GPU Programming in Functional Languages

A Comparison of Haskell GPU Embedded Domain Specific Languages

July 12, 2013
Vrije Universiteit
Amsterdam

Author:
C. Ouwehand

Supervisor:
P. Hijma
W.J. Fokkink

Abstract

Graphical Processing Units (GPUs) are known to be excellent computation accelerators. However, their approach to data processing is very different from regular CPUs. This makes it harder for a regular developer to program these devices. In the past few years, several frameworks were introduced to simplify the programming of GPU devices. Accelerate and Obsidian are two of these frameworks, written in Haskell. Both present an *Embedded Domain Specific Language* (EDSL) to represent GPU computations. They try, at different levels, to abstract GPU details away.

In this paper we will assess different aspects of these EDSLs to see how usable they are for programmers. We will assess CUDA as well, to see how these EDSLs compare to plain CUDA. We will investigate a matrix multiplication use case, to see how well Accelerate and Obsidian perform.

1 Introduction

Many-core devices provide major speedup opportunities for multi threaded applications. *Graphics Processing Units* (GPUs) have become very popular many-core systems. They have a long history of development and are already available in many systems. Historically, they were designed to accelerate the rendering of computer graphics. However, their capability to process large datasets with a high degree of parallelism, resulted in many experiments to create general purpose programming models for these devices. This concept is also known as *General Purpose GPU* (GPGPU) programming. As a response to these experiments, Nvidia released CUDA.

CUDA simplifies GPGPU programming. In the past, DirectX or OpenGL were used to exploit the parallel capabilities of GPU devices. CUDA eliminates the need of such an intermediate layer[6]. It provides a straight forward general purpose interface to the GPU. However, many of the subtleties of GPU programming must be taken care of, to create even the simplest GPU program. Many language constructs, frameworks and *Embedded Domain Specific Languages* (EDSLs) emerged to simplify the interface to the GPU even more.

An EDSL is a domain specific language, embedded in an other language. In the past few years, several GPU EDSLs appeared, implemented in Haskell. We will examine two of these languages: Accelerate[3] and Obsidian[13]. These EDSLs define a restricted set of functions over, possibly multidimensional, arrays. The restricted function set can be translated to CUDA code and compiled by Nvidia's compiler to GPU binaries.

Accelerate and Obsidian differ in many ways. The approach of Accelerate is very high-level. It tries to be a generic interface to parallel array programming, independent of the underlying accelerating hardware. The Accelerate EDSL gives the programmer access to a set of functions, which are similar to the list operations available in Haskell. Although similar, the functions provided by Accelerate will be hardware accelerated at run-time.

Obsidian takes a different approach to GPU programming than Accelerate. It does not abstract all the subtleties away, but instead gives the programmer some control over lower-level constructs. It tries to find a balance in low-level control and high-level abstractions. Obsidian gives the programmer an abstracted interface to GPU programming.

GPUs are used to boost the performance of applications. As stated before, CUDA simplified the interface to the GPU. It eliminated the need to convert a general problem to a graphics problem. However, CUDA is not a high-level programming model. Many low-level details must be considered, when writing a GPGPU program in CUDA. Both Accelerate and Obsidian abstract the details of GPU programming, but can they make GPU accelerated code more accessible to mainstream programmers?

In this paper, we will compare the value of Accelerate and Obsidian to the mainstream programmer. We assume that a mainstream programmer has no prior knowledge about GPU programming and functional languages. We will introduce these concepts and have a look at how Accelerate and Obsidian fit in. Finally, we will do a case study, to see how hard it is to implement matrix multiplication in these EDSLs. Based on this case study, we will assess some aspects of these EDSLs.

First we will have a quick look at how CUDA works in section 2. In section 3, we will look at some basic concepts of functional programming, to get some better understanding of how these EDSLs work. Accelerate and Obsidian will be explained in more detail, respectively, in section 4 and 5. And finally,

we will do a case study to compare the EDSLs in section 6.

2 CUDA GPU programming

Both Accelerate and Obsidian combine GPU programming with functional programming. To get a better understanding of how these EDSLs work, we need an introduction in these subjects. This section will give a global overview of how CUDA works. Section 3 will look at the combination of functional programming and CUDA. These two sections combined will provide the needed introduction to discuss Accelerate and Obsidian.

CUDA is a programming platform and a model for programming GPU devices[6]. It explicitly distinguishes between the CPU machine, also called the *host*, and the GPU *device*. An extended version of C/C++ can be used to write GPU *kernels*. A kernel is a function, meant to be executed on a GPU device. Once launched, the kernel is executed by many threads. These threads are divided in groups, called blocks. The GPU device has many stream processors available. Each processor has many *Arithmetic Logic Units* (ALUs). These blocks of threads are assigned to a stream processor, which executes the kernel instructions in parallel on all its ALUs. This model is better known as *Single Instruction Multiple Data* or SIMD.

The global work flow with a GPU device is as follows. Data must be available before we can launch a kernel. Therefore, we have to allocate some memory on the device. We need to allocate memory for all input and output elements. Once we have allocated enough memory on the GPU, we can copy the input data from host memory to device memory. The device is now ready to execute the kernel. CUDA provides special syntax to launch a kernel. Besides the function name and a parameter list, you will have to provide some dimension information. After the kernel finished its execution, the result can be copied back to host memory and device memory may be freed.

The dimension information is used to decide how to group the threads. As said before, threads are grouped in blocks to be executed on SIMD processors. Each thread has a unique coordinate inside a block. Blocks may be multidimensional. They may have up to three dimensions. However, there is a limit to the amount of threads a block can handle. Therefore, you may have multiple blocks. Blocks are arranged in a grid, which can have up to three dimensions as well. Block and grid dimensions are specified between `<<<` and `>>>`, after the kernel name and before the parameter list, when it is invoked.

A kernel has access to a set of special variables. These variables are defined at runtime and contain dimension and location information. Each thread has a unique coordinate, within its block. Each block has a unique coordinate, within the grid. A thread coordinate is represented by `threadIdx.x`, `threadIdx.y` and `threadIdx.z`. A block has a similar coordinate structure. These coordinate variables express the parallelism of the kernel.

Usually one would use a for-loop, with an increasing index variable `i`, to perform an iterative computation. CUDA allows the programmer to unroll these loops. All iterations are run in parallel, instead of running each iteration sequentially. The unique coordinate, assigned to each thread at run-time, allows a thread to calculate a unique index, which represents the value of `i` in a single iteration.

CUDA GPUs do not actually schedule complete blocks on their stream processors. They split up a block in groups of 32 threads, called *warps*. These warps can be efficiently scheduled. Warps can not be controlled directly, but knowing about their existence can be useful when optimising a CUDA kernel.

Lets have a look at a simple kernel which adds two arrays. Normally we would write such a function as follows:

```
void addArrays ( float *v1
               , float *v2
               , float *r
               , int    l) {
    for (int i = 0; i < l; i++)
      r[i] = v1[i] + v2[i];
}
```

Listing 1: Sequential add array implementation in C

Every iteration, the elements of `v1` and `v2` at index `i` are added. The result is stored in `result` at index `i`. Although simple and effective, a SIMD approach can boost the performance of this function:

```
__global__ void addArrays ( float *v1
                          , float *v2
                          , float *r) {
    unsigned int index = blockIdx.x
                       * blockDim.x
                       + threadIdx.x;
    r[index] = v1[index] + v2[index];
}
```

Listing 2: Parallel add array implementation in CUDA

In this example, we use the block and thread coordinates to calculate the location of the array elements to add. For an array of length n, we allocate n threads, grouped in blocks. Every thread has to perform only one calculation. GPU devices can schedule many threads simultaneously. The result: a major time reduction, compared to the sequential example.

3 Functional programming

Both Accelerate and Obsidian are written in Haskell, which is a functional language. In order to under-

stand these EDSLs, we need to introduce some concepts of functional programming. In section 3.1 we will present an overview of how these concepts are used in Accelerate and Obsidian.

Functional languages use a mathematical approach to programming. Their definition of a function is closely related to the mathematical meaning of a function. Variables do not exist as we know them in the imperative world. They can be seen as names for expressions. Functions are *referential transparent*, meaning that a function can be substituted by its return value. Furthermore, functions do not have side-effects. Haskell adds a very powerful type system to this list of properties. Haskell's type system allows a programmer to prove some properties of a program.

Higher-order functions are essential in functional programming. They introduce composability on the level of a function. Higher-order functions are functions that accept one ore more functions as parameters and/or return a function. They represent abstract processes. These generic functions can be composed to represent a specific algorithm. For instance, looping over an array, to apply some function to every element in that array, can be abstracted as *mapping* a function to each element in the array. This abstraction is captured by the higher order function *map*. Map takes an unary function and a list. It applies the given function to each element in the list, recursively. This relieves the programmer from writing an entire loop construct.

Lazy evaluation is an other tool in the functional toolbox. Lazy evaluation allows the program to evaluate its expressions only when needed. This allows expressions to define infinite structures. For example, one could define an infinite list inductively. When a value is needed by an other expression, the list will evaluate all values up to and including the requested value. Functions are parameterized expressions and can be expressed inductively as well. This concept is also known as recursion.

Complex concepts can be expressed in a highly abstract way. These abstractions can be applied to specific problems by parameterizing the abstraction. When combined with inductive definitions one can apply abstract functions to an infinite problem space, while only computing the parts needed by the algorithm. This combination of higher-order functions and lazy evaluation leads to a high level of *modularisation*.

3.1 Functional EDSL overview

Both Accelerate and Obsidian specify an *Embedded Domain Specific Language*. In both EDSLs the programmer is presented with a restricted set of types and functions. Most functions are simple and composable. Instead of executing, these functions generate parts of an *Abstract Syntax Tree* (AST). One can write a program, which generates a full AST. This AST can be converted to CUDA C code, which can be compiled and run on a GPU device. The functions and types are constrained in such a way that they would only allow correct GPU programs to be written. Since these EDSLs are embedded in Haskell, the type system will discard type incorrect programs at Haskell compile-time. Simple EDSL functions can be composed to form highly complex programs, without worrying about the underlying GPU subtleties.

4 Accelerate

Accelerate[3] is a very high-level GPGPU EDSL. It is based on the GPU.Gen EDSL[8] and Repa[5], a regular, shape-polymorphic, parallel array library for Haskell. Arrays in Accelerate have the type signature: `Array sh e`, where `sh` is the shape type of the array and `e` is the element type. Arrays can be multidimensional in Accelerate. The shape type defines the dimensionality of an array. The shape is represented by a list, containing the extent of each dimension. Accelerate allows zero dimensional arrays. These are represented by the shape type *Z*. A matrix can be represented by a two dimensional array. The shape type of a matrix would look something like `(Z :. Int :. Int)`. However, this only tells us that it is a two dimensional array, but it does not tell us how big it is. If we want a matrix of three by three, filled with floats, we would specify the array as `Array (Z :. 3 :. 3) Float`.

In Accelerate, functions may be *shape-polymorphic*. These are functions that can be applied to an array of any dimension. The shape-polymorphic functions in Accelerate are restricted, they can only operate on the outermost dimensions of an array. The outermost dimension of a shape is the rightmost element in the shape list. A function can specify that it only cares about the outermost dimension, without worrying about the entire shape of the array. Such a function could be applied to any one or higher dimensional array.

All allowed base operations on Accelerate arrays and Accelerate types are purposely limited through the Haskell type system. Therefore, Accelerate arrays benefit from Haskell's compile-time guarantees. Where possible, Accelerate overloads standard operators and functions. This results in a safe subset of Haskell expressions, which can be translated to CUDA code.

Operations on arrays are not directly executed, but are translated to an AST. These ASTs are used to instantiate CUDA C skeletons, which are compiled and loaded back into the Haskell program. This is a runtime process, which means that kernels can be optimised depending on the runtime platform.

The high-level design of Accelerate enables rapid development in an almost native Haskell environment. However, program performance is compiler bound. Performance gains can only be achieved by optimizing the backend. Accelerate has support available for different backends. Currently, the CUDA backend is the only stable backend in Accelerate.

Accelerate is, at the time of writing, under heavy development. One of its major drawbacks is that it generates one kernel for each skeleton. This means that it runs one kernel for each function it wants to execute. For example, the dot product of two vectors can be calculated with `fold (+) 0 (zipWith (*) vec1 vec2)`. The vectors *vec1* and *vec2* are multiplied, element-wise, resulting in a single vector. The dot product is the sum of the elements in the result vector. The functions `fold` and `zipWith` are used to calculate the result. Both functions will generate one kernel each.

Another drawback of Accelerate is the lack of sharing. Assume `x` to be defined as the dot product expression, from the previous paragraph. Now, if we have an expression where `x` is used twice, like `x + x`, than `x` must be calculated twice.

Currently, Accelerate is being upgraded to handle these problems[10]. The first drawback is addressed by the introduction of *kernel fusion*. In this process, generated kernels are categorised and, based on their category, fused with adjacent kernels in the program. The sharing drawback will be handled by *sharing recovery*. Each expression will get an identifier based on their structure. If two expressions are exactly the same, than they will receive the same identifier. This way, duplicate expression can be executed once and share their result.

5 Obsidian

Obsidian[13] approaches the abstraction of GPU computations in a different way. It does not try to abstract all the GPU subtleties away. Instead, it gives an abstract view on GPU programming. It allows a programmer to specify at which level of the GPU hierarchy a function should operate.

Obsidian has changed throughout its development. It started with array types, which distinguished between shared memory and global memory arrays. It provided functions to transfer arrays between shared and global memory. However, the current version[1] has taken some different approaches. It has a rich type system, which enables functions to be specified at thread, block and grid level. It introduces *Push* and *Pull* array types. These types give the programmer more control over the performance of the generated code. We will discuss these array types in more detail, in section 5.1. Obsidian distinguishes between static and dynamic arrays. Static arrays have a fixed size. These are mostly used for thread local computations. Dynamically sized arrays can be used on block and grid level computations. A functions can leverage shared memory by using `force`. This function forces the threads to write their computation to shared memory and synchronise with the other threads, in one block.

The ability to give a function a thread, block or grid level type, allows algorithms to be decomposed. In plain CUDA C code, the role of a block or grid emerges from the kernel code itself. In Obsidian, one can explicitly define the role of blocks and the grid, while still generating a single kernel. In section 6.1.3 we will see that this principle helps us to separate our concerns.

5.1 Pull and Push Arrays

Obsidian knows two types of arrays: push and pull arrays. They were introduced to give the programmer more control over the efficiency of the generated code[4]. Their names refer to their usage scenarios. Pull arrays are used where data needs to be *pulled* from some location. While push arrays are used in scenarios where we want to *push* data in an array.

Push arrays describe where elements should be stored. They store elements as a collection of index/-value pairs. With this addition, the programmer has finer control over how the CUDA code is generated. But the programmer must be aware that more than one item can be assigned to a single location.

Pull arrays, on the other hand, are similar to the general concept of arrays. They describe arrays with an indexing function and a length. The name *pull array* just emphasizes its use cases and the fact that it is the complement of a push array.

Concatenation of arrays is an example of an operation which produces inefficient code, when implemented using pull arrays. For example, imagine two arrays of length 2^k, which we concatenate using pull arrays. The generated code will use a conditional, which uses the thread index to choose an input array to operate on. As stated in section 2, threads are scheduled in units of 32 threads, called warps. Therefore, when the number of threads is not a multiple of 32, half of the threads will execute the true branch and the other half the false branch. Thus, the allocated resources are not fully utilised while executing this code.

To optimise this code, one would use half as many threads and let each thread execute both the true and false branch. In this case, each thread calculates the correct input and output index based on its thread index. Push arrays allow you to do just that. With a push array we can specify what should be done to get the elements at the right index of the concatenated array.

[1] version 0.0.0.10, at the time of writing

The distinction between these two array types give the programmer more control over how the code will be generated. However, it adds some complexity to the type system. The programmer has to think about which kind of array is most suitable for the problem at hand.

6 The comparison

After this brief introduction to Accelerate and Obsidian, we will proceed with a comparison of these EDSLs. In order to compare them, we will do a case study. We will look at a matrix multiplication in both EDSLs and CUDA. The CUDA implementation serves as a reference. It will be used in the comparisons as well, to show the differences with hand-written CUDA code. We will compare the following aspects:

Learning curve
: In order to use any language, you will have to learn it. When a language is easy to learn, you can start using it faster. We will investigate possible roadblocks, while learning any of these languages.

Performance
: The main reason to use GPUs is performance. We will benchmark the case study implementations, to see which performs best.

Compilation and running
: The compilation cycle of a language is a very important part of development. A slow or complex compilation cycle can slowdown development drastically. We will look at what we must do to get from source code to a running program.

Control and abstraction
: We need to control certain details, in order to optimise a program. However, we need to abstract details, to build bigger programs. We want to know how these languages handle this trade off.

Maintainability
: The world changes rapidly. Our code needs to keep up with these changes. We want to know how manageable our code is, once written.

Applications
: Some languages are more suitable for certain tasks than others. This depends on several aspects. Based on our use case we will assess what kind of applications are most suitable for these languages.

6.1 Case study: matrix multiplication

We try to implement matrix multiplication and identify any implementation problems. Based on the results, we will judge the EDSLs.

6.1.1 CUDA implementation

A straight forward approach of implementing matrix multiplication is in terms of dot products. Every cell of the result matrix is the dot product of one row of the first input matrix and one column of the second.

CUDA provides a model where we can group threads in blocks and blocks in a grid. This hierarchy allows us to look differently at matrix multiplication. We can divide a matrix in rows and a row in cells. Each cell will be computed by one thread and each row by one block. As we know, one cell can be calculated using the dot product. Therefore, one thread will implement the dot product, where the block and grid dimensions will specify the dimensions of the output matrix.

```
__global__
void matMul (float *a,
             float *b,
             float *result) {
  unsigned int offset, cellIdx, i;
  float sum;

  offset  = blockDim.x * blockIdx.x;
  cellIdx = offset + threadIdx.x;
  sum     = 0;
  for (i = 0; i < blockDim.x; i++) {
    sum += a[offset + i]
         * b[blockDim.x * i + threadIdx.x];
  }

  result[cellIdx] = sum;
}
```

Listing 3: CUDA matrix multiplication implementiation

In listing 3 the CUDA matrix multiplication implementation is shown. It calculates a dot product in the for-loop. Before the thread can start the for-loop, it has to figure out which cell it computes. The input arrays are just one dimensional C arrays. To find the cell to work with, the thread has to calculate the index of that cell. As we use a block to represent rows, we can use the block dimension and block index to calculate the offset of the row to work with. A thread computes one cell, within a row. Therefore, the thread index combined with the previously calculated row offset gives us our cell index in the resulting array.

These values are used in the for-loop. The row offset of the resulting array is the same as the row offset of the first input array. So, we only need to add the current iteration `i`, to find the cell index. From the second input matrix, we need the column cells. To find these cells, we combine the block dimension and the current iteration to find the correct row, in each iteration. We add the thread index to find the correct column. The product of these cells are aggregated in the variable `sum`, which is written to one cell in the result matrix.

While easy to comprehend, it is not very flexible or well optimised. Rows are indexed using one dimension of the thread index, for both input matrices.

This restricts the input matrices to square matrices of equal size. CUDA limits the maximum length of the block and grid dimensions. This effectively limits the dimensions of the input and output matrices.

Many optimisations are possible, but they will make the algorithm harder to comprehend. Besides, the programmer must have some knowledge about the details of CUDA or even device specific knowledge.

Once written, this code can be easily integrated in a C/C++ program, with just a little bit of glue code. The C program interacts with the Haskell CUDA binding to prepare device memory and launch the kernel.

6.1.2 Accelerate implementation

Accelerate has a very abstract array representation. This allows us to concentrate on the problem at hand. Instead of thinking about which steps needs to be taken, we can approach a problem in an abstract way. Accelerate allows us to use multidimensional arrays. It allows us to increase and reduce the dimensionality of an array. These concepts can be very useful when we have to reason about matrix multiplication. We will use these concepts to visualize our problem.

```
import Data.Array.Accelerate as A

matrixMul a b = sum' (prod aCube bCube)
where
  sum'   = A.fold (+) 0
  prod   = A.zipWith (*)
  t      = A.transpose b
  getRow = indexHead . indexTail
  getCol = indexHead
  rowsA  = getRow (A.shape a)
  colsB  = getCol (A.shape b)
  sliceA = lift (Z :. All :. colsB :. All)
  sliceB = lift (Z :. rowsA :. All :. All)
  aCube  = A.replicate sliceA a
  bCube  = A.replicate sliceB t
```

Listing 4: Accelerate matrix multiplication implementation

Listing 4 shows a matrix multiplication implementation in Accelerate. Type signatures are left out for brevity. The `where` block contains the definitions used in the main function. These definitions can represent values as well as functions. For example, `sum'` evaluates to a function and `t` evaluates to a value.

The main function defines what looks like a dot product. However, it actually builds two cubes from the input matrices, by replicating along a new dimension. Suppose that the first input matrix has m rows and the second input matrix has n columns. When we replicate the first input matrix n times and the second input matrix m times, we end up with two cubes with the same dimensions. Take three adjacent sides on the cube. We know that two of these sides have the same dimension as our input matrices. The third side will have the dimension of the result matrix (m by n). We merge the two cubes by multiplying each corresponding cell of the cube. Finally, we can flatten the cube by reducing one dimension with `sum'`. This leaves us with a matrix of m by n.

There are no GPU details to take in consideration. GPU acceleration is an added bonus. It provides a highly abstract interface to array computations. Instead of worrying about low-level details, the programmer has to think about how to represent a problem. This representation of the matrix multiplication was presented before in the Repa paper[5].

Functions written in Accelerate are easily launched with the `run` function, which is available in the CUDA backend, provided by Accelerate. Before arrays can be used, they must be transfered to the GPU device. Accelerate has a convenient function called `use`, which does exactly that.

6.1.3 Obsidian implementation

Obsidian has a range of types to specify what kind of array you are working with. It does not define dimensionality, but it allows you to create multidimensional arrays by nesting them. Nested arrays can only be created by splitting up flat arrays. Since a matrix is a two dimensional structure, we will introduce a new type. However, Obsidian distinguishes between static and dynamic arrays. This enables us to define a matrix in multiple ways. To aid readability, we will define two types of matrices.

```
type DSMatrix e = DPull (SPull e)
type SMatrix  e = SPull (SPull e)
```

Listing 5: Obsidian Matrix types.

We have defined a dynamic-static matrix (`DSMatrix`) and a static-static matrix (`SMatrix`). A dynamic-static matrix can be viewed as a dynamic array, split up in statically sized parts. We need to distinguish between static and dynamic types, in order to use certain functions, defined by Obsidian.

In listing 6 we implement matrix multiplication in Obsidian. The algorithm is similar to the CUDA implementation. However, its definition is split in multiple functions, each specifying one level of the CUDA GPU hierarchy. The grid level function combines the answer of the block level function. The block level function computes one row of the result matrix by computing the dot product for each cell, using a thread level function.

The Obsidian function `pConcatMap` maps a function over a nested array and concatenates the result, to return a flat array. It describes a function at either block or grid level. The exact level depends on the function it uses to compute its result. For example, a thread level function can be lifted to a block level function, by mapping it in parallel over a nested array, with `pConcatMap`.

```
-- Grid level function
matMul :: (Num e, MemoryOps e)
       => DSMatrix e
       -> DSMatrix e
       -> DPush Grid e
matMul a b = pConcatMap row a
  where
    row x = matMulRow x t
    t     = transpose b


-- Block level function
matMulRow :: (Num e, MemoryOps e)
          => SPull   e
          -> SMatrix e
          -> BProgram (SPush Block e)
matMulRow row mat =
  return $ pConcatMap (dotP row) mat


-- Thread level function
dotP :: (Num e, MemoryOps e)
     => SPull e
     -> SPull e
     -> TProgram (SPush Thread e)
dotP a b = return
         $ sumS
         $ zipWith (*) a b
  where
    sumS = seqReduce (+)
```

Listing 6: Obsidian matrix multiplication

This clear separation of the hierarchy levels makes it easier to comprehend than the plain CUDA example, while it generates roughly equivalent code. Besides, these functions are composable. For example, one could write a function which takes four arrays and produce a tuple of two dot products, at thread level. The difference between pull and push arrays makes it somewhat harder to combine function to form new functions.

Finally, to generate CUDA code from Obsidian, one must call `genKernel name program inputlist`. This will generate a CUDA kernel from `program`, with `name` as its function name. The `inputlist` is used to specify what the argument list looks like. The generated code can be compiled with the `nvcc` compiler. To load it in Haskell, one must compile to *ptx* or *cubin* format. These compiled files can be loaded using the Haskell CUDA binding.

6.2 Results

6.2.1 Learning curve

CUDA is somewhat easy to learn, but hard to master. The concept of blocks and grids is easy to understand and to work with. Just a few extra constructs need to be added to an existing C/C++ program, to extend it with GPU accelerated functions. A naive implementation is not too hard to write. It becomes harder when you want to improve the performance of a naive implementation. Many details must be taken into consideration. This makes CUDA difficult to master.

If you do not have any experience with functional languages, both Accelerate and Obsidian can be hard to get used to. Functional programming requires a different mindset, compared to imperative programming. To assess their value, we have to assume some basic knowledge of functional programming.

Once used to Haskell, Accelerate is not too hard to learn. It overloads many Haskell functions and operators. Its library and syntax are very similar to that of Repa. Many multidimensional array operations from Repa are available in Accelerate. Only the type signature of a function gives away that it is a GPU accelerated function. Its API has not changed much over the years and it has some documentation available.

Obsidian is harder to learn. It has changed a lot over the years and there is not much documentation available. The type system is complex and requires some knowledge of GPU programming. Concepts like `Push` and `Pull` arrays take time to get used to.

6.2.2 Performance

Table 1 presents the results of the performance tests. Accelerate performs worse than Obsidian and CUDA. We have added a Repa implementation, as a sequential reference. The Repa implementation uses the same algorithm as the Accelerate implementation. Although slower than Obsidian and CUDA, Accelerate still is much faster than Repa.

Each test was run ten times to get an average. The Obsidian code was loaded at runtime using the Haskell CUDA binding[2]. We used n by n matrices, to test our kernels. The size of n used in a single test can be found in the leftmost column of the table.

	Repa	Accelerate	Obsidian	CUDA
64	5.21×10^{-2}	7.28×10^{-3}	2.59×10^{-4}	2.62×10^{-4}
128	4.21×10^{-1}	1.27×10^{-2}	3.92×10^{-4}	3.91×10^{-4}
256	3.40	3.50×10^{-2}	9.75×10^{-4}	9.75×10^{-4}
512	2.80×10^{1}	1.68×10^{-1}	4.83×10^{-3}	4.88×10^{-3}
1024	2.25×10^{2}	1.29	3.67×10^{-2}	3.72×10^{-2}

Table 1: Matrix multiplication performance test run on a GTX480

Accelerate produces optimised kernels. However, it generates one kernel for every parallel array computations. Our implementation contains two kernel functions, the `fold` kernel and the `zipWith` kernel. The `replicate` function only transforms index space and thus creates a virtual cube structure, by replicating over a new dimension. We merge the virtual cubes using `zipWith`. The result cube structure cannot be represented by a simple index space transformation. The `zipWith` kernel has to calculate every element of the intermediate cube structure, before we

[2] `https://github.com/tmcdonell/cuda`

can run the `fold` kernel. Therefore, the two n by n matrices used as input, will produce an intermediate cube structure of n by n by n. In other words, the input size is in the order of $2n^2$ and the output size is in the order of n^2, while the intermediate cube structure is in the order of n^3. This explains the major slowdown of Accelerate.

The matrix multiplication implementation in Obsidian generates a single kernel. The kernel calculates the dot product. It uses a sequential loop to calculate the product of two vectors. The sum is aggregated, while calculating the product. This leads to a much faster kernel. The handwritten CUDA kernel is similar to the generated Obsidian kernel and, therefore, performs about as well as the Obsidian kernel.

6.2.3 Compilation and running

CUDA C code can be compiled with the `nvcc` compiler. This compiler separates the CUDA specific code from the regular C code. The regular code is send to the main compiler (e.g. gcc) and the CUDA code is compiled by `nvcc`. All objects are linked back in a single binary, which can be run like any normal binary.

Accelerate has a CUDA function template for every parallel operation. It instantiates all needed templates at runtime and pushes them in the compiler queue. It tries to optimise the parameter list, used for compilation. It has threads running, to compile many pieces at the same time, using `nvcc`. Once compiled, the kernel is added to a hash table. This avoids recompilation of already compiled functions. Functions are loaded and executed using the Haskell CUDA binding. This allows Accelerate to interleave regular Haskell code with GPU accelerated code.

Obsidian just generates code. It does not compile and launch the generated code. However, with a little glue code, one can easily dump the generated code and compile it using `nvcc`. Running it can be a little bit trickier, since it involves some knowledge about the Haskell CUDA binding.

6.2.4 Control and abstraction

Many details must be taken into consideration, when writing a CUDA kernel. CUDA provides a rich set of tools to fine-tune the performance of a piece of code. This gives the programmer a lot of control. Although it provides an abstract layer over GPUs in general, it does not try to abstract array computations in any way.

Accelerate does not provide any control of CUDA details, while working with the EDSL. One can write its own backend, allowing a programmer to add some optimisations. This limits the possible optimisations one can do within the EDSL. However, it enables highly abstract algorithms to be programmed, with GPU acceleration for free.

Obsidian is a lot more flexible than Accelerate, when it comes to control. Some aspects of CUDA programming are abstracted. However, many aspects can still be controlled from within Obsidian. For example, writing to shared memory and synchronising threads can be explicitly controlled. It does not abstract array computations as much as Accelerate does, nor does it attempt to abstract all GPU programming details away. It does present a sort of intermediate language, where GPU computations can be decomposed, while it still generates a single kernel.

6.2.5 Maintainability

We noted in section 6.2.4 that CUDA is a very detailed language. Depending on the level of specialisation of a function, it can be very hard to maintain. It is possible to write simple, (almost) sequential code in CUDA C. But this will result in just a minor performance boost. To get the best performance possible, one must think about memory access patterns, when to use shared memory, when to synchronise and more. Once optimised, it can be very difficult to change the code, when needed.

Both Accelerate and Obsidian enjoy the strictness of the Haskell type system. Accelerate provides the complete package. It provides a rich set of composable function and automatic compilation of the generated CUDA code. Accelerate uses the CUDA driver to investigate its environment. So, whenever the environment changes, for example, the code is ran on a different system, Accelerate tries to optimise its processes according to its environment. Whenever requirements change, existing functions might be reused, depending on the context.

Obsidian is harder to maintain, due to its complex type system. However, Haskell allows easy composition of functions. Since the language is defined by a restricted set of operations in Haskell, it enjoys some of its benefits. The use of `Push` and `Pull` arrays, combined with static and dynamic arrays, can make function composition difficult. But once written, it is easier to maintain than CUDA code.

6.2.6 Applications

CUDA programs are best suited for applications where speed is all that counts. It allows you to fine-tune every bit of your GPU accelerated code. When all effort can be put in code optimisation, without having to care about maintainability, plain CUDA code can bring a major performance boost to your application.

Accelerate allows code that can be boosted, without any knowledge of GPU programming. Applications

where correctness and simplicity of the algorithm is important, Accelerate would be the best choice. It allows the programmer to write highly abstract GPU accelerated code, without actually having to know anything about GPU programming.

Obsidian is more general purpose than the other two. It could be used to quickly prototype GPU accelerated applications. Parts of an existing Obsidian program might be reused in a new program. The abstraction over the GPU hierarchy makes reasoning about an application much easier. The abstracted hierarchy makes it an interesting platform for teaching about GPU programming.

7 Related Work

In the past few years, many frameworks emerged to simplify GPU programming. Accelerate and Obsidian are two examples, written in Haskell. There are many more solutions available, to simplify GPU programming, or data parallel programming in general. For example, Python has a GPU programming framework called Copperhead[2] and Scala has Firepile[11]. Beside frameworks embedded in a language, there are languages dedicated to data parallel programming. Examples of these are Single Assignment C[12] and NESL[1]. However, we will keep our focus on Haskell EDSLs.

We will start with GPU.Gen. GPU.Gen[8] is the predecessor of Accelerate. It is a two level language. It distinguishes between *collective array operations* and *scalar computations*. Scalar computations are used to parameterise collective array functions. GPU.Gen defines a GPU *monad* to aggregate collective GPU operations and encapsulate them. It presents a stable introduction to functional GPU programming.

Nikola[9] is an EDSL which shares some properties with both Accelerate and Obsidian. Like Accelerate, it specifies an EDSL which tries to seamlessly integrate with Haskell. However, it only supports first-order functions. The only higher-order functions available, are embedded in the language. Nikola supports runtime optimisation, compilation and execution of the embedded language. Beside runtime compilation, it supports Haskell compile-time compilation of the embedded language as well. This enables the programmer to decide when to compile the embedded language. Nikola also offers the programmer to embed CUDA C source code within a Haskell program. Like Obsidian, it generates a single kernel.

Finally, we will look at Barracuda. Barracuda[7] is a research EDSL in Haskell. It has been developed to demonstrate some simple optimizations for GPU array languages. The capabilities of the language are limited, due to its specific purpose. However, it presents a method for automatic use of shared memory. This feature is not available in Accelerate and Obsidian.

8 Conclusion

We have looked at two functional GPU EDSLs, Accelerate and Obsidian. We have assessed their value for mainstream programmers, with no prior knowledge about GPU programming. We have looked at a use case, to get a better view of the strengths and weaknesses of these EDSLs.

Many aspects influence the usability of these EDSLs. Both can have major benefits, depending on your requirements. Accelerate is good at abstracting multi-dimensional array problems, with GPU acceleration for free. On the other hand, Obsidian provides more control over the performance of the generated code and allows the code to be optimised by hand, afterward. Both present a new way of writing GPU accelerated code.

With no prior knowledge of GPU programming, Accelerate is the easiest solution to write GPU accelerated abstract array oriented code. It is relatively easy to learn and can be quickly put to work. There might be performance penalties in some situations, but in most cases it will boost the performance of abstract array computations.

This work could be extended by taking more languages into account. We have looked at two EDSLs, while there are more GPU EDSLs available. Every EDSL comes with its own set of pros and cons. With this paper, we try to presents a decent introduction to the world of functional GPU EDSLs.

References

[1] Guy E Blelloch. Nesl: A nested data-parallel language.(version 3.1). Technical report, DTIC Document, 1995.

[2] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pages 47–56. ACM, 2011.

[3] Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14. ACM, 2011.

[4] Koen Claessen, Mary Sheeran, and Bo Joel Svensson. Expressive array constructs in an embedded gpu kernel programming language. In *Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming*, pages 21–30. ACM, 2012.

[5] Gabriele Keller, Manuel MT Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. *ACM Sigplan Notices*, 45(9):261–272, 2010.

[6] David B Kirk and W Hwu Wen-mei. *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann, 2010.

[7] Bradford Larsen. Simple optimizations for an applicative array language for graphics processors. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 25–34. ACM, 2011.

[8] Sean Lee, Manuel MT Chakravarty, Vinod Grover, and Gabriele Keller. Gpu kernels as data-parallel array computations in haskell. In *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods*, 2009.

[9] Geoffrey Mainland and Greg Morrisett. Nikola: embedding compiled gpu functions in haskell. In *ACM Sigplan Notices*, volume 45, pages 67–78. ACM, 2010.

[10] Trevor L McDonell, Manuel MT Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising purely functional gpu programs.

[11] Nathaniel Nystrom, Derek White, and Kishen Das. Firepile: run-time compilation for gpus in scala. In *ACM SIGPLAN Notices*, volume 47, pages 107–116. ACM, 2011.

[12] Sven-Bodo Scholz. *Single Assignment C*. PhD thesis, Citeseer, 1996.

[13] Joel Svensson, Mary Sheeran, and Koen Claessen. Obsidian: A domain specific embedded language for parallel programming of graphics processors. In *Implementation and Application of Functional Languages*, pages 156–173. Springer, 2011.