GPU Accelerating the Ypnos Programming Language

Samuel Pattuzzi

March 2013

Contents

1 Introduction **4**
1.1 Motivation 4
1.2 Related work 4
1.2.1 Ypnos 4
1.2.2 Accelerate 6
1.3 Summary 8

2 Preparation **9**
2.1 Requirements Analysis 9
2.2 Implementation Approach 9
2.3 Choice of Tools 9
2.3.1 Programming Languages 9
2.3.2 Development Tools 9
2.3.3 Libraries 9
2.4 Software Engineering Techniques 10
2.4.1 Iterative Development 10
2.4.2 Test Driven Development 10
2.5 Summary 11

3 Implementation **12**
3.1 Stencil Compilation 12
3.1.1 Type System Approach 13
3.1.2 Centring 14
3.1.3 Run-time Approach 16
3.2 Primitives 17
3.2.1 Non unifying approach 17
3.2.2 Type classes 17
3.2.3 Type class parameter 18
3.2.4 Associated type families 19
3.2.5 Associated data families 20
3.2.6 Final implementation 20
3.3 Usage 20
3.3.1 Constructors and destructors 20

4 Evaluation **21**
4.1 Performance . . . 21
4.1.1 Methodology . . . 21
4.1.2 Overhead . . . 22
4.1.3 Benchmark suite . . . 22
4.1.4 Deducing a model . . . 22
4.2 Correctness . . . 22
4.3 Usability . . . 24
4.3.1 System = Language + Environment . . . 24
4.3.2 Methodology . . . 25
4.3.3 User profiles . . . 25
4.3.4 Conclusions . . . 29
4.4 Summary . . . 29

5 Conclusion **30**
5.1 Accomplishments . . . 30
5.2 Lessons Learnt . . . 30
5.3 Future Work . . . 30

1 Introduction

In this project I created a compiler for the Ypnos programming language that targets modern GPUs 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 they are now able to target machines both with and without compatible GPUs.

1.1 Motivation

TODO: talk about the need for parallelisation on GPUs

1.2 Related work

1.2.1 Ypnos

Stencil computations are an idiom used in parallel programming. They work by defining a *kernel* (or *stencil*) which is applied to each element of an array of data (which I will call a *grid*). The kernel computes a new value for the array location using the old value and the old values of its neighboring cells. In summary the opperation behaves similarly to convolution.

The idiom is particularly useful in the fields of graphical processing where some typical applications include: Gaussian blur, Laplacian of Gaussian, Canny edge detection and many other filter based methods. Stencils can also be used to computer approximations of differential systems of equations. As such they are useful in the simulation of physical systems via fluid, stress or heat dynamics.

Ypnos is an *embedded domain specific language*(EDSL) for stencil computations. Rather than build a language from scratch, it is embedded within the Haskell programming language. 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.

Syntax

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

The syntax for a simple averaging stencil would look as follows:

```
avg2D = [fun| X*Y:|_ a _| |b @c d| |_ e _| -> (a + b + c + d + e )/5|]
```

Ypnos uses the *quasiquoting mechanism* in Haskell to provide its syntax. It essentially allows the programmer to provide a parser and enter the custom syntax in brackets `[parser| ... my custom syntaxt ...|]`. In the case of the Ypnos stencil functions the parser is called `fun`.

The next thing to be noted about the stencil is the syntax `X*Y:`. `X` and `Y` are both dimension variables (as is `Z`). They can be combined using the `*` operator. The syntax defines the dimensionality of the stencil and helps us parse the arguments.

The arguments are enclosed within pipe characters (`|`). Their arrangement in code is typically indented to reflect their grid shape. Arguments can either be named or “don’t care” possitions denoted with either a name or an underscore respectively. The arguments annotated with `@` is the cursor, the central cell whose position is used for the result of the stencil.

The final section of the syntax comes after `->` and is the computation. This can be most Haskell syntax though recursion and function definition is not possible.

Primitives

As well as the syntax for stencil functions, Ypnos provides a library of primative operations. The primatives allow the programmer to combine the stencils with grids to produce the computations they want. The main primative in Ypnos is the *run* primative which applies the stencil computation to a grid.

```
run :: (Grid D a -> b) -> Grid D a -> Grid D b
```

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.

pppTODO: Say something about the argument being Grid D a.

In some locations near the edge of the grid their may not be enough neighbors to satisfy a stencil. In this case Ypnos provides a special syntax for dealing with these *boundaries*. I have considered the implementation of boundaries beyond the scope of this project however I will include a brief description of their behaviour.

For each boundary of the grid outside of which the run primative may need to access we define a behaviour. We may compute a value for these locations using: the current location index, values from the grid (accessed via a specially bound variable). A common boundary is the *mirror* boundary which 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 primative of the Ypnos language is the *reduce* primative whose purpose is to summarise the contents of a grid in one value. It may be used to compute functions such as the mean, sum or minimum/maximum.

```
reduce :: (a -> a -> a) -> Grid D a -> a
```

The primative 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 which support an intermediary type.

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

The *Reducer* data type takes parameters:

- a function reducing an element and a partial value to a partial value,
- a function reducing two partial values,
- a default partial value
- and a conversion from a partial value to the final value.

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

1.2.2 Accelerate

Modern GPUs provide vast amounts of SIMD parallelism via general purpose interfaces (GPGPU). For most users these are hard to use as the interfaces are very low level and require the user to put a lot of effort into writing correct parallel programs.

Matrices are a mathematical model which embodies SIMD parallelism. They are operators over large amounts of data. It is possible to express many parallel calculations and operations as matrix equations.

Accelerate is an EDSL for the Haskell language which implements parallel array computations on the GPU. The primary target GPUs are those which support NVIDIA's CUDA extension for GPGPU programming. Accelerate uses algorithm skeletons for online CUDA code generation.

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

Accelerate contains built in support for stencil computations and so is an excellent intermediary target for this project. While the stencil semantics of Accelerate and Ypnos differ in some respects, the former is powerful enough to represent the later. As such, this project will concern itself not with the generation of CUDA code directly but through the Accelerate language.

GPU computation

Haskell execution happens on the CPU and in main memory whereas GPU execution happens in parallel in a separate memory. 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 processes *copy-on* and *copy-off* respectively.

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` and `Acc (Array sh e)` denotes the same but in GPU memory and may also encapsulate an operation.

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 AST to be compiled by the run primitive. Together use and run form the constructors and destructors of the `Acc` data type.

```
use :: Array sh e -> Acc (Array sh e)
run :: Acc (Array sh e) -> Array sh e
```

A stratified language

While the main type of operation in Accelerate is over arrays. However, we often want to compose arrays out of multiple scalar values or functions over scalars. A classic example of this is the map function to transform an entire array by a function over the individual values[1]. For this reason, in addition to the `Acc` type, Accelerate also provides the `Exp` type where the former represents collective operations and the later represents scalar computations.

The map function would then looks like this:

[1]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.

```
map :: (Exp a -> Exp b) -> Acc (Array sh a) -> Acc (Array sh b)
```

Scalar operations do not support any type of iteration or recursion in order to prevent the asymmetric operation 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, in the support of addition, subtraction and other numerical operations, Accelerate provides an instance of the type class `Num`. This means that operations can be typed as follows:

```
(+) :: Num a => Exp a -> Exp a -> Exp a
1 + 2 + 3 :: Exp Integer
```

Stencil support

I have already mentioned that function over scalars can be applied over a whole grid, the map function being an example of this. Accelerate provides support for stencil computations via the `stencil` function.

```
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))
```

The first parameter is a function which represent the stencil. We see that `sten`, the stencil pattern, 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 deal with an out of bounds access by picking the nearest coordinate from within the array.

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

1.3 Summary

TODO

2 Preparation

TODO: Summarise section

TODO: Make more discursive, story like.

TODO: Why am I doing all this?

2.1 Requirements Analysis

TODO

2.2 Implementation Approach

TODO: Things like type classes, families etc

2.3 Choice of Tools

TODO

2.3.1 Programming Languages

2.3.2 Development Tools

2.3.3 Libraries

Learning

In order to get familiar with the syntax of these two libraries I decided to implement some sample functions in both, the main one being the average function we have already seen. With this I was able to familiarise myself with the embedded languages and prepare for the task of implementation.

Furthermore, this allowed me to become familiar with the other tools I would be using in this project, namely:

- Haskell, the programming language. Having not used this before I had to get familiar with the syntax as well as some of the more advanced type system features and extensions such as: *type classes*, *type families* and *data families*.
- Cabal, a build system for Haskell with automatic dependencies resolution and fetching. The toy functions allowed me to test setting up a build system which I would later use for the rest of the project.
- Git, the version control system. I was already quite familiar with this system before this project but it allowed me to make use of some of Git's more advanced features such as *stashing*, *sub-projects* and *branching*.
- CUDA, the drivers and library. 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.

TODO: Requirements analysis plus other software engineering techniques

2.4 Software Engineering Techniques

TODO

2.4.1 Iterative Development

2.4.2 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 that 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 defacto 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, we specify properties which take the data to be tested as an argument. We then leave the generation of this data up to 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.

2.5 Summary

3 Implementation

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 Quasi Quoting 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 following form:

```
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))
```

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

Haskell's quasiquoting mechanism is a compiler option 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 underling implementation from the user. It works by providing 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 following AST for our simple calculator:

```
data Expr  =  IntExpr Integer
           |  BinopExpr (Integer -> Integer -> Integer) Expr Expr
```

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

We see that the quasiquoter expr allows us to appreviate 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 semblance of the types. I explored each of these approaches in the course of the project and I would like to now describe the benefits and drawbacks of both.

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 a closer look at the precise differences between them in the types of our stencil avg[1]:

```
avgCPU :: (IArray UArray a, Fractional a) =>
          Grid (Dim X :* Dim Y) b dyn a -> a
avgGPU :: Floating (Exp a) =>
          Stencil3x3 a -> Exp a
```

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 to us as we have our own syntax already. 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, the theoretic dual of the of the monad. This restrains the type of the run operation to be of the form:

```
cobind :: (D a -> b) -> D a -> D b
```

Where we let D be a grid of a certain dimension and a and b be the types of that grid.

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

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.

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 program this. This can be translated by padding the stencil handed off to Accelerate such that the cursor has been centred.

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

```
          b
*---------------*
| _ | c | _ | _ |
*---*   *-------*
  a       b-a-1
```

We define the variables a to be the position of the cursor and b to be the length of the stencil. Now we must find how much we must pad at the beginning and end of the stencil to centre the cursor. This is given by the following two equations:

$$pad_{beginning} = max\{a, b\} - a$$

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

This means that after centring we get the following:

```
         coffset roffset
          *---*   *---*
| _ | _ | c | _ | _ |
```

3.1.2 Centring

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 deal with this in two separate cases except that it would be nice to extend the approach to higher dimension eventually. I three principle approaches to doing this: using lists as an intermediary; using arrays as intermediaries; and operating on the grid patterns directly via type classes. Before addressing the approaches I will mention the types we were converting.

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

`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.

The pattern of arguments in Accelerate is a expressed as 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.

Intermediate Approaches

The first approach taken involved converting first from grid patterns into lists, balancing these lists then 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 their 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. This made the point of the intermediary stage of arrays altogether pointless.

Direct Approach

The third and final approach was to operate directly on the lists extracted from the `GridPattern` types. As I already mentioned: the problem with working with lists is that the dimensionality is lost in the type. To retain this information in the type system I designed a class `GridIx` to perform the basic operations - `addBefore`, `addAfter`, `find` and `size` - in a dimension sensitive way while still being polymorphic.

```
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
```

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 at a general level without being concerned about the number of dimensions actually involved.

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 programs type and maintain the theoretic underpinnings of Ypnos: the comonadic structure. In order to achieve this we need to do some run-time type translations. These have an overhead for (TODO: do they?) the performance of the stencil application but I will discuss at the end of this section what approaches can be taken to mitigate this overhead.

As already seen, we would like the run primitive to take the form:

```
run :: Comonad g => (g a -> b) -> g a -> g b
```

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

run :: Comonad g => (g a `arr` b) -> g a -> g b

The data constructor is paramatrized on both g a and b. To build up an instance of arr 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:

stencil :: Comonad g => g (Exp a) -> Exp b stencil' :: Comonad g => g a `arr` b

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

However, we are only half-way there: the type of stencil accepted by Accelerate is still not of the form g (Exp a) -> Exp b. I achieve this stencil 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 TODO: mention deforestation.

3.2 Primitives

TODO: rewrite

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 primary approaches. The first was to re-implement the primitives in a separate module. In this case the user would import whichever implementation they required. This approach had some fatal draw backs in that 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. This approach required the use of some complicated type features in order to make the types unify. However, the final result is much more usable.

3.2.1 Non unifying approach

The initial of run was linked to the compile time implementation of the stencil function. At the highest level this meant that the function `run` had the following type:

```
run :: (Stencil sh x sten) =>
       (sten -> Exp y) -> Grid d Nil Static x -> Grid d Nil Static y
```

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. Further, constraints need to then be added for the types of `x` and `y` to satisfy Accelerates `stencil` function. In the end the end this type becomes unwieldy meaning that 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 need so this incarnation of the `reduce` primitive never saw the light of day.

3.2.2 Type classes

In this project I am aiming both to make and 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 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.

What would be nice is 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. The result is an implementation of the primitive which changes dependant 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 other 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 is the same thing. However, we run into the same problems as with stencil translation (see the section on run-time stencil translation).

3.2.3 Type class parameter

The first approach to solving this problem 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 (semi-)unified type. As the reduce primitive was the first to bring about such issues lets examine how this approach can be applied to it.

```
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
```

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 funtcions of the reducer. These values correspond to different types of functions which is what tells Haskell to use the Accelerate 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 [2]. Unlike the reduceG example, Haskell cannot, without help from the programmer, choose a different quasiquoter (as is required with the static approach). With the run-time approach we may be able to do better but we will see this later. (TODO: ensure this forward reference holds)

So in theory this approach should work however, we encounter problems with the useability of this approach. Let's further examine the the type of the reduceG primitive when applied to GPUGrids:

```
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
```

Notice that both the return value and default value have type Exp which is problematic as *lifting* and *unlifting* is not easy for the user to do and the wrapped value is not particularly useful or meaningful. One approach to changing this would be to introduce dependant type parameters for the functions rather than the values and this could work however, I actually took the following approach.

3.2.4 Associated type families

We already encountered associated type families in the section on [direct translation]{#direct}. Here we will use these same type families to achieve the different function types we are looking for.

The ideal type for the Reducer in the GPU implentation would be:

```
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
```

Clearly this is an improvement to the user as they get a simple value they know what to do with. By examining this we can deduce that there are actually two of abstract function involved: 1 arguments functions of Exps and 2 argument. If we implement these we as two associated type families we get the behaviour we want:

[2]The notation that denotes this in Haskell is grid -> sten. We see this a couple of times in the given example.

```
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
```

Next I wanted to extend this approach to run. However, with the run primitive 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 has the *kind* of `Constraint` instead of the usual ∗ (denoting type). Here is an example of the `RunGrid` class modified in this way:

```
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
```

NEXT: exposed sten. Not very nice or general

3.2.5 Associated data families

3.2.6 Final implementation

TODO: run implementation

TODO: reduce implementation

3.3 Usage

TODO: Example of usage and replacement

3.3.1 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 testing through out 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 to account for this.

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. Which does the timing of the CPU.
- 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*[TODO: link to paper] 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 machines specifications are as follows:

- Ubuntu Linux 12.04 32-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 this I must first discount the effect of copying to and from the GPU. This was done via an `id` function implemented in Accelerate. The effect of which is to copy the data from the CPU to the GPU, perform no operations there then copy the data back. This will allow us to have a baseline measure of how fast our computations could be without this overhead.

4.1.3 Benchmark suite

The benchmark suite must test both the speed-up of both primitives: `run` and `reduce`. For this I have implemented a set of functions representative functions for each 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** and **mean reduction functions** which constitute two of the most common reduction operations over grids.

TODO: Speed up run

TODO: Reduce

4.1.4 Deducing a model

TODO: Model of on/off

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 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 test 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 incorporates both the test data (some times 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 where particularly necessary to check by QuickCheck:

- 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 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 was is same.
- Centres the cursor given the original grid had a cursor.
- Both roffset and coffset are always positive on such a grid. [TODO: section reference]

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 a 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[1] which made me consider other options.

Given that I couldn't 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

[1][TODO: explain bug]

code an the run implementation but this still provided some assurance. Once the bug had been fixed in the CPU implementation I was then 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 an failure is registered if they should differ.

```
write  --->  compile   --->   test   --->   commit
code              |              |              |
  ^---------------+--------------+--------------+
```

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 useability 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*[TODO:ref] to compare the various approaches already discussed.

Cognitive Dmensions 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*, for a complete list of dimensions considered see appendix TODO) 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 verses programming in a visual editor.

For the purposes of discussing only the language changes that I have introduced I will fix 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)
- [TODO: Anything else?]

4.3.2 Methodology

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

- Identify the relevant users of my system and sketch out a basic user profile.
- Select the relevant task of these users on my part of the language.
- Highlight which cognitive dimensions are most important to tasks selected.
- 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 users of Ypnos 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 a writing a graphics transformation for a photo editing package. The photos her user 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 don’t have a compatible GPU. She already has very good 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 it’s 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 task 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 look ahead: allowing the programmer to use one implementation without having to think about later switching.
- Consistency: the programs syntax or usage doesn't change from CPU to GPU.
- Terseness: the syntax to specify the implementation doesn't 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 users mental model for these types of operation.

The various approaches to provide an API to the programmer where discussed in the implementation section (TODO: link). They essentially boiled down to the following three approaches: choosing the different implementation based on importing, using type classes with associated data families, 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.

Cognitive dimensions	Accelerate	Non-unifing	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 sure 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 simple swap out the implementation by importing form different places.	**Best** We do not have imposed lookahead as we can easily swap the constructors.	**Best** There is little imposed look ahead 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 syntax's are different and so 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 don't 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 underneath the users noses due to different type and constraint families. This might affect some programs.
Abstraction gradient	**Best** The abstraction is at it's basic level: that of using 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 4.1: Comparison of the different API's using cognitive dimensions.

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 wouldn't 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

5 Conclusion

5.1 Accomplishments

5.2 Lessons Learnt

5.3 Future Work