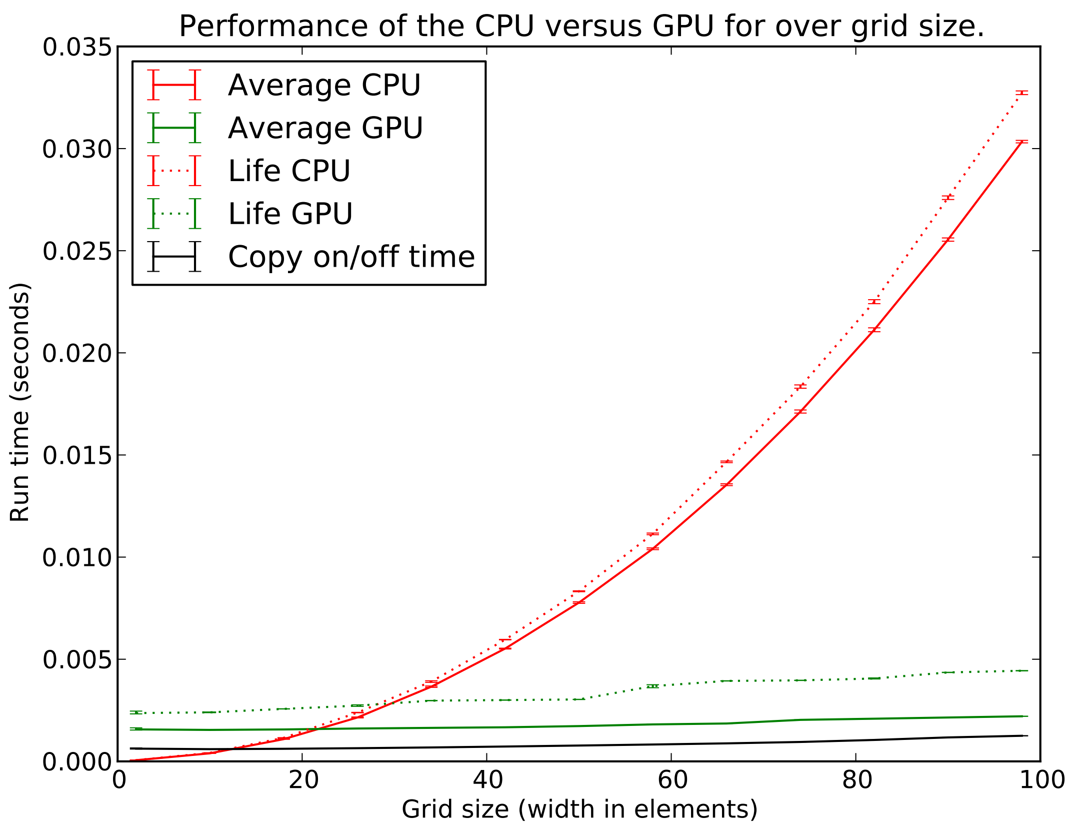

Figure 4.1.: This plot shows the performance of the CPU verse the GPU implementations of the run primitive as it degrades with the grid size increasing. The grids are square in shape and the size given is for one of its dimensions. For comparison, both the average and game of life stencils are depicted. Also shown is the copy-on/off times for the GPU.

scale of analysis the ceiling and actual stencils show little difference in performance signifying that most of the time is spent in copying. On the greater scale we see how the stencil computations actually diverge from the ceiling as GPU computation time starts to become significant. This can be seen more clearly in Figure 4.3 where the copy on/off time has been discounted.

From the graphs it is clear to see that copy on/off time does not account for all the overhead in the GPU computation. I hypothesise that this is due to a copy on/off time for code as well as data (which was not accounted for in my calculations). This is supported by the fact that the Game of Life has a greater overhead at zero than the average function. This is due to the fact that it took more code to implement.

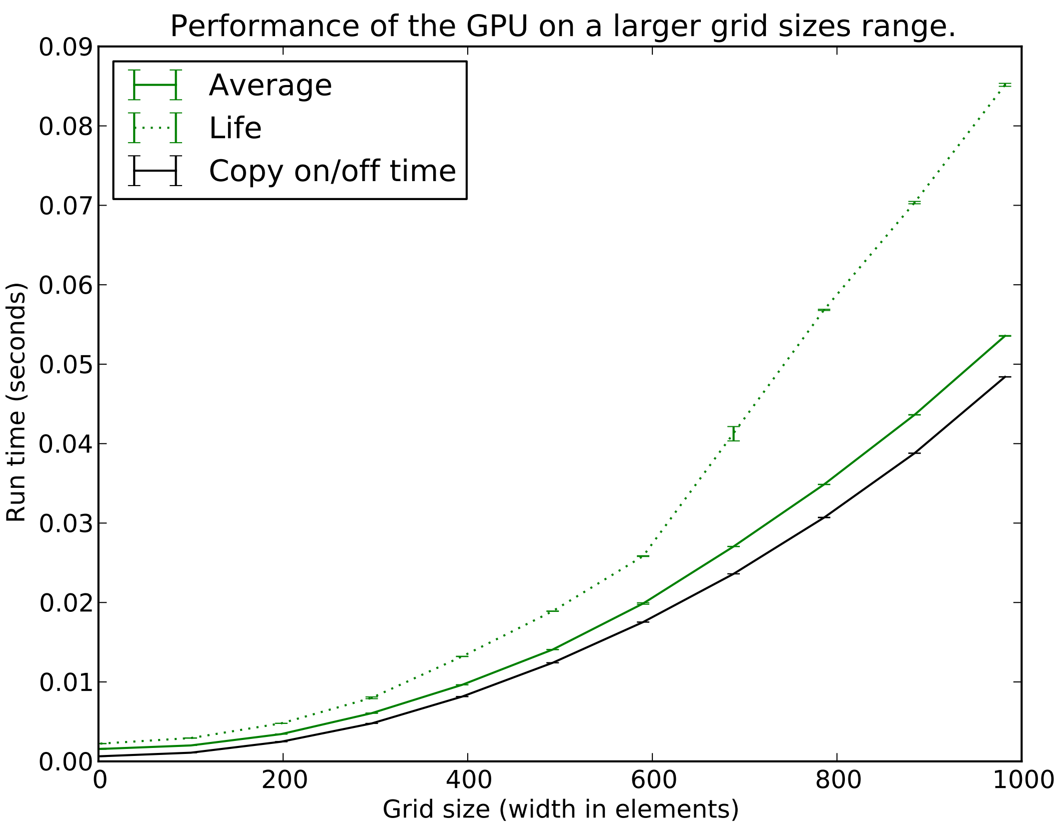

Figure 4.2.: This plot shows the performance of the GPU implementations of the run primitive as it degrades with a larger scale of grid size. Here we can see that both the Game of Life and average function diverge from the ceiling measurement. Game of Life degrades the worst due to function calling overheads in its implementation.

Reduce

As we can see from Figure 4.4, the performance of the reduce primitive on the GPU also exceeds that of the CPU. In this case, however, the cross-over happens latter because the performance of the CPU is already quite acceptable. This is consequence of the fact that the reduction calculations are much less intensive than those of the stencil function or the Game of Life.

In this case we see a cross-over happening at around the 80 by 80 grid size.

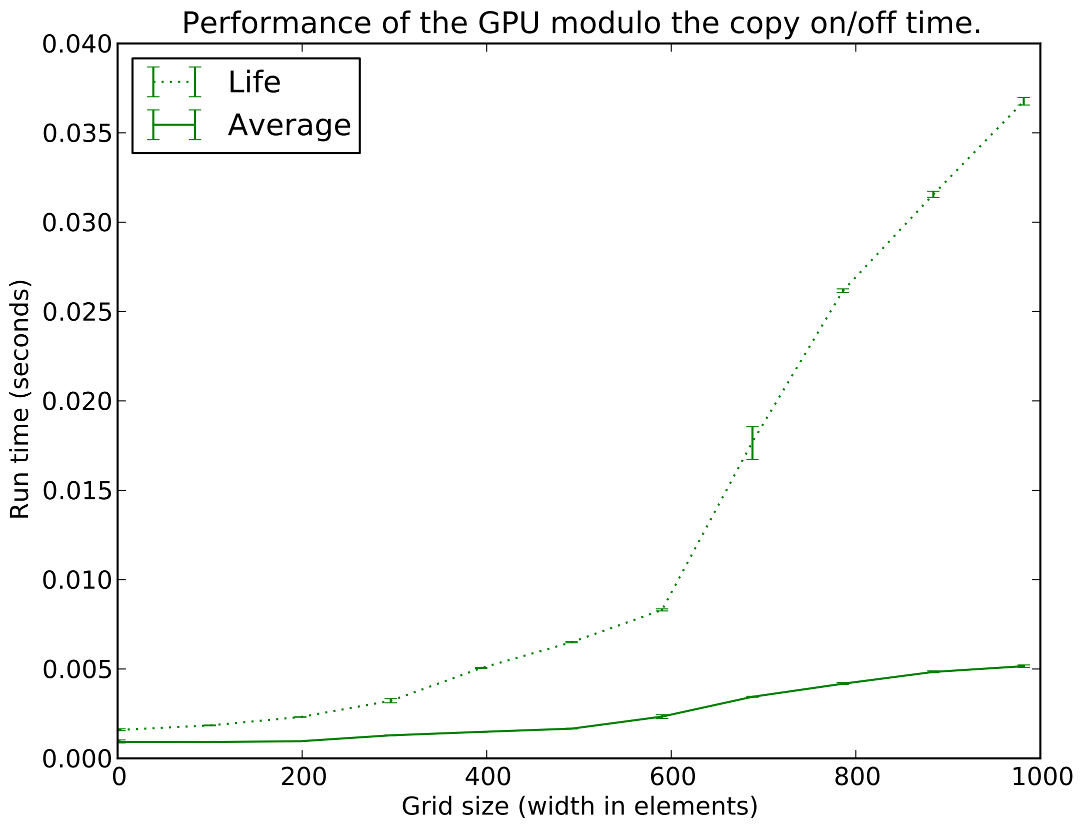

Figure 4.3.: Here we see the performance of the GPU on the same 0-1000 scale. Now the copy on/off time has been discounted by subtraction.

4.1.5. Deducing a model

A future goal of this project might be to automatically decide when a certain workload is best suited to the CPU or GPU. The experiments in this section have already given us an insight into this: once grid size goes beyond a certain size, it is time to move onto the GPU. We also saw that the complexity of the program plays a part in both the overhead and the rate of degradation. However, at small values of grid size the overhead is most significant and so the degradation can be ignored.

In order to deduce a model which could be used for switching between CPU and GPU we must determine (a) an approximation function for the CPU's rate of degradation and (b) the overhead particular to our program on the GPU. We will ignore (b) seeing as the evaluation was not sufficient to determine this. (a) can be found by fitting a quadratic to the curve. This has been done in Figure 4.5 and we can see that the quadratic polynomial fits almost perfectly. The coefficients of this polynomial are given by 4.1

$$y = ax^2 + bx + c \tag{4.1}$$

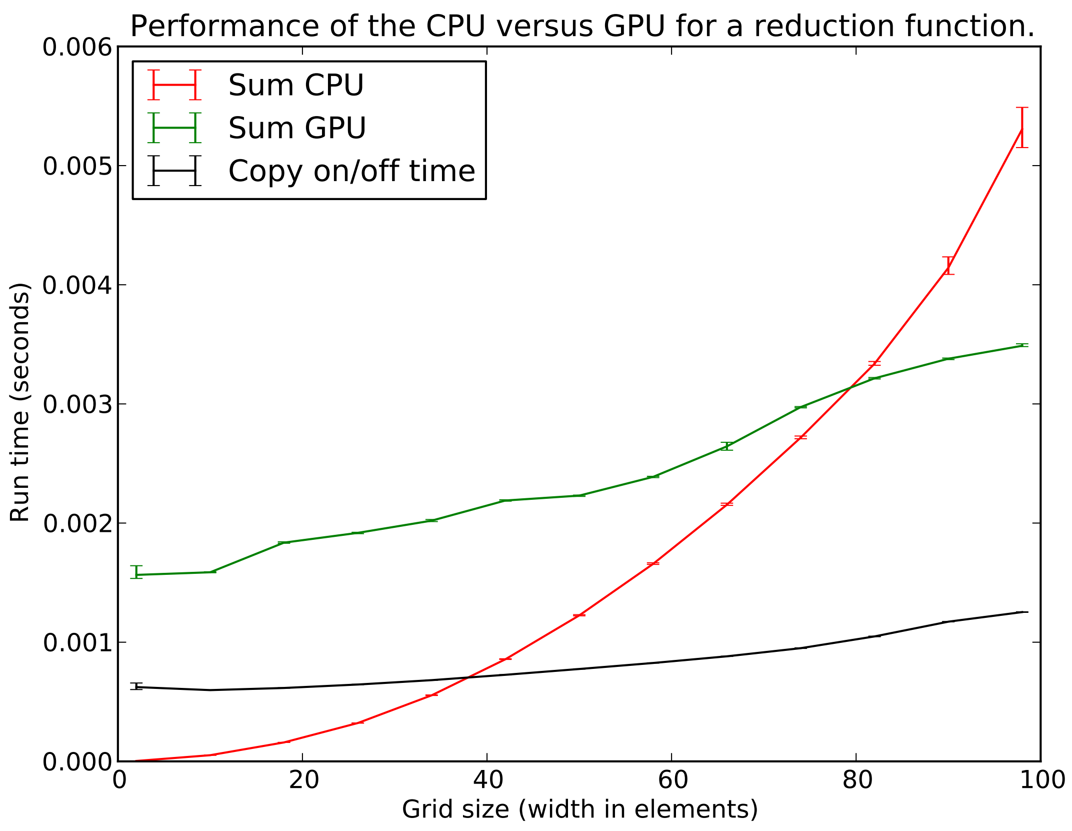

Figure 4.4.: This plot shows the performance of the CPU versus GPU versions of a reduction function. The reduction function given in this case in the sum function.

Where the coefficients are:

$$\begin{aligned} a &= 3.23223432 \times 10^{-6} \\ b &= -9.23861372 \times 10^{-6} \\ c &= 1.56150424 \times 10^{-4} \end{aligned}$$

Knowing the particular overhead of our function (as we have assumed) we are now able to use equation 4.1 to calculate the value of x for which we should switch to using the GPU.

4.2. Correctness

A central goal of the project was to produce a correct translation from Ypnos to Accelerate. Already, by choosing a type safe language such as Haskell I vastly reduced the number of run-time errors possible due to programming errors. To catch the rest I made use of *unit*

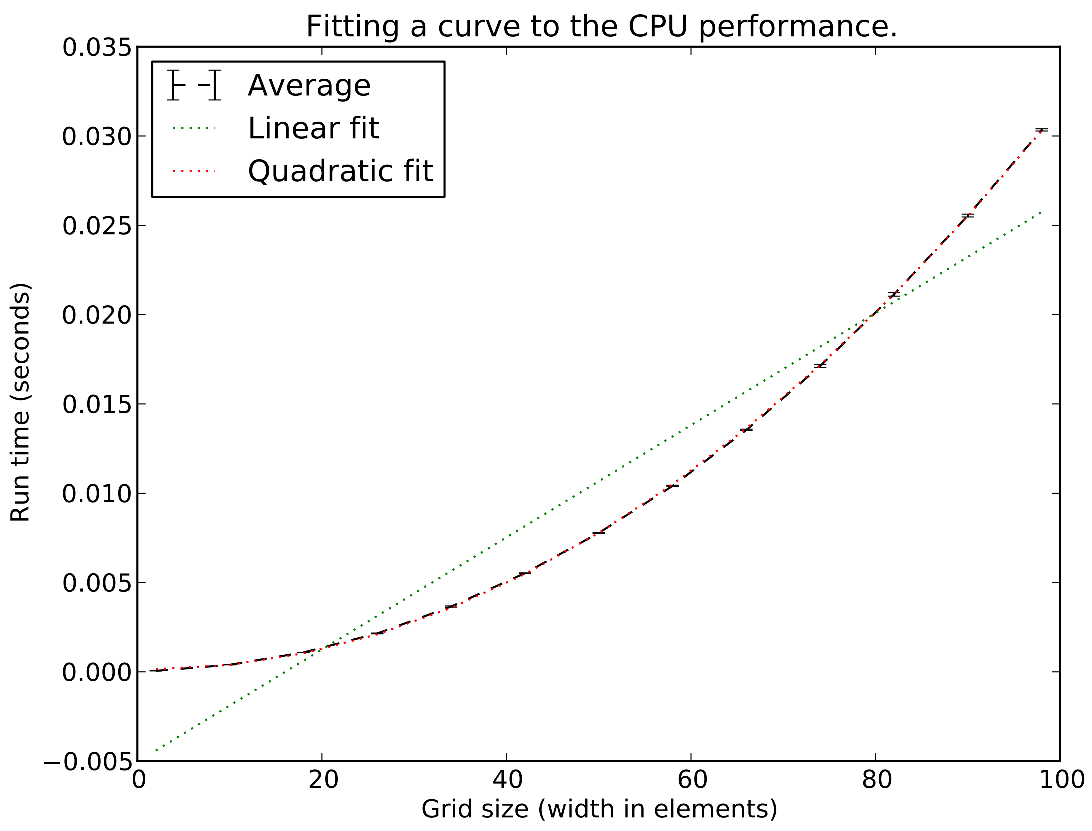

Figure 4.5.: Least square fitting of a linear and quadratic curve to the performance data for the CPU implementation of the average function.

testing and *Test-Driven Development*. Clearly, unit testing can only provide an assurance of correctness and not a guarantee. However, I decided that a formal proof (which could give these guarantees) was beyond the scope of this project. In writing these tests I have assumed that the original CPU implementation was correct and could be compared against as a gold standard.

The testing framework used works slightly differently to other unit testing frameworks. In a standard framework the user provides test cases which incorporate both the test data (sometimes generated) and assertions. In Haskell's *QuickCheck* we only provide axioms about our functions and let the framework provide the data based on the type.

Typically, QuickCheck will generate hundreds of test samples to verify a particular axiom. This provides a better assurance than ordinary unit testing as, via the random process, QuickCheck often comes up with corner cases the programmer may not have devised themselves.

The following sections of my project required unit testing:

- The centring algorithm for grid patterns, as this contains a large part of the translations complexity.

- The `run` primitive.
- The `reduce` primitive.

The approach taken to testing the grid patterns was to ensure that the transformation:

- Starts with a grid that has certain properties (a precondition): regular size, positive size, has a cursor.
- Maintains the regularity of size: the length of each row is the same.
- Centres the cursor, given the original grid had a cursor.
- Both roffset and coffset are always positive on such a grid (see Section 3.1.2 for an explanation of these two values).

The assumption was that grid patterns given to the transformation procedures would be correct to begin with. As such, to improve the amount of test data generated, I enforced these properties at the generation level. This is safe as the grid patterns are generated through the CPU translation which I am assuming to be correct.

To test the primitives I used a standard testing approach of comparing against an existing correct implementation. Both implementations are fed the same data and their results should come out the same. For the `reduce` primitive I compare against Haskell's built-in reduce function as I can safely assume this to be correct. For the `run` primitive I originally indented to test against the Ypnos CPU implementation as I was assuming this to be correct. However, in running my tests I uncovered a bug in the implementation of boundaries which made me consider other options.

Given that I could not trust the results of the CPU implementation I tested the GPU primitive against a hand coded stencil in Accelerate. This was not ideal as it used essentially the same code an the run implementation but this still provided some assurance. Once the bug had been fixed in the CPU implementation, I was able to test against this as well.

The run primitive is tested by running the average function on a randomly generated grid. The grid is passed to the GPU, CPU and Accelerate implementations of `avg`. The resulting grid is then compared between the two and any difference counts as a failure.

The same procedure is used for the reduce primitive. We use a one-dimensional grid for this case as the built-in Haskell function we are comparing against is one-dimensional. The resulting reduced values are compared and a failure is registered if they should differ.

For the large part of the project I have been coding tests and implementation in parallel (also known as Test-Driven Development or TDD). This allowed me to catch errors early on and fix them immediately. TDD allowed for much faster debugging as it provides confidence in the functionality of certain parts of code. This meant that when I encountered bugs I was able to pin-point their origin often without the use of a debugger.

4.3. Usability

While not mentioned in my original proposal, the usability to the programmer is another non-functional requirement. I decided that performing a full usability study would be unnecessary as this was a secondary requirement. Instead, I have chosen to evaluate the usability using the method of *Cognitive Dimensions* [4], to compare the various approaches already discussed.

Cognitive Dimensions of notations (CD) provide a light weight vocabulary for discussing various factors of programming language design. As Ypnos is essentially a programming language (albeit, one embedded in Haskell) it makes sense to use this technique. It works by specifying a number of properties of a notation (*dimensions*, a complete list can be found in Green[4]) which must, in general, be traded off against one another. For this reason it is important to understand the representative tasks and the user that will be performing them. Then design decisions in the language can be compared and evaluated using the dimensions relative to the tasks.

4.3.1. System = Language + Environment

It is important to note that CD relates to a whole system, not just the language. We define the system to be the combination of programming language and the programming environments. For example, programming over the phone versus programming in a visual editor. For the purposes of discussing only the language changes that I have introduced I will fix the environment and assume that it has the following features:

- Screen-based text editor (e.g. Vim, Emacs or TextMate)
- Search and replace functionality (including regular expressions)

4.3.2. Methodology

I used the following procedure in evaluating the changes to Ypnos using CD:

- Identify the relevant users of my system and sketch out a basic user profile.
- Select the relevant task of these users on the part of the language I implemented.
- Highlight which cognitive dimensions are most important to the selected tasks.
- Show a comparison of the various approaches to this implementation.
- Conclude which approach was taken and why.

4.3.3. User profiles

I have decided that given the applications to scientific computing and graphics, the two main types of users would be scientists simulating physical systems and graphics programmers developing graphics algorithms. I have included two user stories for our two representative users:

> “Kiaran is a physical scientist who is writing a simulation of a fluid dynamics system. He has a little Haskell experience already but has mostly used other languages such as Matlab and Fortran. He chose Ypnos/Haskell because he knew it would allow him to easily switch between a CPU implementation on his machine and a GPU implementation on the simulation machine he is using.”

> “Noemi is writing a graphics transformation for a photo editing package. The photos her users edit are typically very large but she still would like to provide real-time performance with her algorithms. Noemi has a GPU in her computer, so she will be writing for this to ensure that her performance is good. However, she also wants her system to degrade well on machines that do not have a compatible GPU. She already has experience in Haskell and is familiar with more complex features and extensions such as type and data families. She has picked Ypnos/Haskell because of its syntax and the ease of degrading.”

We can see that there are many tasks that these users would want to perform with our system: coding up a filter into a stencil (Noemi), writing a complex reduction to determine the state of the system (Kiaran), debugging to find out why they get the wrong values (both). However, I will be ignoring all tasks that involve parts of the system which I did not implement. This leaves us with one central task for the two use cases: converting between GPU and CPU.

The cognitive dimensions relevant to this task are:

Low repetition viscosity	to allow the user to easily change the implementation without changing too many points in code.
Little to no imposed lookahead	allowing the programmer to use one implementation without having to think about later switching.
Consistency	the programs syntax or usage does not change from CPU to GPU.
Terseness	the syntax to specify the implementation does not get in the way of coding the stencils.
Closeness of mapping	the model presented to the user through the API should map well to the user’s mental model for these types of operation.

The various approaches to provide an API to the programmer were discussed in the implementation section (3). They essentially boiled down to the following three approaches: choosing the different implementation based on importing, using type classes with associated data families and using type classes with associated type families. For the sake of comparison I will also include the approach of the programmer re-coding their implementation in Accelerate for the GPU.

I will summarise the results of the evaluation here, for a full discussion see Table B.1 in the Appendix. The re-coding approach was best in the closeness of mapping, abstraction gradient and hidden dependency categories, in all others it was worst. The non-unifying approach was best in the imposed lookahead category and worst in the hidden dependency and closeness of mapping categories. Data families was best in the imposed lookahead and hidden dependencies categories and worst in the closeness of mapping category. Finally, type families was best in repetition viscosity, imposed lookahead, consistency and terseness but worst in hidden dependencies and closeness of mapping.

4.3.4. Conclusions

As we now can see, the best approach for our users is that of associated type families with the data constructor for the stencil function. This approach is best in the viscosity, imposed lookahead, consistency and terseness dimensions. However, for this it has compromised in hidden dependencies, abstraction and closeness of mapping.

The *hidden dependency* problems are mitigated by the Haskell compiler which warns and throws errors when there is a conflict in these dependencies. While a little increase in hidden dependencies is necessary to reduce viscosity, there could be room for improvement here by making the types more consistent. This would help us remove the dependencies due to the changing types and constraints.

Given that our example users are fairly advanced, the increase in *abstraction* should not be a problem, however, we should be aware of this extra difficulty to learning the language. We imagine that Kiaran would not have a problem learning about type families but it is still a learning curve.

The *closeness of mapping* is an issue that is not inherent in the implementation, but rather an artifact of it. With more time on this project I would try to re-introduce the comonadic types to the type family approach. This could require using a lower level implementation rather than using Accelerate. For this reason getting a closer mapping was beyond the scope of this project.

4.4. Summary

In this chapter we saw how the performance of my GPU implementation surpasses that of the CPU for larger grid sizes. This holds for both the run and reduce primitives. I demonstrated the testing approach taken and discussed how this can provide some assurance as to the correctness of the translation. Finally, I conducted usability evaluation using the method of cognitive dimensions.

In summary, this chapter demonstrated all the original goals of the project have been fulfilled (see original proposal in appendix A). Furthermore, I have demonstrated the usability of my API, a goal which was not originally stated.

5. Conclusion

In the dissertation so far I have taken the reader through the preparation, implementation and evaluation of the Ypnos acceleration.

This project has caused me to pick up a large number of new and complicated technologies from a previously unexplored field of computer science. Having only experienced the ML programming from part Ia, diving into Haskell's intricate type system has been an enlightening experience which has taught me much about my personal software engineering approach and the type systems of other languages.

5.1. Accomplishments

In this project I have managed to accomplish all the goals laid out by the original proposal: a correct translation and run primitives which out-perform their CPU counterparts. Furthermore, I conducted usability evaluation which was not originally conceived as a requirement.

Aside from the mechanical goals of the project I have also deepened my own knowledge of computer science and software engineering. I can now add Haskell to the list of languages I am intimately familiar with. My experience with Haskell has given me a better understanding of the type theoretic decisions that underpin other more common languages (such as the various types of polymorphism used in OOP). Furthermore, my software engineering approaches have been improved by often needing to re-factor code and design complex systems such as the centring algorithm.

I consider the project a success in that it both completed the goals required, providing a step forward for the Ypnos programming language, and helped me learn much about Haskell and GPU computation.

5.2. Lessons Learnt

In completing this project I have learnt the importance of having a strong plan to stick to. I realised as I went further into this project that I had not allocated enough time to researching and learning a new programming ecosystem. I had assumed that picking up a new language would be as easy as the previous OOP languages I have taught myself (such as Python). However, I had severely underestimated the time it would take to get familiar with the complex type theory underlying the Haskell compiler.

Furthermore, if I were to repeat the project I would set out better procedures for making notes and documenting the project as it progressed. The time required to compile all the information for the dissertation and understand the different approaches taken in the implementation was significantly more than I had previously anticipated. In future work I will endeavour to keep a more complete diary with frequent reviews of all the work accomplished so far and how it all fits together.

5.3. Future Work

While all the goals of this project have been achieved, the path of accelerating Ypnos on the GPU is by no means complete. A number of extensions from the original proposal still have not been attempted, which include:

- The `iterate` primitive which would allow more efficient execution of pipelined operations.
- The `zip` primitive which would allow implementation of more complicated combinations. This would allow programs such as *Canny edge detection* to be implemented.

Further to the original extensions, during the course of the project more possible avenues for exploration were discovered. Further exploration may be possible in:

- Attempting to expose the comonadic nature of operations in the type given to the programmer.
- Deducing a full model for the GPU computation overhead.
- Automatic selection of the backend dependent on static program analysis.

References

[1] Manuel M.T. Chakravarty et al. “Accelerating Haskell array codes with multicore GPUs”. In: *Proceedings of the sixth workshop on Declarative aspects of multicore programming*. 2011. URL: http://www.cse.unsw.edu.au/~chak/papers/acc-cuda.pdf.

[2] Alistair Cockburn. “Using both incremental and iterative development”. In: *CrossTalk* (May 2008). URL: http://www.crosstalkonline.org/storage/issue-archives/2008/200805/200805-Cockburn.pdf.

[3] Murray I. Cole. *Algorithmic skeletons: structured management of parallel computation.* Pitman London, 1989. URL: http://homepages.inf.ed.ac.uk/mic/Pubs/skeletonbook.pdf.

[4] T.R.G. Green. *An Introduction to the Cognitive Dimensions Framework.* Nov. 1996. URL: http://homepage.ntlworld.com/greenery/workStuff/Papers/introCogDims/index.html.

[5] CUDA Nvidia. *Programming guide.* 2008.

[6] Dominic A. Orchard, Max Bolingbroke, and Alan Mycroft. “Ypnos: declarative, parallel structured grid programming”. In: *Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming.* 2010. URL: http://www.cl.cam.ac.uk/~dao29/publ/ypnos-damp10.pdf.

[7] Dominic Orchard and Alan Mycroft. “Efficient and Correct Stencil Computation via Pattern Matching and Static Typing”. In: *Proceedings IFIP Working Conference on Domain-Specific Languages.* 2011. URL: http://arxiv.org/pdf/1109.0777v1.pdf.

A. Original Proposal

Introduction, The Problem To Be Addressed

In recent years, Moore's law has begun to plateau. As a result the hardware industry has increasingly been turning to MIMD and SIMD architectures as a solution. Some of the highest performing SIMD implementations today are provided by GPUs and can be harnessed via GPGPU languages such as CUDA and OpenCL. However, taking advantage of GPGPUs is hard as it requires knowledge of the low-level concepts of these languages. Also, CUDA and OpenCL are not portable between different hardware and methods of concurrency.

Structured grids are a common computational pattern for scientific parallel computation. They allow us to specify *stencils* or *kernels* which are local computations that compute a new value from neighboring cells. Stencils can be applied to every index such as to make them a computation over the whole array. Many algorithms can be described in this way including the Gaussian blur filter, edge detection as well as scientific applications such as fluid dynamics. The pattern is also highly parallelizable by splitting the array into smaller chunks and running the stencil on each separately.

Ypnos [6] is an Embedded Domain Specific Language in the Haskell programming language that is capable of describing and running these kernels over an array. It defines a syntax and various primitives for structured grid computation.

A kernel is described using a modified Haskell function syntax. The following is a kernel used to compute the local average of an array and has a blurring effect. The arguments are written as an array and the central point is annotated with `@`. It also denotes the location where the kernels return value is written in the new array.

```
ave2D :: Grid (X * Y) Double -> Double
ave2D (X * Y): | _  t _ | = (t+l+c+r+b)/5.0
               | l @c r |
               | _  b _ |
```

Ypnos has a number of primitive array operations. A *run* primitive is used to turn the stencil into an array operation in the manner specified above. An *iterate* primitive simply recursively applies run. A *reduce* primitive allows us to summarise the data so that we can calculate means, sums, minimums or maximums. This result is often used as a stopping condition for iteration.

zip and *unzip* primitives allow us to pair and unpair the values of two arrays respectively. This is useful when dealing with multiple inter-related quantities as is the case in a physical system with force, acceleration and velocity.

Due to the declarative syntax and its purity, the order of application of the stencils is not important. The author of an Ypnos program does not need to worry about the method of concurrency underpinning their program. This allows the implementers of Ypnos to use whichever method is most appropriate for the given work-load and platform. As many computers these days are equipped with very advanced GPUs capable of general purpose computation, the Ypnos language should be able to use these to accelerate its calculations.

Starting Point

- I have had experience of functional programming from the course as well as having completed some Haskell tutorials.
- I have experience of building a compiler from Part IB supervision work.
- During the course of an 11 week internship I learnt to plan, implement, document and test my own project.
- I have already read the Ypnos paper and am familiar with the constructs of the language as well as its primitives.
- At present Ypnos has been partially implemented in a single threaded fashion on the CPU. This implementation can be taken as both the starting point and the benchmark for the new implementation.
- A Haskell ESDL already exists for compiling array computations to CUDA code. The library, 'accelerate' [1], takes an AST and produces code to run on the GPU. I will use this library as a back-end to avoid writing a compiler to CUDA code directly.

Resources Required

- For this project I shall require my own laptop computer that runs Arch Linux for the bulk of development work.
- Backup will be to github, the SRCF and/or the MCS. Should my computer fail I will be able to use the MCS computers for the project.
- I require an Nvidia GPU in order to test the code produced. This will be provided by Dominic Orchard and I will have access to the machine via SSH for testing purposes.

Work to be done

The project breaks down into the following sub-projects:

1. Write unit tests that can be used to check the correctness of my implementation. The tests will cover micro aspects of the programming language such as: constants, unary application, binary application, indexing, conditionals, local let-binding, etc.
2. The implementation of the main compilation. This involves writing a compilation pass that can take the Ypnos AST and produce a correspondent 'accelerate' AST.
3. The implementation of the basic *run* and *reduce* primitives as well as any combinators for constructing and deconstructing arrays from raw data. The combinators may be written in the 'accelerate' language directly.
4. The testing of the implementation to check that it works correctly and is faster than the original modulo data copying. This will need a test bench to be constructed that includes various well know stencil computations. The test bench will include the following programs as a minimum: the Game of Life, Gaussian blur, Canny edge detection, and (if zip/unzip are implemented) the difference of Gaussians.

Success Criterion for the Main Result

The project will be a success if:

1. It can compile Ypnos code and implements the *run* primitive on the GPU.
2. It implements the *reduce* primitive on the GPU.
3. It scales better than the current single threaded implementation on large work-loads. That is to say that when the input size should correlate with the speed-up observed. However, this must take into account the time required to copy data on and off the GPU.
4. The translation is correct, preserving the semantics of stencil computations on the GPU

Possible Extensions

If the main aim for this project is achieved then I shall try to implement further primitives of the Ypnos language. The programmer will then be able to take advantage of the speed gains of the GPU pipeline. I will attempt them in this order:

1. The "iterate" primitive. This will eliminate the need to copy data between the CPU and GPU at each step.
2. The "zip" primitive.

I may also attempt to enhance the compiler to decide at run-time whether to use the GPU or CPU dependant on the size of computation required. I will investigate modelling of the GPU execution times in relation to data size and stencil complexity.

Timetable: Workplan and Milestones to be achieved.

Planned starting date is the 19th of October when the proposal is accepted.

- **19th of October – 4th of November** Learn to write and read Haskell code. Write some programs in Ypnos. Try to understand the existing code base. Start writing a unit testing suite.
- **5th of November – 18th of November** Finish writing the unit testing suite by including Gaussian blur and Game of Life programs. Get familiar with the 'accelerate' ESDL by reading the paper and writing some toy programs.
- **19th of November – 30th of November** Start implementation of the compiler from the Ypnos AST to the 'accelerate' AST. Most basic operations should translate correctly by this point.
- **1st of December – 16th of December (Christmas)** Finish the compiler and begin work on implementing the run and reduce primitives.
- **17th of December – 6th of January (Christmas)** Course revision and Holiday.
- **7th of January – 20th of January (Christmas)** Finish the primitives if necessary. Write the progress report. Start work on the basic test bench.
- **21th of January – 3rd of February** Finalise the main test bench and run experiments. Analyse the performance and scalability of the approach. Make improvements to the code as necessary to achieve the main aim of the project.
- **4th of February – 17th of February** If there is time then the main extensions may be implemented at this point.
- **18th of February – 3rd of March** Write the main chapters of the dissertation.
- **4th of March – 15th of March** Elaborate on the existing tests bench and run final experiments. Complete most of dissertation in draft form.
- **16th of March – 22nd of April (Easter)** Finish dissertation. The rest of the vacation is set aside for course revision.
- **23th of April – 5th of May** A draft must be completed and sent to my supervisor and director of studies by the **23rd of April**. This will be followed by proof reading and then an early submission.

B. Full Cognitive Dimension Analysis

CD	Accelerate	Non-unifying	Data families	Type families (only stencil data type)
Repetition viscosity	**Worst** Clearly here we have a very high viscosity: each function must be re-written in terms of new syntax and run in different ways.	We have improved the viscosity significantly. The user must only implement their stencils in one language but they must still change all the imports and correct type errors.	Data families worsen the viscosity over the import method as we must now change all the data constructors as opposed to the imports. In real code there will be more of these than import locations.	**Best** Here we have the least repetition viscosity of all the approaches. We now only need to change the quasi quoter to change the whole implementation.
Imposed lookahead	**Worst** The user must know ahead of time that they will be writing in two languages to be sure to minimize duplication of code and structure their program correctly.	**Best** There is practially no imposed lookahead as we can simply swap out the implementation by importing from different places.	**Best** We do not have imposed lookahead as we can easily swap the constructors.	**Best** There is little imposed lookahead in theory, though some operations are currently not supported in the GPU implementation. (TODO: link to more) Some types may not be supported easily in both implementations so this should be considered too.

Consistency	**Worst** The syntaxes are different and so are fairly inconsistent. There are, however, some similarities between the two in their stencil representation.	Consistency is improved as the syntax is now uniform but types are not uniform.	The syntax and usage is the same except for changing the constructors which is inconsistent.	**Best** We have eliminated the inconsistency in usage of the data families. Now the approach is almost entirely consistent except for the types.
Terseness	**Worst** A lot of code is written by the user to cope with the two different implementations.	Changing requires a fair bit of code to be changed as we may be importing many different things from the Ypnos libraries and all these things must be changed.	We require a lot of code to express the swap from CPU to GPU.	**Best** The syntax for switching is minimally terse.
Hidden dependencies	**Best** No hidden dependencies. It is very clear and explicit what is going on.	**Worst** Many hidden dependencies are introduced as the types of the different imported functions do not necessarily match. This can cause failures in many different places on changing the import.	**Best** Here the dependencies introduced in the type system by the import approach have been made explicit by data constructors.	**Worst** More hidden dependencies are introduced. Types change without the users knowing due to different type and constraint families. This might affect some programs.

Abstraction gradient	**Best** The abstraction is at its most basic level – choosing between Ypnos or Accelerate. The user must get to grips with these abstractions as a minimum.	The abstraction level is fairly low as it uses only simple Haskell constructs.	The user must now be familiar with the idea of an associated data family and GADT which are quite advanced Haskell type features.	The abstraction here is perhaps highest of all as it uses the most advanced type features.
Closeness of mapping	**Best** This way we preserve the comonadic nature of the operations in the type so that it is obvious to the user what is going on.	**Worst** The comonadicity is lost due to having to change the types to suit the accelerate implementation.	**Worst** The comonadicity is lost also.	**Worst** The comonadicity is lost.

Table B.1.: Comparison of the different API using cognitive dimensions.