Obsidian: GPU Programming in Haskell

Koen Claessen, Mary Sheeran, Joel Svensson
Chalmers

1 Introduction

Obsidian is a language for data-parallel programming embedded in Haskell. As the Obsidian programs are run, C code is generated. This C code can be compiled for an NVIDIA 8800 series GPU (Graphics Processing Unit), or for other high-end NVIDIA GPUs. The idea is that the style of programming used in Lava for structural hardware design [2] can be applied to data-parallel programming as well. Therefore Obsidian programmers use combinators that have much in common with those used in Lava. However, where Lava generates the netlist for a fixed-size circuit, Obsidian can generate GPU programs that are parametric in input size.

2 GPGPU and Data-parallel programming

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 [14]. 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 [15]. Sorting is another area where there are success stories [18, 12].

2.1 The NVIDIA 8800 GPU

The NVIDIA 8800 GPU is described as “a set of SIMD multiprocessors” in the CUDA Programming Manual [7]. Each of the multiprocessors in the set consists of 8 SIMD processing elements and the high end GPUs in the 8800 series have 16 such multiprocessors, giving a total of 128 processing elements.

On each of these groups of 8 SIMD processing elements a number of threads can be executed. Such a group of threads is called a thread *block*. Each of the threads in a block is executing an instance of the same program. Up to 512 threads can be executing within a block. A block is divided into smaller groups that are executed in a SIMD fashion; these groups are called *warps* [7]. This means that within a warp, all threads are progressing in lock-step through the program. There is a scheduler that periodically switches warps. However, between warps; SIMD fashion of execution is not maintained, thus thread synchronisation primitives are needed.

3 Programming in Obsidian

Obsidian can be used to describe computations on arrays. Currently there are some limitations on what computations can be described. As an example the computations must be length homogeneous, that is the input and output arrays must be equal in length. Also programs are currently limited to working only with integers. Another limitation is that currently the generated code can operate on arrays of length up to 512 elements. This is because the generated code is run in one block of threads, with at most 512 threads, and each thread is only operating on one element of the array.

The first example program reverses an array and adds one to each element:

```
rev_incr = rev ->- fun (+1)
```

Here `rev` is an index permutation and `fun` applies a function to each element of the array. The combinator `->-` composes its two arguments into one operation, by feeding the outputs of the first into the inputs of the second. The `rev_incr` program can be run on the GPU using the `execute` function or it can be run on the CPU using the `emulate` function:

```
*Obsidian> emulate rev_incr [1..10]
[11,10,9,8,7,6,5,4,3,2]
```

The functions `execute` and `emulate` both take an Obsidian program and a Haskell list as arguments. The C program is generated and the Haskell list is turned into a C array. The resulting C program is then passed to the NVIDIA compiler and turned into GPU code or, in the emulation case, into code for the CPU. The compiled program is executed and the result read back into the Haskell system and presented.

The C code generated from the previous `rev_incr` program looks as follows:

```
__global__ static void rev_incr(int *values, int n)
{
  extern __shared__ int shared[];
  int *source = shared;
  int *target = &shared[n];
  const int tid = threadIdx.x;
  int *tmp;
  source[tid] = values[tid];
  __syncthreads();
  target[tid] = (source[((n - 1) - tid)] + 1);
  __syncthreads();
  tmp = source;
  source = target;
  target = tmp;

  __syncthreads();
  values[tid] = source[tid];
}
```

In this program, the general structure of a program generated by Obsidian is visible. First the array is loaded into shared memory. This is followed by a CUDA `__syncthreads()` statement making sure the entire array is loaded into the shared memory. When the shared memory is set up, the computation described in the Obsidian program commences. The generated C code then ends with another `__syncthreads()` and the storing of the computed results into the array.

An Obsidian program will bear much resemblance to the corresponding Lava program. In some cases the Lava and Obsidian descriptions will be identical. As an example, here is the description of the *shuffle exchange network*:

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

In the definition of `shex`, the combinators `rep` and `evens` are used along with the index permutation `riffle`. `rep` repeats a program a given number of times and `evens` applies a two-input two-output function to each even numbered input and its direct neighbour. Figure 1 shows a visual representation of the shuffle exchange network.

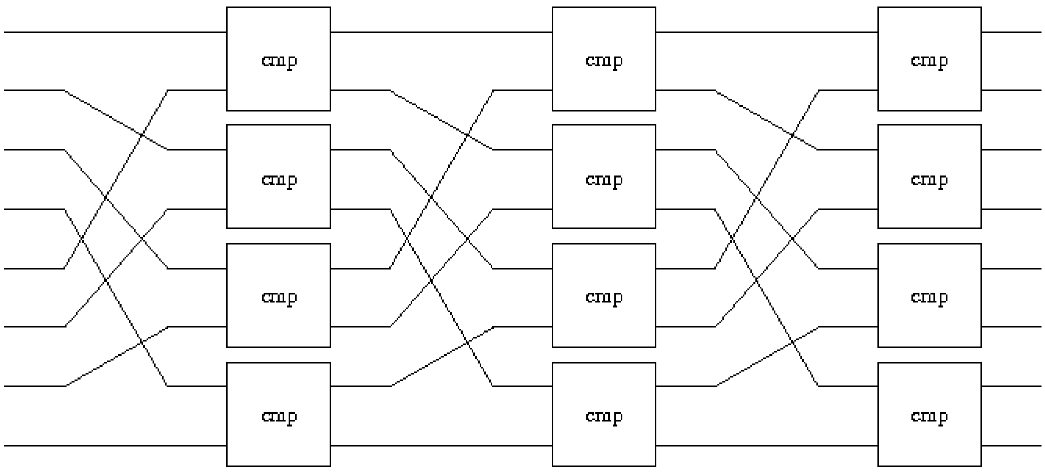

Figure 1: Shuffle exchange network

In Lava as well as Obsidian, `shex` declared above defines the shuffle exchange network. However, the above program will generate a GPU program which is not parametric in the length of the input array, much as Lava generates a netlist of fixed size. This can be fixed by using a different repeat combinator called `repE` that takes an expression as argument instead of an integer:

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

This parametric shuffle exchange network will work for any array of length a power of 2. Both `rep` and `repE` result in a **for loop** in the generated C program.

The following example shows the kind of C code that is generated from an Obsidian program using the combinator `repE`. The example program reverses an array as many times as the array is long and hence it is the identity on even length arrays while it reverses arrays of odd length.

```
revs arr = let n = len arr
           in  repE n rev arr
```

The code generated from the example above looks as follows:

```
__global__ static void revs(int *values, int n)
{
  extern __shared__ int shared[];
  int *source = shared;
  int *target = &shared[n];
  const int tid = threadIdx.x;
  int *tmp;
  source[tid] = values[tid];
  __syncthreads();
  for (int i0 = 0;(i0 < n);i0 = (i0 + 1)){
    target[tid] = source[((n - 1) - tid)];
    __syncthreads();
    tmp = source;
    source = target;
    target = tmp;
  }
  __syncthreads();
  values[tid] = source[tid];
}
```

In Lava, it is common to define circuits recursively. Here is an example showing a butterfly network:

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

Figure 2 shows the butterfly network (which forms the merger in Batcher's well-known *bitonic sort* [1]). At the moment there is no good way of dealing with programs such as `bfly`, in Obsidian. Defining an Obsidian function recursively can lead to C programs with deeply nested conditionals, which is very bad for the performance on the target platform. A way to deal with recursive structures is definitely needed and will be explored as future work.

Since we cannot at the moment generate code for recursive structures, the second example program will be a *periodic sorter*, instead of a recursive one. A periodic sorter works by repeatedly applying a so-called periodic merger to the input. The first component needed is a two-sorter. A two-sorter, here called `cmpSwap`, is a two-input two-output function that sorts its two inputs onto the outputs:

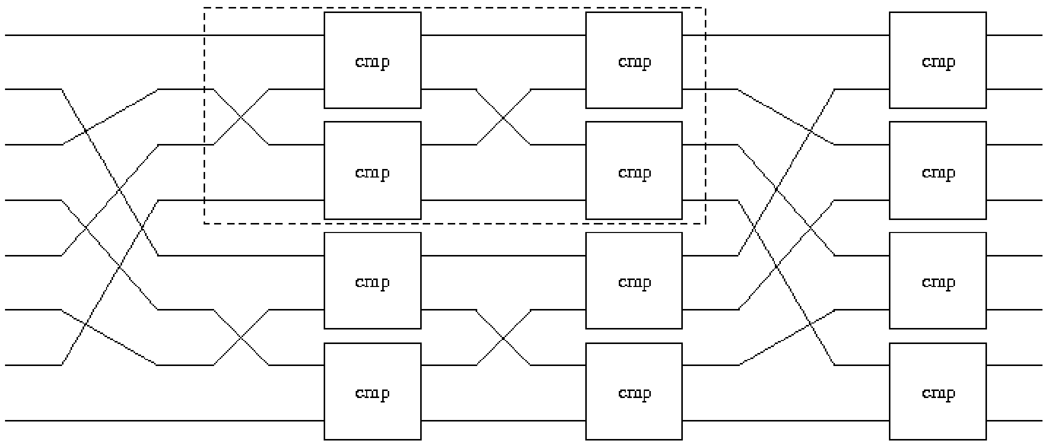

Figure 2: Butterfly network

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

Now the shuffle exchange network can be used to define a bitonic merger:

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

For a given component, such as `cmpSwap`, the functional behaviour of the shuffle exchange network is equivalent to that of the butterfly described above. This can be shown by induction, see for example reference [17]. An earlier Lava paper also showed how to exploit the *zero one principle* in automatic proofs about fixed-size sorting networks [6]. Here, we use that idea (and Lava) to show that the bitonic merger and the merger made using the shuffle exchange network have identical behaviour for 8 inputs. The following property states that using the the component `cmpSwapB`, which is a two-sorter on boolean values, the shuffle exchange network and the butterfly network produce the same output given the same input:

```
prop_shex_bfly =
    forAll (list 8) $ \xs ->
        shex 3 cmpSwapB xs  <==> bfly 3 cmpSwapB xs
```

Now this can be verified using Lava and SMV:

```
Main> smv prop_shex_bfly
Smv: ... (t=0.01system) \c
Valid.
```

Below are two test runs using `bmergeIt`. The first input shown is a bitonic sequence (with first half increasing and second half decreasing,) resulting in sorted output. The second input is not a bitonic sequence and the result remains unsorted.

```
*Obsidian> execute pbmergeIt [1,3,5,7,8,6,4,2]
[1,2,3,4,5,6,7,8]

*Obsidian> execute pbmergeIt [1,7,4,2,6,8,3,5]
[1,2,3,7,4,5,6,8]
```

The next question is how to make a periodic merger. Composing several bitonic mergers in sequence does not give a sorter (as the reader might like to verify). However, Dowd et al have not only introduced the periodic balanced merger, but also shown how it relates to the shuffle exchange (or omega) network [8].From the τ permutation that they introduce, we can derive a related permutation, here called `tau1`, that when composed with a bitonic merger gives a network that behaves identically to the balanced periodic merger.[1] Figure 3 shows the permutation defined by tau1 schematically. The `tau1` index permutation is defined as follows:

```
tau1 = unriffle ->- one rev
```

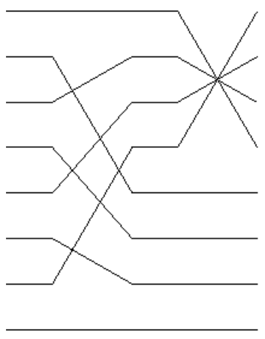

Figure 3: Index permutation defined by `tau1`

Combining `tau1` and bmergeIt results in a merger that has the same behaviour as the balanced periodic merger [8]. Figure 4 shows this merger.

```
dmergeIt = tau1 ->- bmergeIt
```

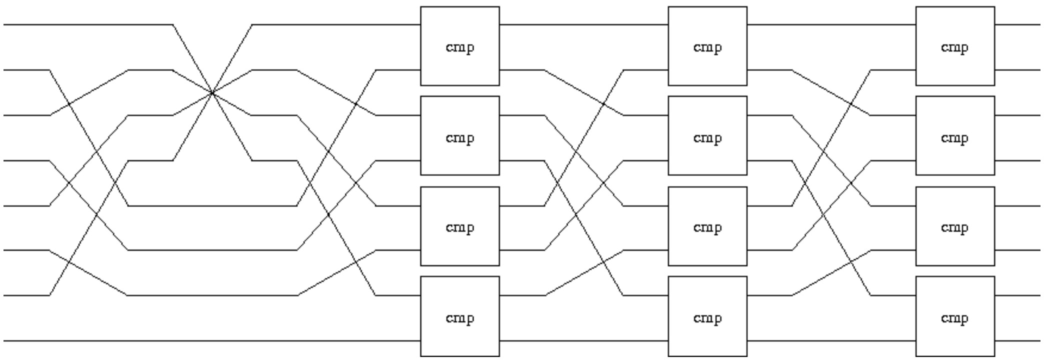

Figure 4: Periodic merger

The sorter called *iterative vsort* is a periodic sorter made from this merger. It is implemented by repeatedly applying `dmergeIt` to the input array.

```
vsortIt arr =
    let n = log2i (len arr)
    in (repE n dmergeIt) arr
```

```
*Obsidian> execute vsortIt [8,1,4,2,3,6,7,5]
[1,2,3,4,5,6,7,8]
```

This section showed how programs are written using Obsidian and also what the generated C code looks like.

4 Results

The experiements shown in this section was performed using an earlier version of Obsidian. In that version, a single Obsidian sync point resulted in two `__synchthreads()` in the generated code. The figures presented here are obtained using the following hardware:

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

[1]Thanks to Eva Suci (Berrgen) for pointing out that Dowd et al contained exactly the information we needed to be able to make a completely iterative description of a periodic sorter

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 5 shows the running time of a number of sorters generated from Obsidian descriptions as well as a sorter running on the CPU and one implemented directly in CUDA. The GPU sorters are running on one of the multiprocessors, using 8 SIMD processing elements. 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 [16].

sortOET Odd Even Transposition sort, is a periodic sorter with depth n.

vsortIt is similar to the sorter described previously, but it is optimized using tables that represent the `tau1` permutation. For 2^n inputs it has depth n^2

vsortIt2 is the same sorter as the above, but with an extra sync inserted into its shuffle exchange network.

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

bitonicSort is the implementation of bitonic sort supplied by the CUDA SDK. For 2^n inputs it has depth $n(n+1)/2$.

The chart in figure 5 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 most efficient sorter in the comparison is the CUDA Bitonic sort. The difference in running time between *bitonicSort* and *vsortIt2* is explained by the difference in depth between them. The sorter called *vsortIt2* is deeper and therefore slower. For example, for 512 inputs *vsortIt2* has depth 81 while *bitonicSort* has depth 45. This indicates that the generated code is acceptably efficient.

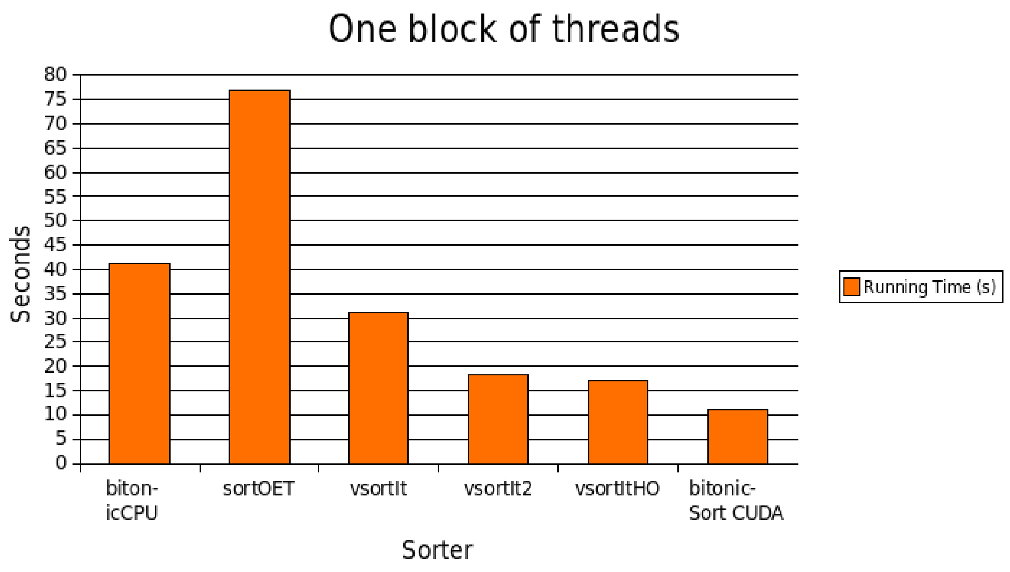

Figure 5: Running time measurements using one block

5 Implementation of Obsidian

Using Obsidian it is possible to describe operations on arrays. To represent an array in Obsidian the type `Arr a` is used:

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

An array in Obsidian is a tuple of an indexing function and an expression representing the length of the array. There are a number of helper functions for working with arrays, `mkArray`, `len` and !, for creating new arrays, getting the length of a given array and indexing in an array. These functions are mostly used inside of the library and not by the application writer.

One of the basic operations on arrays provided by Obsidian is the `rev` function. Its use has been shown in the previous section about programming. In the library `rev` is implemented as follows:

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

This function takes an array and returns a new one with the indexing function transformed. The type of `rev` is monadic, the monad `W` is a variant of the Writer monad with functionality added for generating loop variables. All Obsidian functions are monadic for consistency even though a function such as rev makes no use of the functionality the `W` monad provides.

To be able to generate C Code from an Obsidian program, a representation of C code is at certain points written into the `W` monad. We call these points *sync* points. These sync points are expressed in Obsidian using the `sync` function:

```
sync :: Syncable a => Arr a -> W (Arr a)
```

When a sync is reached an object of type `IxExp -> AbsC` is written into the W Monad. This function is later applied to an index expression representing a thread's thread Id. The result, `AbsC`, of this application is then used to generate the CUDA C code.

A sync point can be inserted into a program as follows, using the shuffle exchange network example from earlier:

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

The operations before and after the `sync` are now sequenced in the generated code and no longer composed into one operation executed in parallel on the target platform. In some cases this insertion of an extra `sync` can have beneficial effect on the performance of the generated code. This is probably due to the simpler expressions returned by `sync`. Perhaps this effect will diminish when we later introduce optimisation of the generated expressions. Currently no optimisation of the generated code is performed; this will be explored as future work.

Let us return to the `rev_incr` example from the previous section. When passing the program `rev_incr` to `emulate` or `execute`, a `sync` is added to it. So in reality the `rev_incr` program is defined as:

```
rev_incr2 = rev ->- fun (+1) ->- sync
```

When using `execute` or `emulate` the two programs `rev_incr` and `rev_incr2` are the same. Each of the `execute` and `emulate` functions analyses its input and decides whether or not to insert a `sync` statement.

Sync points can also be added automatically in the combinators `rep` and `repE`. This is essentially making the program `rep 3 rev` and `rep 3 (rev ->- sync)` equivalent.

To generate the C code from an Obsidian program the first step is to apply the Obsidian program to a symbolic array defined as follows:

```
symArray :: Arr (Exp Int)
symArray = (\ix -> (index (variable ``source'' Int) ix),
            variable ``n'' Int)
```

If all goes well an object of type `IxExp -> AbsC` is created. This object is applied to an index expression representing thread id:

```
threadID = variable ``tid'' Int
```

The `AbsC` type describes the abstract syntax of a subset of C. For example it contains for loops, if statements, variable declarations and assignments. From this representation a CUDA C source file is generated.

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 that the programming style for Obsidian is derived. In Lava combinators are used to describe hardware [2].

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 [9]. Many ideas from the paper "Compiling Embedded Languages", describing the implementation of Pa,n were used in the implementation of Obsidian [11].

The two languages above are those that had a more direct impact on this 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 [3]. NESL is described in [4].

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 [5].

In both NESL and Data Parallel Haskell, 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 [13]. 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 [10]. 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 Future Work

Currently the generated code operates on array with a maximum length of 512 elements. Breaking this barrier in combination with exploiting the full

GPU is high priority for future work.

Even though the results show that the generated code is reasonably efficient, it must be said that implementing the efficient sorter presented in the results demdanded much care in choosing what combinators to use and exactly how they should be implemented. Optimising the generated code might result in less work needed from the application implementor in order to assure efficiency. At the moment no optimisation of the generated code is performed. There are a number of techniques for optimising expressions similar to those generated by Obsidian in "Compiling Embedded Languages" [11].

Recursive Obsidian functions are not recommended, as 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 [4]. Perhaps this would offer a solution to the previously stated shortcoming as well. It looks as though 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 kinds of parallel functions that can be applied to a nested data structure are limited.

8 Acknowledgement

This research is supported by the Swedish Basic Research Agency, Vetenskapsrådet. We would also like to thank our colleague Ulf Assarsson for giving us an introduction to GPU programming.

References

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

[2] 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.

[3] 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.

[4] Guy E. Blelloch. NESL: A Nested Data-Parallel Language. Technical Report CMU-CS-93-129, CMU Dept. of Computer Science, April 1993.

[5] 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.

[6] Koen Claessen, Mary Sheeran, and Satnam Singh. Using lava to design and verify recursive and periodic sorters. *STTT*, 4(3), 2003.

[7] *NVIDIA CUDA Compute Unified Device Architecture: Programming Guide Version 1.0, www.nvidia.com/cuda.*

[8] Martin Dowd, Yehoshua Perl, Larry Rudolph, and Michael Saks. The periodic balanced sorting network. *J. ACM*, 36(4):738–757, 1989.

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

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

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

[12] 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.

[13] 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.

[14] 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.

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

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

[17] Mary Sheeran. Sorts of butterflies. In *IVth Workshop on Higher Order (Banff,1990), ed. Birtwistle*, Workshops in Computing. Springer, 1991.

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