An embedded language for data-parallel programming

Joel Svensson

January 2008

Abstract

This thesis describes the implementation of Obsidian, an embedded language for data-parallel programming. The programming style used in Obsidian borrows many ideas from the hardware description language Lava. In lava combinators are are used to combine circuits into larger circuits. Obsidian uses the idea of combinators from Lava, but instead of circuits; data-parallel computations are described. From the high level descriptions of data-parallel computations expressed in Obsidian, C programs are generated for execution on an NVIDIA GPU (Graphics Processing Unit). Modern GPUs are becomming powerful parallel computers, this is giving rise to an entire field called the GPGPU (General-Purpose Computations on the GPU) field. This field driven by the desire to exploit the power of GPUs for general-purpose computations. Obsidian explores a high level programming style for expressing parallel computataions. The implementation of a number of sorting algorithms using Obsidian is also shown. This thesis touches on the areas of GPGPU, embedded languages and data-parallel programming.

1 Preface

This document is written as a masters thesis in computer science at Gothenburg University. This thesis describes a project carried out during the fall of 2007 supervised by Mary Sheeran and Koen Claessen both at the Department of Computer Science and engineering at Chalmers University of Technology. The subject matter of the project was suggested by Mary Sheeran, when I approached her with an inquiry after taking her course on hardware description and verification.

This project addresses the issue of parallel programming and presents a programming style where combinators are used to express parallel computations. The thesis describes the implementation of a language that we call Obsidian. The name Obsidian was chosen to indicate its relation to the hardware description language Lava (Obsidian being a form of volcanic glass). Obsidian is implemented as an embedded language, using Haskell as host.

I would like to thank Mary Sheeran for giving me the opportunity to work on this very interesting project. I also thank Mary Sheeran and Koen Claessen for all their help and motivating discussions during the project. Thanks also to Ulf Assarsson who gave us an introduction to GPU (Graphics Processing Unit) programming. Lastly thanks to Satnam Singh who supplied two of the figures used in the document.

Contents

List of Figures

2 Introduction

This document describes the implementation of an embedded language for data-parallel programming in Haskell. From high level descriptions of parallel computations low level C programs are generated. These C program can be compiled and run on an NVIDIA GPU (Graphics Processing Unit). The programming style of this embedded language borrows many ideas from Lava. Lava is an embedded language for structural hardware description and verification developed by Koen Claessen and Mary Sheeran at Chalmers and Satnam Singh at Xilinx.

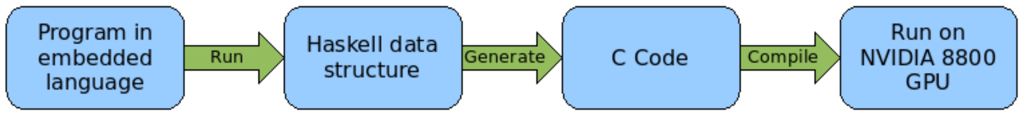

Figure 1: Steps in generating CUDA C code

2.1 GPGPU

GPUs designed to produce the fast paced graphics in modern games are now interesting for general purpose computations as well. GPUs are designed for graphical computations of highly data-parallel nature. In comparison to CPUs (Central Processing Units), GPUs devote more of their transistor budget to computation, where CPUs need to devote much effort to extracting instruction-level parallelism [25]. The GPGPU (General-Purpose Computations on the GPU) field is driven by the desire to use the computational power of GPUs for general-purpose computations.

GPUs have been successfully applied to several areas such as physics simulation, bioinformatics and computational finance [26]. Sorting is another area where there are success stories [29, 18].

2.2 CUDA and the NVIDIA 8800 GPU

With NVIDIA's release of CUDA (Compute Unified Device Architecture) the availability of GPUs to the general purpose programmer is improved greatly. Earlier, GPGPU programming was done through the graphics API and expressing a non-graphical computation using graphics vocabulary is awkward [25]. With CUDA you are programming using a general purpose

API, in fact a subset of the C standard libraries. Now GPUs can be viewed as highly parallel co-processors. Another attempt to make the GPU more accessible to the general-purpose programmer is Microsoft Accelerator, which supplies libraries for data-parallel programming on the GPU.

The NVIDIA 8800 GPU is described as “a set of SIMD multiprocessors” in the CUDA Programming Manual [1]. SIMD (Single Instruction Multiple Data) is one of the categories of computer architectures proposed by Micheal J. Flynn [3]. The NVIDIA 8800 GPU has up to 128 processing elements organised in groups of eight. There are up to 16 such “multiprocessors” within the single GPU. Figure 2 shows a schematic picture of the architecture.

On one of these groups of 8 SIMD processing elements a number of threads are executed. Such a group of threads is called a thread *block*. A thread is the execution of a sequential list of instructions. Each of the threads in a block is executing an instance of the same program. A block is divided into smaller groups that are executed in a SIMD fashion; these groups are called *warps*[1]. This means that within a warp, all threads are progressing at the same pace through the program. There is a scheduler that periodically switches warps. Between warps the SIMD fashion of execution is not maintained, thus thread synchronisation primitives are needed.

To get the most benefit out of running on the GPU a program has to be highly parallel. Also it should have a large number of arithmetic operations compared to memory operations so as to be able to hide latencies. Another issue that can degrade the performance is flow control instructions. A flow control instruction such as `if`, `for`, `while` or any other instruction that would be turned into branch instructions in the compiled code has the potential to negatively impact performance. Branching may cause threads to follow different paths of execution. When this happens there is a risk that the execution paths need to be serialised. In the case of an `if` statement, the execution of the `then` and the `else` blocks of code might be sequenced [4].

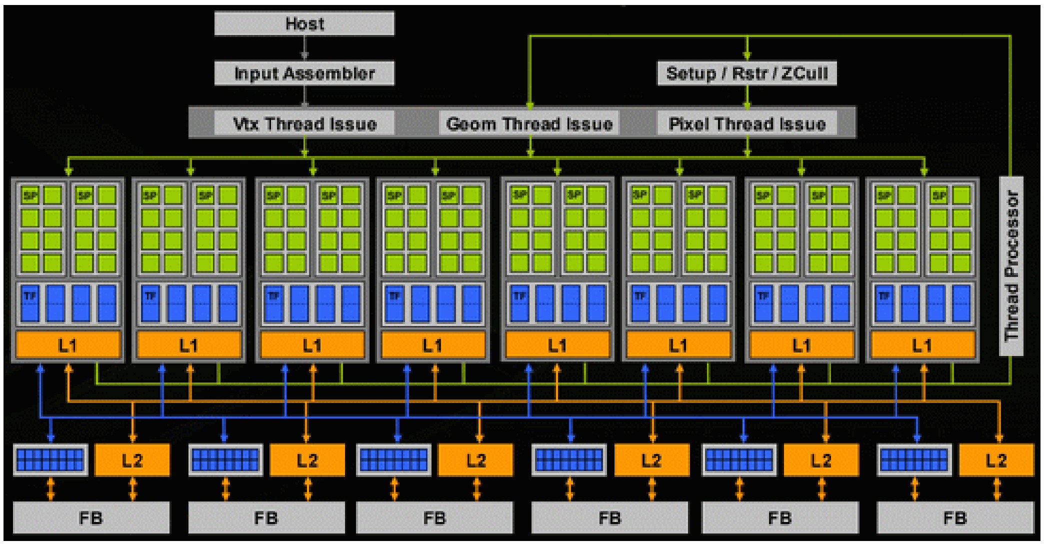

Figure 2: Schematic diagram of the NVIDIA 8800 GPU

2.3 Data-parallel programming

In the data-parallel programming model operations can be performed on elements of some data structure, for example an array or matrix, in parallel. Programs are often written in a sequential style with little difference to normal single threaded programs. But in the data-parallel programming model, the program is executed by a number of processing elements or virtual processors in parallel [19]. The SIMD architecture is particularly well suited for the data-parallel programming model. SIMD systems have several processing elements all executing the same instruction at any given moment, but using different data [3].

The following is a short list of data-parallel operations [24, 27]:

elementwise operations are operations that apply some function to each element in a collection of data.

reductions such as summing up all the elements in an array of integers.

prefix, or scan, operations are related to the reduction operations. but give as result all partial reductions.

restricted, or conditional operations. An array of boolean values can be used to turn operations on or off at certain locations.

communication such as get communication or send communication.

2.3.1 Data-parallel programming in CUDA

CUDA programs are written in C extended with some new syntax for amongst other things invoking a computation on the GPU. Programs written to execute on the GPU are called *kernels*. These kernels are executed by a specified number of threads on the GPU. Invoking such a computation on the GPU is done via:

```
myComputeKernel<<<n,m,k>>>(arg1,arg2);
```

The code above sets up `myComputeKernel` to be executed by n blocks of m threads, using up to k bytes of shared memory per block. An important issue here is that one block of threads will execute on only one of the multiprocessors described above. Hence there are eight SIMD processors working on each block of threads. It is only within one of these blocks that communication and synchronisation between threads is possible [1].

The threads executing in a block communicate by reading and writing from a shared memory. This shared memory is located on the GPU chip and is sometimes also referred to as a parallel data cache. Declaring a variable in the shared memory space is done using the storage qualifier `__shared__` [1].

There are also a number of built-in variables that can be accessed by each thread. It is not allowed to assign values to any of these variables:

`threadIdx` contains the thread's thread id within the block. The variable threadIdx is a vector of three elements accessed by threadIdx.x, threadIdx.y and threadIdx.z.

`blockIdx` contains the block id; this together with the thread id identifies the thread uniquely.

`blockDim` specifies the dimensions of the block.

`gridDim` specifies the dimensions of the grid of blocks,

Heres a small example where each thread is in charge of reading a value from the array `values`, located in device memory, into the array `shared` in shared memory:

```
shared[threadIdx.x] = values[threadIdx.x]
```

Below is an example of a data-parallel program written in CUDA. The example shows bitonic sort from the CUDA SDK (Software Development Kit) [2]. In this example the functions `bitonicSort` is run by as many threads as there are elements in the array to be sorted. The example uses 512 threads to sort an array of length 512:

```
__device__ inline void swap(int & a, int & b)
{
// Alternative swap doesn't use a temporary register:
// a ^= b;
// b ^= a;
// a ^= b;

    int tmp = a;
    a = b;
    b = tmp;
}

__global__ static void bitonicSort(int * values)
{
    extern __shared__ int shared[];

    const int tid = threadIdx.x;

    // Copy input to shared mem.
    shared[tid] = values[tid];

    __syncthreads();

    // Parallel bitonic sort.
    for (int k = 2; k <= NUM; k *= 2)
    {
        // Bitonic merge:
        for (int j = k / 2; j>0; j /= 2)
        {
            int ixj = tid ^ j;

            if (ixj > tid)
            {
                if ((tid & k) == 0)
                {
                    if (shared[tid] > shared[ixj])
                    {
                        swap(shared[tid], shared[ixj]);
                    }
                }
                else
                {
                    if (shared[tid] < shared[ixj])
                    {
                        swap(shared[tid], shared[ixj]);
                    }
                }
            }

            __syncthreads();
        }
    }
    // Write result.
    values[tid] = shared[tid];
}
```

The innermost loop in the program above is the bitonic Merger, a bitonic merger sorts an array where the first half of the array is sorted in increasing order and the second half is sorted in decreasing order. There is more information on this algorithms in [10].

2.3.2 synchronisation in CUDA

To avoid the problems that can occur when several concurrently running threads are accessing the same memory, such as the RAW (Read After Write), WAR (Write After Read) and WAW (Write After Write) hazards, thread synchronisation is needed. In CUDA, synchronisation is done by calling the function:

```
__syncthreads();
```

The CUDA programming manual describes this as a barrier synchronisation. Only when all threads have reached the synchronisation barrier will the computation proceed [1]. As a result of this the synchronisation primitive can only be used in conditional code if all threads choose the same path.

2.4 Lava

Lava is an embedded language for structural hardware description and verification developed by Koen Claessen and Mary Sheeran at Chalmers and Satnam Sing at Xilinx [6]. In Lava, circuits are designed using combinators that capture common connection patterns. Combinators are higher order functions, used to combine circuits into larger circuits. The combinators used in Lava have been shown well suited for designing sorting networks [12]. A sorting network is different from the more commonly known sorting algorithms like quick-sort in that the number of comparisons is data independent. This makes sorting networks suitable for hardware and parallel implementations [22].

2.4.1 Batcher's bitonic sort in Lava

An example of a sorting network is Batcher's bitonic sorter [5]. One of the building blocks needed to implement bitonic sort is the bitonic merger. In Lava, such a merger is constructed from a *butterfly* of two-sorters. A butterfly is a network with a recursive structure indicated in figure 3. A two-sorter, is a two-input two-output circuit which sorts its two inputs onto the outputs.

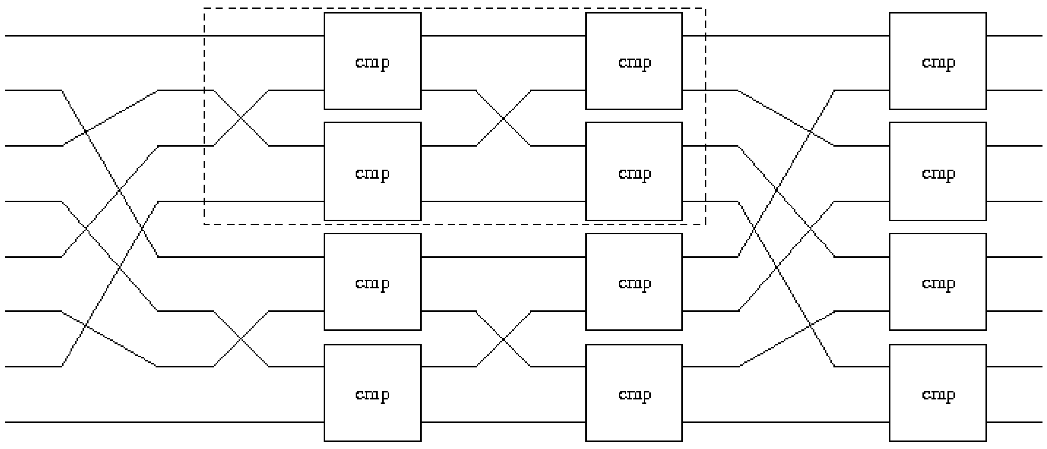

Figure 3: An 8 input butterfly network

The lava code for the butterfly is:

```
bfly 1 f = f
bfly n f = ilv (bfly (n-1) f) ->- evens f
```

Figure 3 shows what a butterfly generated by the program `bfly 3 cmp` looks like where `cmp` is a two-sorter. The following program gives a merger useable in implementing bitonic sort:

```
bfly n cmp
```

The butterfly defined above sorts arrays where the first half is sorted increasingly and the second half decreasingly. The butterfly of two-sorters performs the same operation as the inner loop of the CUDA example earlier. The following two simulations of a butterfly show that it can be used to merge a bitonic sequence:

```
> simulate (bfly 3 cmp) [1,3,5,7,8,6,4,2]
[1,2,3,4,5,6,7,8]
> simulate (bfly 3 cmp) [8,2,6,4,1,5,7,3]
[1,2,3,6,4,7,5,8]
```

In the implementation of the butterfly above the combinators `ilv`, `evens` and `->-` were introduced. Sequential composition of circuits, `->-`, connects the outputs of the circuit to the left of it to the inputs on the circuit to the right. The combinator `evens` places a two-input circuit between each even numbered input and its following neighbour:

```
evens f []         = []
evens f (a:b:cs)  = f [a,b] ++ evens f cs
```

The `ilv` combinator connects all the even numbered inputs to one instance of a circuit and all the odd numbered inputs to another instance of the same circuit.

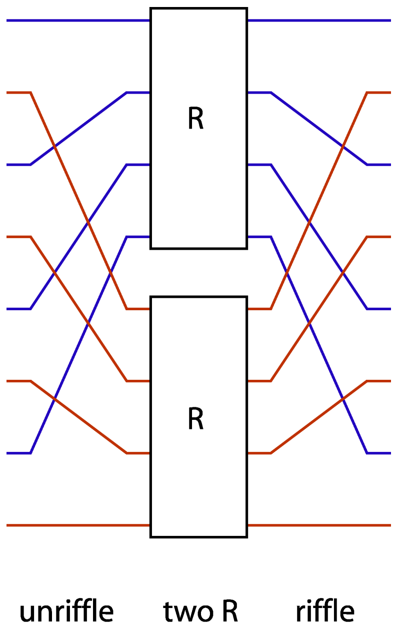

Figure 4: Interleave

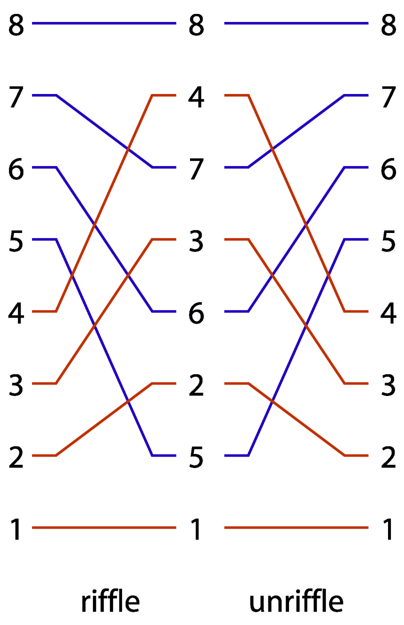

Figure 5: Riffle and unriffle

Figures 4 and 5 show the wiring patterns `riffle`, `unriffle` and how `ilv` is implemented using `riffle`, `unriffle` and the combinator `two`. The combinator `two` places two copies of an n input circuit in parallel, giving a $2n$ input circuit. The combinators and wiring patterns mentioned above have some interesting properties. Two such properties are that n successive applications of `riffle` of a list of length 2^n returns it to its original order or that the combinator `two` distributes over sequential composition:

```
two (R ->- S) = two R ->- two S
```

It is also the case that applications of `two` and `ilv` commute:

```
two (ilv R) = ilv (two R)
```

These properties are described in much more detail in [17].

In Lava, the sorting network can be described like this using the `bfly` from above:

```
sortB 0 cmp = id
sortB n cmp = parl (sortB k cmp) (sortB k cmp ->- reverse) ->- bfly n cmp
  where
    k = n - 1
```

In `sortB` the combinator `parl` is used to recursively generate two smaller sorters of size $n/2$, the outputs from these two sorters are then merged by the bitonic merger implemented using the butterfly from above. The combinator `parl` takes two circuits with m and n inputs and places the first above the second yielding a $m + n$ input circuit. The `sortB` function above takes two parameters deciding the size and what comparator to use. The application `sortB m cmp`, where `cmp` is a two-sorter, gives an instance of bitonic sort for arrays of length 2^m. Below two simulations of the sorter defined in this section are shown:

```
> simulate (sortB 3 icmpSwap) [8,7,6,5,4,3,2,1]
[1,2,3,4,5,6,7,8]
> simulate (sortB 3 bcmpSwap) [high,low,high,low,high,low,high,low]
[low,low,low,low,high,high,high,high]
```

The two two-sorters used in the simulation above, `icmpSwap` and `bcmpSwap` are implemented as follows:

```
icmpSwap [a,b] = [imin (a,b), imax (a,b)]
bcmpSwap [a,b] = [and2 (a,b), or2 (a,b)]
```

From the Lava programs above a netlist can be generated to enable implementation of the design in hardware, or logic formulae can be output for verification using a model checker or SAT solver [12].

2.5 Embedding a language in Haskell

Embedding a language in Haskell is done by creating a data type that represents programs in the embedded language, a library of functions to manipulate and create objects of that type and functions that execute such programs [11].

2.5.1 Embedding a language of integer expressions

This small example embeds an interpreter for integer expressions. The integer expressions are of a type called `Exp` defined as follows:

```
data Exp = Literal Int
         | Add Exp Exp
         | Sub Exp Exp
         | Mul Exp Exp
         | IfThenElse Exp Exp Exp
         | Exp :>: Exp
         | Exp :<: Exp
         | Exp :==: Exp
           deriving(Eq,Show)
```

To keep the example small this data-type combines integer and boolean expressions. The boolean values *True* and *False* are represented by the objects `Literal 1` and `Literal 0`.

Overloading infix operators $+,-$ and $*$ makes working with the `Exp` data structure easy and beautiful. In Haskell, types that you can use $+,-$ and $*$ on are members of the type class `Num`. Making an instance of that class for the `Exp` type is shown below:

```
instance Num Exp where
    (+) = Add
    (-) = Sub
    (*) = Mul
    signum a = IfThenElse (a :<: 0) (-1) 1
    abs    a = IfThenElse (a :<: 0) (-a) a
    negate a = 0 - a
    fromInteger = Literal . fromInteger
```

With this machinery in place we can switch from writing `Add (Literal 7) (Literal 3)` to simply writing $7 + 3$. Unfortunately overloading $<$,$>$ and

== is not possible because of how the type class `Ord` is implemented. But having :<:, :>: and :==: instead is not that much worse.

The last piece is the actual interpreter, here called `run`, “runs” an expression and presents its result.

```
run :: Exp -> Int
run (Literal i) = i
run (Add i j)   = run i + run j
run (Sub i j)   = run i - run j
run (Mul i j)   = run i * run j
run (IfThenElse a b c) = if (run a == 1) then run b
                                          else run c
run (a :<: b)   = if (run a < run b) then 1 else 0
run (a :>: b)   = if (run a > run b) then 1 else 0
run (a :==: b)  = if (run a == run b) then 1 else 0
```

Below is an example of what using these integer expressions looks like in an interactive session using for example ghci or hugs.

```
> run ((5+2) * 3)
21
> (5+2) * 4 :: Exp
Mul (Add (Literal 5) (Literal 2)) (Literal 4)
```

This section showed a few of the features that make Haskell a suitable host language. Overloading and infix constructors are features that make the embedded language clean and elegant.

3 An embedded language for data-parallel programming

This section describes the implementation of an embedded language for data-parallel programming in Haskell. From now on this language will be called Obsidian. CUDA C programs are generated when the Obsidian programs are executed. The CUDA code that is generated is designed to be executed by one thread block in the GPU. This means that the generated code will run on only one of the multiprocessors, 8 SIMD processors, in the 8800 GPU. The Programs generated use one thread per array element and are thus limited to array sizes of 512 elements which is the maximum number of threads in a block [1]. Each thread in the generated code has a duty, the duty to store a value into an array at a position decided by that thread's thread id. This means that for example a swap operation, that could really have been performed by a single thread, is done cooperatively by two threads. This does result in some duplication of work, but allowing a single thread to perform a swap operation would mean that the choice of what task a thread performs need to change as well. It would be necessary for threads to be able store values in two different memory locations. There are no limitations in the hardware that would make that view impossible. Exploring this approach is left as future work.

During the implementation of Obsidian several design choices have been made. The most interesting choice was concerning the synchronisation primitive. In CUDA there is only one synchronisation function and it synchronises across all the threads. The result is that CUDA does not allow the synchronise function in conditional code, unless that conditional evaluates identically across all threads [1]. This means that if Obsidian is supposed to have a more general view on synchronisation, there is need for a transformation step, turning the program into a form that can be executed on the GPU. There is an example of this in a coming subsection about synchronisation.

3.1 Programming style

The idea of Obsidian, is to use a programming style similar to the one in Lava for expressing data-parallel computations. Combinators will be used to piece together programs into larger programs; where in Lava they are used to piece together circuits into larger circuits. The first most visible difference to Lava programs are the occurrences of a primitive called `sync` that appears often in Obsidian programs. To illustrate this, a Lava program and its Obsidian counterpart are given. This program defines the *shuffle exchange network* that will be revisited later in the section about applications.

First the Lava program that defines the shuffle exchange network:

```
shex c n = rep n (riffle ->- evens c)
```

And below is the corresponding Obsidian program:

```
shex f n = rep n (riffle ->- evens f ->- sync)
```

Most of the combinators and functions used in the two programs above have already explained in the Lava section. The repeat combinator, `rep`, repeats a program a given number of times. In the code generated by Obsidian this results in a for loop wrapped around the program. The `sync` primitive will be explained in more detail later and can here be thought of as the operation that finalises the changes made to the array.

A difference between Lava and Obsidian is that Lava generates a fixed size circuit. Using Obsidian it is possible to generate programs for a fixed array size or to generate programs that are parametric in the length of the array. As an example the following program that reverses an array, can be applied to arrays of any length (<= 512):

```
reverse = rev ->- sync
```

From this Obsidian program the following CUDA C program is generated:

```
__global__ static void reverse(int *values, int n)
{
  extern __shared__ int shared[];
  const int tid = threadIdx.x;
  int tmp;
  shared[tid] = values[tid];
  __syncthreads();

  tmp = shared[((n - 1) - tid)];
  __syncthreads();
  shared[tid] = tmp;
  __syncthreads();

  values[tid] = shared[tid];
}
```

And as an example of a Obsidian program that generates code for fixed size instances the `shex` program from above can be used.

This section showed a brief introduction to the programming style that is aimed for with Obsidian. The following sections show selected parts of its implementation. More details on the implementation are also given throughout the section on applications.

3.2 Arrays

Obsidian describes programs operating on arrays. To represent an array the type `Arr a` is used:

```
type Arr a = (IxExp -> a, IxExp)
```

The first element of this tuple is a function from indices to values. The second element represents the length of the array. In most cases the `a` in the type above will be of an expression type. The indexing function is where most of the semantics of the Obsidian program will be accumulated when executed. As an example the function `rev` that reverses an array is implemented as:

```
rev :: Arr a -> W (Arr a)
rev arr =
    let n = len arr
    in  return $ mkArray (\ix -> arr ! ((n - 1) - ix)) n
```

The function `mkArray` creates an array with a given indexing function and a length. The operator `!` is used to index into an array and the function `len` returns the expression that represents the length of the array. The return type of `rev` above, `W (Arr a)`, is Monadic [21]. This is the case for all Obsidian functions. A Monad is a type constructor with the following operations defined on it: bind (>>=), sequence (>>), return and fail. These operations should also obey a few laws known as the monad laws. These laws state that return is left and right identity of bind and that bind is associative. Monads are used in Haskell for amongst other things stateful computations and IO. The monad `W` in the example above is a special implementation of the Writer monad.

Below is the implementation of the the apply function, in Obsidian called `fun`:

```
fun :: (a -> b) -> Arr a -> W (Arr b)
fun f arr = return $ mkArray (\ix -> f (arr ! ix)) (len arr)
```

The above two functions both make changes to the indexing function of the array representation. The reverse function changes how elements are indexed and the apply function operates on the element indexed using it. Some of the Obsidian functions also alter the length of an array. An example of such an operation is `halve`:

```
halve :: Arr a -> W (Arr a, Arr a)
halve arr =
    let n = len arr
        nhalf = idiv n 2
        h1 = mkArray (\ix -> arr ! ix)  (n - nhalf)
        h2 = mkArray (\ix -> arr ! (ix + (n - nhalf))) nhalf
    in  return (h1,h2)
```

The function `halve` above returns a pair of arrays of half the length of the original one. An important issue here is that Obsidian programs must be length homogeneous. This means that even a program that reduces an array to a single value, must be designed so that the program produces a resulting array of the same length as the given input. The desired value can be stored for example at index 0 in the result array. There is an example of a reduction like this in the section describing applications. It is also not possible to write programs that take as input an array of length n and produce as result an array of length $2n$. A easy way around that is of course to use an array of length $2n$ to start with but where n of the elements are only used as target.

Related to `halve` is the function `conc` that concatenates two arrays into a single one. This is an example of a operation that introduces a conditional into the generated expression:

```
conc :: Choice a => (Arr a, Arr a) -> W (Arr a)
conc (arr1,arr2) =
    let (n,n') = (len arr1,len arr2)
    in return $ mkArray (\ix -> ifThenElse (ix <* n)
                                (arr1 !  ix)
                                (arr2 !  (ix - n))) (n+n')
```

3.3 Expressions

To represent expressions in Obsidian a type called `DExp` is used. It is called `DExp` for dynamic expression since expressions of different types can be represented by the same data type. This follows the approach from Compiling

Embedded Languages [16].

```
data DExp = LitInt Int
          | LitBool Bool
          | LitFloat Float
          | Op2 Op2 DExp DExp
          | Op1 Op1 DExp
          | If DExp DExp DExp
          | Variable Name Type
          | Index DExp DExp
          deriving(Eq,Show)
```

To be able to work in a safer typed environment the phantom types technique is used [20]. This is where a new type is created with a parameter “a”:

```
data Exp a = E DExp
```

Now lifting the operations to the `Exp` level gives a typed environment to work in. The example below shows the integer operations div and mod lifted into the `Exp` type:

```
idiv,imod :: Exp Int -> Exp Int -> Exp Int
idiv = lift2 (Op2 Div)
imod = lift2 (Op2 Mod)
```

Making `Exp Int` an instance of `Num` and overloading the operators $+$,$-$ and $*$ gives them the proper types as well. That is, the type system will protect us from adding something of type `Exp Int` to something of type `Exp Float`. There is also a similar instance declaration for `Exp Float`.

3.4 Functions

Obsidian functions are Haskell functions defined on the types supplied by Obsidian. This is an example of where a feature of the host language is directly used by the embedded language. In Haskell, functions are often defined recursively. But in Obsidian defining functions recursively should be done very sparingly. Recursive functions lead to large expressions and depending on what operations are involved in the recursion, a possibility of expressions with a high number of conditionals. The number of conditionals should generally be kept to a minimum in GPU programs.

3.5 Abstract C code

When an Obsidian program is executed, an internal representation of C code is generated. This abstract representation of C is later used to generate the actual CUDA code that can be compiled for the NVIDIA 8800 GPU. The data structure that carries the representation of C is called `AbsC`:

```
data AbsC = Skip
          | For AbsC DExp AbsC AbsC
          | NewVar DExp DExp
          | AbsC :>> AbsC
          | IfThenElse DExp AbsC AbsC
          | Synchronize
          | DExp := DExp
```

An element of the above type is not generated directly. First an element of a type called `Code` is generated. The `Code` type represents functions from index expressions to `AbsC`:

```
type Code = IxExp -> AbsC
```

3.6 Synchronisation

In Obsidian synchronisation is done using the `sync` primitive. Here is a small example showing its use:

```
rev ->- sync
```

The above program reverses an array. When sync is reached, a representation of the program up to that point is written into the `W` monad.

To illustrate how the CUDA synchronise primitive has influenced the design of Obsidian, two very similar Obsidian programs will be used. The first of these programs is directly executable on the GPU, while the second needs a transforming step to be executable. These programs use the combinator `two`, that split an array in two halves and applies a function to each half. Depending on what operation is passed to `two` it may generate code containing an if statement. Below is the definition of two in terms of `parl`:

```
two f = parl f f
```

In the first program sync is used in a way that directly corresponds to CUDA's view on synchronisation. From this program CUDA code can be generated directly:

```
two rev ->- sync
```

From the Obsidian program above, the following CUDA program is generated:

```
__global__ static void reverses(int *values, int n)
{
  extern __shared__ int shared[];
  const int tid = threadIdx.x;
  int tmp;
  shared[tid] = values[tid];
  __syncthreads();
  tmp = ((tid < (n - (n / 2))) ?
         shared[(((n - (n / 2)) - 1) - tid)] :
         shared[((((n / 2) - 1) - (tid - (n - (n / 2)))) + (n - (n / 2)))]);
  __syncthreads();
  shared[tid] = tmp;
  __syncthreads();

  values[tid] = shared[tid];
}
```

Below is the second, similar but forbidden, program:

```
two (rev ->- sync)
```

This program will result in CUDA code with a call to the synchronisation function within a conditional statement which CUDA does not allow.

```
__global__ static void reverses(int *values, int n)
{
  extern __shared__ int shared[];
  const int tid = threadIdx.x;
  int tmp;
  shared[tid] = values[tid];
  __syncthreads();
  if ((tid < (n - (n / 2)))){
    tmp = shared[(((n - (n / 2)) - 1) - tid)];
    __syncthreads();
    shared[tid] = tmp;
    __syncthreads();

  }
  else {
    tmp = shared[((((n / 2) - 1) - (tid - (n - (n / 2)))) + (n - (n / 2)))];
    __syncthreads();
    shared[tid] = tmp;
    __syncthreads();
  }

  values[tid] = shared[tid];
}
```

During the development of Obsidian there was a version capable of transforming the above function to a form executable on the GPU. This was done through a post processing step on the abstract representation of C code. This approach also needed additional information stored in the array representation. The attempts to finish this approach was discontinued due to lack of time. For future work I think that what kind of `sync` primitive to use should be investigated further. It is possible that there is a way to do without the `sync` entirely, but that is not considered here.

3.7 Generating CUDA code

This section shows how CUDA code is generated from an Obsidian program. To keep it concrete an example will be used. Code will be generated from the program:

```
rev ->- sync
```

The first step is to run this Obsidian program and in the process generate an object of type `Code`. To run a program like this, it is applied to a symbolic array. The symbolic array represents indexing into a named concrete array at an index.

```
symArray :: Arr (Exp Int)
symArray = (\ix -> (index (variable ‘‘shared’’ Int) ix),
            variable ‘‘n’’ Int)
```

When sync is reached an element of type `Code` is generated and written into the `W` monad. The CUDA code to be run on the GPU is generated from this `Code` object. The function of type `Code` is applied to an index expression that symbolises thread id:

```
threadID = variable ‘‘tid’’ Int
```

In the CUDA code generated, the variable `tid` is assigned a value representing the thread's id. When applying a function of type `Code` to `threadID` the result is an object of type `AbsC`.

Now the object of type `AbsC` can be used to generate the CUDA code. Having a complete program as an object in Haskell opens some doors for optimisations. It is possible to apply transformations to the `AbsC` object to generate a more efficient but equivalent program. In this version no such transformations are applied; instead the CUDA code is generated directly from the `AbsC` representation.

The CUDA code generated from `rev ->- sync` looks like this:

```
__global__ static void reverse(int *values, int n)
{
  extern __shared__ int shared[];
  const int tid = threadIdx.x;
  int tmp;
  shared[tid] = values[tid];
  __syncthreads();

  tmp = shared[((n - 1) - tid)];
  __syncthreads();
  shared[tid] = tmp;
  __syncthreads();

  values[tid] = shared[tid];
}
```

From the C program above the kind of programs generated by Obsidian can be explained. As hinted in the introduction to this section each thread has the duty to store a value to a location decided by its thread id. Each thread reads a value into the variable `tmp`. This is directly followed by a synchronize statement; making sure that all threads has computed their local `tmp`. Following this is the update of memory and another sync that makes sure that all memory locations are updated before progressing further. The result is a C program that contains 3 of CUDA's `__syncthreads()` statement where the Obsidian program it is generated from contains only a single `sync`.

In general the Obsidian `sync` operation generates code as follows:

```
tmp = ...;
__syncthreads();
shared[tid] = tmp;
__syncthreads();
```

This accounts for 2 of the 3 `__syncthreads()` above, the one occuring first in the listing is part of the boilerplate code and is there to make sure the entire array is in shared memory before starting any computations on it.

3.8 Interactive sessions

A nice feature of many Haskell systems is the interactive sessions. These interactive sessions are excellent for testing and prototyping ideas quickly and easily. Lava made strong use of this feature. Obsidian has two functions that are useful when working in ghci, called `execute` and `emulate`. The function `execute` runs a program on the GPU while `emulate` runs the program on the computers CPU. Both functions take as argument a program to run and a list. The list is turned into an array in the generated code. The generated program is executed on the GPU or the CPU and the result is turned into a Haskell list again and is presented.

```
*Obsidian> execute (rev ->- sync) [1..8]
[8,7,6,5,4,3,2,1]
*Obsidian> emulate (rev ->- sync) [1..8]
[8,7,6,5,4,3,2,1]
```

4 Testing framework

QuickCheck is a system for testing Haskell programs on random data developed by Koen Claessen and John Hughes at Chalmers. QuickCheck defines two embedded languages in Haskell, one language for formal specifications and one for test data generation [13]. This section describes a framework for testing the implementation of Obsidian and for testing applications implemented using it. Throughout this section a select set of representative properties are tested against parts of Obsidian and programs implemented using it. In the coming section on applications the testing framework is used again to gain confidence in the implementations.

4.1 Testing by executing on the GPU

The functions `execute` and `emulate` give the opportunity to test Obsidian programs using regular Haskell lists. As an example the reverse program from above will be tested using the `emulate` function.

```
rev ->- sync
```

The above program will be tested against the Haskell standard reverse function. The following property states that the result of emulating `rev ->- sync` on a list should be the same as the reverse of that same list.

```
prop_reverse =
    forAll (sized vector) $ \xs ->
        emulate (rev ->- sync)  xs == reverse xs
```

Now it is possible to check this property using QuickCheck:

```
> quickCheck prop_reverse
OK, passed 100 tests.
```

Because of the way `emulate` and `execute` are implemented running 100 tests using QuickCheck can take several minutes. This is because the program is generated and compiled at each call to `emulate` or `execute`. An easy way around this would be to have other run functions that generate the program and compiles it once, then running it on different data. This approach is not followed here and is left as future work. Another way that is described in the following section is to test properties using the expressions contained in the `DExp` type.

4.2 Testing at the DExp level

To be able to test properties of combinators without running them on the GPU a `DExp` evaluator was written. Since most of the semantics of a program is carried by the expression built during execution, being able to test properties on these expressions is important. This way of testing properties is only applicable to programs that do not use the `sync` primitive. Applying this technique to programs that use `sync` would give vacuous results.

To allow QuickCheck to create arbitrary indices, a generator `arbIx` is written:

```
arbIx (E (LitInt len)) = do
  n <- choose (0,(len - 1))
  return (E (LitInt n))

arbLength = do
  n <- choose (1,max_length)
  return (E (LitInt n))
```

Indices must be within the bounds of the array at hand. The index generator therefore takes the length of an array as argument. When testing properties this way the arrays used are defined by their lengths. The indexing function is initially just the identity. The `arbLength` generator is used to ensure that the values used as array lengths are positive.

Now QuickCheck can be used to test properties such as that `rev ->>- rev` is equal to the identity function:

```
prop_revrev =
    forAll arbLength $ \len  ->
        forAll (arbIx len) $ \ix ->
            let arr = mkArray (\ix -> ix) len
            in prop_programsEqual (rev ->>- rev) (idM) arr ix
```

In `prop_revrev` above the property `prop_programsEqual` is used. This property uses the evaluate function on the two programs and tests that the resulting values are identical:

```
prop_programsEqual p1 p2 arr ix =
    evaluate p1 arr ix == evaluate p2 arr ix
```

Another property that can be tested at this moment is that n riffles gives the identity function on arrays of length 2^n. This is tested in the following property:

```
prop_nriffle =
    forAll lengths $ \(len,loglen) ->
       forAll (arbIx len) $ \ix ->
          let arr = mkArray (\ix -> ix) len
              n   = toInt len
          in  prop_programsEqual (composeN loglen riffle) idM arr ix
       where composeN 0 f = idM
             composeN n f = f ->- composeN (n-1) f
             lengths = do i <- choose (1,10)
                          return (E(LitInt (2^i)),i)
```

To be able to test properties of the more advanced combinators such as `two` and `ilv` it is necessary to generate arbitrary programs as well. To do this a data type that represents programs built using the standard combinators is created. After making this data type an instance of Arbitrary, QuickCheck can generate programs to use in the testing phase. There is also a function called `toProgram` that goes from this representation of a program to a real program.

```
data ProgramExp = Id
                | Seq ProgramExp ProgramExp
                | Rev
                | Sort2
                | Riffle
                | Unriffle
                | Two ProgramExp
                | Ilv ProgramExp
                | Parl ProgramExp ProgramExp
                  deriving (Show)
```

Now it is possible to test for example that `two` distributes over sequential composition:

```
prop_distTwo x y =
    forAll arbLength  $ \len ->
       forAll (arbIx len) $ \ix ->
          let (x',y') = (toProgram x, toProgram y)
              arr = mkArray (\ix -> ix) len
          in  (prop_programsEqual (two (x' ->- y')) (two x' ->- two y')) arr ix
```

When testing this property, QuickCheck supplies two arbitrary elements of the type `ProgramExp`. These are turned into regular programs using the `toProgram` function.

In the section about Lava, it was stated that applications of `ilv` and `two` commute. In Lava this property is true without reservations, but in Obsidian the property has to be stated in the following contrived fashion:

```
prop_twoIlv x =
       forAll lengths  $ \len ->
          forAll (arbIx len) $ \ix ->
            let x' = toProgram x
                arr = mkArray (\ix -> ix) len
            in  (prop_programsEqual (two (ilv x')) (ilv (two x'))) arr ix
        where lengths = do i <- choose (1,100000)
                           return (E (LitInt (2^(2+(depth x))*i)))
```

This is because of differences in the semantics of the combinators `ilv` and `two` in Obsidian and Lava. In Obsidian an array is given as input to a program, the combinators `two` and `ilv` split the given array in two and apply a function to each halve. In Lava the combinator `two` takes an n input circuit and the result is a $2*n$ input circuit. The property given above `prop_twoIlv` states that `two` and `ilv` does commute, given the array used as input is of a particular size. The size of array needed is decided by analysing the depth of the argument given to `two` and `ilv` in the property:

```
depth (Seq a b)  = max (depth a) (depth b)
depth (Two a)    = 1 + depth a
depth (Ilv a)    = 1 + depth a
depth (Parl a b) = 1 + (max (depth a) (depth b))
depth _ = 0
```

This section showed how QuickCheck can be used to test programs implemented using Obsidian as well the implementation of Obsidian itself.

5 Applications

This sections describes the implementation of a small number of applications in Obsidian. The section begins with the implementation of a few sorters followed by an investigation of their efficiency. Lastly the implementation of a reduction operation is described.

5.1 Odd even transposition sort

A periodic sorter is implemented by repeatedly applying a function to an input array. This section describes the implementation of a periodic sorter that is known as odd even transposition sort [12]. The sorter is constructed by composing copies of the following function:

```
sortOETCore = sortEvens ->- sortOdds
   where
     sortEvens = evens (cmpSwap (<*)) ->- sync
     sortOdds  = odds (cmpSwap (<*)) ->- sync
```

The function `cmpSwap` is defined as follows:

```
cmpSwap op (a,b) = ifThenElse (op a b) (a,b) (b,a)
```

It is used in `sortOETCore` above as a two-sorter. The two combinators `evens` and `odds` both apply a function, in this case the two-sorter `cmpSwap <*`, to neighbouring pairs of array elements. They are defined as follows:

```
evens f arr =
    let n = len arr
    in  return $ mkArray (\ix -> ifThenElse ((imod ix 2) ==* 0)
                          (ifThenElse ((ix + 1) <* n)
                                          (fst (f (arr ! ix,arr ! (ix + 1))))
                                          (arr ! ix))
                          (ifThenElse (ix <* n)
                                          (snd (f (arr ! (ix - 1),arr ! ix)))
                                          (arr !  ix))) n



odds f = endS 1 (evens f)
```

In the listing above, `odds` is implemented as applying `evens` to an endsegment of the array. The two figures 6 and 7 shows how the two combinators `evens` and `odds` can be visualised.

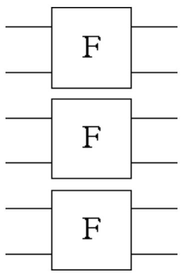

Figure 6: Evens

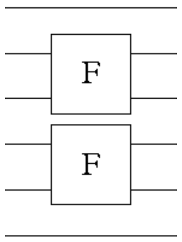

Figure 7: Odds

To construct the odd even transposition sort for arrays of length $2n$, n copies of the function `sortOETCore` above are composed. This composition is done using the `rep` combinator that repeats a program a number of times. Figure 8 shows a small instance of the periodic sorter. The sorter is implemented as follows:

```
sortOET n = rep (div n 2) sortOETCore
```

And the combinator `rep` is implemented like this:

```
rep m prg arr =
    do
      a <- newInteger
      let n = (E (LitInt m))
          update = ((unE a) := (Op2 Add (unE a) (LitInt 1)))
      (_,c) <- local (prg arr)
      write (\ix -> (for (declare a 0) (a  <* n) update (c ix)) )
      return arr
```

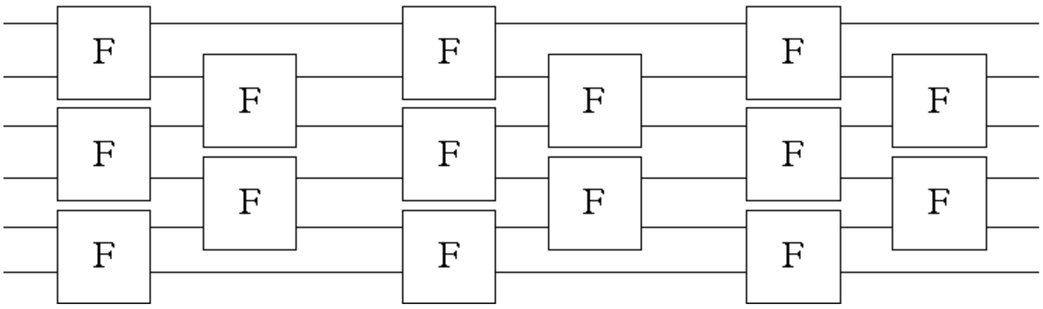

Figure 8: Odd even transposition sort for arrays of length six.

QuickCheck is used to test this implementation of odd even transposition sort. The following properties are defined to test the `sortOET` program using either the CPU or the GPU:

```
-- test sortOET using the CPU
prop_sortOETEmu =
    forAll (sized vector) $ \xs ->
      emulate (sortOET (length xs)) xs == sort xs

-- test sortOET using the GPU
prop_sortOET =
    forAll (sized vector) $ \xs ->
      execute (sortOET (length xs)) xs == sort xs
```

Testing the above properties with QuickCheck fails and the counter example presents a list of odd length. The problem is that this implementation does not generate sorters for odd length arrays correctly. Here is a modified version that generates a sorter for odd length arrays as well:

```
sortOET n = rep (div n 2) sortOETCore ->-
            if (mod n 2 /= 0)
            then evens (cmpSwap (<*)) ->- sync
            else idM
```

With these additions both odd and even length sorters can be generated.

Performing the test in a GHCI interactive session looks like this:

```
> quickCheck prop_sortOET
OK, passed 100 tests.
> quickCheck prop_SortOETEmu
OK, passed 100 tests.
```

The generated sorter operates on arrays of fixed length. It is also possible to write an Obsidian program that generates a sorter wich is parametric in the length of the array. This is done using a combinator very similar to `rep` called `repE` which is defined as follows:

```
repE :: Exp Int -> Program a a -> Program a a
repE n prg arr =
    do
      a <- newInteger
      let update = ((unE a) := (Op2 Add (unE a) (LitInt 1)))
      (_,c) <- local (prg arr)
      write (\ix -> (for (declare a 0) (a  <* n) update (c ix)) )
      return arr
```

The only differance compared to `rep` is that in `repE` the argument that decided the number of time to repeat is an expression. Using `repE` a parametric version of odd even transposition sort can be implemented as follows:

```
sortOETParam arr =
    let n = len arr
    in (repE (idiv (n+1) 2) sortOETCore) arr
```

Using $n + 1$ to make sure it repeats `sortOETCore` an extra time when operating on an array of odd lenght. By doing this the sorter will perform a little unnecessary work but given the nature of the target platform this is probably better than introducing another conditional into the program. Below is a listing of the CUDA code generated from `sortOETParam`:

```
__global__ static void sortOET(int *values, int n)
{
  extern __shared__ int shared[];
  const int tid = threadIdx.x;
  int tmp;
  shared[tid] = values[tid];
  __syncthreads();
  for (int i0 = 0;(i0 < ((n + 1) / 2));i0 = (i0 + 1)){
    tmp = (((tid % 2) == 0) ?
           (((tid + 1) < n) ?
            ((shared[tid] < shared[(tid + 1)]) ?
             shared[tid] :
             shared[(tid + 1)]) :
            shared[tid]) :
           ((tid < n) ?
            ((shared[(tid - 1)] < shared[tid]) ?
             shared[tid] :
             shared[(tid - 1)]) :
            shared[tid]));
    __syncthreads();
    shared[tid] = tmp;
    __syncthreads();
    tmp = ((tid < 1) ?
          shared[tid] :
           ((((tid - 1) % 2) == 0) ?
            ((((tid - 1) + 1) < (n - 1)) ?
             ((shared[((tid - 1) + 1)] < shared[(((tid - 1) + 1) + 1)]) ?
              shared[((tid - 1) + 1)] :
              shared[(((tid - 1) + 1) + 1)]) :
              shared[((tid - 1) + 1)]) :
            (((tid - 1) < (n - 1)) ?
             ((shared[(((tid - 1) - 1) + 1)] < shared[((tid - 1) + 1)]) ?
              shared[((tid - 1) + 1)] :
              shared[(((tid - 1) - 1) + 1)]) :
             shared[((tid - 1) + 1)])));
    __syncthreads();
    shared[tid] = tmp;
    __syncthreads();

  }
  values[tid] = shared[tid];
}
```

It is obvious that the CUDA code listed above can be optimised. At this time no such optimisations are performed by Obsidian but rather it is left up to the CUDA compiler. Experiments indicate that performing some optimisations on the code before passing it to the CUDA compiler can have positive effect. Introducing such optimisations into Obsidian is left as future work.

5.2 Bubblesort

Bubble sort is another example of a sorting network, figure 9 shows a small instance of the network. To implement bubble sort, new combinators are needed. As a first try the combinator `stairD` was implemented, it is described in figure 10.

Using `stairD` it is possible to implement bubble sort by repeatedly applying the function:

```
stairCore n = stairD n (cmpSwap (<*))
```

To make bubble sort for arrays of length n, n instances of `stairCore` is sequenced:

```
stairSort n = rep n (stairCore n)
```

The resulting `stairSort` can be visualised as in figure 11. This differs from the figure 9 in several ways. First there are three unnecessary applications of `cmpSwap` in the particular case that is described in the figures. But even worse is that there is no parallelism in the figure. Only two threads will be working at any time in the `stairSort` program. Figure 11 shows the unnecessary two-sorters. To make a more parallel version of bubble sort A new combinator `bubblenet` was implemented, using two for loops. With this combinator a parametric version of bubble sort can be implemented as:

```
bubbleSort = bubblenet (cmpSwap (<*))
```

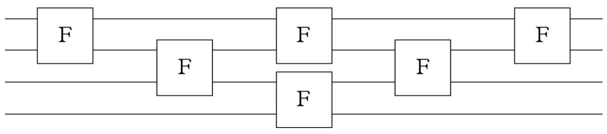

Figure 9: The bubble sort network

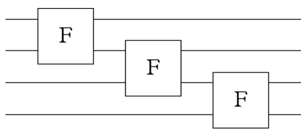

Figure 10: The stairD combinator

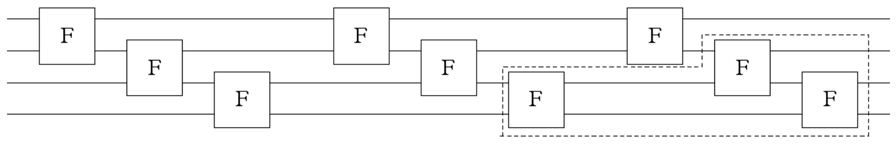

Figure 11: StairSort for arrays of length 4

5.3 Iterative vsort

This section describes the implementation of a more efficient sorter using Obsidian. This sorter has a depth of $log^2(n)$, where the two other sorters described, odd even transposition sort and bubble sort, have a depth of n. Vsort is built around the shuffle exchange network mentioned earlier. This network is equivalent to the butterfly network described in the Lava section, as shown in [17]. The following listing defines the shuffle exchange network:

```
shex f n = rep n (riffle ->- evens f ->- sync)
```

It is also possible to define the shuffle exchange network in another way that gives a parametric version of the network. Implemented like this the shuffle exchange network is applicable to arrays of length a power of two:

```
pshex f arr =
    let n = log2i (len arr)
    in  repE n (riffle ->-  evens f ->- sync) arr
```

The merger needed for vsort is implemented as:

```
bmergeIt n = shex (cmpSwap (<*)) n
```

Figure 12 shows an instance of the merger defined above.

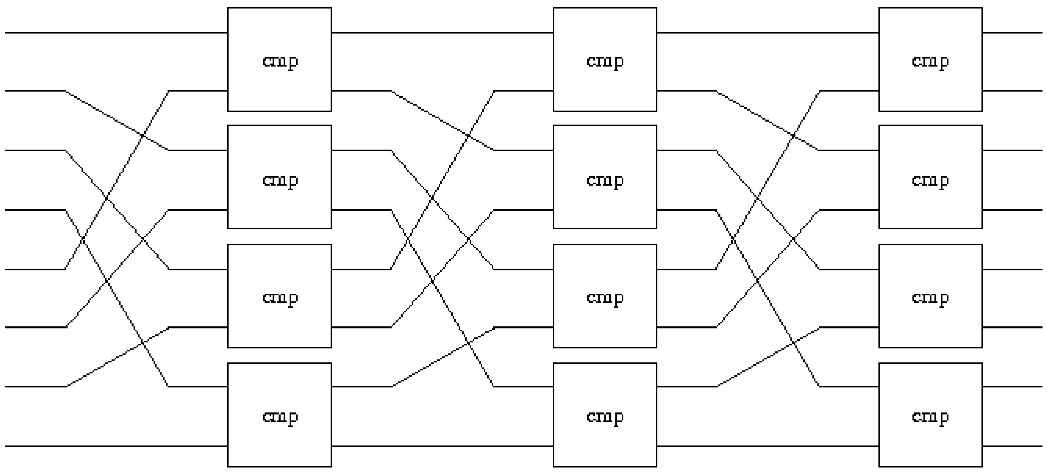

Figure 12: Merger implemented using a shuffle exchange network.

Now vsort is implemented using the merger defined above and an index permutation called tau.

```
one f = parl idM f

tau 0 = idM
tau n = one rev ->- two (tau (n-1))

vmergeIt n = tau (n-1) ->- sync ->- bmergeIt n

vsortIt n = rep n (vmergeIt n)
```

The application `vsortIt 4` generates a sorter for arrays of length 16. Unfortunately 4 is the largest argument that can be used in generating a sorter of this kind. This is because of the size of the expression that is generated during recursion of the function `tau`. An argument larger than 4 generates a CUDA program that fails during compilation.

The solution to this problem is given by using a pregenerated table that represents the `tau` permutation of the array. A new Obsidian function is needed that transforms an index by using a table. This function is called tblLook:

```
tblLook table arr =
    return $ mkArray (\ix -> arr ! (table ! ix)) (len arr)
```

Using the tblLook a new shuffle exchange combinator can be implemented using a riffle table:

```
shexT f n = rep n (tblLook rifftab ->- sync ->- evens f ->- sync)
    where
      rifftab = mkArray (\ix -> index (variable "riffle_table" Int) ix) (variable "n" Int)
```

Optimised versions of bmergeIt and vmergeIt can be implemented as follows:

```
bmergeItTab n = shexT (cmpSwap (<*)) n


vmergeItTab n = tblLook tautab ->- sync ->- bmergeItTab n
    where
      tautab = mkArray (\ix -> index (variable "tau_table" Int) ix) (variable "n" Int)
```

And now an optimised version of vsort that uses tables is implemented:

```
vsortItTab n = rep n (vmergeItTab n)
```

Now to be able to generate CUDA code from the Obsidian program above, new code generation routines are implemented. In the generated code, the tables are stored in what is called the "constant memory" which is a cached area of the graphic cards memory. A new execute function called executeTab is implemented as well. The function executeTab takes as an extra argument, a list of (Name,[Int]) pairs containing table name and a list of integers representing the table.

Now all that is needed are the actual tables, *riffle_table* and *tau_table*. These tables are generated by ordinary Haskell functions operating on lists. Here the functions that generates the tables are called t_tau and t_riffle.

Now vsortItTab can be run using executeTab as follows:

```
> executeTab (vsortItTab 2) [("riffle_table",t_riffle 4),("tau_table",t_tau 2)] [4,3,2,1]
[1,2,3,4]
```

Vsort is another example of an Obsidian program from which a fixed size instance of a sorter can be generated. In this case I think it would be considerably harder to write a parametric version. The major obstacle here are the tables; that are pregenerated and of fixed size. But using another permutation than the *tau* permutation used above, it is possible to generate a less efficient but parametric version of vsort. This sorter can be used on arrays of length a power of two:

```
pbmergeIt = pshex (cmpSwap (<*))

pdmergeIt = tauDowd' ->- sync ->- pbmergeIt

pvsortIt arr =
    let n = log2i (len arr)
    in (repE n pdmergeIt) arr
```

This sorter uses the parametric shuffle exchange network mentioned in the beginning of this section together with a new index permutation called `tauDowd'` that is implemented as follows:

```
tauDowd' = unriffle ->- sync ->- one rev
```

This results in a parametric sorter, but it is less efficient than the previous `vsortIt`. The previous sorter is more efficient because of how the tables are hiding a lot of the work, such as occurences of `parl` which have a negative impact on performace.

5.4 Efficiency of generated sorters

The figures presented here are obtained using the following hardware:

```
CPU: Intel Core 2 Duo 2,4GHz
GPU: NVIDIA 8800 GTS 1.2GHz
```

The dataset used in the tests was 288MB of random data. This dataset was split into batches of 512 32bit elements. Each batch of 512 elements was then sorted individually. Figure 13 shows the running time of the sorters presented earlier, Odd Even Transposition Sort and three different versions of Vsort, using one block of threads. The sorters are thus running on one of the multiprocessors, 8 SIMD processing elements, in the GPU. All running times where obtained using the Unix `time` command. Below are short descriptions of all the sorters used in the comparison:

bitonicCPU is an implementation of bitonic sort for the CPU. The implementation is adapted from one shown in [28].

sortOET is implemented in a previous section, Odd even transposition sort.

vsortIt is the sorter described in the previous section, Iterative vsort.

vsortIt2 is very similar in appearance to the previously described "vsortIt" except that an extra `sync` has been inserted in **shexT** as follows:

```
shexT f n = rep n (tblLook rifftab ->- sync ->- evens f ->- sync)
```

It is interesting to see that this has a positive effect on execution time.

vsortHO is a hand optimised version of vsort. This version is actually quite different from the different versions of vsort generated by Obsidian. In this version each swap operation is done by a single thread.

bitonicSort is the implementation of bitonic sort supplied by the CUDA SDK.

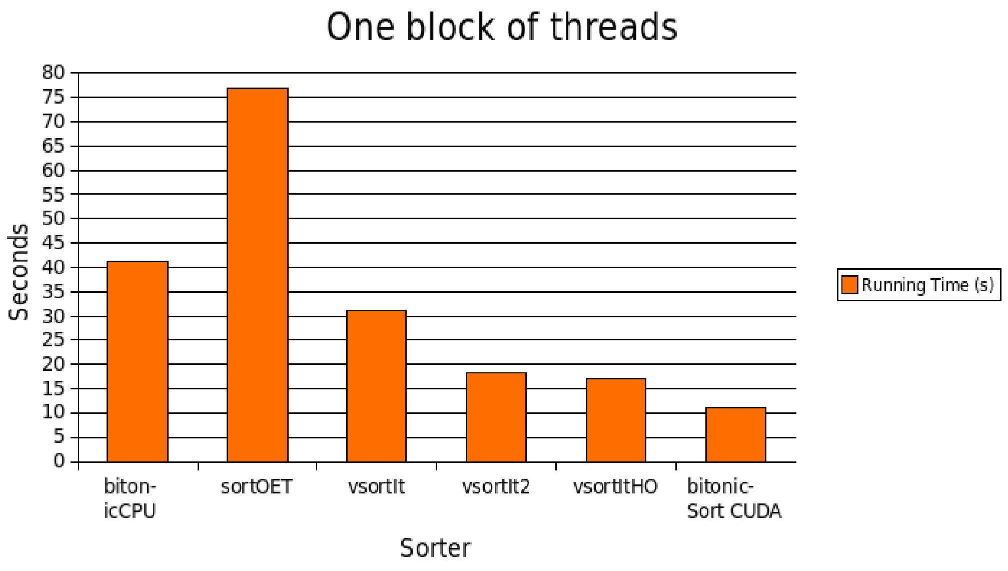

Figure 13: Running time measurements using one block

The chart in figure 13 shows that it is possible to generate a sorter using Obsidian that is close in performance to its hand optimised counterpart.

The difference in performance between the very similar sorters *vsortIt* and *vsortIt2* is a result of the much less complicated expressions in the latter.

The second chart, in figure 14, shows the same sorters again but this time running as many blocks as there are multiprocessors in the GPU used in the experiment. As expected, sorting 12 batches in parallel lead to a big speedup. Since Obsidian generates programs that run in a single block this result is mostly a novelty. But it shows that there is potential to exploit by future versions of Obsidian.

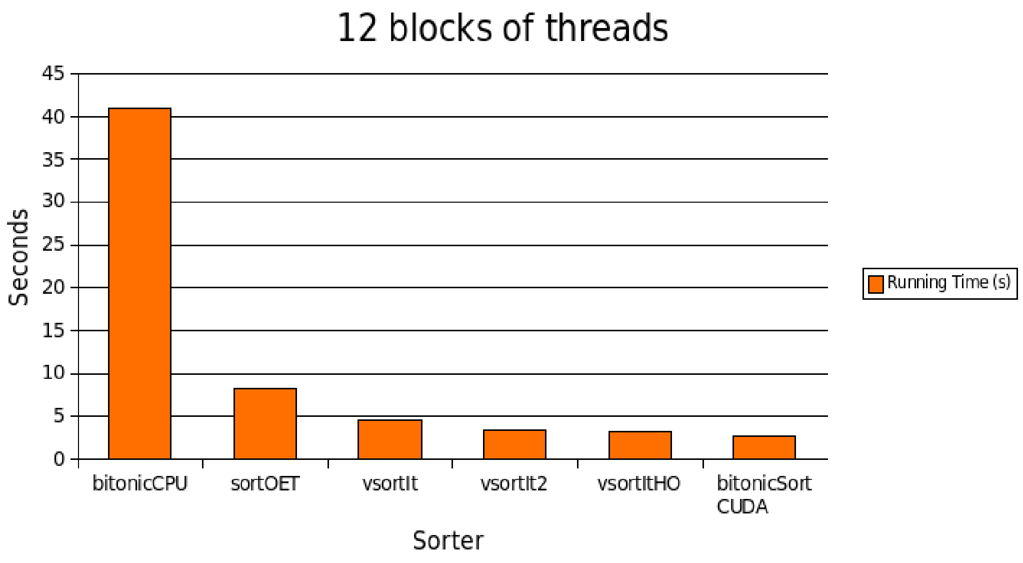

Figure 14: Running time measurements using 12 blocks

The target platform is very sensitive to small differences in the generated code. The number of conditional statements and in what way they are nested has much impact on the performance of the generated program. This is not explored very thoroughly so far in this project. But the following experiment, using four very similar implementations of the sorter called vsortIt, shows that there is considerable difference in performance between very similar implementation of a sorter.

Figure 15 shows the running time of four implementations of vsort. The first two called *vsortIt* and *vsortIt2* are the same as in the description above. The two new sorters, *vsortItS* and *vsortItS2* differs only in that they use a function called `sort2` where the previously defined sorters use `evens (cmpSwap (<*))`. The `sort2` function is defined as follows:

```
sort2 = leis (pair ->- fun (cmpSwap (<*)) ->- unpair)
```

This difference results in generated code with deeper nesting conditionals as a result from using the more general building blocks, `pair` and `unpair`. The combinator `evens` is a primitive in Obsidian and targets a more specific use.

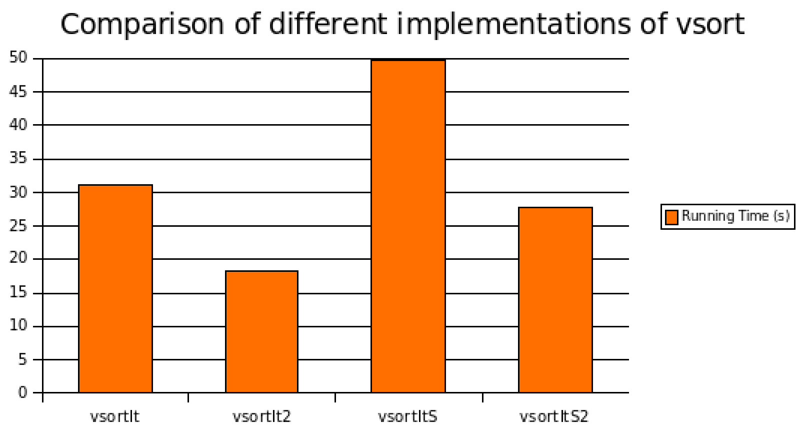

Figure 15: Running time for different implementations of vsort

This section showed the results of measuring the running time of programs generated by Obsidian. The results show that programs generated by Obsidian can be efficient but that it is vital to choose the correct combinators. As future work I think it should be investigated more deeply how combinators should be implemented to fit the target platform better. Apparently small changes in the generated code can affect the performance greatly.

5.5 Reductions

A reduction is an operation that computes a scalar value from an array. This section describes two different implementations of a reduce primitive in Obsidian. Figure 16 shows the concept of reducing an array using a function `F`.

The first attempt to implement the reduction operation uses a symbolic array where the length is replaced with a variable that changes over the execution of the main loop. The program applied to the array in each repetition of the

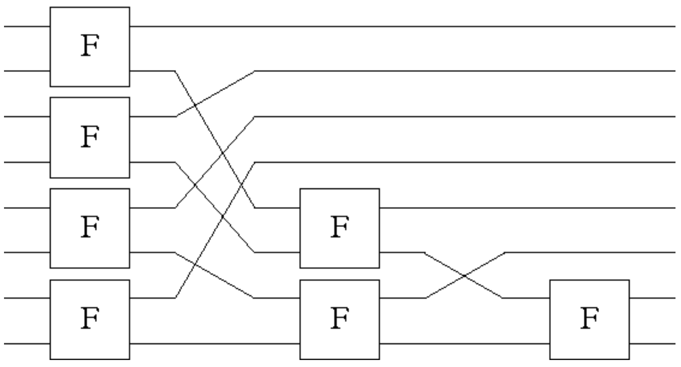

Figure 16: Reduce an array using a function F

loop is built using `evens` and `unriffle`.

```
reduce :: Syncable (Exp a) => ((Exp a,Exp a) -> (Exp a,Exp a)) -> Program (Exp a) (Exp a)
reduce f arr@(ixf,n) =
      do
        i <- newInteger
        s <- newInteger
        let array = mkArray ixf s
            f' = evens f ->- unriffle ->- sync
        (_,c) <- local (f' array)
        write (\ix -> (declare s n) :>>
               for (declare i 1) (i <* n) ((unE i) := (Op2 Mul) (LitInt 2) (unE i))
                 (
                   (c ix) :>> ((unE s)  := ((Op2 Sub) (unE s)
                                           ((Op2 Div) (unE s) (LitInt 2)) ))
                 ) )
        return arr
```

The second attempt uses the `initS` combinator. This combinator applies a program to an initial segment of an array. The argument to `initS` is supplied a variable that is updated each iteration through the loop.

```
reduce2 :: Syncable (Exp a) => ((Exp a,Exp a) -> (Exp a,Exp a)) -> Program (Exp a) (Exp a)
reduce2 f arr@(ixf,n) =
      do
        i <- newInteger
        s <- newInteger
        let f' = (initS s (evens f ->- unriffle)) ->- sync
        (_,c) <- local (f' arr)
        write (\ix -> (declare s n) :>>
               for (declare i 1) (i <* n) ((unE i) := (Op2 Mul) (LitInt 2) (unE i))
                 (
                   (c ix) :>> ((unE s)  := ((Op2 Sub) (unE s)
                                           ((Op2 Div) (unE s) (LitInt 2)) ))
                 ) )
        return arr
```

The function given to `reduce` should follow a certain pattern. It should be a function that operates on a pair of elements giving its result as the first element of the result pair. Here are two functions for use with reduce:

```
addpair (a,b) = (a+b,b)

maxpair (a,b) = (ifThenElse (a >* b) a b, b)
```

A QuickCheck property was defined to check that the two versions of `reduce`, using the `addpair` function, produce identical results:

```
prop_reducesEq =
    forAll lengths $ \len ->
        forAll (vector len) $ \xs ->
            execute (reduce addpair) xs == execute (reduce2 addpair) xs
            where lengths = do n <- choose (0,512)
                               return n
```

This last property tests that a reduction using the `addpair` function gives the same result as the Haskell function `sum`:

```
prop_reduceOk =
    forAll lengths $ \len ->
        forAll (vector len) $ \xs ->
            (head (execute (reduce addpair) xs)) == sum xs
            where lengths = do n <- choose (1,512)
                               return n
```

The reduction operations described here uses the `sync` primitive. Because of this the `reduce` and `reduce2` operations can only be applied at top-level.

6 Related work

This project touches a number of different areas, such as embedded languages, data-parallel programming and the GPGPU area.

Lava is an example of an embedded language written in Haskell. It is from Lava the programming style for Obsidian is derived. Lava has been described in more detail earlier and is only mentioned here.

Pan is an embedded language for image synthesis developed by Conal Elliot. Because of the computational complexity of image generation, C code is generated. This C code can then be compiled by an optimising compiler. Pan is described in the paper [14]. Many ideas from the paper "Compiling Embedded Languages", describing the implementation of Pan where used in the implementation of Obsidian [16].

The two languages above are those that had a more direct impact on this current project. The programming style using combinators has much in common with Lava while the implementation is in debt to Pan.

NESL is a functional data-parallel language developed at Carnegie Mellon university. NESL offers a kind of data-parallelism known as nested data-parallelism. Nested data-parallelism allows a parallel function to be applied over nested data structures, such as arrays of arrays, in parallel. NESL is compiled into a intermediate language called VCode that in turn can be used to generate code for numerous parallel architecture [7]. NESL is described in [8].

Data Parallel Haskell takes the ideas from NESL and incorporates them into the Glasgow Haskell Compiler. Data Parallel Haskell adds a new built-in type of parallel arrays to Haskell. Data parallel programs are expressed as operations on objects of this type. The implementation of Data Parallel Haskell is not complete but is showing promise [9].

NESL and Data Parallel Haskell are examples of where the data-parallel programming model is implemented in a functional setting. Both implement nested data parallelism.

PyGPU is a language for image processing embedded in Python. PyGPU uses the introspective abilities of Python and is in that way bypassing the need to implement new loop structures and conditionals for the embedded language. In Python it is possible to access the bytecode of a function and from that extract information about loops and conditionals [23]. Programs written in PyGPU can be compiled and run on a GPU.

Vertigo is another embedded language by Conal Elliot. Vertigo is a language for 3D graphics that targets the DirectX 8.1 shader model. Vertigo can be used to describe geometry, shaders and to generate textures. Each sublanguage is given formal semantics [15]. From programs written in

Vertigo assembly language programs are generated for execution on a GPU.

Like Obsidian, PyGPU and Vertigo generate code that can be run on GPUs. Though PyGPU and Vertigo are aimed at graphics applications, not GPGPU applications as Obsidian.

7 Discussion and future work

Obsidian, an embedded language for data-parallel programming was implemented using Haskell. A number of applications were successfully implemented using this embedded language. Obsidian presents a high level programming interface where testing an idea is quick and easy. Programs written in Obsidian are short and concise, where the counterpart in the lower level CUDA language can be lengthy. This project has also shown that the style of programming borrowed from Lava, is suitable for data-parallel programming as well. The programming style, using combinators that capture patterns of computations, results in small and elegant programs that really are enjoyable to write.

Throughout the latter half of the development, QuickCheck has been used to test the implementation of Obsidian as well as the applications implemented using it. This has been a valuable aid in the implementation process.

The section on efficiency of generated sorters showed that it is possible to generate efficient code from Obsidian programs. The experiment involving 4 different implementations of the sorter called vsort showed that the choice of and implementation of the combinators used has significant impact on the performance of the generated code. At the moment no optimisations are applied during the code generation. There are a number of techniques for optimising expressions similar to those generated by Obsidian in “Compiling Embedded Languages” [16]. Using optimisation techniques during the code generation is left as future work.

At the moment the generated code operates on arrays with a maximum length of 512 elements. Breaking this barrier in combination with exploiting the full GPU is high priority for future work. Now the code generated by Obsidian is run on a single block of threads and are thus only using a small part of the GPU. Targeting the full GPU will improve the performance of the generated programs. It is possible that a change in programming style is needed when stepping up to the full GPU. There are many issues here to consider, such as that there is no synchronisation between blocks.

Recursive Obsidian functions are not recommended, due to how the expressions generated by such functions tend to be large. Because of this, Obsidian is lacking some expressive power. In the future some control structure to replace recursion is needed. A possible approach is to use combinators that capture common recursive patterns, such as a divide-and-conquer combinator.

Another possible path to investigate in the future is nested data parallelism. Nested data parallelism allows the implementation of divide-and-conquer algorithms [8]. Perhaps this would offer a solution to the previously stated shortcoming as well. I believe a limited form of nested data parallelism could be implemented rather easily using the "block of threads" structure supported by CUDA. However, that direct approach would probably mean that the kind of parallel functions that can be applied to a nested data structure is limited.

References

[1] *NVIDIA CUDA Compute Unified Device Architecture: Programming Guide Version 1.0.*

[2] NVIDIA CUDA SDK v1.0, www.nvidia.com/cuda.

[3] *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann.

[4] *Technical Brief: NVIDIA GeForce 8800 GPU Architecture Overview.*

[5] Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, pages 307–314, 1968.

[6] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware Design in Haskell. In *International Conference on Functional Programming, ICFP*, pages 174–184. ACM, 1998.

[7] G. E. Blelloch and S. Chatterjee. VCODE: A data-parallel intermediate language. In *Proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 471–480, 1990.

[8] Guy E. Blelloch. NESL: A Nested Data-Parallel Language. Technical Report CMU-CS-93-129, April 1993.

[9] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon P. Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. In *DAMP '07: Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 10–18, New York, NY, USA, 2007. ACM Press.

[10] Thomas W. Christopher. Bitonic sort. http://www.toolsofcomputing.com/tc/CS/Sorts/bitonic_sort.htm.

[11] Koen Claessen. Embedded languages for describing and verifying hardware, April 2001. Dept. of Computer Science and Engineering, Chalmers University of Technology. Ph.D. thesis.

[12] Koen Claessen, Mary Sheeran, and Satnam Singh. The design and verification of a sorter core. In *Proc. of Conference on Correct Hardware Design and Verification Methods (CHARME)*, Lecture Notes in Computer Science. Springer Verlag, 2001.

[13] Koen Cleassen and John Hughes. Specification based testing with quickcheck. In *The Fun of Programming*, Cornerstones of Computing, pages 17–40. Palgrave, 2003.

[14] Conal Elliott. Functional images. In *The Fun of Programming*, "Cornerstones of Computing" series. Palgrave, March 2003.

[15] Conal Elliott. Programming graphics processors functionally. In *Proceedings of the 2004 Haskell Workshop*. ACM Press, 2004.

[16] Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(2), 2003.

[17] G. Jones and M. Sheeran. *Collecting Butterflies.* Oxford University Programming Research Group, 1991. ISBN 0-902928-69-4.

[18] Alexander Greß and Gabriel Zachmann. GPU-ABiSort: Optimal parallel sorting on stream architectures. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rhodes Island, Greece, 25–29 April 2006.

[19] Daniel W. Hillis and Guy L. Steele. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, December 1986.

[20] Ralf Hinze. Fun with phantom types. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, pages 245–262. Palgrave Macmillan, 2003. ISBN 1-4039-0772-2 hardback, ISBN 0-333-99285-7 paperback.

[21] Paul Hudak, John Peterson, and Joseph Fasel. A gentle introduction to Haskell 98. http://www.haskell.org/tutorial/, 1999.

[22] Mihai F. Ionescu. Optimizing parallel bitonic sort. In *IPPS*, pages 303–309, 1997.

[23] Calle Lejdfors and Lennart Ohlsson. Implementing an embedded gpu language by combining translation and generation. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1610–1614, New York, NY, USA, 2006. ACM.

[24] Björn Lisper. Data parallelism and functional programming. In *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, pages 220–251, London, UK, 1996. Springer-Verlag.

[25] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[26] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation.* Addison-Wesley Professional, 2005.

[27] H. Richardson. High performance fortran: history, overview and current developments, 1996.

[28] Robert Sedgewick. *Algorithms in C: Parts 1-4, Fundamentals, Data Structures, Sorting, and Searching.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[29] Erik Sintorn and Ulf Assarsson. Fast parallel gpu-sorting using a hybrid algorithm. October 2007. Workshop on General Purpose Processing on Graphics Processing Units.