Introduction

Preparation

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 differencial 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 Ypnos syntax as it is a pure functional language. 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.

TODO: Say something about the argument being Grid D a. n 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.

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:

```
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:

[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.

```
(+) :: 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.

Learning the libraries

Test driven development

TTD with QuickCheck

Implementation

Compilation of the stencils

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`, and

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.

Accelerate stencils and Ypnos stencils are different in that Ypnos allows the programmer to annotate the cursor location however

intermediary list approach cd9813c198a98135967ed6434415dfb388c10657

intermediary GridPattern' 8179d475bc9f192b4350b7df844be04dcffd1eb8

Directly with lists via type classes

Dealing with 1d and 2d

Implementation of the primatives

Initial implementation of run

Proliferation of constraints

Reduce implementation and the trick to it.

Generalization of functions

Type generalization approaches

Associated types vs type synonyms

How would it be used

Example of usage and replacement

Evaluation

Approach

Measure of difficulty

measuring copy on/off

Results

Speed up run

Reduce

Deducing a model

Model of on/off

Useability to the programmer

User having to write types

Conclusion