THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Embedded Languages for Data-Parallel Programming

Bo Joel Svensson

CHALMERS | GÖTEBORG UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY AND GÖTEBORG UNIVERSITY
Göteborg, Sweden 2013

Embedded Languages for Data-Parallel Programming
Bo Joel Svensson
ISBN 978-91-7385-939-4

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 3620
ISSN 0346-718X

Technical Report 101D
Department of Computer Science and Engineering
Functional Programming Research Group

Chalmers University of Technology and Göteborg University
SE–412 96 Göteborg, Sweden
Phone: +46 (0)31–772 1000

Printed in Sweden
Chalmers Reproservice
Göteborg, Sweden 2013

Abstract

Computers today are becoming more and more parallel. General purpose processors (CPUs) have multiple processing cores and Single Instruction Multiple Data (SIMD) units for data-parallelism. Graphics processors (GPUs) bring massive parallelism at the cost of being harder to program than CPUs. This thesis applies embedded language methodology to data-parallel programming. Two embedded languages are presented, Obsidian for general purpose GPU programming and EmbArBB for data-parallel programming across platforms.

CPUs and GPUs get more parallel resources with each new generation. The question of how to efficiently program these processors arises. We are after efficiency both in programmer productivity and in application performance. Using embedded languages allows us to experiment with what abstractions to present to the programmer at relatively little effort.

Obsidian is an embedded language for general purpose programming of GPUs. We try to strike a balance between high level, productivity increasing abstractions and low-level control needed for performance. The Obsidian programming model mirrors the GPU architecture and the programmer is constrained into writing GPU-friendly code. Hierarchy level polymorphic library functions are supplied to make these constraints feel less obtrusive. Obsidian programs are compiled into CUDA C code. This compilation is based on a simple and elegant monad reification technique.

In cases where the programmer is not interested in low-level details or wants the program to run over a range of hardware, a higher level language can be used. EmbArBB is a Haskell embedding or the Intel ArBB system. EmbArBB relies on the ArBB system to generate code (via a Just-In-Time compiler) to a range of hardware.

EmbArBB embeds a preexisting library for data-parallelism into Haskell and we obtain very good performance at little implementation effort. This performance comes from the expertise and effort put into the ArBB system and that we get for free. Embedding ArBB is a way to provide these benefits to the Haskell programmer and a way to increase usefulness of an existing system by opening it up to a wider audience. Obsidian is very different; it is not based on a set of high-level parallel primitives. The Obsidian programmer can implement these primitives in different ways and then select the best one. We have obtained very good performance in case studies involving reductions. Obsidian programs are also more terse and composable, compared to CUDA.

Keywords: Data-parallelism, Embedded languages, Functional Programming, Graphics Processing Units

Acknowledgments

I would like to thank my supervisor Mary Sheeran. She is a very supportive, motivating and understanding supervisor. I also thank my co-supervisor Koen Claessen for all the great advice and tips he has provided over the years. I cannot imagine that this would have worked with any other supervisors. Thanks to Josef Svenningsson, a late addition to my troop of supervisors. Working with Josef has been fun and rewarding.

I have often turned to Emil Axelsson with technical questions, and he has always been happy to help. For this I am grateful.

I thank Manuel Chakravarty for letting me visit his group at the University of New South Wales and Trevor McDonell and Sean Lee for taking care of me during my visit. I look back very fondly to my stay in Australia.

For being a great supervisor during my three month internship at Intel, I thank Ryan Newton. These where three very intense months away from family but at the same time a very rewarding experience.

Thanks to my thesis opponent Stephen Edwards for valuable comments on a draft of this thesis.

I want to thank Erik, Ola, Markus and Viktor for our occasional lunches and GPU discussions. I also thank my office mates, Michal and Nicholas, for their company. I thank Eva Axelsson, who I have felt most comfortable to turn to for help and information, even when she was not the appropriate channel.

Finally I want to thank my family and friends. Thanks to my sisters and parents for their support. I thank my friends for being understanding during times when I have been very self-absorbed. I am very grateful to my wife Andrea for supporting and believing in me and to our daughter Noemi for bringing much joy to my breaks from work while writing this thesis; soon she is old enough for real LEGO.

This research has been funded by the Swedish Foundation for Strategic Research (which funds the Resource Aware Functional Programming (RAW FP) Project) and by the Swedish Research Council.

Publications

This thesis includes the following publications:

A. Josef Svenningsson and Bo Joel Svensson. Simple and Compositional Reification of Monadic Embedded Languages, 2013. *18th ACM SIGPLAN International Conference of Functional Programming, ICFP 2013.*

B. Bo Joel Svensson, Mary Sheeran and Koen Claessen. Obsidian: A Domain Specific Embedded Language for General Purpose Parallel Programming of Graphics Processors. In *Proc. of Implementation and Applications of Functional Languages (IFL)*, Lecture Notes in Computer Science, Springer Verlag, March 2009.

C. Bo Joel Svensson, Koen Claessen and Mary Sheeran. GPGPU kernel implementation and refinement using Obsidian. *Practical Aspects of High-level Parallel Programming, PaPP 2010.* Procedia Computer Science.

D. Koen Claessen, Mary Sheeran and Bo Joel Svensson. Expressive Array Constructs in an Embedded GPU Kernel Programming Language. In *Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming*, DAMP '12. ACM.

E. Josef Svenningsson, Bo Joel Svensson and Mary Sheeran. Efficient Counting Sort Implementations using an Embedded GPU Programming Language. *2nd Workshop on Functional High-Performance Computing, FHPC '13.*

F. Bo Joel Svensson and Mary Sheeran. A High-Level Embedded Language for Low-Level GPU Kernel Programming. ***This work has not yet been published.***

G. Bo Joel Svensson and Ryan Newton. Programming Future Parallel Architectures with Haskell and ArBB. *Future Architectural Support for Parallel Programming (FASPP), in conjunction with ISCA '11.*

H. Bo Joel Svensson and Mary Sheeran. Parallel Programming in Haskell Almost for Free: an embedding of Intel's Array Building Blocks. *Proceedings of the 1st ACM SIGPLAN Workshop on Functional High-Performance computing, FHPC '12.*

Personal contributions

The following describes my personal contributions to the papers included in this thesis.

Paper A

I intuitively applied the monad reification method as part of the implementation of Obsidian. Josef Svenningsson identified its importance and suggested we write about it. The paper has two distinct parts. Sections one and two describe the method as I used it. And section four makes the compositional aspect of the method explicit.

The code and examples in section one and two are my work. The contents of section four was developed by Josef Svenningsson and me. For section four we also got very much and valuable help from Emil Axelsson.

For the rest of the paper, just substituting my name for Benny and Josef's for Björn will give a good indication of our contributions.

Papers B, C, D and E

The Obsidian language has grown out of discussions between Mary Sheeran, Koen Claessen and me. When it comes to implementation of the ideas we had in these discussions, I have been responsible. Push arrays, first appearing in paper D, were invented by Koen Claessen. However, the implementation of them in the setting of Obsidian was done by me. Paper D also has large contributions by Mary Sheeran when it comes to examples and implementing them.

Paper E was the first I wrote with Josef Svenningsson, who had been thinking about an interesting variation of counting sort and wanted to implement it. Together we added atomic operations to Obsidian and I modified the code generator to accommodate them. Sections 1 and 2 in the paper are written by Josef. Sections 3 through 7 are mostly my work, including the benchmarking. Section 8 is joint work.

Papers G and H

The work behind paper G, work was performed by me under supervision of Ryan R. Newton who is also mostly responsible for the writing of the paper. Programming and running time experimentation was performed by me, while getting much valuable advice and guidance from Ryan.

The work behind paper H was done by me as a hobby project and my aspirations were not high. I just wanted to experiment with some embedded language techniques that we had been avoiding (had no need for) in Obsidian. Mary discovered what I was doing and applied the right amount of pressure, leading to us encompassing

much more of ArBB functionality than I had in mind. The paper's text is joint work; most examples as well as benchmarking are my work, but the sparse matrix vector multiplication example is entirely Mary's.

Contents

1 Embedded Languages for Data-Parallel Programming **1**

1.1 Introduction 1

1.2 Obsidian: An embedded language for GPU kernel implementation . 3

1.2.1 GPU hardware 3

1.2.2 GPU programming 4

1.2.3 Writing efficient GPU Kernels 11

1.2.4 Obsidian Implementation 13

1.2.5 Results 18

1.3 Embedding Intel Array Building Blocks 19

1.3.1 EmbArBB: Embedding ArBB in Haskell 20

1.3.2 Current status of ArBB and EmbArBB 22

1.4 Related work 22

1.4.1 Languages for data-parallel programming 22

1.4.2 Embedded domain specific languages 25

1.4.3 Compiling Embedded Languages 26

1.5 Future work 26

1.6 Discussion 27

2 EDSL Implementation Papers **33**

2.1 Simple and Compositional Reification of Monadic Embedded Languages 35

2.1.1 Introduction 35

2.1.2 Compilation of the monadic robot EDSL 39

2.1.3 Meeting in Björn's office 42

2.1.4 Composing reifiable monadic languages 46

2.1.5 Epilogue 51

Bibliography 53

3 GPU Programming Papers **55**
3.1 Obsidian: A Domain Specific Embedded Language for Parallel Programming of Graphics Processors . . . 57
3.1.1 Introduction . . . 57
3.1.2 Connection patterns for hardware design and parallel programming . . . 58
3.1.3 Graphics Processing Units, accessible high performance parallel computing . . . 60
3.1.4 Obsidian: a domain specific embedded language for GPU programming . . . 63
3.1.5 Results . . . 71
3.1.6 Discussion . . . 72
3.1.7 Future work . . . 75
3.1.8 Conclusion . . . 75
Bibliography . . . 78
3.2 GPGPU Kernel Implementation and Refinement using Obsidian . . 79
3.2.1 Introduction . . . 79
3.2.2 Programming in Obsidian . . . 82
3.2.3 Outline of Code Generation . . . 86
3.2.4 Case Studies . . . 87
3.2.5 Future work . . . 90
3.2.6 Related work . . . 92
3.2.7 Discussion and conclusion . . . 92
Bibliography . . . 95
3.3 Expressive Array Constructs in an Embedded GPU Kernel Programming Language . . . 97
3.3.1 Introduction . . . 97
3.3.2 Push Arrays . . . 104
3.3.3 Application . . . 108
3.3.4 Discussion . . . 119
3.3.5 Appendix . . . 122
Bibliography . . . 124
3.4 Counting and Occurrence Sort for GPUs using an Embedded Language 125
3.4.1 Introduction . . . 125
3.4.2 Counting Sort . . . 126
3.4.3 GPUs and CUDA . . . 130
3.4.4 Obsidian . . . 131
3.4.5 Implementing counting sort in Obsidian . . . 139
3.4.6 Implementing occurrence sort in Obsidian . . . 141

3.4.7 Performance evaluation . . . 142
3.4.8 Discussion . . . 145
Bibliography . . . 149
3.5 A High-Level Embedded Language for Low-Level GPU Kernel Programming . . . 151
3.5.1 Introduction . . . 151
3.5.2 A taste of Obsidian programming . . . 154
3.5.3 Obsidian internals . . . 157
3.5.4 Case studies . . . 167
3.5.5 Future Work . . . 182
3.5.6 Conclusion . . . 183
Bibliography . . . 186

4 Retargetable Parallel Programming Papers **187**
4.1 Programming Future Parallel Architectures with Haskell and Intel ArBB . . . 189
4.1.1 Introduction . . . 189
4.1.2 Embedded Domain-Specific Languages . . . 190
4.1.3 Harbb = ArBB + Accelerate . . . 191
4.1.4 Intel Array Building Blocks (ArBB) . . . 192
4.1.5 Implementation of Harbb . . . 194
4.1.6 Preliminary Results . . . 195
4.1.7 Related Work . . . 196
4.1.8 Discussion and Conclusions . . . 199
Bibliography . . . 202
4.2 Parallel Programming in Haskell Almost for Free . . . 203
4.2.1 Introduction . . . 203
4.2.2 Related Work . . . 205
4.2.3 Motivation . . . 207
4.2.4 Programming in ArBB . . . 207
4.2.5 Programming in EmbArBB . . . 208
4.2.6 Implementation . . . 224
4.2.7 Benchmarks . . . 231
4.2.8 Future Work . . . 236
4.2.9 Conclusion . . . 237
Bibliography . . . 239

Chapter 1

Embedded Languages for Data-Parallel Programming

1.1 Introduction

This thesis applies embedded language methodology to data-parallel programming of massively parallel hardware. An embedded language is a language that is implemented as a library for some existing host language. In this case, the functional programming language Haskell is used as host language.

Computers today are increasingly parallel. Each new generation of general purpose processors (CPUs) and of graphics processors (GPUs) brings more parallelism. The question of how to program these computers most efficiently, both from a productivity and a performance point of view, arises. Explicit parallel programming is hard and automatic means of parallelisation, while giving a sequential programming model to the programmer, often misses the target when it comes to performance. The book "Structured Parallel Programming" by Michael McCool et al argues the need for explicit parallel programming models [MM12].

The embedded language approach to programming language implementation is beneficial, since we are not entirely sure of how to program current and future highly parallel computers. Implementing an embedded language allows us more rapid prototyping of ideas. We can experiment with what abstractions to expose to the programmer, without much effort. There are many other benefits of embedded languages: making use of the host language's library functions (if they are general enough), the host language can be used as a powerful macro language for the embedded language, providing a more levelled learning curve to programmers already

familiar with the host language. For programmers who are already proficient in the host language, learning an embedded language is more like picking up a new library.

This thesis focuses on data-parallelism, as opposed to task or control parallelism. Data-parallelism is applicable where the input data can be divided into chunks and operated upon independently. Data-parallelism is scalable; the potential for parallelism increases with the problem size. Two different embedded languages for data-parallel programming are presented in this thesis: Obsidian, for general purpose computations on GPUs and EmbArBB for parallel programming across platforms, such as CPUs, GPUs and other accelerators.

Graphics Processors trade programmer convenience for increased parallelism. For example, where CPUs dedicate large portions of chip area to caches, branch prediction and instruction level parallelism (ILP), GPUs put additional cores. This comes at a price of increased programmer effort. GPUs have programmer managed shared memories, conditionals are troublesome from a performance perspective and memory access patterns can make or break the performance of an application. With Obsidian, we try to find the right level of abstraction for programming GPUs. We try to strike a balance between low-level control of details that influence performance and high-level abstractions to increase productivity and make programs more terse. The end goal with Obsidian is a language for writing high performance GPU programs that compare well to code written by CUDA [NVIb] (a C-dialect for GPU programming) experts, provided that both programmers have similar GPU expertise. The Obsidian programmer would however write programs in a more composable manner. Obsidian programs are compiled into CUDA.

Obsidian is geared towards the programmer who is interested in GPU hardware. Making good use of Obsidian requires that the programmer is willing to make an effort to learn about GPUs and how to program them efficiently. However, another case is a programmer who does not want to invest that effort but wants to write a high-level parallel program and have it run reasonably well on different parallel architectures. In this case, a language with even higher level of abstraction is preferable. It then becomes the job of a more advanced compiler to turn that into code for the targeted platforms. The EmbArBB language targets this programmer. EmbArBB exposes a set of high-level operations on arrays, such as `map`, `reduce` and `scan`. The programmer has no power over the parallel strategy that should be used to implement these operations but instead relies on a JIT-compiler (Just In Time-compiler). EmbArBB is a Haskell embedding of Intel ArBB [NSL+11] that was released as an embedded language in C++. EmbArBB relies entirely on the ArBB system for code generation and optimisation.

1.2 Obsidian: An embedded language for GPU kernel implementation

In 2006, NVIDIA released their first Compute Unified Device Architecture (CUDA) GPU. With CUDA, NVIDIA also made it a lot easier for programmers to use GPUs for general purpose programming [NVIc]. A dialect of C, called CUDA C, was released specifically to simplify the implementation of general purpose computations on GPUs.

Before CUDA, programmers still tried (very ingeniously) to make use of GPUs for non-graphics applications via the graphics API. The desired computation was expressed in graphics terms and the result computed through an act of rendering. Of course, since the available abstractions targeted the graphics domain, this added extra burden on the programmer. CUDA was NVIDIA's first response to this problem. CUDA is a dialect of C, extended with a parallel programming model suitable for GPUs.

Having a language specifically made for general purpose programming on the GPU, such as CUDA, is a big step up from earlier. But CUDA still offers a very low-level interface to programming the GPU, which is preferred in a situation where the programmer needs to think about hardware specific details to get the best performance. CUDA provides this, but at the price that it is hard to build upon, abstract and reuse prior programming effort. This is where we are trying to go one step further with Obsidian.

1.2.1 GPU hardware

Before going into programming of GPUs, it is important to have some background. The GPU programming model exposed by CUDA very much mirrors the underlying hardware. Some of the details that make GPU programming hard are more apparent when looking at the underlying hardware.

A CUDA GPU is built around a single kind of processor (as opposed to the different kinds of processors found in earlier GPUs). The processors in the GPU (called MPs, MultiProcessors) all contain a number of cores called SPs (Streaming Processors). Each MP also contains local memory, called shared memory since it can be accessed by all of the SPs in that MP. The number of MPs varies over the available GPUs; cheaper GPUs have as few as one MP, and as you go up in price the number of MPs increases.

Each MP of the GPU can manage a large number of threads; on today's GPUs up to 2048 threads can run on a single MP. The GPU schedules threads in groups of 32, called *Warps*, that are executed in lock-step (SIMD style execution). Threads

are also divided into *Blocks*; the threads within a block can communicate using the shared memory. The maximum block size is currently 1024 threads. Threads within a warp can communicate via the shared memory without using any synchronisation primitive. However, if communication takes place across warps synchronisation is necessary. A barrier synchronisation mechanism exists to ensure that all threads within a block have reached the same position in the code. Blocks are also grouped into a grid that is the collection of blocks executing the same program.

1.2.2 GPU programming

The CUDA programming model reflects the hierarchical architecture of the GPU. The programmer writes what are called *kernels*; in CUDA jargon, a kernel refers to a sequential C program that is parameterised on a thread's identity. The kernel program is executed in parallel across all the threads of a block and many such blocks of threads can execute in parallel on the available MPs. Kernels may use shared memory that is local to a block; the programmer needs to decompose the algorithm and data in such a way that communication will be local to the blocks.

The program below is a very simple example of a CUDA kernel that computes elementwise sums.

```
__global__ void vecAdd(int32_t *i1, int32_t *i2, int32_t *r) {

  unsigned int gid = blockIdx.x *
                     blockDim.x +
                     threadIdx.x;

  r[gid] = i1[gid] + i2[gid];

}
```

The kernel takes two input arrays, `i1` and `i2`; the result is stored in `r`. Note how the kernel makes use of three variables, `threadIdx.x`, `blockIdx.x` and `gridDim.x`. These variables identify a thread uniquely within a grid. Here they are used to obtain an index into the provided arrays. The dimensions of the blocks and of the grid are set by a controlling program upon launch of the kernel. This controlling program is run by the CPU of the computer housing the GPU. The kernel is launched using the following syntax:

```
vecAdd<<<1,32,0>>>(d_i1,d_i2,d_r);
```

This line of code launches the kernel on one block of 32 threads. The zero indicates that the kernel uses no shared memory. If the kernel did use shared memory, the

amount needed should have been specified there. Figure 1.1 shows the complete sequence of commands for a minimal example using the kernel specified above.

In Obsidian, the same kernel (the local computation) is implemented as follows:

```
vecAdd :: Num a => SPull a -> SPull a -> SPull a
vecAdd = zipWith (+)
```

The `vecAdd` program takes two `SPull` input arrays and gives one as result. The Obsidian `vecAdd` is more general than the CUDA version. From it we can generate CUDA kernels for any element type that supports the `(+)` operation. In Haskell we know that types that belong to the `Num` type class has a `(+)` operation.

There are some important differences between CUDA and Obsidian programs. One is that with Obsidian we generate kernels for fixed size local computations. This is to avoid doing out-of-bounds checks which would mean conditionals in the code. Conditionals are problematic from a performance point of view. Another difference is that in Obsidian the programmer specifies the entire computation, including how it is mapped over the blocks of the GPU. This is done implicitly in CUDA, based on the indexing arithmetic used in the kernel and the number of blocks and threads provided at kernel launch. This means that the Obsidian programmer needs to write a program that performs this mapping:

```
vecAddG :: Num a => DPull a -> DPull a -> DPush Grid a
vecAddG i1 i2  =
  pConcat $
  fmap push $
  zipWith vecAdd
          (splitUp 32 i1)
          (splitUp 32 i2)
```

The `vecAddG` program specifies a computation on large arrays; the large input arrays are split up into pieces that are operated upon by the local computation, `vecAdd`, specified above. Each input array passed to this function must have length a multiple of 32. Since the `splitUp` operation splits the arrays into chunks of 32 elements, the body of the generated kernel will be constructed specifically for this size. `zipWith` is used to map the local computation onto each chunk of the input arrays. The result is an array of arrays that is flattened with the `pConcat` function.

```
int main(void) {

  int32_t h_i1[32], h_i2[32], h_r[32];

  int32_t *d_i1, *d_i2, *d_r;


  for (int i = 0; i < 32; ++i) {
    h_i1[i] = i;
    h_i2[i] = 32 - i;
  }

  // Allocate device memory
  cudaMalloc((void**)&d_i1,32*sizeof(int32_t));
  cudaMalloc((void**)&d_i2,32*sizeof(int32_t));
  cudaMalloc((void**)&d_r,32*sizeof(int32_t));

  // Copy inputs to device
  cudaMemcpy(d_i1,
             h_i1,
             32*sizeof(int32_t),
             cudaMemcpyHostToDevice);
  cudaMemcpy(d_i2,
             h_i2,
             32*sizeof(int32_t),
             cudaMemcpyHostToDevice);

  // Launch the kernel
  vecAdd<<<1,32,0>>>(d_i1,d_i2,d_r);

  // Copy result to host
  cudaMemcpy(h_r,
             d_r,
             32*sizeof(int32_t),
             cudaMemcpyDeviceToHost);

  for (int i = 0; i < 32; ++i) {
    printf("%d ",h_r[i]);
  }

  return 0;
}
```

Figure 1.1: Host code for launching a simple CUDA kernel on the GPU.

The CUDA code below is generated from the `vecAddG` program:

```
extern "C" __global__ void vecAdd(int32_t* input0, uint32_t n0,
                                  int32_t* input1, uint32_t n1,
                                  int32_t* output2)
{
    uint32_t tid = threadIdx.x;

    if (blockIdx.x < min(n0 / 32U, n1 / 32U)) {
        output2[blockIdx.x * 32U + tid] =
            input0[blockIdx.x * 32U + tid] +
            input1[blockIdx.x * 32U + tid];
    }
}
```

There are some superficial differences between the code generated using Obsidian and the handwritten example above. One such difference is that the index computation is not shared. Obsidian relies on the CUDA compiler to discover such sharing via a Common Subexpression Elimination (CSE) optimisation pass. Another difference is that Obsidian has inserted a conditional that only executes the body for certain blocks. Discussion of the causes for that conditional is postponed until section 1.2.4. A real difference is that the kernel is generated for a fixed size; here there is a constant (32) where in the handwritten code we used `blockDim.x`. If those differences are set aside, the two kernels are identical. Figure 1.2 shows how the `vecAddG` kernel can be launched from within Haskell, using Obsidian's rudimentary support for launching GPU computations.

An alternative way to implement `vecAddG` is to inline the local computation directly. This makes the entire program look like this.

```
vecAddG :: Num a => DPull a -> DPull a -> DPush Grid a
vecAddG i1 i2  =
  pConcat $
  fmap push $
  zipWith (zipWith (+))
          (splitUp 32 i1)
          (splitUp 32 i2)
```

The generated code looks exactly as before.

```
performVecAdd =
  withCUDA $
  do
    kern <- capture 32 (vecAddG :: DPull EInt32
                                -> DPull EInt32
                                -> DPush Grid EInt32)

    useVector (V.fromList [0..32::Int32]) $ \i1 ->
      useVector (V.fromList [0..32::Int32]) $ \i2 ->
        allocaVector 32 $ \(o :: CUDAVector Int32) ->
        do
          fill o 0
          o <== (1,kern) <> i1 <> i2

          r <- peekCUDAVector o
          lift $ putStrLn $ show r
```

Figure 1.2: Host code for launching a simple Obsidian kernel on the GPU. `capture` compiles an Obsidian program into CUDA. At this point the types of the Obsidian program need to be made concrete. The number, 32, used as an argument to `capture` specifies that generated code should be specialised for 32 threads per block.

A CUDA kernel that uses shared memory takes the form:

```
__global__ void kernel(int32_t *i, int32_t *r) {

  extern __shared__ int32_t sm[];

  unsigned int tid = threadIdx.x;
  unsigned int gid = blockIdx.x *
                     blockDim.x +
                     threadIdx.x;

  sm[tid] = i[gid];
  __syncthreads();

  /* Compute on sm */

  __syncthreads();
  r[gid] = sm[tid];

}
```

A portion of the input array is stored into shared memory. A barrier synchronisation is used to ensure that each thread within the block has stored its value into shared memory (only necessary if threads will communicate across warp boundaries). After computing on the data locally, it is written back into global memory. This kernel sets up two aliases for thread identities. A thread's global identity is called `gid` and its local identity is called `tid`. The global ID is used to index into the large global array; indexing into local shared memory is done using the local ID.

In Obsidian, local computations can also use shared memory. The Obsidian arrays we have seen in the examples, `Pull` and `Push` arrays do not directly correspond to actual data in memory. Pull and push arrays rather represent ways to compute arrays. More information about these array representations is available in section 1.2.4. The Obsidian programmer can store intermediate arrays in shared memory by using a `force` operation.

The following kernel multiplies each element in an array by two and then adds one:

```
kernel :: Num a => SPull a -> SPull a
kernel = fmap (+1) . fmap (*2)
```

This kernel computes its result without any storage of intermediate results; the code is equivalent to `fmap ((+1).(*2))`. In the generated CUDA code below, it can be seen that the (*2) and (+1) operations are applied without any intermediate storage.

```
extern "C" __global__ void kernel(int32_t* input0, uint32_t n0,
                                  int32_t* output1)
{
    uint32_t tid = threadIdx.x;

    if (blockIdx.x < n0 / 32U) {
        output1[blockIdx.x * 32U + tid] =
          input0[blockIdx.x * 32U + tid] * 2 + 1;
    }
}
```

Storing an intermediate result between the two operations can be accomplished by changing the kernel as follows.

```
kernelF :: (MemoryOps a, Num a)
        => SPull a -> BProgram (SPull a)
kernelF = liftM (fmap (+1)) . force . fmap (*2)
```

This change introduced some noise. The reason for this is that Obsidian programs that use memory are monadic. The `BProgram` in the result type is a monad. Another difference is that we need to be able to write the array elements to memory; in Obsidian this means that the element type must be in the `MemoryOps` class.

The code below is generated using the `kernelF` program:

```
extern "C" __global__ void kernel(int32_t* input0, uint32_t n0,
                                  int32_t* output1)
{
    extern __shared__ uint8_t sbase[];
    uint32_t tid = threadIdx.x;

    if (blockIdx.x < n0 / 32U) {
        ((int32_t*) sbase)[tid] =
           input0[blockIdx.x * 32U + tid] * 2;
        __syncthreads();
        output1[blockIdx.x * 32U + tid] =
           ((int32_t*) sbase)[tid] + 1;
    }
}
```

This code uses a shared memory array called `sbase`. There is also a call to `__syncthreads` to ensure that all the writes have completed before proceeding. In this case the synchronisation is not really necessary since all threads belong to the same warp. There is an `unsafeforce` operation that naively removes synchronisations if the array being forced is short enough. More advanced analysis could also

be performed to remove synchronisations whenever communication is entirely warp local. We do not apply that optimisation, but work on warp level computations is in progress and would put this ability in the hands of the programmer.

1.2.3 Writing efficient GPU Kernels

Writing efficient GPU kernels is hard. There are many details that the CUDA programmer must be aware of in order to make optimal use of the GPU. The NVIDIA "CUDA C Best Practices Guide" [NVIa], is a good source to learn more about these details. Here is a list, with explanations of some of the practices that the guide deems most important. The list also contains information about how these best practices influence our work on Obsidian.

- **Minimise data transfers between host and device:** The bandwidth between device memory and GPU is much higher than the bandwidth between host memory and the GPU device via the PCIe bus. This means that, first of all, one must evaluate if the computation to be performed is significant enough to warrant a transfer to the GPU. If the computation is not arithmetic intensive, it may be better to do the work on the CPU. However, if the data is already in device memory and the operation that is about to be performed on it is more efficiently handled by the CPU, it may still be better to let the GPU do the work. In other words, compute near the data unless transferring it pays off.

  Data transfers between the host memory and device (GPU) memory is out of Obsidian's scope. If Obsidian is used to generate kernels for a computation, the programmer has already decided that a GPU should be used. However, even after making the decision to use the GPU the programmer may need to consider how to transfer the data. In some cases it is possible to overlap computation on the GPU and data transfer. These kinds of optimisations are considered when writing the C program that launches the CUDA kernels. Obsidian does have rudimentary support for launching kernels from within Haskell but currently we consider this mostly a tool for quickly being able to run a kernel and see what it does, not as a means to implement applications. This part of Obsidian is future work.

- **Ensure that global memory accesses are coalesced:** Certain access patterns into GPU device memory can be *coalesced* (combined into few memory transactions). Unfortunately what these access patterns are varies somewhat depending on the GPU. For a typical model of GPU, the rule is that concurrent memory accesses (by threads within a warp) will be coalesced to as many

transactions as cache lines touched (128 Byte l1 cache lines). This means that strided accesses from within a warp are not favoured.

Inputs to an Obsidian program are pull arrays. Pull arrays can be arbitrarily permuted; the permutation used will decide the access pattern. In section 1.2.4, there is more information about pull arrays. If global memory accesses can be coalesced or not partly also depends on the algorithm being implemented. The next paragraph suggests a trick for coalescing memory accesses, in some cases, even if the algorithm is unsuitable. Then it becomes a trade between bad memory access pattern and using more shared memory.

- **Minimise use of global memory:** Accessing global device memory can take as many as 400 to 600 clock cycles if the data is not cached, this is 10 - 20 times as many cycles as an access to shared memory. The programmer should use shared memory as much as possible to avoid redundant loading from global memory. Shared memory can also be used to coalesce reads from global memory. Shared memory is a limited resource in the MP, so if a kernel uses very much shared memory, fewer such blocks can be active on the MP.

  Intermediate arrays are stored into shared memory when the Obsidian programmer uses the `force` function.

- **Use a multiple of 32 threads per block:** Groups of 32 threads, warps, are the scheduled unit on a GPU. If a warp accesses memory (and will be waiting for it arrive during 600 cycles) it will be swapped out and another warp will take its place. This means that the number of available warps decides how well memory latency can be hidden. A block that is not a multiple of 32 leads to wasting GPU resources; some processing elements will stand idle during the execution of the incomplete warps.

  Obsidian requires regularity even more than CUDA does. Since we focus on generating high performance kernels, we rule out generation of kernels that work on different sizes (per block). The choice of how many threads to use per block is in the hands of the Obsidian programmer. The method for making that choice has changed over time. Earlier, the number of threads needed by a kernel was decided by the size of the arrays it computed. This has recently changed and is now set by a parameter passed to the code generation process.

- **Avoid different execution paths within a block:** Any branching instruction that causes the execution to branch into different code (in CUDA jargon called a diverging branch) within a warp will affect performance. When the execution paths within a warp diverge, the computation is serialised.

Sometimes divergent code cannot be avoided. However, it is important not to have threads diverge unnecessarily. When using pull arrays alone, this was often a problem; with the addition of push arrays, the situation improved. Using push arrays, some divergent computations can be explicitly split up into phases, each phase with all threads following the same execution path. This means that fewer threads are required to do the same work. Push arrays are explained in more detail in section 1.2.4.

- **Do not use `__synchthreads()` in diverging code:** Extreme care must be taken when synchronising within any conditionally executed code; every thread must reach this barrier. Failing to ensure this will likely lead to the kernel producing incorrect results. Note that this is a major limitation to the composability of CUDA functions. Wheather or not it is safe to call some function in a certain place in your code can be decided only by inspecting that function's implementation.

  In Obsidian, it is impossible for a barrier synchronisation to appear within diverging conditional code. Conditionals can be introduced either by an `ifThenElse` function at the element level or by a `Cond` statement that is an operation in the `Program` monad; we have seen an example of the program monad in section 1.2.2. The `Program` monad is parameterised on the GPU hierarchy level. There are thread programs (`TProgram`), block programs (`BProgram`) and grid programs (`GProgram`). the `Cond` conditional chooses between two thread programs while a synchronisation can only appear in an Obsidian block program. More information about programs and the GPU hierarchy is available in section 1.2.4.

The list above provides some very general guidelines on what to think about when writing GPU kernels. The lesson is that fast memory near the processing element should be preferred over slow memory far away. Favouring fast memory demands the effort of decomposing computation and data in a way that makes this possible.

1.2.4 Obsidian Implementation

Obsidian is implemented as an embedded language using Haskell as host. Obsidian is a compiled embedded language; a very good paper that is highly relevant to our approach is "Compiling Embedded Languages" by Conal Elliott et al [EFdM03]. This paper describes how to embed a compiler backend and generate efficient code in some target language. One key aspect of this approach is that the embedded language is a library of functions that create and compose Abstract Syntax Trees (ASTs). Obsidian (chapter 3) as well as EmbArBB (chapter 4) are implemented in a similar way.

An embedding that builds ASTs is called a deep embedding. Shallow embeddings, on the other hand, do not build ASTs. In "Combining Deep and Shallow Embedding for EDSL" Josef Svenningsson and Emil Axelsson combine the two methods, shallow and deep, in order to simplify the AST data type (fewer constructors) and make more extensible embedded languages [SA13]. Obsidian also uses a combination of shallow and deep embeddings.

The embedded language approach has become popular to use as a way of raising the level of abstraction in fields where the default is to use low-level languages. For example, Lava and Wired [KC07, ACS05] raise the level of abstraction in the fields of hardware design and verification and Feldspar does the same in the area of digital signal processing [ACD+10]. For GPU programming, there are numerous embedded approaches using various host languages. Two languages for GPU programming embedded in Haskell are Accelerate and Nikola [CKL+11, MM10]. And there is Intel ArBB, embedded in C++, with the goal of being parallel across platforms [NSL+11]. Related work and information about the tools, libraries and languages that make the wider context surrounding Obsidian can be found in section 1.4.

Deep embedding: The `Program` data type

Obsidian uses a deeply embedded `Program` data type that represents GPU Kernels. A CUDA GPU has a hierarchy of parallel resources. At the bottom there are threads, executing sequential programs. There are groups of threads (of 32) called *Warps* that execute in lock-step. Warps are the scheduled unit of work on a GPU. There are *Blocks* of threads, a set of threads that run as a group and share local (shared) memory. And lastly there is a *Grid* of blocks that specifies the total number of threads involved in a computation (Number_of_Blocks * Number_of_Threads). The deeply embedded program data type of Obsidian models this hierarchy by parameterising programs on a hierarchy level type parameter (the `t` parameter below).

```
data Program t a where
```

The `t` parameter can be either `Thread`, `Warp`, `Block` or `Grid`. These types are related to each other via a `Step` type constructor that represents going upward one step in the hierarchy.

```
data Step a -- A step in the hierarchy
data Zero
```

The `Thread` type is level `Zero` (`type Thread = Zero`). Then `Block` is `Step Thread` and `Grid` is `Step Block`. Currently `Warp` is not included in the hierarchy, leading to warp programming being a special case. One benefit a warp

has is that threads can communicate via shared memory without using synchronisation primitives. There is no programmer support for warp level programming in CUDA (as there is for block level programs). The programmer needs to take care to ensure that the communication patterns are within a warp and is then free to remove synchronisations. If warps were to be put in the program level hierarchy of Obsidian, its place would be between the thread and block level. This would force the programmer to go via warps when putting together a block computation even if the communication pattern is not suited for that, leading to inconvenience with no gain.

As an example of how the hierarchy level parameter is used, I show the `ForAll` constructor from the `Program` data type. This constructor represents parallelism either over threads or blocks.

```
ForAll :: EWord32
            -> (EWord32 -> Program t ())
            -> Program (Step t) ()
```

`ForAll` takes a number of parallel iterations, and a body represented by a function from an index to a program. The body is a `Program` at some level `t` while the resulting program is a step above. This means that a thread program can be turned into a block program by using `ForAll`, or that a block program can be turned into a grid program.

Information about the `Program` data type can be found in paper F, A High-Level Embedded Language for Low-Level GPU Kernel Programming, in section 3.5.

Scalars

Scalars and operations on scalars are represented by an expression data type (`Exp a`) in Obsidian. This is another example of a deep embedding; the expression data type contains constructors for literal values, arithmetic operations, and conditionals.

Haskell's type class system allows us to overload arithmetic operations such as `(+)`, `(-)` and `(*)` by making expressions an instance of `Num`. This allows the Obsidian arithmetic expressions to look exactly like corresponding native Haskell arithmetic.

The approach taken to embed the scalar language is very similar to what is described in "Compiling Embedded Languages" [EFdM03]; but we use a Generalised Algebraic Datatype (GADT) to obtain typed expressions. An alternative to using a GADT is to use phantom types, which earlier versions of Obsidian did (section 3.1).

Arrays

Obsidian has two array representations. These array representations are implemented as shallow embeddings on top of the expression and program data types. This means that these arrays will disappear during Haskell evaluation of the Obsidian program.

Pull arrays: Pull arrays have been part of Obsidian from the very beginning [Sve08] but called either `Array` or just `Arr`. A pull array represents an array as an indexing function; given an index it provides an element:

```
data Pull s a = Pull {pullLen :: s,
                      pullFun :: EWord32 -> a}
```

Pull arrays are parameterised on both element (`a`) and length (`s`) type. The length parameter is used to be able to represent arrays with static (known at Haskell runtime) or dynamic (length represented by an expression, an unknown length). When creating block level computations, we want to know the exact lengths in order to avoid generating code riddled with conditionals concerning array lengths. In CUDA, it is possible to write kernels that are not specialised for a certain size. This is accomplished by adding bounds checks. Since conditionals are problematic from a performance point of view, fixed size kernels are recommended when performance is crucial. In Obsidian, we have chosen to disallow implementation of kernels that operate on different sizes locally. Dynamic lengths are used for grid level computations; this allows a local computation of a fixed size to be run in parallel over arrays a multiple of the size that the local computation handles. In generated code, this is currently visible as a conditional on `blockIdx.x`. However, this is still considered work in progress; the conditional is only really necessary for Obsidian kernels that have more than one output array and the outputs are of different length.

One benefit of pull arrays is that they give fusion of all operations on them for free. For example, `map f` is implemented on pull arrays by composing `f` with the indexing function:

```
map f (Pull n ixf) = Pull n (f . ixf)
```

Now, we see that `map f . map g` is the same as `map (f . g)` since both result in `f` and `g` becoming composed onto the indexing function. No intermediate array is built in memory.

Another benefit of pull arrays is that they are parallel. Simply put, early Obsidian was pull arrays in addition to a way to compute the array and store its elements to memory (called `sync` in early versions of Obsidian and later `force`). A very direct way to compile a pull array to a GPU is to launch as many threads as there

are elements in the array and apply the indexing function to the thread id. Now each thread will have computed an element of the array and can store that to memory.

Push arrays: Push arrays were added to Obsidian as a complement to pull arrays. Pull arrays are good because they are so simple to understand and easy to parallelise. However, certain operations on pull arrays yield code that is not suitable for GPU execution. Concatenation and interleaving of pull arrays results in conditionals in the indexing function. The interleaving case, which is worst, leads to diverging branching in every warp. If branches diverge, the GPU turns the processing elements taking one path off and allow the others to progress; then turn to the other branch. The computation of the branches is serialised and compute resources are wasted.

Push arrays, like pull arrays, are functions. A Push array is a higher order function, taking a function from value and index to a thread program and giving as result a program at some level in the hierarchy. The function passed to the push array function we call a write-function. Below is the definition of push arrays that we use in Obsidian. Push arrays are parameterised on element type (`a`), size type (`s`) and program hierarchy level (`p`).

```
data Push p s a =
  Push s ((a -> EWord32 -> TProgram ()) -> Program p ())
```

When working with pull arrays, the consumer of such an array decides how to iterate over the array and what elements to compute. This is done by applying the push array function to those indices that are interesting to the consumer. Push arrays work in the opposite way. The Push array (the result program) contains the iteration schema; the consumer of the push only decides what write function to supply. The iteration schema used is decided upon when creating a push array. The code below is an example of how to create a push array from a pull array.

```
convertToPush :: Pull Word32 e
               -> Push Block Word32 e
convertToPush (Pull n ixf) =
  Push n $ \wf ->
    ForAll (fromIntegral n) $ \i -> wf (ixf i) i
```

The function above takes a pull array and creates a push array with a parallel iteration schema over threads within a block. There are other iteration schemas to chose from such as sequential, combinations of sequential and parallel across both threads and blocks (two level parallelism).

Push arrays give many of the same benefits as pull arrays. Using push arrays, we also get fusion of operations as default. The implementation of map (shown below)

illustrates how mapping a function `f` over a push array results in an application of `f` in the write-function.

```
map f (Push s p) =
  Push s $ \wf -> p (\e ix -> wf (f e) ix)
```

Compilation to CUDA

When compiling Obsidian programs to CUDA, we only need to be concerned with the `Program` datatype. Given the hierarchy level type parameters to the program datatype, we know that the AST can be quite directly translated to CUDA. We know that there will be at most one level of nestedness in the parallelism (several blocks, each of several threads). This means that we do not need to do any transformations on the AST before generating CUDA code. Paper F (section 3.5) contains technical details of the compilation process.

1.2.5 Results

We have implemented basic algorithms such as sorting, merging, fractal generation, reductions and parallel prefix (scans). Often we have managed to get very good performance, close to that of hand optimised CUDA code. GPU execution of scan operations is something that has been worked on extensively in the GPGPU community and very efficient implementations exist [MG09, HS07, BOA09, SHG08]. In Paper F (section 3.5), we only study scan kernels (scan of small arrays) and measure their speed relative to each other. Compared to a scan kernel supplied with the CUDA API, our fastest kernel is slightly more than three times slower. Performance needs to be improved at the kernel level before we attempt composing them into large scan operations. We hope to improve this situation with the addition of the warp level abstractions in Obsidian (this is future work, section 1.5).

Paper F (section 3.5) tells a story about how to implement and tune a reduction kernel in Obsidian. Seven reduction kernels are implemented, the first very simple and naive. Each following kernel tweaks some aspect of the previous one. Applying these changes is pleasingly simple using Obsidian; some of these changes would mean a major rewrite in CUDA. The final kernel is almost 5 times as fast as the first, and reaches a throughput of 140GB/s on a GPU with 192GB/s memory bandwidth. The speed of any kernel will always be limited by how fast the data can be pushed through the GPU (see reference [MG09] for an in depth discussion of kernel performance); this performance is very satisfactory. In Paper F, a large reduction algorithm based on this kernel is implemented that runs slightly faster than Accelerate's fold skeleton. Even being just nearly as fast as Accelerate would in this

case have been fine, since the fold skeleton is hand-tuned CUDA code. With this reduction case study, we have pushed Obsidian as far as possible with the current abstractions. However, it is possible that with more work on the warp level abstractions, this could be improved further. Our reduction case study was based on lecture slides from NVIDIA [Har].

In paper E (section 3.4), we implement counting sort and occurrence sort using Obsidian. We use very low-level features of Obsidian and end up producing code that performs very well.

Paper D (section 3.3) introduces push arrays and uses them in the implementation of sorting networks, with quite good results. With push arrays, we were able to experiment with more detailed decompositions (what thread computes what value) of the algorithms.

1.3 Embedding Intel Array Building Blocks

Intel ArBB [NSL+11] is a system for high-level data-parallel programming with the ability to generate code for a variety of different hardware configurations. It is implemented as an embedded language in C++. One motivation for ArBB that is mentioned in [NSL+11] is that while high-performance computing specialists are highly competent at implementing kernels using low-level programming techniques, mainstream developers are not. ArBB offers a more composable way to write programs that make use of core and vector (SIMD) parallelism and doing this while using a familiar language, C++.

One of the greatest strengths of ArBB, as I see it, is that it also comes with a low-level C interface. The main purpose of this C interface is to make it easier to use the ArBB system from other languages. Many languages have foreign function interfaces that are geared towards C. This gives ArBB a level of language independence on top of its cross platform abilities.

ArBB is based on a set of parallel primitives (or structures) on dense one, two and three dimensional vectors and nested vectors:

- **Reductions:** `add_reduce`, `mul_reduce`, `max_reduce` ...
- **Scans:** `add_scan`, `mul_scan`, `max_scan` ...
- **Sorting:** `sort`
- **Permutations:** `gather`, `scatter`, `reverse` ...
- **Sequential loops:** `_for`, `_while`
- **Map, zipWith, stencil:** are all supported via a single primitive called `map`.

These operations (and many more, there are about 120 operations in total) can be used by the programmer when writing programs using ArBB. Note that there are

sequential looping constructs with special names, `_for` and `_while`. ArBB is a deeply embedded language and calling the ArBB functions does not actually execute them, it just adds a node to some internal representation. Therefore, the use of normal C++ for loops leads to unrolled ArBB programs and a special `_for` loop is needed to get a loop in the generated program. This is exactly the way these things work in Haskell embedded languages, but it may be even more surprising to a C++ programmer. Before an ArBB function can be computed, it needs to be captured. For this, ArBB has a `call` function that is used to run functions using ArBB functionality. The first time `call` is used on a function pointer, that function is executed and the ArBB functions used in it build an AST (we say the function has been captured). The AST is then compiled and specialised for the hardware available. Any subsequent calls of a function using ArBB functionality do not lead to a recompilation; as the first call caches this compiled code.

The first step of our work with embedding Intel ArBB functionality was to implement very direct and low-level bindings to the C interface. This provides a Haskell function (in the IO Monad) for each of the functions in the ArBB C interface. More details and results of this work can be found in paper G in section 4.1.

1.3.1 EmbArBB: Embedding ArBB in Haskell

The ArBB Haskell bindings provide a very rudimentary interface to ArBB functionality and are not useful for application implementation. We need a higher-level interface to ArBB to offer to the Haskell programmer. Our first attempt was to implement an Accelerate backend using the ArBB bindings. However, this had some hard problems to solve. Accelerate is a richer language and there was an API mismatch between Accelerate and ArBB. These problems are mentioned in paper G, section 4.1. As a way to offer ArBB functionality to the Haskell programmer in a way that is useful and simpler to implement, we started working on EmbArBB.

EmbArBB improves the interfacing with Haskell by implementing a deeply embedded language. Now, programs in EmbArBB create ASTs. These ASTs are compiled via the ArBB bindings, creating function objects that can then be run.

Dense and nested arrays

ArBB supports one, two and three dimensional arrays, called dense vectors. In EmbArBB, we represent these with a data type, `DVector`, parameterised by dimensionality and element type. The dimension parameter can be either `Dim1`, `Dim2` or `Dim3`. There are also nested vectors, in EmbArBB called `NVector`, parameterised on element type.

Operations on arrays

EmbArBB provides many of the ArBB operations on vectors via a shape-polymorphic interface. For example, a reduction operation can take a two dimensional vector and reduce all rows or columns and provide a one dimensional result. It can also take a one dimensional vector and return a single value (zero dimensional vector).

```
addReduce :: Num a
          => Exp USize
          -> Exp (DVector (t:.Int) a)
          -> Exp (DVector t a)
```

The `addReduce` function takes an input vector that is at least one dimensional (`t:.Int`) and returns a vector that has one less dimension (`t`).

Deep embedding

EmbArBB is a traditional deep embedding; there is an AST data type with constructors representing each ArBB language feature. It is true that a deep embedding of ArBB leads to duplication of effort. The low-level operations exported by the ArBB C interface already construct an AST within the ArBB system. One possible benefit of the deep embedding is that it buys independence from the ArBB backend. It is possible to compile the AST generated on the Haskell side to some other target language.

When compiling EmbArBB to ArBB, a sharing detection pass is performed. This pass detects sharing in the AST using the method of A. Gill [Gil09]. One benefit of this pass is that it reduces the number of calls into the ArBB C library. The efficiency of code generated is probably not affected, since ArBB would discover that sharing in a Common Subexpression Elimination (CSE) pass. Another potential benefit is that sharing detection on the Haskell side results in a simpler AST being built on the ArBB side, potentially making the job for the ArBB optimisation passes simpler. What effects these operations truly had on performance has not been investigated. Actually, applying a sharing detection technique was one of my personal motivations for working on EmbArBB. In Obsidian, we have not seen the need for this technique. In Obsidian, sharing is in the hands of the programmer, via the `force` operation. On a GPU, communication is more expensive than computation, so whether to share a computation or not becomes a tool for the programmer to use.

One benefit of a deep embedding is that transformations can be performed on the AST. Applying any optimisations in EmbArBB would very likely duplicate effort already put into ArBB. Therefore, we apply no further transformations after sharing

detection. However, if EmbArBB would be used to generate code in some other target language a set of optimisations would be needed.

Evaluation

In paper H, section 4.2, we benchmark EmbArBB against ArBB in C++ and the Haskell Repa library. EmbArBB compares favourably to Repa and is often a lot faster. The ArBB C++ performance is not distinguishable from EmbArBB's, indicating that our Haskell embedding adds no extra overhead. Of course, this is expected, since once the ArBB system has generated the code, that code is identical no matter if the AST was built via Haskell or C++ calls. A comparison of how well EmbArBB and C++ compares in regards to the time it actually takes to go through the process of making those ArBB C API calls has not been performed. However, if the generated code is used many times, ArBB's caching of generated code means that the initial code generation cost can be amortised.

1.3.2 Current status of ArBB and EmbArBB

Since our work with ArBB, Intel has unfortunately retired the ArBB system. This leaves EmbArBB without a functioning code generating backend. This makes the choice of a deep embedding (in hindsight) the correct one. EmbArBB could be resurrected by implementing a new code generating backend. However, a complete reimplementation of the ArBB API would be a massive undertaking.

1.4 Related work

1.4.1 Languages for data-parallel programming

In figure 1.3, our work is placed in relation to other languages, embedded languages and libraries for parallel programming. The languages and libraries considered are data-parallel; languages for control parallelism or distributed message passing systems are not considered. The systems are roughly divided into three groups, based on their level of abstraction. There are low-level languages; here I place languages that are imperative and C-like. In the middle, layer I place languages that have higher level abstractions and are more easily composable. at the highest level, I place languages that completely abstract away from details of the hardware they run on. The division is not free from personal bias and the placement of languages in boxes was not always easy.

The figure also divides the languages according to what hardware they support. There are CPU specific languages, GPU specific languages and languages that support or aspire to support CPU,GPU and accelerator (Larrabee, Xeon Phi) execution.

With Obsidian, we try to target the sparsely occupied area of mid-level GPU programming. With CUDA [NVIc] and OpenCL [TKG], the programmer has full control of how to divide the computation amongst threads and blocks. Using Accelerate [CKL+11], Nikola [MM10] and Thrust [NVId], the programmer uses high-level patterns, without direct insight into how the application is mapped onto the threads and blocks of the GPU. Using Obsidian, we want to bridge that gap by both being more composable than CUDA and giving the programmer control of what the computation will look like on the GPU. This is done by allowing the programmer to specify and compose thread-level and block-level code. Recently, the CUB library arrived that is built on a similar idea, that GPU programmers need to have block/thread level control to get maximum performance, while maintaining a higher level and more Thrust-like programmer interface [Mer]. Before CUB, and as far as I am aware, we were alone at targeting this level.

An in depth comparison of Obsidian and Accelerate is hard to perform. The reason for this is that Obsidian is a language for kernels, while Accelerate is a language for applications. Any larger GPU application consists of a number of kernels that are cooperating. Obsidian only has very basic support for launching and coordinating kernels for such applications from within Haskell, and Accelerate is limited in specifying new specialised kernels.

The ArBB and EmbArBB systems also provide a set of parallel primitives that the programmer composes to build his application. I place ArBB and EmbArBB in a box slightly lower than Accelerate, since while ArBB/EmbArBB also have built in reduction primitives, just like Accelerate, they are not general primitives. Where Accelerate has a single higher order reduction operation, ArBB/EmbArBB has `reduce_add, reduce_mul` and so on.

Microsoft Accelerator [TPO06] is a language embedded in C#. Accelerator has data-parallel arrays and a set of aggregate operations (operations on entire arrays). The operations on data-parallel arrays are JIT-compiled to GPU code. Accelerator can also compile to multithreaded CPU code and make use of SIMD units of modern CPUs [Mic]; this makes the Accelerator system related to ArBB.

There are many embedded languages implemented in Python that make use of GPUs to improve performance. One example is Copperhead [CGK11]. Copperhead supports flat and nested data-parallelism via primitives such as `map` and `reduce`.

Repa (Regular Shape-polymorphic Arrays) [KCL+10] is a Haskell library for data-parallel programming. Repa uses the same array representation as Obsidian, in Repa called a delayed array, which is a function from index to element. However,

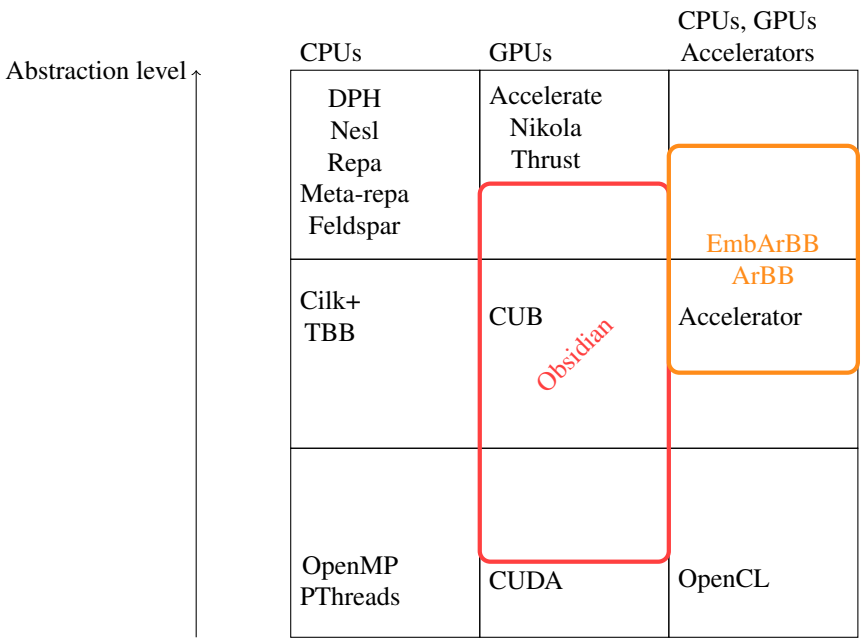

Figure 1.3: Placing our work in the landscape of languages and libraries for parallel general purpose programming of CPUs, GPUs or both.

in Repa arrays have shape; they can be of any dimensionality and there are shape-polymorphic functions on these. For example, `map` can be applied to an array of any shape.

Meta-Repa [AS13] is a reimplementation of the Repa library using deep embedded language techniques. Meta-Repa programs build ASTs that are then compiled (using template Haskell) to Haskell code that uses low-level parts of the Repa library for its parallelism. The Meta-Repa approach to implementing the Repa library gives inlining of operations for free, where Repa programs need to be annotated with pragmas to ensure inlining. On the other hand, in Meta-Repa the programmer needs to explicitly state when something should not be inlined (using a `force` primitive). This is related to our work on EmbArBB that also uses an existing library (ArBB) for its parallelism. In reference [KN13], a case is made for this style of embedding preexisting DSLs.

Data-Parallel Haskell [JLKC08] and Nesl [Ble96] are languages for nested data-parallelism. Both run on CPUs, but there is also work towards a GPU version of Nesl [BR12].

Delite [BSL+11] is an infrastructure for implementation of domain specific embedded languages. There is an extensible internal representation (IR,AST); the programmer extends some base IR with domain specific constructs. The Delite system has been used to implement domain specific embedded languages for machine learning, rendering, physics and other domains. Delite, being a framework and the languages implemented using it very domains specific, was impossible to place in the matrix (Figure 1.3).

1.4.2 Embedded domain specific languages

Feldspar is an embedded language for digital signal processing [ACS+11]. Apart from using the syntactic library [Axe12], it is quite similar to Obsidian. There is a deeply embedded core language (called Core) and vectors as a shallow embedding on top of that, just like in Obsidian.

When we first started working on Obsidian, we used Lava [BCSS98] as inspiration. We wanted to mimic the Lava programming style but generate GPU code. At first this worked quite well and most Obsidian programs superficially looked quite like Lava programs. We have diverged from this aspiration since realising that GPUs are quite a different story and performance requires more control of GPU architecture specifics.

1.4.3 Compiling Embedded Languages

In paper A (section 2.1), we describe our method for compiling monadic embedded languages. This work is closely related to a method described in "The Constrained Monad Problem" by Sculthorpe and Gill [SBGG13]. In that paper, they solve a more general problem than just compilation, but our method is nice because it is easy to understand and solves a single problem and does it well. In "Generic monad constructs", Persson et al also solve the problem of compiling monads using a continuation monad.

"Compiling Embedded Languages" by Elliott and Finne has been a constant source of information and something we have referred to in almost all our work. This is, in my opinion, the paper to turn to for anyone starting out with compiled embedded languages.

1.5 Future work

Obsidian is a language for GPU kernel implementation, not complete applications. It is not as well integrated into Haskell as for example Accelerate. There are ways to execute Obsidian kernels from within Haskell (we have seen an example of this in section 1.2.2); the recommended work flow, however, is to generate the CUDA as text and write the host code by hand. This can be seen as both a shortcoming and a feature. It is hard to integrate Accelerate programs in larger applications written in other languages; when using Obsidian a kernel is generated and can be used as part of for example a library.

Obsidian is also quite a low-level language. The programmer needs to know GPU details and is allowed to write programs at a level almost as low as CUDA, if it is desired. Paper E (section 3.4) contains examples of programming at this low level. One potential use of Obsidian could be as the code generating backend of high-level and very domain specific embedded languages. This could be a niche for Obsidian.

Obsidian constrains the programmer into writing programs that we can easily compile to CUDA. This is done by modelling the GPU programming in types attached to programs. A currently ongoing master's project is investigating what effects removing these types have. It is clear that if the programmer is not constrained to writing flat (or very limited nested) programs, transformations are needed on the AST before generating CUDA code.

Obsidian's support for warp level computations is work in progress. Currently, warp level computations are not as well integrated into the hierarchy as the block, thread and grid levels are. The warp abstractions offered also require more work in the code generating backend than those that correspond directly to CUDA concepts.

The problem is that CUDA does not have a warp abstraction. Warps needs to be better integrated with the rest of Obsidian and the code generation required for warps needs more testing before being trustworthy and useful. When operational, warp level abstractions will help the programmer make better use of the GPU.

1.6 Discussion

This thesis contains two approaches to embedding a language for data-parallel programming in Haskell. EmbArBB embeds a preexisting library for data-parallelism in Haskell and, at little effort from us, very good performance is gained. The performance of EmbArBB is entirely due to all the optimisation expertise and effort that was invested in the ArBB system. We get all that expertise for free. What EmbArBB provides is a way for Haskellers to benefit from the ArBB system. ArBB and EmbArBB are based on a set of parallel primitives for the programmer to use and compose. Obsidian takes a different approach. Built in primitives are very few and very low level, such as sequential and parallel for loops. On top of the low-level capabilities, a library of high-level primitives can be built. The key is that we want the programmer to be able to specify how to implement these higher level abstractions.

Looking at the related work, we see that almost all high level GPU programming languages provide abstractions of parallel primitives, aggregate operations, maps, folds and reductions, giving justification to our investigation of GPU programming without these. We believe that the GPU architecture is still volatile enough that there is no "the right" way of implementing these primitives. As of today, the programmer still needs to explore the design space and find a solution that performs well on a particular GPU.

Performance of Obsidian kernels is often good and the Obsidian code that they are generated from is often shorter than corresponding CUDA programs. Obsidian programs differ from CUDA programs by describing whole programs. A CUDA programmer needs to take a leap from the single threaded program parameterised on thread ID, to what actually will happen when this program is run by hundreds or thousands of threads at once. The extensive indexing arithmetic often present in CUDA programs can make this leap a hard one. Similar programs in Obsidian could use operations on whole arrays such as `map` or explicitly bounded parallel loops.

Push arrays were invented by Koen Claessen and first implemented in Obsidian to provide more control over where computation takes place (which thread computes which element). Push arrays has since been experimented on by others and show up in Nikola [Mai] and Feldspar [APS]. Push arrays appear in StreamHS [KN13] and in [Els] as part of a multi-dimensional array calculus in Standard ML.

Bibliography

[ACD+10] Emil Axelsson, Koen Claessen, Gergely Dévai, Zoltán Horváth, Karin Keijzer, Bo Lyckegård, Anders Persson, Mary Sheeran, Josef Svenningsson, and Andras Vajda. Feldspar: A domain specific language for digital signal processing algorithms. In *8th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, pages 169–178. IEEE Computer Society, 2010.

[ACS05] Emil Axelsson, Koen Claessen, and Mary Sheeran. Wired: Wire-aware circuit design. *Correct Hardware Design and Verification Methods*, pages 5–19, 2005. Springer.

[ACS+11] Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. The Design and Implementation of Feldspar an Embedded Language for Digital Signal Processing. In *Proceedings of the 22nd international conference on Implementation and application of functional languages*, IFL'10, pages 121–136, Berlin, Heidelberg, 2011. Springer-Verlag.

[APS] Emil Axelsson, Anders Persson, and Josef Svenningsson. Push arrays in Feldspar. `github.com/Feldspar/feldspar-language/blob/d93eafce62afc7a32945a6b9eefe6368310f414f/src/Feldspar/Vector/Push.hs`.

[AS13] Johan Ankner and Josef David Svenningsson. An EDSL approach to high performance Haskell programming. In *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell*, Haskell '13, pages 1–12, New York, NY, USA, 2013. ACM.

[Axe12] Emil Axelsson. A generic abstract syntax model for embedded languages. In *17th ACM SIGPLAN Conf. on Functional Programming*. ACM, 2012.

[BCSS98] 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.

[Ble96] Guy Blelloch. Programming Parallel Algorithms. *Communications of the ACM*, 39(3), 1996.

[BOA09] Markus Billeter, Ola Olsson, and Ulf Assarsson. Efficient stream compaction on wide SIMD many-core architectures. In *Proc. Conf. on High Performance Graphics*, HPG '09, pages 159–166. ACM, 2009.

[BR12] Lars Bergstrom and John Reppy. Nested data-parallelism on the GPU. In *17th ACM SIGPLAN Conf. on Functional Programming*. ACM, 2012.

[BSL+11] Kevin J. Brown, Arvind K. Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 89–100, Washington, DC, USA, 2011. IEEE Computer Society.

[CGK11] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. In *Proc. of Principles and practice of parallel programming*, PPoPP '11, pages 47–56, New York, NY, USA, 2011. ACM.

[CKL+11] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell Array Codes with Multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, DAMP '11, pages 3–14, New York, NY, USA, 2011. ACM.

[EFdM03] Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(2), 2003.

[Els] Martin Elsman. Multi-dimentional array calculus in standard ml. `github.com/melsman/MoA`.

[Gil09] Andy Gill. Type-Safe Observable Sharing in Haskell. In *Proceedings of the 2009 ACM SIGPLAN Haskell Symposium*, 09/2009 2009.

[Har] Mark Harris. Optimizing parallel reduction in CUDA. `"http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf"`.

[HS07] Mark Harris and Shubhabrata Sengupta. Parallel Prefix Sum (Scan) with CUDA. In *GPU Gems 3*, 2007.

[JLKC08] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M T Chakravarty. Harnessing the multicores: Nested data parallelism in

haskell. In Ramesh Hariharan, Madhavan Mukund, and V Vinay, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2008)*, volume 2 of *Leibniz International Proceedings in Informatics*, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[KC07] Mary Sheeran Koen Claessen. A slightly revised tutorial on lava: A hardware description and verification system. Technical report, Chalmers University of Technology, 2007.

[KCL+10] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 261–272, New York, NY, USA, 2010. ACM.

[KN13] Abhishek Kulkarni and Ryan R. Newton. Embrace, Defend, Extend: A Methodology for Embedding Preexisting DSLs, 2013. Functional Programming Concepts in Domain-Specific Languages (FPCDSL'13).

[Mai] Geoffrey Mainland. Push arrays in Nikola. `github.com/mainland/nikola/blob/d86398888c0a76f8ad1556a269a708de9dd92644/src/Data/Array/Nikola/Repr/Push.hs`.

[Mer] Duane Merrill. CUB. `http://nvlabs.github.io/cub/`.

[MG09] Duane Merrill and Andrew Grimshaw. Parallel Scan for Stream Architectures. *University of Virginia, Department of Computer Science, Charlottesville, VA, USA, Technical Report CS2009-14*, 2009.

[Mic] Microsoft Research. Accelerator v2. `http://research.microsoft.com/en-us/projects/accelerator/`.

[MM10] Geoffrey Mainland and Greg Morrisett. Nikola: Embedding Compiled GPU Functions in Haskell. In *Proceedings of the third ACM Haskell symposium*, pages 67–78. ACM, 2010.

[MM12] James Reinders Michael MCCool, Arch D. Robinson. Structured Parallel Programming: Patterns for Efficient Computation, 2012. Morgan Kaufmann.

[NSL+11] Chris J. Newburn, Byoungro So, Zhenying Liu, Michael McCool, Anwar Ghuloum, Stefanus Du Toit, Zhi Gang Wang, Zhao Hui Du, Yongjian Chen, Gansha Wu, Peng Guo, Zhanglin Liu, and Dan Zhang. Intel's array building blocks: A retargetable, dynamic compiler and embedded language. In *Proceedings of the 2011 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 224–235, Washington, DC, USA, 2011. IEEE Computer Society.

[NVIa] NVIDIA. CUDA C Best Practices Guide. `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/`.

[NVIb] NVIDIA. CUDA C Programming Guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[NVIc] NVIDIA. NVIDIA CUDA. `http://www.nvidia.com/cuda`.

[NVId] NVIDIA. NVIDIA Thrust Library. `https://developer.nvidia.com/thrust`.

[SA13] Josef Svenningsson and Emil Axelsson. Combining Deep and Shallow Embedding for EDSL. In Hans-Wolfgang Loidl and Ricardo Peña, editors, *Trends in Functional Programming*, volume 7829 of *Lecture Notes in Computer Science*, pages 21–36. Springer Berlin Heidelberg, 2013.

[SBGG13] Neil Sculthorpe, Jan Bracker, George Giorgidze, and Andy Gill. The Constrained-Monad Problem, 2013. 18th ACM SIGPLAN international conference on Functional programming, ICFP 2013.

[SHG08] Shubhabrata Sengupta, Mark Harris, and Michael Garland. Efficient parallel scan algorithms for gpus. Technical report, 2008.

[Sve08] Joel Svensson. An embedded language for data-parallel programming, 2008. Master's thesis, Göteborg University, `http://www.cse.chalmers.se/~joels/writing/joelmscthesis.pdf`.

[TKG] The Khronos Group. OpenCL - The open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`.

[TPO06] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. volume 34, pages 325–335, New York, NY, USA, October 2006. ACM.

Chapter 2

EDSL Implementation Papers

2.1 Paper A: Simple and Compositional Reification of Monadic Embedded Languages

Josef Svenningsson, Bo Joel Svensson

Abstract

When writing embedded domain specific languages in Haskell, it is often convenient to be able to make an instance of the `Monad` class to take advantage of the do-notation and the extensive monad libraries. Commonly it is desirable to compile such languages rather than just interpret them. This introduces the problem of monad *reification*, i.e. observing the structure of the monadic computation. We present a solution to the monad reification problem and illustrate it with a small robot control language. Monad reification is not new but the novelty of our approach is in its directness, simplicity and compositionality.

2.1.1 Introduction

Benny is a computer science student working in a project involving programming robots in a low-level imperative language. However, Benny has a budding interest in functional programming using Haskell and has read "The Haskell school of expression" [Hud99]. He gets the idea to implement a language for robot control embedded in Haskell. Benny realises that the capabilities of the robot hardware are very similar to those of the robots that Hudak evaluates graphically in a grid world. There is however a very important difference; the embedded language needs to be compiled into some form that is understood by the robot.

Guided by the capabilities of the target robot, Benny designs an API for robot programming. The robot can perform operations such as *move* that steps the robot forward and *turn* left and right. The robot also has the capability to execute program loops and conditionals and it has a forward facing `sensor` with which it can query the world.

The API that Benny designs is simple:

```
move      :: Program ()
turnLeft  :: Program ()
turnRight :: Program ()
sensor    :: Program Bool
cond      :: Program Bool
             -> Program ()
             -> Program ()
             -> Program ()
while     :: Program Bool
             -> Program ()
             -> Program ()
```

Now, Benny wants to express robot programs using the Haskell *do* notation. The motivation behind this is the imperative look and feel of the operations he identified and the potential to use all the control structures in the *Control.Monad* library. For this, a `Monad` instance is needed.

To be able to create a first-order representation of a computation described using monads, Benny needs to solve the problem of *reifying* monads. This is the story of how Benny came up with a particularly simple solution to this problem.

Example programs

Benny is quite happy with his language design and decides to write some programs to try it out.

The first program is called `sMove`, which implements a safe move operation. This operation can be executed by the robot even though it is facing an obstacle, without risk of harming robot or obstacle.

```
sMove :: Program ()
sMove = cond sensor turnRight move
```

The second program moves the robot forward until it stands directly in front of an obstacle such as a wall.

```
moveToWall :: Program ()
moveToWall = while ((liftM not) sensor) move
```

Based on these examples, Benny is quite satisfied with the design of his language and turns to implementing it.

Implementation and data structures

Enthused by the prospect of compiling these programs to the target language and seeing some robot action, Benny goes to work on the data structures.

Since the language is going to be compiled, Benny realises that the booleans in his language cannot be regular Haskell booleans. The booleans needs to be replaced by boolean typed expressions.

```
type Name = String

data BoolE = Lit Bool
           | Var Name
           | (:||:) BoolE BoolE
           | (:&&:)  BoolE BoolE
           | Not BoolE
```

Following this, the operations that should go into the `Program` data type feel straightforward.

```
data Program a where
  Move      :: Program ()
  TurnLeft  :: Program ()
  TurnRight :: Program ()
  Sensor    :: Program BoolE
  Cond      :: Program BoolE
               -> Program ()
               -> Program ()
               -> Program ()
  While     :: Program BoolE
               -> Program ()
               -> Program ()
```

Benny continues by naively adding constructors for `Return` and `Bind` to the `Program` data type.

```
data Program a where
  ...
  Return :: a -> Program a
  Bind   :: Program a
            -> (a -> Program b)
            -> Program b
```

Then he writes down the `Monad` instance.

```
instance Monad Program where
  return = Return
  (>>=)  = Bind
```

Proud of his accomplishments, Benny sends his Haskell module in an email to Prof. Björn. Benny knows that Björn is teaching an introductory Haskell course and should be able to provide feedback.

Problem statement

Meanwhile in Professor Björn's Office

Prof. Björn notices an email from Benny in his inbox and opens it.

> Dear Professor Björn
>
> I am CS student working in a robot control project. Usually, we program our robots in C but I have developed an interest in Haskell programming and thought it'd be natural to try implementing an EDSL. I know you teach an FP course and thought I would ask for your input. Attached to this mail is a file containing an outline of the data types I want to use. Does this look sensible to you?
>
> Thank you
> Benny

Prof. Björn opens the attachment and takes a look at the data type. He particularly notices the constructors `Return` and `Bind` and the `Monad` instance. He shakes his head at Benny's naiveté and writes an email back:

Hello Benny

I'm afraid your implementation of the monadic primitives can never work. The constructor `Return` can take any arbitrary value that has nothing to do with the language you're designing. These values may be strings, binary trees or higher-order functions from zygomorphisms to histomorphisms. The same problem goes for `Bind`; it has to be able to handle arbitrary values. It cannot possibly work! This is a very difficult problem and such a naive solution is bound to fail.

Prof. Björn
Dept. of Computer Science and Engineering
Chalmers University of Technology

While waiting for feedback from Björn, Benny has carried on trying to implement a compiler for his language.

When Björn's email does arrive, Benny is confused. He feels he has already managed to compile the `Program` data type into a representation closer to that which is executed by the robot hardware.

He writes an apprehensive email back to Prof. Björn.

Hello Prof. Björn

Thanks for your feedback. But I think it does work! Attached to this email you find a file containing my attempt at compiling the `Program` data type into a representation closer to that which is executed by our robots.

I have also attached two example programs that you can compile and then run in the graphical simulator (see figure 2.1)

Benny

2.1.2 Compilation of the monadic robot EDSL

Björn receives Benny's latest email

```
spiralIn :: Int -> Program ()
spiralIn 0 = return ()
spiralIn n = do
  replicateM_ 2 $ do
    replicateM_ n move
    turnLeft
  spiralIn (n-1)
```

```
followWall :: Program ()
followWall =
  while (return true) $
    cond checkLeft sMove $
             do turnLeft
                move

checkLeft :: Program BoolE
checkLeft = do
  TurnLeft
  s <- Sensor
  TurnRight
  return s
```

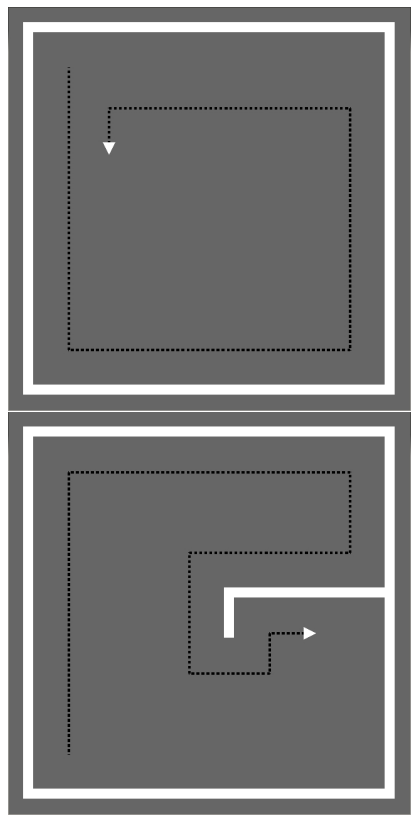

Figure 2.1: The `spiralIn` and `followWall` programs sent from Benny to Björn in an email

```
newNameSupply :: NameSupply
split2 :: NameSupply ->
          (NameSupply,NameSupply)
split3 :: NameSupply ->
          (NameSupply,NameSupply,NameSupply)
supplyValue :: NameSupply -> Int
```

Figure 2.2: The interface of the splittable name supply used in the implementation of the compile functions

Björn looks through the `Compiler` module and finds a data type describing a first-order representation of the robot language.

```
data Prg = PMove
         | PTurnRight
         | PTurnLeft
         | PSensor Name
         | PCond BoolE Prg Prg
         | PWhile Name Prg Prg
         | PSeq Prg Prg
         | PSkip
         | PAssign Name BoolE
```

From the `Prg` data type, Björn presumes that the `PSeq` constructor will be the result of compiling `Bind` and that `PSensor` binds a variable which can be used in `PAssign` and `PWhile`. The conclusion is that the representation looks sensible, but he is very interested in seeing the `Program a -> Prg` transformation. The compile function assumes a splittable name supply with the interface described in figure 2.2.

```
runCompile :: Program a -> Prg
runCompile prg = snd $ compile s prg
  where
    s = newNameSupply

compile :: NameSupply -> Program a -> (a, Prg)
compile s Move = ((),PMove)
compile s TurnRight = ((),PTurnRight)
compile s TurnLeft  = ((),PTurnLeft)
```

```
compile s Sensor    = (Var nom,PSensor nom)
  where
    v   = supplyValue s
    nom = "v" ++ show v
compile s (Cond b p1 p2) =
    ((),bp `PSeq` PCond b' p1' p2')
  where
    (s1,s2,s3) = split3 s
    (b',bp)    = compile s1 b
    (a1,p1')   = compile s2 p1
    (a2,p2')   = compile s3 p2
compile s (While wp prg) = ((),PWhile nom nwp prg')
  where
    (s1,s2,s3) = split3 s
    (b,wp')    = compile s1 wp
    (c,prg')   = compile s2 prg
    nom        = "v" ++ (show $ supplyValue s3)
    nwp        = (wp' `PSeq` PAssign nom b)
compile s (Return a) = (a,PSkip)
compile s (Bind pa f) = (b, prg1 `PSeq` prg2)
  where
    (s1,s2)  = split2 s
    (a,prg1) = compile s1 pa
    (b,prg2) = compile s2 (f a)
```

Björn studies the code in disbelief, saves the module and tries it out on some examples. The code does indeed seem to work and Björn's disbelief is replaced with enthusiasm; this appears to be a simple and convenient way to reify monads.

Björn sends an email to Benny, inviting him to a meeting.

2.1.3 Meeting in Björn's office

BJÖRN: Hello, come in.

BENNY: Thanks.

BJÖRN: So, about this robot language! You gave me a bit of a surprise there. I was entirely sure that what you did was impossible.

BENNY: Oh, But I just did the first thing that came to mind. There was no deep thought behind it.

BJÖRN: The problem is [*Björn approaches his whiteboard*] when you try to reify the bind constructor of your representation.

```
Bind :: Program a -> (a -> Program b) -> Program b
```

BJÖRN: You need to come up with an element of type `a` to pass to the function.

BENNY: I didn't realise it was problem.

BJÖRN: But it is! Your solution is that you are careful about the return types of all operations in your language. They are either of unit type or some type which can be guaranteed to be reified. [*Björn scribbles on his whiteboard*]

```
Move :: Program ()

Sensor :: Program BoolE
```

BJÖRN: This is different from how such constructs are normally implemented. If I had implemented the language I would have given `Sensor` the type `Program Bool` so that I could write an evaluator of type `Program a -> a`, or perhaps `Program a -> M a`. But then adding a monad to the language and reifying it becomes much harder.

BENNY: Yes, now that you mention it, I have seen that style used for defining DSLs.

BJÖRN: Having constructors like `Sensor` return `BoolE` instead of `Bool` is a crucial part of why your method works. It allows the `compile` function to be written in such a way that it can generate the return values statically and pass them as arguments to `Bind`. In a sense you're doing evaluation at the same time as compilation. There's just enough evaluation to remove the monadic constructs. It's a very neat trick!

BENNY: Thank you.

BJÖRN: Another way to think about your trick is that you are using a writer monad to evaluate your language and produce the first-order syntax tree as a side effect. Then you could have used a state monad transformer on top of that for the fresh name supply. But that's a stylistic choice.

BENNY: You seem to be making a big deal out of this. I just did what I thought was the most straightforward thing to do.

Related work

BJÖRN: Others have solved this problem before but I have never seen a solution as simple as yours and was quite surprised that it works. For example, in this paper [*Björn shows Benny the paper [PAS12]*], the authors use a continuation monad to be able to reify the monadic constructs of their language. If they were to implement your robot language they would use the same `Program` data type as you do. However, they would not

expose that to the user of the language. Instead they would create a type like this:

```
data P a =
  P (forall r. ((a -> Program r) -> Program r))
```

BJÖRN: The type `P` is a continuation monad so the monad instance comes naturally. Operations on `P` are defined using `Bind` like this:

```
while :: P BoolE -> P () -> P ()
while c b =
  P (\k -> While (runP c) (runP b) `Bind` k)

cond :: BoolE -> P () -> P () -> P ()
cond b t e =
  P (\k -> Cond b (runP t) (runP e) `Bind` k)

runP :: P a -> Program a
runP (P f) = f (\a -> Return a)
```

BENNY: I see. Continuations are quite magical to me. I could never have come up with that technique. How do they deal with transforming higher-order programs to first-order programs?

BJÖRN: That's a good question. The transformation to first order programs is dealt with by a library called Syntactic. It is described in a separate paper [*Björn pulls out the paper [Axe12] from a pile of papers*]. It's hard to do an apples to apples comparison between Syntactic and your technique. Syntactic solves a much bigger problem than what you're doing so it is naturally more complicated.

BENNY: Ok.

BJÖRN: There are also the papers [FG12, SBGG13] which solves the problem in a manner that is more similar to yours. They also have explicit constructors `Bind` and `Return` but give them a slightly different type which guarantees that all applications of `Bind` are *normalised*, so that all `Bind`s are right-associated. Unexpectedly, this allows them to make instances of the `Monad` class but at the same time constrain the arguments of `Bind`. Their technique is more complicated than yours, but just as with the case of Syntactic, they solve a more general problem. However, there is one particular thing I like about your technique.

BENNY: What is that?

BJÖRN: Your technique is compositional.

Compositionality

BENNY: What does it mean that my technique is compositional?

BJÖRN: The compile function is compositional because there is one case for each constructor and each case deals with exactly one constructor. In particular, the constructors `Return` and `Bind` are handled completely separately from all other language constructs.

BENNY: And that is good ?

BJÖRN: Absolutely! Compositional definitions are nice because they imply that there is no weird semantical interaction between the constructs as they are defined independently. But it also means that it should be possible to factor out the constructors `Return` and `Bind` into a data type of their own. Then, it could be combined with other data types using techniques like Data Types á la Carte [Swi08] or CompData [BH12]. This way, a language can be designed piece by piece.You can then select the set of pieces required for a particular task or that suits a particular brand of robots.

BENNY: Oh! Interesting.

The Monad laws

BJÖRN: There is still one problem with your method though.

BENNY: What's that?

BJÖRN: When we make instances of the monad class we expect certain laws to hold. [*Björn writes the laws on his whiteboard*]

```
m >>= return    = m
return a >>= f  = f a
(m >>= f) >>= g = m >>= \a -> f a >>= g
```

BJÖRN: These laws clearly don't hold for your instance. Take the first law for instance. The left hand side will have extra `Bind` and `Return` constructors compared to the right hand side.

BENNY: Hmmm. I hadn't really thought about that. But adding an extra return at the end of a computation shouldn't make a difference in my implementation.

BJÖRN: So, you're saying that your implementation actually obeys the first monad law?

BENNY: Well, at least when I run my programs I will never see any difference between the left hand side and the right hand side.

BJÖRN: Aha, so what you're saying is that if we compare the semantics of programs rather than comparing the programs themselves then we get some useful laws. [*Björn scribbles some new laws on the whiteboard*]

```
eval (m >>= return)     = eval m
eval (return a >>= f)   = eval (f a)
eval ((m >>= f) >>= g) = eval (m >>= \a -> f a >>= g)
```

BJÖRN: These laws are morally the same as the monad laws, especially if we don't let the user of the robot language ever compare terms in the language.

BENNY: Yes, that captures my intuition very well.

BJÖRN: Ok, good. Can you prove these equations?

BENNY: No, I don't have any experience proving programs correct.

BJÖRN: Well, it shouldn't be that difficult. Let me see what I can come up with.

[*Björn scribbles frenetically on a piece of paper for a couple of minutes.*]

Aha! Your technique is quite general. It can reify monads into any kind of structure which is a monoid. Return translates into the monoid unit and bind translates into the monoid operation.

For example, in your compilation function it is important that the semantics of `PSkip` is the identity of the semantics of `PSeq` and that `PSeq` is associative.

BENNY: Ok, that sounds good. I take that as meaning the method is quite general?

BJÖRN: Yes, requiring a monoid is a very mild restriction.

Look at the time! This was interesting, I got quite carried away. We must round off but please get back to me if you make any more progress on your robot language.

BENNY: Thanks very much for your time. Bye!

BJÖRN: Thank you.

2.1.4 Composing reifiable monadic languages

At Benny's computer

Benny was really intrigued by the idea of compositionally building embedded languages and after having read the papers Björn showed him he decides to try his own approach to the problem.

```
data (e1 :+: e2) x a
  = InjL (e1 x a) | InjR (e2 x a)
infixr :+:

class sub :<: sup where
  inj :: sub x a  -> sup x a

instance f :<: f where
  inj = id

instance (f :<: (f :+: g)) where
   inj = InjL

instance (f :<: h) => (f :<: (g :+: h)) where
  inj = InjR . inj
```

Figure 2.3: Data types from CompData for composing languages.

Inspired by CompData he starts out by adding the code in figure 2.3 to his file. The data type `:+:` is used for composing languages, and the `:<:` typeclass provides coercions so that the programmer doesn't have to worry about using the right sequence of the `InjL` and `InjR` constructors to inject terms into the composed language.

Benny then starts to add data types for the different language constructs in his robotic language. The different operations for the robot are pleasingly easy to add.

```
data MoveOp x a where
  Move :: MoveOp x ()

data TurnOp x a where
  TurnLeft  :: TurnOp x ()
  TurnRight :: TurnOp x ()

data SensorOp x a where
  Sensor :: SensorOp x BoolE

data CondOp x a where
  Cond :: x BoolE -> x () -> x () -> CondOp x ()

data WhileOp x a where
  While :: x BoolE -> x () -> WhileOp x ()
```

In order to test out these definitions, Benny next turns to implementing a compiler. He realises that the compiler now needs to be implemented as a class with one instance per compilable sub-language.

```
runCompile :: Compile f => f a -> Prg
runCompile prg = snd $ compile s prg
  where s = newNameSupply

class Compile f where
  compile :: NameSupply -> f a -> (a, Prg)

instance Compile (MoveOp x) where
  compile _ Move = ((), PMove)

instance Compile (TurnOp x) where
  compile _ TurnLeft = ((), PTurnLeft)
  compile _ TurnRight = ((), PTurnRight)

instance Compile (SensorOp x) where
  compile s Sensor = (Var nom, PSensor nom)
    where v   = supplyValue s
          nom = "v" ++ show v

instance Compile x => Compile (CondOp x) where
  compile s (Cond b p1 p2) =
    ((),bp `PSeq` PCond b' p1' p2')
    where (s1,s2,s3) = split3 s
          (b',bp)    = compile s1 b
          (a1,p1')   = compile s2 p1
          (a2,p2')   = compile s3 p2

instance Compile x => Compile (WhileOp x) where
  compile s (While wp p) = ((),PWhile nom nwp p')
    where (s1,s2,s3) = split3 s
          (b,wp')    = compile s1 wp
          (c,p')     = compile s2 p
          nom        = "v" ++ (show $ supplyValue s3)
          nwp        = wp' `PSeq` PAssign nom b

instance (Compile (e1 f), Compile (e2 f))
  => Compile ((e1 :+: e2) f) where
  compile s (InjL a) = compile s a
  compile s (InjR a) = compile s a
```

But when Benny tries to add the monadic operations he finds that they are quite resistant to a compositional treatment. After much struggle he comes up with the following data type definition:

```
data Mops f x a where
  Oper   :: f x a -> Mops f x a
  Return :: a -> Mops f x a
  Bind   :: x a -> (a -> x b) -> Mops f x b
```

The recursion is provided by another data type that Benny calls `MonadExp`.

```
data MonadExp f a = In (Mops f (MonadExp f) a)
```

Benny thinks of the `MonadExp` data type as a representation of a monadic language parameterised over operations `Oper` that are constructed using the constructors of some type `f`. The monad instance for this data type is only slightly more complicated compared to the earlier, non-compositional setting.

```
instance Monad (MonadExp f) where
  return = In . Return
  (>>=) a f = In (Bind a f)
```

Benny can now write the type of his robotic language, built from independent pieces. It is noteworthy that the monadic expressions have to be added on top of all the other language constructs.

```
type Robot = MonadExp (MoveOp :+: TurnOp  :+:
                       CondOp :+: WhileOp :+:
                       SensorOp)
```

Given the definition of his language Benny can now write an injection function which helps writing smart constructors for all the different robot operations.

```
inject :: (sub :<: f)
       => sub (MonadExp f) a
       -> MonadExp f a
inject a = In $ Oper $ inj $ a

move :: (MoveOp :<: f) => MonadExp f ()
move =  inject Move

turnL :: (TurnOp :<: f) => MonadExp f ()
turnL = inject TurnLeft
```

```
turnR :: (TurnOp :<: f) => MonadExp f ()
turnR = inject TurnRight

sensor :: (SensorOp :<: f) => MonadExp f BoolE
sensor = inject Sensor

cond :: (CondOp :<: f)
     => MonadExp f BoolE
     -> MonadExp f () -> MonadExp f () -> MonadExp f ()
cond b p1 p2 = inject $ Cond b p1 p2

while :: (WhileOp :<: f)
      => MonadExp f BoolE
      -> MonadExp f () -> MonadExp f ()
while pb p = inject $ While pb p
```

Things are starting to fall into place. The only remaining bit is to complete the compiler for monadic expressions.

```
instance (Compile x, Compile (f x))
  => Compile (Mops f x) where
  compile s (Oper o) = compile s o
  compile s (Return a) = (a,PSkip)
  compile s (Bind m f) = (b,prg1 `PSeq` prg2)
    where (s1,s2)  = split2 s
          (a,prg1) = compile s1 m
          (b,prg2) = compile s2 (f a)

instance (Compile (f (MonadExp f)))
  => Compile (MonadExp f) where
  compile s (In a) = compile s a
```

Now, Benny has all the functions he needs to compile a simple example program.

```
test1 :: Robot ()
test1 = do
  move
  turnL
  move
  turnL
```

Compiling the program above gives the expected result.

```
PSeq PMove (PSeq PTurnLeft (PSeq PMove PTurnLeft))
```

Benny decides to contact Björn again to show him what he has done.

Dear Prof. Björn

I have attached a file which demonstrates that the monadic constructs can be factored out and compiled separately, just as you suggested.

Thanks
Benny

Benny,

Fascinating! You've managed to factor out the monadic operations so that they can be compiled once and for all. Users of your library don't have to be concerned at all with the semantics of bind and return. Very nice!

I note that the data type for monadic operations cannot be composed as freely as the other operations, but instead has to be applied separately as a final step. Your solution is very similar to how variable binding is handled in the Syntactic library [Axe].

Prof. Björn
Dept. of Computer Science and Engineering
Chalmers University of Technology

2.1.5 Epilogue

This paper shows a simple method of implementing monadic EDSLs. It shows a naïve approach to the monad reification problem which has the additional benefit of being compositional. In section 2.1.4, the language is reimplemented in an extensible way and the compile function is shown explicitly to be compositional by treating sub parts of the language separately.

The example language used throughout the story is small and quite limited. But the technique presented in this paper does scale up to larger languages and more complicated language constructs. It is currently used in the implementation of Obsidian, an EDSL for general purpose GPU programming [CSS12]. Obsidian has language constructs which are considerably more advanced than the robot language is this paper. One example is the sequential for-loop which is higher-order (the type parameter `t` is not relevant to the current discussion):

```
SeqFor :: EWord32 -> (EWord32 -> Program t ())
       -> Program t ()
```

There does, however, seem to be restrictions on what kind of language constructs the reification technique presented in this paper can deal with. For instance, the functions from the `MonadPlus` class has resisted our attempts at adding them to an EDSL in the same way as we've added `Return` and `Bind`. The exact limits of the expressiveness of our method is currently unknown to us.

Acknowledgments

We would like to thank Emil Axelsson for very valuable help during the implementation of the methods used in this pearl. We thank Mary Sheeran for suggesting the names Björn & Benny. The ICFP reviewers provided many helpful suggestion which has improved the paper.

This research has been funded by the Swedish Foundation for Strategic Research (which funds the Resource Aware Functional Programming (RAW FP) Project) and by the Swedish Research Council.

Bibliography

[Axe] Emil Axelsson. Syntactic. http://hackage.haskell.org/package/syntactic.

[Axe12] Emil Axelsson. A generic abstract syntax model for embedded languages. In *17th ACM SIGPLAN Conf. on Functional Programming*. ACM, 2012.

[BH12] Patrick Bahr and Tom Hvitved. Parametric compositional data types. In *MSFP*, pages 3–24, 2012.

[CSS12] Koen Claessen, Mary Sheeran, and Bo Joel Svensson. Expressive Array Constructs in an Embedded GPU Kernel Programming Language.

In *Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming*, DAMP '12, pages 21–30, New York, USA, 2012. ACM.

[FG12] Andrew Farmer and Andy Gill. Haskell DSLs for interactive web services. In *1st International Workshop on Cross-model Language Design and Implementation*, Sep 2012. (published on workshop website, http://workshops.inf.ed.ac.uk/xldi2012/).

[Hud99] Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, New York, NY, USA, 1999.

[PAS12] Anders Persson, Emil Axelsson, and Josef Svenningsson. Generic monadic constructs for embedded languages. In *Proceedings of the 23rd international conference on Implementation and Application of Functional Languages*, IFL'11, pages 85–99, Berlin, Heidelberg, 2012. Springer-Verlag.

[SBGG13] Neil Sculthorpe, Jan Bracker, George Giorgidze, and Andy Gill. The Constrained-Monad Problem, 2013. 18th ACM SIGPLAN international conference on Functional programming, ICFP 2013.

[Swi08] Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423, 2008.

Chapter 3

GPU Programming Papers

3.1 Paper B: Obsidian: A Domain Specific Embedded Language for Parallel Programming of Graphics Processors

Bo Joel Svensson, Mary Sheeran, Koen Claessen

Abstract

We present a domain specific language, embedded in Haskell, for general purpose parallel programming on GPUs. Our intention is to explore the use of *connection patterns* in parallel programming. We briefly present our earlier work on hardware generation, and outline the current state of GPU architectures and programming models. Finally, we present the current status of the *Obsidian* project, which aims to make GPU programming easier, without relinquishing detailed control of GPU resources. Both a programming example and some details of the implementation are presented. This is a report on work in progress.

3.1.1 Introduction

There is a pressing need for new and better ways to program parallel machines. Graphics Processing Units (GPUs) are one kind of such parallel machines that provide a lot of computing power. Modern GPUs have become extremely interesting for *general purpose* programming, i.e. computing functions that have little or nothing to do with graphics programming.

We aim to develop high-level methods and tools, based on functional programming, for low-level general purpose programming of GPUs. The methods are high-level because we are making use of common powerful programming abstractions in functional programming, such as higher-order functions and polymorphism. At the same time, we still want to provide control to the programmer on important low-level details such as how much parallelism is introduced, and memory layout. (These aims are in contrast to other approaches to GPU programming, where the programmer expresses the intent in a high-level language, and then lets a smart compiler try to do its best.)

Based on our earlier work on structural hardware design, we plan to investigate whether or not a *structure-oriented programming style* can be used in programming modern GPUs. We are developing Obsidian, a domain specific language embedded in Haskell. The aim is to make extensive use of higher order functions capturing *connection patterns*, and from these compact descriptions to generate code to run

on the GPUs. Our hardware-oriented view of programming also leads us to investigate the use of algorithmic ideas from the hardware design community in parallel programming.

In the rest of the paper, we first briefly review our earlier work on hardware description languages (Sect. 3.1.2), and introduce the GPU architecture we are working with (Sect. 3.1.3). After that, we describe the current status of the language Obsidian, and show examples (Sect. 3.1.4) and experimental results (Sect. 3.1.5). Obsidian is currently very much work in progress; we discuss motivations and current shortcomings (Sect. 3.1.6) and future directions (Sect. 3.1.7).

3.1.2 Connection patterns for hardware design and parallel programming

Connection patterns that capture common ways to connect sub-circuits into larger structures have been central to our research on functional and relational languages for hardware design. Inspired by Backus' FP language, Sheeran's early work on μFP made use of *combining forms* with geometric interpretations [She84]. This approach to capturing circuit regularity was also influenced by contact with designers of regular array circuits in industry – see reference [She05] for an overview of this and much other work on functional programming and hardware design.

Later work on (our) Ruby considered the use of binary relations, rather than functions in specifying hardware [JS90]. Lava builds upon these ideas, but also gains much in expressiveness and flexibility by being embedded in Haskell [BCSS98, Cla01]. The user writes what look like circuit descriptions, but are in fact circuit *generators*. Commonly used *connection patterns* are captured by higher order functions.

For example, an important pattern is parallel prefix or scan. Given inputs $[x_0, x_1 \ldots x_{n-1}]$, the prefix problem is to compute each $x_0 \circ x_1 \circ \ldots \circ x_j$ for $0 \leq j < n$, for $\circ$ an associative, but not necessarily commutative, operator. For example, the prefix sum of `[1..10]` is `[1,3,6,10,15,21,28,36,45,55]`. There is an obvious sequential solution, but in circuit design one is often aiming for a circuit that exploits parallelism, and so is faster (but also larger). In a construction attributed to Sklansky, one can perform the prefix calculation by first, recursively, performing the prefix calculation on each half of the input, and then combining (via the operator) the last output of the first of these recursive calls with each of the outputs of the second. For instance, to calculate the prefix sum of `[1..10]`, one can compute the prefix sums of `[1..5]` and `[6..10]`, giving `[1,3,6,10,15]` and `[6,13,21,30,40]`, respectively. The final step is to add the last element of the output of the first recursive call (`15`) to each element of the output of the second.

To express the construction in Lava, we make use of two connection patterns.

`two :: ([a] -> [b]) -> [a] -> [b]` applies its component to the top and bottom halves of the input list, concatenating the two sub-lists that result from these applications. Thus, `two (sklansky plus)` applied to `[1..10]` gives `[1,3,6, 10,15,6,13,21,30,40]`. Left-to-right serial composition has type `(a -> b) -> (b -> c) -> a -> c` and is written as infix `->-`. The description of the construction mixes the use of connection patterns, giving a form of reuse, with the naming of "wires".

```
sklansky :: ((t, t) -> t) -> [t] -> [t]
sklansky op [a] = [a]
sklansky op as = (two (sklansky op) ->- sfan) as
  where
    sfan as = a1s ++ a2s'
      where
        (a1s,a2s) = splitAt ((length as + 1) `div` 2) as
        a2s'      = [op(last a1s,a) | a <- a2s]

*Main> simulate (sklansky plus) [1..10]
[1,3,6,10,15,21,28,36,45,55]
```

Lava supports simulation, formal verification and netlist generation from definitions like this. Circuit descriptions are *run* (in fact symbolically evaluated) in order to produce an intermediate representation, which is in turn written out in various formats (for fixed size instances). So this is an example of *staged programming* [Tah04].

The Sklansky construction is one way to implement parallel prefix, and there are many others, see for instance Hinze's excellent survey [Hin04]. Those who develop prefix algorithms suitable for hardware implementation use a standard notation to represent the resulting networks. Data flows from top to bottom and the least significant input is at top left. Black dots represent operators. For example, Figure 3.1 shows the recursive Sklansky construction for 32 inputs.

In this work, we plan to investigate the use of connection patterns, and more generally an emphasis on *structure*, in parallel programming. We have chosen to target GPUs partly because of available expertise among our colleagues at Chalmers, and partly because reading papers about General Purpose GPU (GPGPU) programming gave us a sense of déjà vu. Programs are illustrated graphically, and bear a remarkable resemblance to circuit modules that we have generated in the past using Lava. We see an opportunity here, as there is an extensive literature, going back to the 1960s, about implementing algorithms on silicon that may provide clues about implementing algorithms on GPUs. This literature does not seem to have yet been scrutinised by the Data Parallel Programming or GPGPU communities. This is possibly because GPUs are moving closer to simply being data parallel machines, and

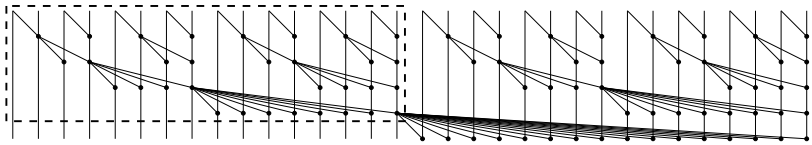

Figure 3.1: The Sklansky construction for 32 inputs. It recursively computes the parallel prefix for each half of the inputs (corresponding to the use of `two` in the definition) and then combines the last output of the lower (left) half with each of the outputs of the upper (right) half. The dotted box outlines the recursive call on the lower half of the inputs.

so work on library functions has taken inspiration from earlier work on Data Parallel Programming, such as Blelloch's NESL [Ble96]. But some of the restrictions from the early data parallel machines no longer hold today; for instance broadcasting a value to many processors was expensive in the past, but is much easier to do on modern GPUs. So a construction like Sklansky, which requires such broadcasting, should now be reconsidered, and indeed we have found it to give good results in our initial experiments (writing directly in CUDA). In general, it makes sense to spread the net beyond the standard data parallel programming literature when looking for inspiration in parallel algorithm design. We plan to explore the use of "old" circuit design ideas in programming library functions for GPUs.

Below, we briefly review modern GPUs and a standard programming model.

3.1.3 Graphics Processing Units, accessible high performance parallel computing

In the development of microprocessors, the addition of new cores is now the way forward, rather than the improvement of single thread performance. Graphics processing units (GPUs) have moved from being specialised graphics engines to being suitable to tackle applications with high computational demands. For a recent survey of the hardware, programming methods and tools, and successful applications, the reader is referred to [OHL$^+$08]. Figure 3.2, taken from that paper, and due to NVIDIA, shows the architecture of a modern GPU from NVIDIA. It contains 16 multiprocessors, grouped in pairs that share a texture fetch unit (TF in the figure). The texture fetch unit is of little importance when using the GPU for general purpose computations. Each multiprocessor has 8 stream processors (marked SP in the figure). These stream processors has access to 16kB of shared memory.

See reference [OHL$^+$08] for information about the very similar AMD GPU ar-

chitecture. We have used the NVIDIA architecture, but developments are similar at AMD. Intel's Larrabee processor points to a future in which each individual core is considerably more powerful than in today's GPUs [SCS+08].

The question of how to program powerful data-parallel processors is likely to continue to be an interesting one. Unlike for current multicore machines, the question here is how to keep a large number of small processors productively occupied. NVIDIA's solution has been to develop the architecture and the programming model in parallel. The result is called CUDA – an extension of C designed to allow developers to exploit the power of GPUs. Reference [Lue08] gives a very brief but illuminating introduction to CUDA for potential new users. The idea is that the user writes small blocks of straightforward C code, which should then run in thousands or millions of threads. We borrow the example from the above introduction. To add two $N \times N$ matrices on a CPU, using C, one would write something like

```
// add 2 matrices on the CPU:
void addMatrix(float *a, float *b, float *c, int N)
{
  int i, j, index;
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
      index = i + j * N;
      c[index]=a[index] + b[index];
    }
  }
}
```

In CUDA, one writes a similar C function, called a *kernel*, to compute one element of the matrix. Then, the kernel is invoked as many times as the matrix has elements, resulting in many threads, which can be run in parallel. A predefined structure called `threadIdx` is used to label each of these many threads, and can be referred to in the kernel.

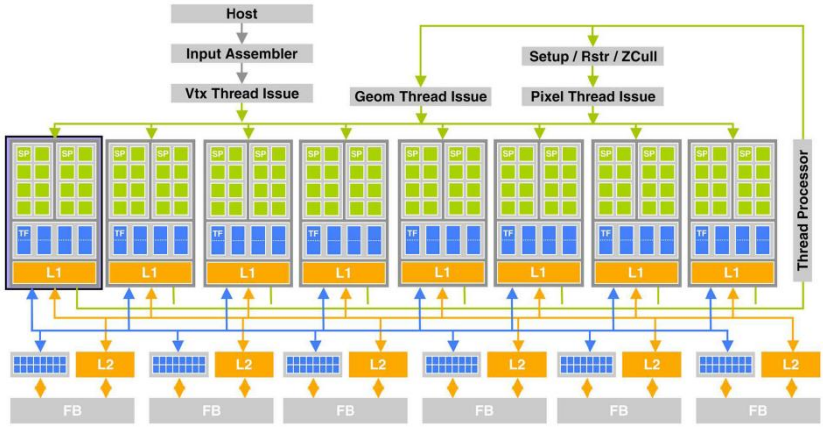

Figure 3.2: The NVIDIA 8800GTX GPU architecture, with 8 pairs of multiprocessors. Diagram courtesy of NVIDIA.

```
// add 2 matrices on the GPU (simplified)
__global__ void addMatrix(float *a,float *b, float *c, int N)
{
  int i= threadIdx.x;
  int j= threadIdx.y;
  int index= i + j * N;
  c[index]= a[index] + b[index];
}

void main()
{
  // run addMatrix in 1 block of NxN threads:
  dim3 blocksize(N, N),
  addMatrix<<<1, blocksize>>>(a, b, c, N);
}
```

Here, a two dimensional *thread block* of size $N \times N$ is created.

CUDA uses *barrier synchronisation* and *shared memory* for introducing communication between threads. Contents of shared memory (16kB per multiprocessor in the architecture shown in Figure 3.2) is visible to all threads in a thread block. It is very much faster to access this shared memory than to access the global device memory. We shall see later that Obsidian provides users with both shared and global

arrays, giving the user control over where data is to be stored.

Since many threads are now writing and reading from the same shared memory, it is necessary to have a mechanism that enables the necessary synchronisation between threads. CUDA provides a barrier synchronisation mechanism called `__syncthreads()`. Only when all threads in a block have reached this barrier can any of them proceed. This allows the programmer to ensure safe access to the shared memory for the many threads in a thread block.

Now, a *grid* is a collection of thread blocks. Each thread block runs on a single multiprocessor, and the CUDA system can schedule these individual blocks in order to maximise the use of GPU resources. A complete program then consists not only of the kernel definitions, but also of code, to be run on the CPU, to launch a kernel on the GPU, examine the results and possibly launch new kernels. In this paper, we will not go into details about how kernels are coordinated, but will concentrate on how to write individual kernels, as this is the part of Obsidian that is most developed. In Obsidian, we write code that looks like the Lava descriptions in section 3.1.2, and we generate CUDA code like that shown above. This is a considerably more complex process than the generation of netlists in Lava.

3.1.4 Obsidian: a domain specific embedded language for GPU programming

To introduce Obsidian, we consider the implementation of a parallel prefix kernel. The implementation bears a close resemblance to the Lava implementation from section 3.1.2:

```
sklansky :: (Choice a, Syncable (Arr s a)) =>
             (a -> a -> a) -> Arr s a -> W (Arr s a)
sklansky op arr
   | len arr == 1 = return arr
   | otherwise = (two (sklansky op) ->- sfan ->- sync) arr
    where sfan arr = do
            (a1,a2) <- halve arr
            let m = len a1
                c = a1 ! (fromInt (m-1))
            a2' <- fun (op c) a2
            conc (a1,a2')
```

The most notable differences between the two implementations are that Obsidian functions are monadic and that a datatype `Arr` is used where Haskell lists are used in Lava. The pattern matching on the list used in Lava is here replaced by guards.

The function `len :: Arr s a -> Int` gives the length of the array. These differences lead to a slightly different programming style.

The Obsidian version of the `sklansky` function implements the sought recursive parallel prefix algorithm, but it contains no information about where in the memory hierarchy the intermediate results are to be held. The following program uses `sklansky` from above but turns it into a concrete kernel that computes all the partial sums of an array of integers:

```
scan_add_kernel :: GArr IntE -> W (GArr IntE)
scan_add_kernel = cache ->- sklansky (+) ->- wb ->- sync
```

The function `cache` specifies that if the array is stored it should be stored in the on-chip shared memory. Actually storing an array is done using the `sync` function, which functionally is the identity function, but has the extra effect of synchronising all processes after writing their data in shared memory, such that they can exchange intermediate results. Using `sync` here allows for computations or transformations to be performed on the data as it is being read in from global device memory. In the `scan_add_kernel` above this means that the first `sklansky` stage will be computed with global data as input, putting its result into shared memory. A kernel computing the same thing but using the memory differently can be implemented like this:

```
scan_add_kernel2 :: GArr IntE -> W (GArr IntE)
scan_add_kernel2 = cache ->- sync ->- sklansky (+) ->-
                   wb ->- sync
```

Here the array is first stored into shared memory. The `sklansky` stages are then computed entirely in shared memory. The write-back function, `wb`, works in a very similar way but specifies that the array should be moved back into the global memory.

Now, `scan_add_kernel` can be launched on the GPU from within a GHCI session using a function called `execute`:

```
execute :: ExecMode
         -> (GArr (Exp a) -> W (GArr (Exp b))
         -> [Exp a] -> IO [Exp b]
```

Here `ExecMode` can be either `GPU` for launching the kernel on the GPU or `EMU` for running it in emulation mode on the system's CPU. Below is the result[1] of launching the scan kernel on example input:

```
*Main> execute GPU scan_add_kernel [1..256]
[0,1,3,6,10,15, ... ,32131,32385,32640]
```

[1] The output has been shortened to fit on a line.

Beyond the combinators described so far, we have experimented with combinators and permutations needed for certain iterative sorting networks. Amongst these are `evens` that applies a function to each pair of elements of an array. Together with `rep` that repeats a computation a given number of times and a permutation called `riffle` a shuffle exchange network can be defined:

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

The shuffle exchange network can be used to implement a merger useful in sorters.

Implementation

As seen in the examples, an Obsidian program is built from functions between arrays. These arrays are of type `Arr s a`. There are also type synonyms `GArr` and `SArr` implemented as follows:

```
data Arr s a = Arr (IxExp ->  a, Int)
type GArr a = Arr Global a
type SArr a = Arr Shared a
```

In Obsidian an array is represented by a function from indices to values and an integer giving the length of the array.

In most cases the `a` in `Arr s a` will be of an expression type:

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

The above expressions are dynamic in the sense that they can be used to represent values of `Int`, `Float` and `Bool` type. This follows the approach from Compiling Embedded Languages [EFdM03]. However, to obtain a typed environment in which to operate, phantom types are used.

```
type Exp a = E DExp

type IntE   = Exp Int
type FloatE = Exp Float
type BoolE  = Exp Bool
type IxExp  = IntE
```

As an example consider the program:

```
add_one :: GArr IntE -> W (GArr IntE)
add_one = fun (+1) ->- sync
```

This program adds 1 to each element of an array of integers. The function `fun` has type `(a -> b) -> Arr s a -> Arr s b`. `fun` performs for arrays what `map` does for lists. The `sync` function used in the example has the effect that the array being synced upon is written to memory. At this point the type of the array determines where in memory it is stored. An array of type `SArr` will end up in the *shared memory* (which is currently 16KB per multi-processor). In this version of Obsidian it is up to the programmer to make sure that the array fits in the memory. An array of type `GArr` ends up in the global device memory, roughly a gigabyte on current graphics cards. Using `sync` can have performance implications since it facilitates sharing of computed values.

To generate CUDA code from the Obsidian program `add_one`, it is applied to a symbolic array of a given concrete length (in this example 256 elements):

```
input :: GArr IntE
input = mkArray (\ix -> (index (variable "input") ix)) 256
```

Applying `fun (+1)` to this input array results in an array with the following indexing function:

```
(\ix -> (E (BinOp Add (Index (Variable "input") ix)
                      (LitInt 1))))
```

At the code generation phase this function is evaluated using a variable representing a thread Id. The result is an expression looking as follows:

```
E (BinOp Add (Index (Variable "input") (Variable "tid"))
             (LitInt 1))
```

This expression is a direct description of what is to be computed.

Importantly, the basic library functions can be implemented using the `Arr s a` type and thus be applicable both to shared and global arrays. The library function `rev` that reverses an array is shown as an example of this:

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

The function `rev` uses `mkArray` to create an array whose indexing function reverses the order of the elements of the given array `arr`. The Obsidian program `rev ->- sync` corresponds[2] to the following lines of C code:

```
arrx[ThreadIdx.x] = arry[n - 1 - ThreadIdx.x];
__syncthreads();
```

When an Obsidian function, such as the `scan_add_kernel` from the previous section, is run, two data structures are accumulated into a monad called `W`. The first is intermediate code `IC` and the second a symbol table. The `W` monad is a writer monad extended with some extra functionality for generating identifier names and to maintain the symbol table. You can think of the `W` monad as:

```
type W a = WriterT (IxExp -> IC) (State (SymbolTable,Int)) a
```

The IC used here is just a list of statements, (less important statements have been removed to save space (`...`)). The `IC` contains a subset of CUDA. In this version of Obsidian not much more than the `Synchronize` and assignment statements of CUDA are used.

```
data Statement = Synchronize
               | DExp ::= DExp
               -- used later in code generation
               | IfThenElse BoolE [Statement] [Statement]
                 deriving (Show,Eq)

type IC = [Statement]
```

The symbol table is a mapping from names to types and sizes:

```
type SymbolTable = Map Name (Type,Int)
```

Information needs to be stored into the SymbolTable whenever new intermediate arrays are created. We have chosen to put this power in the hands of the programmer using the `sync` function. The `sync` function is overloaded for a number of different array types:

[2] in the real IC `arrx` and `arry` are replaced by identifiers generated in the `W` monad

```
class Syncable a where
    sync :: a -> W a
    commit :: a -> W a
```

The types of the `sync` and the related `commit` function are shown in the class declaration above. To illustrate what `sync` does, one instance of its implementation is shown:

```
instance TypeOf (Exp a) => Syncable (GArr (Exp a)) where
    sync arr = do
      arr' <- commit arr
      write $ \ix -> [Synchronize]
      return arr'
    commit arr = do
      let n = len arr
      var  <- newGlobalArray (typeOf (arr ! (E (LitInt 0))))  n
      write $ \ix -> [(unE (index var ix)) ::= unE (arr ! ix)]
      return $ mkArray (\ix -> index var ix) n
```

The `sync` function commits its argument array and thereafter writes `[Synchronize]` into the W monad. To see what this means, one should also look at the `commit` function, in which a new array is created of the same size and type as the given array. In the next step, an assignment statement is written into `W` monad (added to the intermediate code). It assigns the values computed in the given array to the newly created array. From the intermediate code and the symbol table accumulated into the `W` monad, C code is generated following a procedure outlined in figure 3.3.

The first step depicted in the figure is the running the Obsidian program. This builds two data structures `IC` and `SymbolTable`. The IC goes through a simple liveness analysis where for each statement information about what data elements, in this case arrays, are alive at that point is added. An array is alive if it is used in any of the following statements or if it is considered a result of the program. The result of this pass is a new *IC* where each statement also has a set of names pointing out arrays that are alive at that point.

```
type ICLive = [(Statement, Set Name)]
```

Now, the symbol table together with the `ICLive` object is used to lay out the arrays in memory. Arrays that had type `SArr` are assigned storage in the shared memory and arrays of type `GArr` in Global memory. The result of this stage is a *Memory Map*. This is a mapping from names to positions in memory. The picture also shows that another output from this stage is intermediate code annotated with

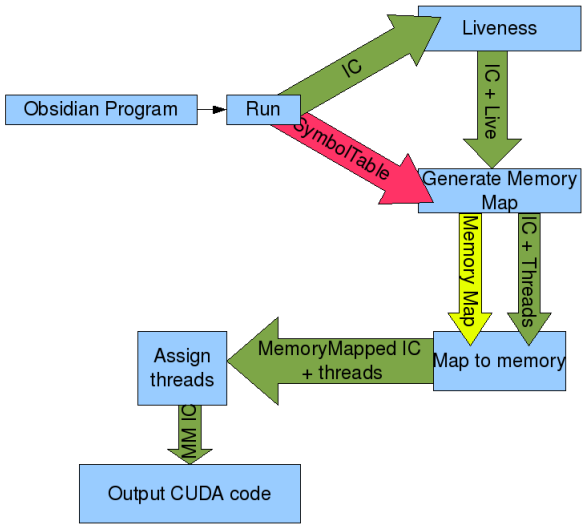

Figure 3.3: Steps involved in generating CUDA C code from Obsidian description. The boxes represent functions and the arrows represent data structures.

```
__global__ static void rev_add(int *source0,char *gbase){
extern __shared__ char sbase[] __attribute__ ((aligned(4)));
const int tid = threadIdx.x;
const int n0 __attribute__ ((unused)) = 256;
((int *)(gbase+0))[tid] = source0[((256 - 1) - tid)];
__syncthreads();
 ((int *)(gbase+1024))[tid] = (((int *)(gbase+0))[tid] + 1);
__syncthreads();
```

Figure 3.4: Generated CUDA code.

thread information, call it `ICT`. This is now done in a separate pass over the IC but it could be fused with the memory mapping stage, saving a pass over the IC. The ICT is just a list of statements and the number of threads assigned to compute them:

```
type ICT = [(Statement,Int)]
```

This enables the final pass over the IC to move thread information into the actual IC as conditionals. The resulting IC is used to output CUDA code.

To illustrate this, the code generated from a simple Obsidian program is shown. The example is very artificial and uses sync excessively in order to create more intermediate arrays :

```
rev_add :: GArr IntE -> W (GArr IntE)
rev_add = rev ->- sync ->- fun (+1) ->- sync
```

Figure 3.4 shows the CUDA C code generated from the `rev_add` program. Here it is visible how intermediate arrays are assigned memory in global memory. The global memory is pointed to by `gbase`:

The code in figure 3.4 does however not show how the `ICT` is used in assigning work to threads. To show this a small part of the generated CUDA code from the `scan_add_kernel` is given in figure 3.5. Notice the conditional `if (tid < 128)`. This line effectively shuts down half of the threads. It can also be seen here how shared memory is used, pointed to by `sbase`. Moreover, from the line with `(63 < 64) ? ...` it becomes clear that there is room for some obvious optimisations.

```
__syncthreads();
if (tid < 128){
  ((int *)(sbase+1520))[tid] = ((tid < 64) ?
  ((tid < 64) ?
  ((int *)(sbase+496))[tid] :
  ((int *)(sbase+752))[(tid - 64)]) :
  (((63 < 64) ?
  ((int *)(sbase+496))[63] :
  ((int *)(sbase+752))[(63 - 64)]) + ((tid < 64) ?
  ((int *)(sbase+496))[tid] :
  ((int *)(sbase+752))[(tid - 64)])));
}
```

Figure 3.5: A small part of the CUDA code generated from the recursive implementation of the Sklansky parallel prefix algorithm.

3.1.5 Results

Apart from the parallel prefix algorithm shown in this paper we have used Obsidian to implement sorters. For the sorters, the generated C code performs quite well. One periodic sorting network, called *Vsort*, implemented in Obsidian in the style of the sorters presented in reference [CSS01], has a running time of 95μsecs for 256 elements (the largest size that we can cope with in a single kernel). This running time can be compared to the 28μsecs running time of the bitonic sort example supplied with the CUDA SDK. However, for 256 inputs, bitonic sort has a depth (counted in number of comparators between input and output), of 36 compared to Vsort's 64. So we feel confident that we can make a sorter that improves considerably on our Vsort by implementing a recursive algorithm for which the corresponding network depth is less. (We implemented the periodic sorter in an earlier version of the system, in which recursion was not available.) The point here is not to be as fast as hand-crafted library code, but to come close enough to allow the user to quickly construct short readable programs that give decent performance. The results for sorting are promising in this respect. Sadly, the results for the Sklansky example are rather poor, and we will return to this point in the following section.

The programs reported here were run on an NVIDIA 8800GTS and timed using the CUDA profiler.

3.1.6 Discussion

Our influences

As mentioned above, our earlier work on Lava has provided the inspiration for the combinator-oriented or hardware-like style of programming that we are exploring in Obsidian. On the other hand, the *implementation* of Obsidian has been much influenced by Pan, an embedded language for image synthesis developed by Conal Elliot [Ell03]. Because of the computational complexity of image generation, C code is generated. This C code can then be compiled by an optimising compiler. Many ideas from the paper "Compiling Embedded Languages", describing the implementation of Pan have been used in the implementation of Obsidian [EFdM03].

Related work on GPU and GPGPU programming languages

We cannot attempt an exhaustive description of GPU programming languages here, but refer the reader to a recent PhD thesis by Philipp Lucas, which contains an enlightening survey [Luc08]. Lucas distinguishes carefully between languages (such as CG and HLSL) that aim to raise the level of abstraction at which graphics-oriented GPU programs are written, and those that attempt to abstract the entire GPU, and so must also provide a means to express the placing of programs on the GPU, feeding such programs with data, reading the results back to the CPU, and so on, as well as deciding to what extent the programmer should be involved in stipulating those tasks. In the first group of graphics-oriented languages, we include PyGPU and Vertigo. PyGPU is a language for image processing embedded in Python [LO06]. PyGPU uses the introspective abilities of Python and thus bypasses 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. Programs written in PyGPU can be compiled and run on a GPU. Vertigo is another embedded language by Conal Elliot [Ell04]. It is a language for 3D graphics that targets the DirectX 8.1 shader model, and can be used to describe geometry, shaders and to generate textures.

The more general purpose languages aim to abstract away from the graphics heritage of GPUs, and target a larger group of programmers. The thesis by Lucas presents CGiS, an imperative data-parallel programming language that targets both GPUs and SIMD capable CPUs – with the aim being a combination of a high degree of abstraction and a close resemblance to traditional programming languages [Luc08]. BrookGPU (which is usually just called Brook) is an example of a language [BFH+04] designed to raise the level of abstraction at which GPGPU programming is done. It is an extension of C with embedded kernels, aimed at arithmetic-intense data par-

allel computations. C is used to declare streams[3], CG/HLSL (the lower level GPU languages) to declare kernels, while function calls to a runtime library direct the execution of the program. Brook had significant impact in that it raised the level of abstraction at which GPGPU programming can be done. The language Sh also aimed to raise the level of abstraction at which GPUs were programmed [MQP02]. Sh was an embedded language in C++, so our work is close in spirit to it. Sh has since evolved into the RapidMind development platform [McC06], which now supports multicores and Cell processors as well as GPUs. The RapidMind programming model has arrays as first class types. It has been influenced by functional languages like NESL and SETL, and its program objects are pure functions. Thus it supports both functional and imperative styles of programming. A recent PhD thesis by Jansen asserts that there are some problems with RapidMind's use of macros to embed the GPU programming language in C++, including the inability to pass kernels (or shader programs) as classes [Jan07]; the thesis proposes GPU++ and claims improvement over previous approaches, particularly through the exploitation of automatic partitioning of the programs onto the available GPU hardware, and through compiler optimisations that improve runtime performance.

Microsoft's Accelerator project moves even closer to general purpose programming by doing away with the kernel notion and simply expressing programs in a data parallel style, using functions on arrays [TPO06]. Data Parallel Haskell [JLKC08] incorporates Nested Data Parallelism in the style of NESL [Ble96] into Haskell. GPUGen, like Obsidian, aims to support GPGPU programming from Haskell [Lee08]. It works by translating Haskell's intermediate language, Core, into CUDA, for collective data operations such as scan, fold and map. The intention is to plug GPUGen into the Nested Data Parallel framework of the Glasgow Haskell Compiler. Our impression is that we wish to expose considerably more detail about the GPU to the programmer, but we do not yet have sufficient information about GPUGen to be able to do a more complete comparison. Finally, we mention the Spiral project, which develops methods and tools for automatically generating high performance libraries for a variety of platforms, in domains such a signal processing, multiplication and sorting [PMJ$^+$05]. The tuning of an algorithm for a given platform is expressed as an optimisation problem, and the domain specific mathematical structure of the algorithm is used to create a feedback-driven optimiser. The results are indeed impressive, and we feel that the approach based on an algebra of what we would call combinators will interest functional programmers. We hope to experiment with similar search and learning based methods, having applied similar ideas in the simpler

[3]Brook is referred to as a "stream processing" language, but this means something different from what the reader might expect: a stream in this context is a possibly multi-dimensional array of elements, each of which can be processed separately, in parallel.

setting of arithmetic data-path generation in Lava.

Lessons learned so far in the project

Our first lesson has been the gradual realisation that a key aspect of a usable GPU programming language that exposes details of the GPU architecture to the user is the means to express where and when data is placed in and read from the memory hierarchy. We are accustomed, from our earlier experience in hardware design, to describing and generating networks of communicating components – something like data-flow graphs. We are, however, unused to needing to express choices about the use of the various levels in a memory hierarchy. We believe that we need to develop programming idioms and language support for this. It seems likely, too, that such idioms will not be quite as specific to GPU programming as other aspects of our embedded language development. How to deal with control of access to a memory hierarchy in a parallel system seems to be a central problem that must be tackled if we are to develop better parallel programming methods in general. A typical example of a generic approach to this problem is the language Sequoia, which aims to provide programmers with a means to express how the memory hierarchy is to be used, where a relatively abstract description of the platform, viewed as a tree of processing nodes and memories, is a parameter [FKH$^+$06]. Thus, programmers should write very generic code, which can be compiled for many different platforms. This kind of platform independence is not our aim here, and we would like to experiment with programming idioms for control of memory access for the particular case of a CPU plus some form of highly parallel co-processor that accelerates some computations.

A second lesson concerns ways to think about synchronisation on the GPU. We naively assumed that `sync` would have nice compositional behaviour, but we have found that in reality one can really only sync at the top level. The reason why the CUDA code generated from the `sklanky` example works poorly on the GPU is that it uses `syncs` in a way that leads to unwanted serialisation of computations. Looking at our generated code, we see that it may be possible to make major improvements by being cleverer about the placement of `syncs`. For instance, the semantics of `two` guarantees that the two components act on distinct data, and this can be exploited in the placing of `syncs` in the generated code.

Finally, we have found that we need to think harder about the two levels of abstraction: writing the kernels themselves and kernel coordination. This paper concerned the kernel level. We do not yet have a satisfactory solution to the question of how best to express kernel coordination. This question is closely related to that about how to express memory use.

3.1.7 Future work

The version of Obsidian described here is at a very experimental stage. The quality of the C code generated needs to improve to get performance on par with the previous version. The previous version however, was very limited in what you could express. This older version is described in [Sve08]. There is a clear opportunity to perform classic compiler optimisations on the IC formed by running an Obsidian program. Currently this is not done at all.

Ways to describe the coordination of kernels in code that is still short and sweet are also needed. Some experiments using methods similar to Lava's netlist generation have been performed, but the resulting performance is not yet satisfactory. In CUDA, Kernel coordination is in part described in the actual kernel code. Kernels decide which parts of the given data to use. As future work we will approach the kernel coordination problem at a lower more CUDA-like level. We will, of necessity, need to develop programming idioms or combinators that express how data is placed in the memory hierarchy. The isolation of this as a central question is one of the more unexpected and interesting results of the project. Right now work is focused on developing combinators that are more clever in their treatment of `sync`s. This leads to new data structures that allow the merging of `sync`s. This new approach seems to make efficient implementations of combinators such as `two` possible.

3.1.8 Conclusion

Obsidian provides a good interface for experimenting with algorithms on GPUs. The earlier version described in [Sve08] showed that it is possible to generate efficient CUDA code from the kind of high level descriptions we are interested in. For the kernel level, the work in progress described in this paper enhances the expressive power of Obsidian, extending the range of algorithms that can be described, as well as the degree of control exercised by the user. Future work will concentrate on improving the performance of the resulting applications, as well as on support for the kernel coordination level.

Bibliography

[BCSS98] 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.

[BFH+04] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, , and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. In *SIGGRAPH*, 2004.

[Ble96] Guy Blelloch. Programming Parallel Algorithms. *Communications of the ACM*, 39(3), 1996.

[Cla01] Koen Claessen. *Embedded Languages for Describing and Verifying Hardware*. PhD thesis, Chalmers University of Technology, 2001.

[CSS01] Koen Claessen, Mary Sheeran, and Satnam Singh. The Design and Verification of a Sorter Core. In *Proc. Int. Conf. on Correct Hardware Design and Verification Methods (CHARME)*, volume 2144 of *Springer LNCS*, pages 355–369, 2001.

[EFdM03] Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(2), 2003.

[Ell03] Conal Elliott. Functional Images. In *The Fun of Programming*, "Cornerstones of Computing" series. Palgrave, March 2003.

[Ell04] Conal Elliott. Programming graphics processors functionally. In *Proceedings of the 2004 Haskell Workshop*. ACM Press, 2004.

[FKH+06] Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[Hin04] Ralf Hinze. An algebra of scans. In *Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, pages 186–210. Springer, 2004.

[Jan07] Thomas C. Jansen. *GPU++ An Embedded GPU Development System for General-Purpose Computations*. PhD thesis, Technische Universitäat München and Forschungsinstitut caesar in Bonn, 2007.

[JLKC08] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M T Chakravarty. Harnessing the multicores: Nested data parallelism in haskell. In Ramesh Hariharan, Madhavan Mukund, and V Vinay, editors, *IARCS Annual Conference on Foundations of Software Technology*

and Theoretical Computer Science (FSTTCS 2008), volume 2 of *Leibniz International Proceedings in Informatics*, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[JS90] G. Jones and M. Sheeran. Circuit design in Ruby. In J. Staunstrup, editor, *Formal Methods for VLSI Design*, pages 13–70. North-Holland, 1990.

[Lee08] Sean Lee. Bringing the power of gpus to haskell, Sept. 2008. Slides from Galois Tech Talk.

[LO06] 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.

[Luc08] Philipp Lucas. *CGiS: High-Level Data-Parallel GPU Programming*. PhD thesis, Saarland University, Saarbrücken, January 2008.

[Lue08] D. Luebke. CUDA: Scalable parallel programming for high-performance scientific computing. In *Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008. 5th International Symposium on*, pages 836 – 838. IEEE, May 2008.

[McC06] M.D. McCool. Data-Parallel Programming on the Cell BE and the GPU using the RapidMind Development Platform. In *GSPx Multicore Applications Conference*, Oct. 2006.

[MQP02] Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. Shader Metaprogramming. In *SIGGRAPH/Eurographics Graphics Hardware Workshop*, Sept. 2002.

[OHL+08] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[PMJ+05] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE special issue on Program Generation, Optimization, and Adaptation*, V93(2):232–275, 2005.

[SCS+08] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27:18:1–18:15, August 2008.

[She84] M. Sheeran. muFP, A Language for VLSI Design. In *LISP and Functional Programming*, pages 104–112. ACM, 1984.

[She05] Mary Sheeran. Hardware design and functional programming: a perfect match. *Journal of Universal Computer Science*, 11(7):1135–1158, July 2005.

[Sve08] Joel Svensson. An embedded language for data-parallel programming, 2008. Master's thesis, Göteborg University, `http://www.cse.chalmers.se/~joels/writing/joelmscthesis.pdf`.

[Tah04] Walid Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation (Dagstuhl Seminar, 2003)*, volume 3016 of *Lecture Notes in Computer Science*, pages 30–50. Springer, 2004.

[TPO06] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. volume 34, pages 325–335, New York, NY, USA, October 2006. ACM.

3.2 Paper C: GPGPU Kernel Implementation and Refinement using Obsidian

Bo Joel Svensson, Koen Claessen, Mary Sheeran

Abstract

Obsidian is a domain specific language for data-parallel programming on graphics processors (GPUs). It is embedded in the functional programming language Haskell. The user writes code using constructs familiar from Haskell (like `map` and `reduce`), recursion and some specially designed combinators for combining GPU programs. NVIDIA CUDA code is generated from these high level descriptions, and passed to the `nvcc` compiler [NVI08]. Currently, we consider only the generation of single kernels, and not their coordination.

This paper is focussed on how the user should work with Obsidian, starting with an obviously correct (or well-tested) description of the required function, and refining it by the introduction of constructs to give finer control of the computation on the GPU. For some combinators, this approach results in CUDA code with satisfactory performance, promising increased productivity, as the high level descriptions are short and uncluttered. But for other combinators, the performance of generated code is not yet satisfactory. Ways to tackle this problem and plans to integrate Obsidian with another higher-level embedded language for GPU programming in Haskell are briefly discussed.

3.2.1 Introduction

Multicore and manycore processors are becoming increasingly common. Modern graphics processing units (GPUs) are examples of manycore processors. Today GPUs come with hundreds of processing elements capable of managing thousands of threads [NVI06]. The Obsidian project is about exploring ways to program these new machines.

Obsidian is a domain specific language for general purpose programming on GPUs (GPGPU), capable of generating code for modern NVIDIA GPUs. In [SSC09] we describe a version of Obsidian that uses a monadic interface. In the current version of Obsidian the monad has been replaced by another data structure, closely related to Arrows [Hug00], representing GPU programs. Although the details are left out because of space limitations, the use of this GPU program representation can

be seen in subsection 3.2.2. For a description of the implementation see [SCS10] and for more general information on embedded language implementation see [EFdM03].

GPU design is driven by the performance demands of graphics applications. The kind of processing that is common in graphics falls in the data-parallel category [PF05].

NVIDIA GPUs and CUDA

Starting with the 8000 series of GPUs (the G80 architecture), NVIDIA's GPUs came with a unified architecture, meaning that all the processing elements on the GPU are of the same kind. This was different from the previous generation's GPUs, which typically had two kinds of processing elements, fragment and vertex processors. Now these two kinds of processors are replaced by a single kind with capabilities surpassing both of the old ones. This new unified architecture together with development tools for GPGPU programming go under the name CUDA (Compute Unified Device Architecture) [NVI08]. CUDA offers the GPGPU programmer a C compiler and libraries for CUDA enabled GPUs (NVIDIA 8000 series and above).

Figure 3.6, shows a conceptual picture of a CUDA enabled GPU. The GPU has a number of *Multiprocessors* (MP in the picture). These MPs contain a number of small processing elements called *Streaming Processors* (SP). These SPs operate in an SIMD fashion. All SPs in a given MP execute the same instruction each clock cycle.

Each MP is capable of maintaining a large number of threads in flight at the same time. A group of threads running on an MP is referred to as a *block*. A block can contain more threads then there are processing elements; today a block can hold up to 1024 threads. There is a scheduler mapping these threads over the SPs available. Threads are scheduled in groups called *warps* consisting of 32 threads that are executed in an SIMD fashion on the MPs. Conditionals that take different paths within a warp have a negative effect on performance; the two diverging paths will in fact be executed sequentially [NVIa].

The MPs also have local memory that is shared between all the SPs of that MP, and thus referred to as *Shared Memory* in the figure. It can be used to exchange information between threads running on the SPs; it can also be used as a software managed cache. Threads within a warp can safely communicate using shared memory without the need for any synchronisation. Threads from different warps, however, need to use a synchronisation primitive to ensure a coherent view of the shared memory. In CUDA C this primitive is called `__syncthreads()`. The `__syncthreads()` primitive provides a barrier that all threads of a block must reach before any is allowed to proceed.

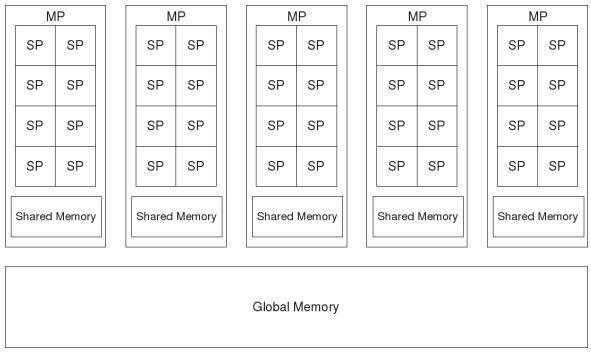

Figure 3.6: Conceptual image of a CUDA enabled GPU.

The *Global memory* is located off chip and is accessible by all MPs. In current GPUs, accesses to this memory are uncached.

Aims of Obsidian

The initial goal of Obsidian is to simplify the development of GPU kernels, the building blocks of larger GPU programs. In the future, ways to combine kernels into larger algorithms will also be explored.

When implementing an algorithm in CUDA, what to compute and how to compute it become codependent. Choices such as how many elements the kernel operates on and how many threads it uses affect each other greatly. Once a kernel is designed with a particular number of elements per thread, this becomes hard to tweak. In CUDA the programmer writes a single program parameterised over a thread identity. This program is then executed by several threads. Taking this point of view often means that what elements to use needs to be computed from the threadIDs. Coming up with these indexing computations is not always trivial.

In Obsidian the program is described as a computation between arrays, and combinators are used instead of direct indexing into structures. Obsidian provides an environment where it is easy to experiment with different partitionings and choices when implementing an algorithm. It is possible to write a simple running first prototype version of a kernel without thinking about architectural details. The prototype implementation can then be refined into a more efficient implementation. The aim of Obsidian is to raise the level of abstraction for the GPGPU programmer and to relieve the programmer of details such as laying things out in memory. Performance affecting decisions should be easy to make and change without major rewrites of the

code.

3.2.2 Programming in Obsidian

Obsidian is a language for GPGPU programming embedded in Haskell. Many of the language features resemble those of Lava, a hardware description and verification language [BCSS98]. The justification to use language constructs similar to that of a hardware description language came from the observation that GPGPU algorithms often were explained using circuit-like pictures, see for example [Ngu07].

Obsidian can be explained as two sub-languages. First there is a language of arrays and operations on arrays, and second, a language that enables mapping of the array language programs onto the GPU.

Array Language

Arrays in Obsidian do not, like in C, name an area of memory. Instead, an array is represented by an abstract data type, called `Arr`, supporting the following operations: `(!)` for indexing, `len` returns the length of an array and `mkArr` that given a function from indexes to elements and a length gives an array. For example using these operations reversing an array can be accomplished as follows:

```
rev :: Arr a -> Arr a
rev arr  = mkArr ixf n
    where
        ixf ix = arr ! (fromIntegral (n-1) - ix)
        n = len arr
```

Array reversal is polymorphic in the element type which gives the `rev` program the type `Arr a -> Arr a`. Internally an array is represented by the computation that gives its elements. The array type consists of two parts, a function from indices to elements and an integer representing its length:

```
data Arr a = Arr (IndexE -> a) Int
```

The length of the array is static, known at compile time, and is represented by an `Int`. The elements of an array can be `Int`, `Float` or `Bool` valued expressions, represented by the types `IntE`, `FloatE` and `BoolE`. Arrays can also contain arrays and tuples as elements.

Obsidian provides a number of functions on this array type. For example, a function can be mapped over an array using `fmap`.

```
fmap :: (a -> b) -> Arr a -> Arr b
```

The array type is also an instance of `Foldable`, so there is a function `foldr` (often called `reduce`) defined on arrays:

```
foldr :: (a -> b -> b) -> b -> Arr a -> b
```

Two other basic functions on arrays that are available are `pair` and `unpair`:

```
pair :: Arr a -> Arr (a,a)
unpair  :: Choice a => Arr (a,a) -> Arr a
```

The function `pair` takes an array and returns an array of pairs where the first element of the input array is paired up with the second, the third with the forth and so on. The `unpair` function does the opposite.

The `Choice` class contains those types that have an `ifThenElse` function defined on them:

```
ifThenElse :: Choice a => BoolE -> a -> a -> a
```

Another example of a function given in the array library is `zipp` of type `(Arr a, Arr b) -> Arr (a, b)`. This function performs on arrays what the normal Haskell `zip` does on lists. However, the input to `zipp` is a pair of arrays.

GPU Programs

The second layer of Obsidian offers a data type that represents a GPU program with input `a` and output `b`:

```
data a :-> b = ...
```

Informally we can think of `a :-> b` as representing programs that operate as illustrated in figure 3.7. This figure shows a program that performs some computation using a number of threads followed by a barrier synchronisation, and so on. The contents of the boxes marked with *Pure* can be thought of as containing an array program.The type constructor `:->` is a variant of an arrow, which in turn is a generalisation of a monad [SCS10]. One way to create a GPU program is by using the function `pure` of type `(a -> b) -> a :-> b`. For example the array language program `fmap (+1)`, that increments every element of an array, can be lifted to a GPU program:

```
incr :: Arr IntE :-> Arr IntE
incr = pure $ fmap (+1)
```

GPU programs can be executed on the GPU from *GHCi* (the Haskell interpreter) using the function `execute`:

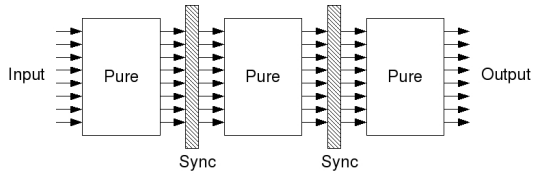

Figure 3.7: A GPU program of type `a :-> b` can be thought of as pure computations interspersed by syncs.

```
execute :: (Flatten a, Flatten b)
         => (Arr a :-> Arr b) -> [a] -> IO [b]
```

Instances of `Flatten` are all the types that can be stored in the GPU memory. Examples of types that are in `Flatten` are `IntE`, `FloatE`, `BoolE`. Arrays and pairs of things that are in Flatten are also instances of `Flatten`. Here, `execute` is used in order to run an instance of the `incr` program on the GPU:

```
*Obsidian> execute incr [0..9]
[1,2,3,4,5,6,7,8,9,10]
```

The elements of the Haskell list given to `execute` are used to create an input array to the kernel. Following this, the kernel is executed on the GPU and the result is read back and presented as a Haskell list.

In the example above, the `execute` function generates the following kernel corresponding to the given GPU program:

```
__global__ void generated(word* input,word* result){
  unsigned int tid = (unsigned int)threadIdx.x;
  extern __shared__ unsigned int s_data[];
  word __attribute__((unused)) *sm1 = &s_data[0];
  word __attribute__((unused)) *sm2 = &s_data[0];
  ix_int(result,tid) = (ix_int(input,tid) + 1);
}
```

The generated kernel has two arguments, an input array of words and an output array of words. Words represent 32-bit quantities that can be either floating point or integer valued. This particular kernel does not use any shared memory; the incremented values are stored directly into the result array that resides in global memory.

Given two GPU programs, `f` and `g`, of suitable types, a composite GPU program can be created by passing the output of `f` to the input of `g`. In Obsidian this is done using the composition operator (or combinator) `(->-)`:

```
(->-) :: (a :-> b) -> (b :-> c) -> (a :-> c)
```

The following illustrates the use of `(->-)` by implementing a program that increments every element of an array but also reverses the array:

```
increv :: Arr IntE :-> Arr IntE
increv = pure (fmap (+1)) ->- pure rev

*Obsidian> execute increv [0..9]
[10,9,8,7,6,5,4,3,2,1]
```

The code generated from the `increv` program is very similar to that of `inc` but the indexing is reversed:

```
__global__ void generated(word* input,word* result){
  unsigned int tid = (unsigned int)threadIdx.x;
  extern __shared__ unsigned int s_data[];
  word __attribute__((unused)) *sm1 = &s_data[0];
  word __attribute__((unused)) *sm2 = &s_data[0];
  ix_int(result,tid) = (ix_int(input,(9 - tid)) + 1);
}
```

The code generated for the `incr` and `increv` examples uses 10 threads to compute the resulting array. By default, the result will be computed using a number of threads equal to the number of elements in the return array.

The `increv` program can also be specified with an explicit storing of intermediate values between the `rev` and the `fmap (+1)`. This is accomplished using a primitive GPU program called `sync`:

```
sync  :: Flatten a => Arr a :-> Arr a

increv :: Arr IntE :-> Arr IntE
increv = pure (fmap (+1)) ->- sync ->- pure rev
```

This version of `increv` computes the same result as the previous one. However, it does so by computing `fmap (+1)` on the array, storing the intermediate result in shared memory followed by computing the reverse. The CUDA C code for this version of `increv` looks like this. Notice how the shared memory is used and the call to `__syncthreads()`:

```
__global__ void generated(word* input,word* result){
  unsigned int tid = (unsigned int)threadIdx.x;
  extern __shared__ unsigned int s_data[];
  word __attribute__((unused)) *sm1 = &s_data[0];
  word __attribute__((unused)) *sm2 = &s_data[8];
  ix_int(sm1,tid) = (ix_int(input,tid) + 1);
  __syncthreads();
  ix_int(result,tid) = ix_int(sm1,(7 - tid));
}
```

3.2.3 Outline of Code Generation

This subsection briefly reviews the code generation process using a hyphothetical program as example. The program under consideration is this:

```
prg = pure (fmap f) ->- sync -> rev ->- sync ->- pure (fmap g)
```

This program applies some function `f` to all elements of an array. The array is then reversed and lastly a function `g` is applied to all elements. Between every operation there is a `sync`. The program `prg` is an element of the datatype `(a :-> b)`. To see what happens during code generation, it is necessary to know the implementation of this type.

```
data a :-> b
   = Pure (a -> b)
   | Sync (a -> Arr FData) (Arr FData :-> b)
```

The type `(a :-> b)` is essentially a list of computations. The function `pure` is directly implemented using the constructor `Pure`. The function `sync` is implemented in terms of the constructor `Sync`:

```
sync :: Flatten a => (Arr a :-> Arr a)
sync = Sync (fmap toFData) (pure (fmap fromFData))
```

The example program `prg` would be represented as follows:

```
Sync (fmap (toFData . f))
     (Sync ((fmap toFData) . rev . (fmap fromFData))
           (Pure (fmap g)))
```

From this representation, which is essentially the list `[fmap f, rev, fmap g]`, the CUDA code is generated as outlined in figure 3.8. An input array is created and applied to the first stage of the computation, `(fmap f)`, resulting in:

```
Arr (\ix -> f(index (variable ``input'') ix)) n.
```

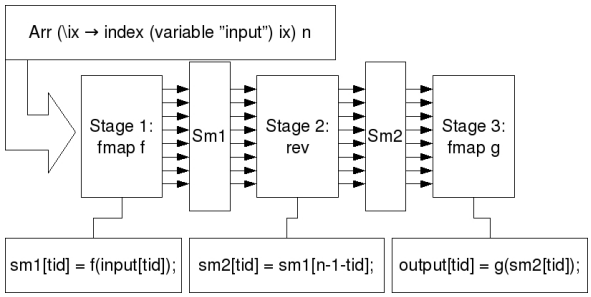

Figure 3.8: Outline of the code generation procedure.

The information given by this array is used to construct the C assignment statement `sm1[tid] = f(input[tid])` as seen in the figure.

Following this a new Obsidian level array that indexed into `sm1` is created and used as input to the second stage of the computation.

3.2.4 Case Studies

This subsection describes a few slightly larger Obsidian programs. The main purpose of this subsection is to show how we want to write GPGPU programs using Obsidian.

Sorting

Sorting is a popular function to place on the GPU, see [SA08, SHG09]. *Odd-Even merge sort*, due to Batcher, uses a merger component called the Odd-Even merger. This merger merges two sorted arrays. In Obsidian this merger and sorter can be implemented using the combinators `two`, `ilv`, `evens` and `odds`. For a detailed description of these combinators see [CSS01].

```
mergeOE :: (Choice a, Flatten a)
         => Int -> ((a,a) -> (a,a)) -> (Arr a :-> Arr a)
mergeOE 1 f = pure (evens f)
mergeOE n f = ilv (mergeOE (n-1) f) ->- sync ->- pure (odds f)
```

This is actually a pattern into which any two-input two-output function can be plugged. However, we choose the following *comparator* function (a compare and swap operation). The resulting merger can be executed on the GPU.

```
cmp :: (Ordered a, Choice (a, a)) => (a, a) -> (a, a)
cmp (a,b) = ifThenElse (a <* b)  (a,b) (b,a)

*Obsidian> let input = ([1,3,5,7,2,4,6,8] :: [IntE])
*Obsidian> execute GPU (mergeOE 3 cmp) input
[1,2,3,4,5,6,7,8]
```

Given this merger, the sorter is implemented by recursively sorting two halves of the input array followed by an application of the merger:

```
sortOE :: Int -> (Arr IntE :-> Arr IntE)
sortOE  0 =  pure id
sortOE  n =  two (sortOE (n-1)) ->- sync ->- mergeOE n cmp

*Obsidian> execute GPU (sortOE 3) [6,0,1,3,4,2,5,7]
[0,1,2,3,4,5,6,7]
```

The performance of the code generated from this sorter specification is poor. It runs about three times slower than the example Bitonic sorter distributed with the CUDA SDK. The reason is that nested applications of the `two` and `ilv` combinators are hard to optimise. We expect that single (parameterised) combinators corresponding to particular nestings of these combinators will bring significant performance improvement.

Parallel Prefix

This subsection shows the implementation of a parallel prefix (also called scan) kernel, known as `sklansky` after J. Sklansky [Skl60]. This kernel will then be optimised step-by-step using Obsidian. This optimisation effort uses an experimental feature of Obsidian called `syncHow`. This is different from the normal `sync` in the way it assigns work to threads. For a more thorough explanation of `syncHow` see [SCS10]. For an explanation of the parallel-prefix operation see [Ble90].

The sklansky parallel prefix algorithm is implemented by splitting the inputs in two halves and recursively applying sklansky to both halves. The two sub-results are then joined by applying the operation between the highest index of the first sub-result and all the elements in the second sub-result, this is done using a function called `fan`:

```
fan op arr = conc (a1, (fmap (op c) a2))
    where (a1,a2) = halve arr
          c       = a1 ! (fromIntegral (len a1 - 1))
```

The `sklansky` function is now implemented using `two` and `fan`:

```
sklansky :: (Flatten a, Choice a)
         => Int -> (a -> a -> a) -> (Arr a :-> Arr a)
sklansky 0 op = pure id
sklansky n op = two (sklansky (n-1) op) ->-
                pure (fan op) ->- sync
```

Executing (`sklansky 3 (+)`) on the GPU given input `[0..7]` computes the prefix sum as expected, returning `[0,1,3,6,10,15,21,28]`

If `sklansky` is used to generate code for an array of 512 elements, it uses 512 threads to calculate the prefix sums. However, the number of applications of the `op` operator that is needed in any stage of the algorithm is only 256. This indicates that using 256 threads to compute the result would give more efficient use of the GPU's resources.

Since the code generated by Obsidian is not by default *in-place* with regard to shared memory, each thread in the 256 threaded program needs to both perform the operation between two elements and copy one value unchanged. This is desired because then each thread performs the exact same operations, which means that there is no risk for divergence within a warp.

This perfect division of labour is not obtainable with the current implementation of the `How` argument to `sync`. However, one division of the work that has experimentally been shown to perform well, see table in next subsection, is to let each thread tid perform the work of tid and $tid + 256$. This program is shown below:

```
sklansky1 :: (Flatten a, Choice a)
          => Int -> (a -> a -> a) -> (Arr a :-> Arr a)
sklansky1 0 op = pure id
sklansky1 n op = two (sklansky1 (n-1) op) ->-
                 pure (fan op) ->- syncHow (pairNth 256)
```

Another optimisation to apply to the sklansky algorithm is the removal of unnecessary `__syncthreads()` calls from the generated code, this is done by using `syncWarp`. It is up to the programmer to ensure that it is safe to use `syncWarp`. For a sklansky network of size 32, it should be safe to leave out the `__syncthreads` since all of the communication stays within a warp. Using this information leads to the following code where the sklansky networks of size 32 or smaller use `syncWarpHow`.

```
sklansky2 :: (Flatten a, Choice a)
          => Int -> (a -> a -> a) -> (Arr a :-> Arr a)
sklansky2 0 op = pure id
sklansky2 n op = two (sklansky2 (n-1) op) ->- pure (fan op)
                   ->-  if n <= 5
                        then syncWarpHow (pairNth 256)
                        else syncHow (pairNth 256)
```

A last tweak to force the algorithm to compute *in-place*. This is accomplished using the primitives `syncIP` and `syncIPHow`, which are variants of `sync` that enforce *in-place* computation.

Parallel Prefix Sums on large arrays

The Sklansky kernels given above can be used in an algorithm that computes the parallel prefix of a large array. This is done in the same way as in [HS07]. The table below shows the results of using the different parallel prefix kernels from above in an algorithm for computing the prefix sums of 2^{20} elements.

Kernel	In-place	Sync in Warp	Threads	ms
NVIDIA SDK	Yes	N/A	256	0.64
Hand Optimised	Yes	No	256	0.74
sklansky	No	yes	512	1.06
sklansky1	No	yes	256	0.89
sklansky2	No	No	256	0.86
sklansky3	Yes	No	256	0.79

The table above shows the running times of six different Sklansky kernels. All runtime measurements where performed on an NVIDIA 9800GX2 using one GPU. The code labeled *Hand Optimised* was written directly in CUDA. This kernel was the result of two afternoons of optimisation effort by two people. The three following kernels *sklansky* to *sklansky2* are generated from the given Obsidian programs. The version called *sklansky3* is identical to `sklansky2` except is in-place. Even the very first version, *sklansky* performs reasonably well, but by using the experimental features the performance can be pushed quite close to the hand optimised version. The fastest kernel, called *NVIDIA SDK*, is the one supplied with the CUDA SDK. This kernel is based on the Brent-Kung network and its implementation is shown in [HS07]. The `NVIDIA SDK` kernel is highly optimized with regards to its memory access pattern and so the code is considerably more complicated than that of any of the Obsidian parallel-prefix kernels.

3.2.5 Future work

Obsidian is work in progress and as such it changes a lot. Future work will both explore improvements to code generation for kernels and ways to coordinate kernels.

In order to get to the quite efficient version of the `sklansky` kernel in subsection 3.2.4, the experimental `How` argument to `sync` was needed. This needs to be explored further. `How` as it is today, any function of type `IndexE -> [IndexE]`,

offers too much freedom and with that the risk of introducing errors. We are searching for an elegant model for expressing how and what to compute, and for ways to integrate this with the language. This is our biggest research challenge.

Subsection 3.2.4, on case studies, showed that it is possible to generate quite efficient code from Obsidian programs. The current version generates efficient code for very specific uses of the `two` combinator, for example the kind used in the `sklansky` parallel prefix network. More work needs to be done in order to find a good way to produce efficient code for a larger set of combinators. This may very well involve designing new combinators that do not need to be nested as the current ones do.

The CUDA code generated by Obsidian uses very few registers. In fact, the generated code is under-utilizing the resources of the GPU. The generated code also contains many common subexpressions. This hints that applying a step of common subexpression elimintaion would be beneficial, increasing the register use and reducing the repeated computation of values. Performing common subexpression elimination would also lower the instruction count, which is quite high to begin with, because loops are unrolled. The register usage and instruction count values on which these conclusions are based where obtained using the CUDA profiler supplied as part of the CUDA toolkit [NVIc].

Obsidian can currently only be used to generate kernels, the small building blocks used to form larger GPU algorithms. As future work, methods of describing kernel coordination in a high level fashion will be investigated.

At the University of New South Wales, Manuel Chakravarty et. al. are developing another language for GPGPU programming called Accelerate. Accelerate is also embedded in Haskell, but intends to be at a higher level of abstraction than Obsidian. Accelerate supplies the programmer with a collection of basic building blocks for data parallel programming. Amongst these building blocks are `map`, `zipWith` and `scan`. With the Accelerate team at UNSW, we are looking into ways of combining the two complementary approaches. In this setting Obsidian could be used to generate the underlying kernels (the building blocks) that the Accelerate programmer uses to build a GPGPU application.

A limitation of Obsidian is the inability to express algorithms where the length of the output array is dependent on the input data. An example of such a data dependent algorithm is `filter`. The `filter` function takes a sequence of elements and a predicate and produces a sequence of those elements for whom the predicate holds. Future work will also consist of searching for a suitable minimal language construct to add that enables expressing such algorithms.

3.2.6 Related work

GPUs are becoming more and more interesting to use in non-graphical applications. A modern GPU is a manycore machine with, today, hundreds of processing elements. The question of how to program these machines arises. NVIDIA's answer is CUDA [NVI08]. CUDA supplies a slightly extended version of C in which the programmer can specify GPU kernels and the controlling CPU program in the same language. In CUDA the programmer writes a single program parameterised over a thread identity. Compare this to Obsidian where the program describes a computation over an array.

There are a number of other C/C++ based languages that target GPGPU programmers, for example Brook[BFH+04] and RapidMind[McC06]. Brook, CUDA and RapidMind are major improvements from what was previously available for the programmer interested in general purpose computations on the GPU.

Higher level approaches are also being investigated. PyGPU embedds a GPGPU programming language in Python[LO06]. PyGPU makes use of Python's introspective abilities to generate efficient code.

Like Obsidian, Accelerate is embedded in Haskell [LCGK09]. Accelerate however, is higher level language than Obsidian. Where the purpose of Obsidian is to implement basic algorithmic building blocks such as reductions and prefix sums, Accelerate provide these building blocks as primitives.

Vertigo is another GPU programming language embedded in Haskell[Ell04]. Vertigo can be used to describe procedureal surfaces and textures and generates efficient DirectX9 shader code.

There are also many examples of domain specific languages (DSLs) that target not only GPUs but also other platforms, such as multicore machines. Of particular interest to us is Microsoft's Accelerator project [TPO06]. Interestingly, Accelerator includes the same restriction to "hardware-like" algorithms as Obsidian does.

3.2.7 Discussion and conclusion

Obsidian is work in progress and there are many loose ends to tie up and paths left to explore. In subsection 3.2.4, the strengths of Obsidian show; it is possible to express quite complex algorithms using short and elegant programs. The case studies also show that it is possible to generate quite efficient code from these high level descriptions. However, this needs more work in order to more reliably produce efficient code and to broaden the available set of combinators for combining GPU programs.

Obsidian offers the GPGPU programmer a higher level language, while trying not to sacrifice too much performance. When Programming in CUDA C, the indexing arithmetic often gets quite complex. This is a common trait of data-parallel programming in C-like languages. One goal of Obsidian is to be able to express these

algorithms without the complex index manipulations; instead the data access pattern is captured in the use of functions such as `pair` and `two` or in the recursive structure of the Obsidian program.

This paper presented Obsidian an embedded language for GPGPU programming that offers a higher level of abstraction compared to languages such as CUDA. Obsidian allows the programmer to think more of the algorithm and less of architectural details of the GPU. The contributions of Obsidian to the GPGPU field is a higher level programming environment that eases experimentation.

Acknowledgments

This research is funded by the Swedish Research Council.

Bibliography

[BCSS98] 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.

[BFH+04] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, , and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. In *SIGGRAPH*, 2004.

[Ble90] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

[CSS01] Koen Claessen, Mary Sheeran, and Satnam Singh. The Design and Verification of a Sorter Core. In *Proc. Int. Conf. on Correct Hardware Design and Verification Methods (CHARME)*, volume 2144 of *Springer LNCS*, pages 355–369, 2001.

[EFdM03] Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(2), 2003.

[Ell04] Conal Elliott. Programming graphics processors functionally. In *Proceedings of the 2004 Haskell Workshop*. ACM Press, 2004.

[HS07] Mark Harris and Shubhabrata Sengupta. Parallel Prefix Sum (Scan) with CUDA. In *GPU Gems 3*, 2007.

[Hug00] John Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111, 2000.

[LCGK09] Sean Lee, Manuel M. Chakravarty, Vinod Grover, and Gabriele Keller. GPU Kernels as Data-Parallel Array Computations in Haskell. In *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods (EPHAM)*, 2009.

[LO06] 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.

[McC06] M.D. McCool. Data-Parallel Programming on the Cell BE and the GPU using the RapidMind Development Platform. In *GSPx Multicore Applications Conference*, Oct. 2006.

[Ngu07] Hubert Nguyen. *GPU Gems 3: 3D and General Programming Techniques for GPUs*. Addison-Wesley Professional, August 2007.

[NVIa] NVIDIA. CUDA C Best Practices Guide. `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/`.

[NVIb] NVIDIA. NVIDIA CUDA. `http://www.nvidia.com/cuda`.

[NVI06] NVIDIA. Technical brief: Nvidia geforce 8800 gpu architecture overview, 2006.

[NVI08] NVIDIA. NVIDIA CUDA Programming Guide 2.0, 2008.

[PF05] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2005.

[SA08] Erik Sintorn and Ulf Assarsson. Fast parallel GPU-sorting using a hybrid algorithm. *J. Parallel Distrib. Comput.*, 68(10):1381–1388, 2008.

[SCS10] Joel Svensson, Koen Claessen, and Mary Sheeran. GPGPU Kernel Implementation using an Embedded Language: a Status Report, 2010. Dept. of Computer Science and Engineering, Chalmers, Tech. Report Number 2010:1.

[SHG09] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *Proc. IEEE Int. Symp. on Parallel & Distributed Processing*, pages 1–10, 2009.

[Skl60] J. Sklansky. Conditional Sum Addition Logic. *Trans. IRE*, EC-9(2), June 1960.

[SSC09] Joel Svensson, Mary Sheeran, and Koen Claessen. Obsidian: A Domain Specific Embedded Language for General-Purpose Parallel Programming of Graphics Processors. In *Proc. of Implementation and Applications of Functional Languages (IFL)*, Lecture Notes in Computer Science. Springer Verlag, March 2009.

[TPO06] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. volume 34, pages 325–335, New York, NY, USA, October 2006. ACM.

3.3 Paper D: Expressive Array Constructs in an Embedded GPU Kernel Programming Language

Koen Claessen, Mary Sheeran, Bo Joel Svensson

Abstract

Graphics Processing Units (GPUs) are powerful computing devices that with the advent of CUDA/OpenCL are becomming useful for general purpose computations. Obsidian is an embedded domain specific language that generates CUDA kernels from functional descriptions. A symbolic array construction allows us to guarantee that intermediate arrays are fused away. However, the current array construction has some drawbacks; in particular, arrays cannot be combined efficiently. We add a new type of *push arrays* to the existing Obsidian system in order to solve this problem. The two array types complement each other, and enable the definition of combinators that both take apart and combine arrays, and that result in efficient generated code. This extension to Obsidian is demonstrated on a sequence of sorting kernels, with good results. The case study also illustrates the use of combinators for expressing the structure of parallel algorithms. The work presented is preliminary, and the combinators presented must be generalised. However, the raw speed of the generated kernels bodes well.

3.3.1 Introduction

Graphics Processing Units (GPUs) are parallel computers with hundreds to thousands of processing elements. The CUDA and OpenCL languages make available the power of the GPU to programmers interested in general purpose computations. In CUDA and OpenCL, the programmer writes *kernels*, Single Program Multiple Data (SPMD) programs that are executed by groups of threads on the available processing elements of the GPU.

CUDA and OpenCL are general purpose programming languages, mirroring the increased capabilities of a modern GPU to target that domain. However, these languages lack compositionality. Also, being based in C/C++ means that the core idea in a program may not be easily visible.

Embedded DSLs for GPGPU programming

We are aiming for a GPU programming language that is more concise than mainstream languages such as CUDA and OpenCL. Obsidian is a domain specific embedded language (DSEL) implemented in Haskell. When an Obsidian program is run, a representation of the program is created as a syntax tree. For more information on EDSL implementation see [EFdM03]. The program representation generated when running an Obsidian program is compiled into CUDA code. We are also working on an OpenCL backend.

Our approach is different from that of other Haskell DSELs targetting GPUs [CKL+11, MM10, Lar11]. We do not try to abstract away from all the peculiarities of GPU programming, but rather provide a higher level language in which to experiment with them. For instance, Accelerate provides a standard set of basic operations such as `map`, `reduce` and `zipWith` as built in skeletons, implemented with the help of small, predefined, hand-tuned CUDA kernels [CKL+11]. Obsidian, on the other hand, allows the user to experiment with the *generation* of small kernels for fixed size array inputs from higher level descriptions. It is intended to allow the user to play with the kinds of tradeoffs that are important when writing such high performance building blocks; in this paper, the main consideration is the number of array elements of the input and output that are manipulated by a single thread in the generated CUDA code. An important aspect of Obsidian is the symbolic array representation used, along with its associated `sync` operation. As we shall see, the `sync` operation allows the programmer to guide code generation and control parallelism and thread use [JS11].

In Obsidian, a kernel that sums an array can be expressed as:

```
sum :: Array IntE -> Kernel (Array IntE)
sum arr | len arr == 1 = return arr
        | otherwise    = (pure (fmap (uncurry (+)) . pair)
                           ->- sync
                           ->- sum) arr
```

The result of running this kernel on an eight element input array, `runKernel sum (namedArray "input" 8)`, is an intermediate representation of the computation (shown in slightly pretty-printed form):

```
arr0 = malloc(16)
par i 4 {
arr0[i] = ( +  input[( *  i 2 )] input[( +  ( *  i 2 ) 1 )] );
}Sync
arr1 = malloc(8)
par i 2 {
arr1[i] = ( +  arr0[( *  i 2 )] arr0[( +  ( *  i 2 ) 1 )] );
}Sync
arr2 = malloc(4)
par i 1 {
arr2[i] = ( +  arr1[( *  i 2 )] arr1[( +  ( *  i 2 ) 1 )] );
}Sync
```

The named intermediate arrays in this representation are then laid out in GPU shared memory and CUDA code can be generated (here for arrays of length eight)[4]:

```
__global__ void sum(int *input0,int *result0){
  unsigned int tid = threadIdx.x;
  unsigned int bid = blockIdx.x;
  extern __shared__ unsigned char sbase[];
  (( int *)sbase)[tid] =
    (input0[((bid*8)+(tid*2))]+
     input0[((bid*8)+((tid*2)+1))]);
  __syncthreads();
  if (tid<2){
    (( int *)(sbase + 16))[tid] =
      ((( int *)sbase)[(tid*2)]+
       (( int *)sbase)[((tid*2)+1)]);
  }
  __syncthreads();
  if (tid<1){
    (( int *)sbase)[tid] =
      ((( int *)(sbase+16))[(tid*2)]+
       (( int *)(sbase+16))[((tid*2)+1)]);
  }
  __syncthreads();
  if (tid<1){
    result0[(bid+tid)] = (( int *)sbase)[tid];
  }
}
```

Arrays in Obsidian

An array is represented by an indexing function and a length:

```
data Array a = Array (UWordE -> a) Word32
```

This array representation has served us well. It has these properties:

- Fusion of operations is automatic.
- It naturally describes a data-parallel computation suitable for CUDA/OpenCL generation.
- Many basic operations can be implemented: `map`, `zipWith` etc.

Using this array representation in a DSEL is not new; the first occurence that we know of is in Pan [Ell03]. Similar array representations have also later been used in Feldspar [ACS+11], and more recently also in the Repa library [KCL+10]. Functions for indexing and getting the length of arrays are as follows:

```
(!) :: Array a -> UWordE -> a
(Array ixf _) ! ix = ixf ix

len :: Array a -> Word32
len (Array _ n) = n
```

[4] An alignment qualifier for shared memory has been omitted to save space in the listings showing generated code

A *Functor* instance for the `Array` datatype is

```
instance Functor Array where
  fmap f arr = Array (\ix -> f (arr ! ix)) (len arr)
```

Now, composed applications of `fmap` will be automatically fused. This is illustrated in the example program below and the CUDA generated from it.

```
mapFusion :: Array IntE -> Kernel (Array IntE)
mapFusion = pure (fmap (+1) . fmap (*2))
```

```
__global__ void mapFusion(int *input0,int *result0){
  unsigned int tid = threadIdx.x;
  unsigned int bid = blockIdx.x;

  result0[((bid*32)+tid)] = ((input0[((bid*32)+tid)]*2)+1);
}
```

Both of these code listings need explanation. In the Haskell code, `mapFusion` has type `Array IntE -> Kernel (Array IntE)`; `Kernel` is a state monad that accumulates CUDA code as well as provides new names for intermediate arrays. Neither of these features of the monad is activated by this example though. The `pure` function is defined using the monad's `return` as `pure f a = return (f a)`. In this case, it lifts a function of type `Array IntE -> Array IntE` into a kernel.

The generated CUDA code computes the result array using a number of threads equal to the length of that array. In this case, the kernel was generated to deal with arrays of length 32. The important detail to notice in the CUDA code is that there is no intermediate array created between the `(*2)` and the `(+1)` operations.

The `mapFusion` example could just as well have been implemented using the kernel sequential composition combinator, `->-`.

```
mapFusion :: Array IntE -> Kernel (Array IntE)
mapFusion = pure (fmap (*2)) ->- pure (fmap (+1))
```

Exactly the same CUDA code is then generated.

In some cases, it is necessary to force computation of intermediate arrays. This can be used to share partial computations between threads and to expose parallelism. In Obsidian, the tool for this is called `sync`, a built-in kernel. Using `sync` as follows prevents fusion of the two operations:

```
mapUnFused :: Array IntE -> Kernel (Array IntE)
mapUnFused = pure (fmap (*2)) ->- sync ->- pure (fmap (+1))
```

The generated CUDA code now stores an intermediate result in local shared memory before moving on.

```
__global__ void mapUnFused(int *input0,int *result0){
  unsigned int tid = threadIdx.x;
  unsigned int bid = blockIdx.x;
  extern __shared__ unsigned char sbase[];
  (( int *)sbase)[tid] = (input0[((bid*32)+tid)]*2);
  __syncthreads();
  result0[((bid*32)+tid)] = ((( int *)sbase)[tid]+1);
}
```

Intermediate arrays are laid out in the `sbase` array in shared memory. Since we may store arrays of many different types in the same locations of the shared memory at different times during the execution, the type casts used in the code above are necessary.

Sync and parallelism

The `sync` operation also enables the writing of parallel reduction kernels. A reduction operation is an operation that takes an array as input and produces a singleton array as output.

First, we define `zipWith` and `halve` on Obsidian arrays.

```
zipWith :: (a -> b -> c) -> Array a -> Array b -> Array c
zipWith op a1 a2 = Array (\ix -> (a1 ! ix) `op` (a2 ! ix))
                   (min (len a1) (len a2))

splitAt :: Word32 -> Array a -> (Array a, Array a)
splitAt n arr =
  (Array (\ix -> arr ! ix) n ,
   Array (\ix -> arr ! (ix + fromIntegral n)) (len arr - n))

halve arr = splitAt ((len arr) `div` 2) arr
```

A reduction kernel that takes an array whose length is a power of two and gives an array of length one can be defined recursively. Defining kernels recursively results in completely unrolled CUDA kernels, and kernel input size must be known at compile time. The approach to reduction taken here is to split the input array into two halves and then apply `zipWith` of the combining function to the two halves, repeating the process until the length is one.

```
reduceS :: (a -> a -> a) -> Array a -> Kernel (Array a)
reduceS op arr | len arr == 1 = return arr
               | otherwise    =
                 (pure ((uncurry (zipWith op)) . halve)
                  ->- reduceS op) arr
```

Since the output of this kernel is of length one, and the number of elements in the output array specifies the number of threads used to compute it, this function, `reduceS`, defines a sequential reduction. The generated code for arrays of length eight is

```
__global__ void reduceSAdd(int *input0,int *result0){
  unsigned int tid = threadIdx.x;
  unsigned int bid = blockIdx.x;

  result0[(bid+tid)] =
    (((input0[((bid*8)+tid)]+
       input0[((bid*8)+(tid+4))])+
      (input0[((bid*8)+(tid+2))]+
       input0[((bid*8)+((tid+2)+4))]))+
     ((input0[((bid*8)+(tid+1))]+
       input0[((bid*8)+((tid+1)+4))])+
      (input0[((bid*8)+((tid+1)+2))]+
       input0[((bid*8)+(((tid+1)+2)+4))])));
}
```

Sequential reduction is not very interesting for GPU execution, but the fix is simple. A well placed use of `sync` indicates that we want to compute, after each `zipWith` phase, the intermediate arrays using as many threads as that intermediate array is long. The effect is shown in the code below.

```
reduce :: Syncable Array a
          => (a -> a -> a) -> Array a -> Kernel (Array a)
reduce op arr | len arr == 1 = return arr
              | otherwise    =
                (pure ((uncurry (zipWith op)) . halve)
                 ->- sync
                 ->- reduce op) arr
```

The CUDA code for reduction with addition on eight elements is

```
__global__ void reduceAdd(int *input0,int *result0){
  unsigned int tid = threadIdx.x;
  unsigned int bid = blockIdx.x;
  extern __shared__ unsigned char sbase[];
  (( int *)sbase)[tid] =
    (input0[((bid*8)+tid)]+
     input0[((bid*8)+(tid+4))]);
  __syncthreads();
  if (tid<2){
    (( int *)(sbase + 16))[tid] =
      ((( int *)sbase)[tid]+
       (( int *)sbase)[(tid+2)]);
  }
  __syncthreads();
  if (tid<1){
    (( int *)sbase)[tid] =
      ((( int *)(sbase+16))[tid]+
       (( int *)(sbase+16))[(tid+1)]);
  }
  __syncthreads();
  if (tid<1){
    result0[(bid+tid)] = (( int *)sbase)[tid];
  }
}
```

In this generated CUDA, three phases can be identified. The first uses four threads to compute a four element intermediate array; the second uses two threads, and so on. At the very end, a single thread copies the result from local shared memory to global memory.

Drawbacks of Obsidian Arrays

The previous subsection described positive aspects of the array representation that we have used so far. There are, however, circumstances in which this Array representation is too restricted.

Take the problem of concatenating two arrays. Using the array representation described above, the only way to concatenate two arrays is to introduce a conditional into the indexing function. If `f` and `g` are the indexing functions of two arrays that

are to be concatenated, and `n1` is the length of the first array, the indexing function of the result must be

```
new ix = if (ix < n1)
         then f ix
         else g (ix - n1)
```

The following program concatenates two arrays:

```
catArrays :: (Array IntE, Array IntE) -> Kernel (Array IntE)
catArrays = pure conc
```

When it is used to generate a CUDA kernel that concatenates two arrays of length 16, the following code is the result:

```
__global__ void catArrays(int *input0,int *input1,int *result0){
  unsigned int tid = threadIdx.x;
  unsigned int bid = blockIdx.x;

  result0[((bid*32)+tid)] =
    (tid<16) ? input0[((bid*16)+tid)] :
               input1[((bid*16)+(tid-16))];
}
```

Now, conditionals like these are *bad* in code to execute on a GPU, with its wide-SIMD data-parallel model. Separating the operation into two assignments and using half as many threads gives much higher performance.

```
__global__ void catArraysByHand(int *input0,
                                int *input1,
                                int *result0){
  unsigned int tid = threadIdx.x;
  unsigned int bid = blockIdx.x;

  result0[((bid*32)+tid)] = input0[((bid*16)+tid)];
  result0[((bid*32)+tid+16)] = input1[((bid*16)+tid)];
}
```

There are cases where code with conditionals is not that bad. An expert on NVIDIA GPUs in particular may say that code with the condition `(tid < 32)` is fine, since 32 is the SIMD width of those GPUs. However, any number that is not a multiple of 32 would lead to poor performance, so in general this is a problem. Worse still, *zipping* two arrays together and then *unpairing* (to get an array of elements) leads to code that takes two different paths depending on odd or even *thread id*. When a GPU executes such code, it shuts down half of the threads and computes the two paths in sequence.

```
zippUnpair :: (Array IntE, Array IntE) -> Kernel (Array IntE)
zippUnpair = pure (unpair . zipp)
```

The `zipp` and `unpair` operations are defined as follows:

```
zipp :: (Array a, Array b) -> Array (a, b)
zipp (arr1,arr2) =
     Array (\ix -> (arr1 ! ix, arr2 ! ix))
           (min (len arr1) (len arr2))
```

```
unpair :: Choice a => Array (a,a) -> Array a
unpair arr =
    let n = len arr
    in  Array (\ix -> ifThenElse ((mod ix 2) ==* 0)
                      (fst (arr ! (ix `shiftR` 1)))
                      (snd (arr ! (ix `shiftR` 1)))) (2*n)
```

Code generated from the `zippUnpair` program exhibits really poor performance; at any time half of the threads are shut down.

```
__global__ void zippUnpair(int *input0,
                           int *input1,
                           int *result0){
  unsigned int tid = threadIdx.x;
  unsigned int bid = blockIdx.x;

  result0[((bid*64)+tid)] =
    ((tid%2)==0) ? input0[((bid*32)+(tid>>1))] :
                   input1[((bid*32)+(tid>>1))];
}
```

If we wrote this CUDA program by hand, we would, again, split it up into two phases so that all threads can progress in parallel.

The arrays described so far, with an indexing function and a length, have been nicknamed *Pull arrays* for how they describe how to compute an element by *pulling* data from a number of places. Using just Pull arrays, we have been unable to solve the problems described so far in this section. The solution is to add a complementary array type to Obsidian.

3.3.2 Push Arrays

In order to improve low level control for the programmer, *Push arrays* are added to Obsidian. The old Pull arrays are still available, along with the new array type.

Some operations, typically involving taking arrays apart, are easily described using Pull arrays, giving efficient code. In those cases, using a Push array would add complexity in the implementation for no performance benefit. Other operations cannot be implemented efficiently with Pull arrays, but Push arrays then provide the solution. This duality is apparent when looking at operations on Pull arrays such as `halve` and `conc` (for concatenate). The `halve` function is efficient since it introduces no diverging conditionals. The `conc` function, on the other hand, introduces conditionals. Concatening two arrays using the `concP` combinator, implemented on Push arrays, allows us to generate the desired code:

```
catArrayPs :: (Array IntE, Array IntE) -> Kernel (ArrayP Int)
catArrayPs = pure concP

__global__ void catArrayPs(int *input0,int *input1,int *result0){
  unsigned int tid = threadIdx.x;
```

```
  unsigned int bid = blockIdx.x;

  result0[((bid*32)+tid)] = input0[((bid*16)+tid)];
  result0[((bid*32)+(16+tid))] = input1[((bid*16)+tid)];

}
```

Compared to the CUDA code for `catArrays`, this kernel uses only 16 threads instead of 32. At each step of the computation, all the threads are fully busy doing exactly the same thing, which is the preferred mode of execution on the target platform.

What are Push Arrays?

The idea behind Push arrays is to have a way to describe where elements are supposed to end up. In some sense, a Push array produces a collection of Index/Value pairs. This makes Push arrays complementary to Pull arrays. For example, it is possible for a Push array to output several elements at the same index (which we probably need to control carefully). Push arrays should permit us to provide more expressive operations on arrays to the user, including an operation similar to Haskell's `filter` on lists. Here, we consider a different advantage of adding push arrays: finer control over patterns of thread use in generated code.

A Push array consists of three parts: a function in continuation passing style, a `Program` datatype and an array datatype.

```
type P a = (a -> Program) -> Program
```

For another example of using continuations and a more complete description of their meaning and application, see [Cla99].

The `Program` datatype has now been adopted as Obsidian's internal representation of CUDA programs.

```
data Program
  = Skip
  | forall a. Scalar a => Assign Name UWordE (Exp a)
  | Par (UWordE -> Program) Word32
  | Allocate Name Word32 Type
  | Synchronize
  | ProgramSeq Program
               Program
```

Even Obsidian programs that never explicitly uses a Push array will also be represented by this datatype.

Note that the `Par` constructor, the *parallel for loop*, could potentially introduce nesting, which would lead to *nested data-parallelism*. We do not compile nested data parallelism into CUDA, and right now this is guaranteed by taking care not to

introduce any nesting in the library functions provided. Some of the simpler cases of nestedness should be possible to take care of quite easily. For example, one extra level of nesting could be done by sequential execution in each thread of the GPU; using sequential computations per thread has been shown to be beneficial [BOA09]. But for the general case of arbitrary nesting, some method of flattening is needed. We also assume that both `Allocate` and `Synchronize` occur only at the top level in objects of type `Program`.

Now, a Push array is a function in continuation passing style coupled with a length.

```
data ArrayP a = ArrayP (P (UWordE, a)) Word32
```

There is a function that takes an array and turns it into a Push array, called `push`. This function is defined for both Pull and Push arrays:

```
class Pushable a where
  push :: a e -> ArrayP e

instance Pushable ArrayP where
  push = id

instance Pushable Array where
  push (Array ixf n) =
     ArrayP (\func -> Par (\i -> func (i,(ixf i))) n) n
```

Going in the other direction, from a Push array to a Pull array, is a costly operation; it involves writing all the elements to GPU memory followed by creating a Pull array that represents reading them. The task of writing intermediate values to memory has traditionally been up to the `sync` operation in Obsidian. Therefore, in this version, `sync` is overloaded to operate on both Pull and Push arrays. This means that the `sync` operation can be used both on arrays of type `Array` and of type `ArrayP`. The result type, however, is always `Array`.

When a Push array is synced, it is applied to a continuation that writes the elements into a named array in memory. The name to use is obtained through the `Kernel` monad.

```
targetArray :: Scalar a => Name -> (UWordE,Exp a) -> Program
targetArray n (i,a) = Assign n i a
```

After applying the Push array to `targetArray <name>`, the `sync` operation proceeds by storing away a representation of the program that computes the array called `<name>`; it returns a Pull array that reads elements from that same array.

Now we have seen enough of the implementation of Push arrays to be able to look at some operations. Earlier, we saw that the array concatenation function `conc` on Pull arrays leads to inefficient code. The Push version of this operation, called `concP` can be implemented as follows:

```
concP :: (Pushable arr1,
          Pushable arr2) => (arr1 a, arr2 a) -> ArrayP a
concP (arr1,arr2) =
  ArrayP (\func -> f func
                  *>*
                  g (\(i,a) -> func (fromIntegral n1 + i,a)))
         (n1+n2)
  where
     ArrayP f n1 = push arr1
     ArrayP g n2 = push arr2
```

The function `concP` takes two arrays, that can be Push or Pull arrays, and concatenates them into a single Push array. It does so by creating a sequential program, using the `*>*` operator for `Program` sequential composition. An example use of this combinator has already been displayed in the `catArrayPs` example.

The `zippUnpair` example shows a drawback similar to that of `catArrays` using Pull arrays. In this case, the problem is that the `unpair` function introduces a conditional that takes different paths depending on odd or even *thread id*. A Push array implementation of the `unpair` operation called `unpairP` can be given as follows:

```
unpairP :: Pushable arr => arr (a,a) -> ArrayP a
unpairP arr =  ArrayP (\k -> f (everyOther k)) (2 * n)
  where
    ArrayP f n = push arr

everyOther :: ((UWordE, a) -> Program ())
              -> (UWordE, (a,a)) -> Program ()
everyOther f  = \(ix,(a,b)) -> f (ix * 2,a) *>* f (ix * 2 + 1,b)
```

Just like `concP`, this function takes either a Push or Pull array as input, and produces a Push array as result.

Rewriting the example from earlier using `unpairP` gives:

```
zippUnpairP :: (Array IntE, Array IntE) -> Kernel (ArrayP IntE)
zippUnpairP = pure (unpairP . zipp)
```

In this case, the generated code looks as follows:

```
__global__ void zippUnpairP(int *input0,int *input1,int *result0){
  unsigned int tid = threadIdx.x;
  unsigned int bid = blockIdx.x;

  result0[((bid*64)+(tid*2))] = input0[((bid*32)+tid)];
  result0[((bid*64)+((tid*2)+1))] = input1[((bid*32)+tid)];
}
```

Again, we get CUDA code that uses half as many threads as the inefficient version, but all threads are occupied at all times. This uses the resources more efficiently.

Being able to generate the kind of code that we have just seen is something we have desired for a long time. We believe that Push arrays are an important tool for obtaining high performance kernels. The results in section 3.3.3 bear this out.

3.3.3 Application

Sorting on a GPU

In this section, we introduce combinators that express patterns of computation on whole arrays, and show their application to the development of fast sorting kernels. By kernel, we mean a computation that is performed by multiple threads, each performing the same computation, in a single block. The computation is performed entirely on the GPU, operating on a short array, which has been placed into shared memory. On the GPU on which we perform measurements, the maximum number of threads in a single block is 512. Thus, we will build and benchmark a sequence of kernels that sort 512 inputs. Our first kernels are implemented using one thread per array element. Next, we show how Push arrays allow us to move to having each thread operate on two array elements, giving a substantial performance improvement.

Sorting kernels are typically used as building blocks in larger programs to sort much larger sequences of inputs. In section 3.3.3, we show how to build a sorter for large arrays from small building blocks, including small kernels for sorting and merging that are generated from Obsidian. Our small kernels are constructed in the form of sorting and merging *networks*, building on Batcher's bitonic merger [Bat68] and on the periodic balanced merger [DPLR89]. We chose also to implement the large sorter used in benchmarking the small kernels as a sorting network. However, large sorters that are not themselves sorting networks (with typical examples being radix sort and quicksort) often call small sorting networks when they need to sort small arrays during their execution. Thus, small, fast sorting kernels have a variety of uses.

Describing Batcher's bitonic merger

The bitonic merger is typically presented as a recursive construction and we have earlier explored ways to describe and analyse it in both (our) Ruby and in Lava [She89, CSS01]. Here, we consider iterative descriptions using similar combinators.

Figure 3.9 illustrates the merger for 16 inputs. Data flows from left to right. The vertical lines indicate components that operate on two array elements, placing the minimum onto the lower (abstract) wire, and the maximum onto the other output of the component. The leftmost *stage* operates (for $n = 16$) on elements that are 8 apart. the next stage deals with elements that are 4 apart, and so on.

We introduce a combinator `ilv1`, for *interleave*, that captures this pattern. `ilv1 i f g` applies `f` to elements 2^i apart, producing the output at the lower of the two input indices; it applies `g` to the same pairs of elements, producing the output on the upper index. Defining `stage i` to be `ilv1 i min max`, the four stages in the diagram are simply `stage` applied to3, 2 1 and 0. The definition of `ilv1` makes

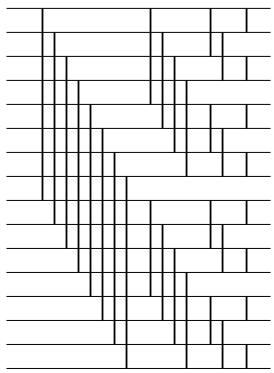

Figure 3.9: A diagram of a 16 input bitonic merging network, using a style that is standard in the literature. Note that in each stage containing 8 min/max or comparator components, all 8 operate on independent parts of the input and so can proceed in parallel.

use of the fact that flipping the bit i of an index (using the function `flipBit`) gives the index of the element that will be combined with it using the functions`f` and `g`. The decision about whether to apply `f` or `g` is made by looking at the value of bit i. As we shall see, the use of the Obsidian `ifThenElse` produces conditionals in the resulting CUDA.

```
lowBit :: Int -> UWordE -> Exp Bool
lowBit i ix = (ix .&. bit i) ==* 0

flipBit :: Bits a => Int -> a -> a
flipBit = flip complementBit

ilv1 :: Choice a =>
        Int -> (b -> b-> a) -> (b -> b -> a) ->
        Array b -> Array a
ilv1 i f g arr = Array ixf (len arr)
  where
    ixf ix = let l = arr ! ix
                 r = arr ! newix
                 newix = flipBit i ix
             in (ifThenElse (lowBit i ix) (f l r) (g l r))
```

Expressing `ilv1` using bit-flipping may seem strange, but it has the advantage that it actually applies the desired pattern of computation repeatedly over larger input arrays. Now, for 2^n inputs, a Haskell list containing the n calls of this interleave combinator are built:

```
bmerge :: Int -> [Array IntE -> Array IntE]
bmerge n = [istage (n-i) | i <- [1..n]]
  where istage i = ilv1 i min max
```

Finally, the `compose` function makes each element of the list into a kernel (using `map pure`) and places a `sync` between each kernel (using `composeS`).

```
compose :: (Scalar a) =>
           [Array (Exp a) -> Array (Exp a)]
           -> Array (Exp a) -> Kernel (Array (Exp a))
compose = composeS . map pure

runm k = putStrLn$ CUDA.genKernel "bitonicMerge"
         (compose (bmerge k)) (namedArray "inp" (2^k))
```

Note that `bmerge k` works on inputs of length 2^{k+j}, for $j > 0$, applying the merger to sub-sequences of length 2^k. The CUDA code for `bmerge 4` on 16 inputs (with some newlines inserted) is

```
*Main> runm 4
__global__ void bitonicMerge(int *input0,int *result0){
  unsigned int tid = threadIdx.x;
  unsigned int bid = blockIdx.x;
  extern __shared__ unsigned char sbase[];
  (( int *)sbase)[tid]
     = ((tid&8)==0)
      ? min(input0[((bid*16)+tid)],input0[((bid*16)+(tid^8))])
      : max(input0[((bid*16)+tid)],input0[((bid*16)+(tid^8))]);
  __syncthreads();
```

```
  (( int *)(sbase + 64))[tid]
     = ((tid&4)==0)
      ? min((( int *)sbase)[tid],(( int *)sbase)[(tid^4)])
      : max((( int *)sbase)[tid],(( int *)sbase)[(tid^4)]);
  __syncthreads();
  (( int *)sbase)[tid]
     = ((tid&2)==0)
      ? min((( int *)(sbase+64))[tid],(( int *)(sbase+64))[(tid^2)])
      : max((( int *)(sbase+64))[tid],(( int *)(sbase+64))[(tid^2)]);
  __syncthreads();
  (( int *)(sbase + 64))[tid]
     = ((tid&1)==0)
      ? min((( int *)sbase)[tid],(( int *)sbase)[(tid^1)])
      : max((( int *)sbase)[tid],(( int *)sbase)[(tid^1)]);
  __syncthreads();
  result0[((bid*16)+tid)] = (( int *)(sbase+64))[tid];
```

Modifying the bitonic merger

The bitonic merger for which we have just generated a kernel is known to sort so-called bitonic sequences, which include sequences whose first half is sorted in one direction and whose second half is sorted in the other direction. This fact can be used to build the well-known bitonic sorting network. However, a GPU implementation typically needs to check, for each comparator, whether or not it should sort upwards or downwards, see for instance the simple CUDA implementation shown in Appendix A. We choose here to modify the merger so that it sorts two concatenated sequences that are sorted in the *same* direction. We do this by using a well-known trick, reversing half of the input to the merger. It turns out that we can also reverse the same half of the output of the first stage of the network, without affecting overall behaviour. The resulting network, `tmerge`, shown in Figure 3.10, encourages us to develop a new combinator to describe the characteristic V-shaped pattern that results in the first stage. The combinator is modelled on `ilv1`. The only difference is that the "partner" of an index is found not by flipping bit i, but by flipping bits 0 to i, using function `flipLSBsTo`. The implementation of `vee1` is got from that for `ilv1` by replacing the call of `flipBit` by one of `flipLSBsTo` (and we could also have chosen to make a more generic function that is parameterised on this *partner* function).

```
flipLSBsTo :: Int -> UWordE -> UWordE
flipLSBsTo i = (`xor` (oneBits (i+1)))

vee1 :: Choice a =>
        Int -> (b -> b-> a) -> (b -> b -> a) ->
        Array b -> Array a
vee1 i f g arr = Array ixf (len arr)
  where
    ixf ix = let l = arr ! ix
                 r = arr ! newix
                 newix = flipLSBsTo i ix
             in (ifThenElse (lowBit i ix) (f l r) (g l r))


tmerge :: Int -> [Array IntE -> Array IntE]
tmerge n = vstage (n-1): [istage (n-i) | i <- [2..n]]
```

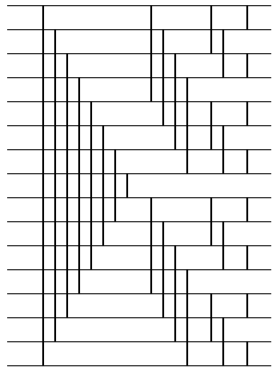

Figure 3.10: 16 input merging network, *tmerge*. The first stage is made with a new combinator that we call vee, while the remaining stages are as in the bitonic merger. This network sorts an input that consists of two half-sized sorted sequences, giving the opportunity to build a tree-shaped sorting network.

```
  where
    vstage i = vee1 i min max
    istage i = ilv1 i min max
```

Now that we have a merger that sorts sub-sequences containing two concatenated sorted sequences, it is easy to make a tree of them. The list of kernels to be composed now becomes

```
tsort1 :: Int -> [Array IntE -> Array IntE]
tsort1 n = concat [tmerge i | i <- [1..n]]
```

The resulting sorting network is shown in Figure 3.11.
The following call writes the resulting CUDA to file `tsort1.cu` and this is the first *generated* CUDA kernel whose performance is measured in section 3.3.3.

```
runs1 k
  = writeFile "tsort1.cu" $ CUDA.genKernel "tsort1"
      (compose (tsort1 k)) (namedArray "inp" (2^k))
```

The generated code uses one thread per array element (just as the `bitonicMerge` kernel shown above did). The next step is to move to using Push as well as Pull arays, so as to be able to generate more efficient code from essentially the same sorter construction.

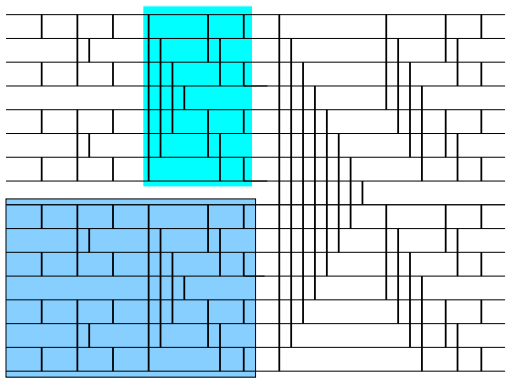

Figure 3.11: A 16-input sorter made from a tree of *tmerge* mergers. 8 2-input mergers feed 4 4-input mergers, followed by 2 8-input and one 16-input merger. A sorter on 8 inputs is shaded, as is a merger on 8 inputs, above it.

New combinator implementations using Push arrays

It is in combining the results of the `f` and `g` functions that we run into difficulty using just Pull arrays. Earlier, we saw how to use Push arrays to implement `concP`, which concatenates two arrays. Here, we use exactly the same approach to make a new version of the *interleave* combinator. The results of applying the `fs` and `gs` are combined into a Push array, in the right order.

```
ixMap :: (UWordE -> UWordE) -> ArrayP a -> ArrayP a
ixMap f (ArrayP p n) = ArrayP (ixMap' f p) n

ixMap' :: (UWordE -> UWordE)
          -> P (UWordE, a)
          -> P (UWordE, a)
ixMap' f p = \g -> p (\(i,a) -> g (f i,a))

insertZero :: Int -> UWordE -> UWordE
insertZero 0 a = a `shiftL` 1
insertZero i a
  = a + (a .&. fromIntegral (complement (oneBits i :: Word32)))

ilv2 :: Choice b =>
        Int -> (a -> a -> b) -> (a -> a -> b) ->
        Array a -> ArrayP b
ilv2 i f g (Array ixf n)
   = ArrayP (\k -> app a5 k *>* app a6 k) n
  where
    n2 = n `div` 2
    a1 = Array (ixf . left) (n-n2)
    a2 = Array (ixf . right) n2
```

```
    a3 = zipWith f a1 a2
    a4 = zipWith g a1 a2
    a5 = ixMap left (push a3)
    a6 = ixMap right (push a4)
    left = insertZero i
    right = flipBit i  . left
    app (ArrayP f _) a = f a
```

This new combinator can now replace `ilv1` in the bitonic merger, giving a kernel that runs considerably faster. We will use that kernel to build a large sorter later.

The implemenation of `vee2` is almost identical to that of `ilv2`, with `flipBit i` replaced by `flipLSBsTo` as before (so that, again, one would in fact make a more generic function for building such combinators). Now, we just need to replace the `ilv1` and `vee1` combinators in the tree sorter with `ilv2` and `vee2` , to get a verrsion that uses half as many threads:

```
tmerge2 :: Int -> [Array IntE -> ArrayP IntE]
tmerge2 n = vstage (n-1) : [ istage (n-i) | i <- [2..n]]
  where
    vstage i = vee2 i min max
    istage i = ilv2 i min max


tsort2 :: Int -> [Array IntE -> ArrayP IntE]
tsort2 n = concat [tmerge2 i | i <- [1..n]]
```

As we shall see in section 3.3.3, the resulting code is significantly faster. This is because it uses one thread per two array elements, and the code no longer contains any conditionals.

In order to go faster still, we resort to building a different sorting network, of exactly the same size as the bitonic sorter, but based instead on the balanced period merger of Dowd et al [DPLR89]. This involves the introduction of one new combinator that can be seen as a mixture of the `ilv` and `vee` combinators already introduced.

A sorter built from the balanced periodic merger

First, we note that the balanced periodic merger contains multiple uses of the now familiar vee-shaped pattern, see Figure 3.12.

```
bpmerge2 :: Int -> [Array IntE -> ArrayP IntE]
bpmerge2 n = [vstage (n-i) | i <- [1..n]]
  where vstage i = vee2 i min max
```

Now Dowd et al proved that the balanced periodic merger sorts two *interleaved* sorted sequences. So, taking an iterative view of the resulting sorter, we want to build a tree of mergers as before, but the smaller mergers should be interleaved, rather than operating on adjacent sub-sequences. There should be one merger on the right hand

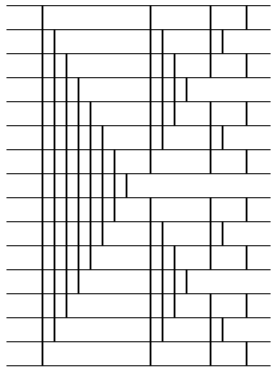

Figure 3.12: 16 input periodic balanced merging network

end of the network; left of that, there should be two mergers that operate on the odd and even elements, and each of them should in turn be fed by two interleaved mergers, and so. The sorter is illustrated, for 16 inputs in Figure 3.13. Just to the left of the final balanced merger, one of the two interleaved mergers is shown using dotted lines. It operates on a completely different set of inputs from the other 8-input merger in the same part of the tree.

The most straightforward way to give an iterative description of this sorter is to introduce a new combinator that is a combination of `ilv` and `vee`. The only thing that we need to change is the *partner* function. This time we will flip not the least significant bits from position 0 to position i but from position i to $i + j$.

```
-- flip bits from position i to position i+j inclusive
flipBitsFrom :: Bits a => Int -> Int -> a -> a
flipBitsFrom i j a = a `xor` (fromIntegral mask)
  where
    mask = (oneBits (j + 1):: Word32) `shiftL` i

ilvVee1 :: Choice a =>
           Int -> Int ->
           (b -> b-> a) -> (b -> b -> a) ->
           Array b -> Array a
ilvVee1 i j f g arr = Array ixf (len arr)
  where
    ixf ix = let l = arr ! ix
                 r = arr ! newix
                 newix = flipBitsFrom i j ix
             in (ifThenElse (lowBit (i+j) ix) (f l r) (g l r))

ilvVee2 :: Choice b => Int -> Int ->
```

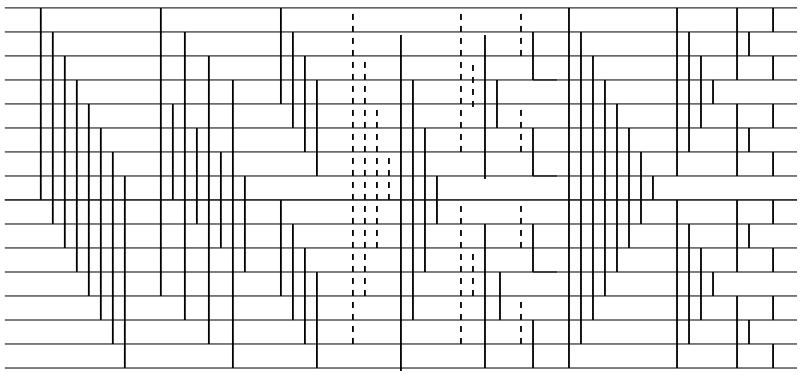

Figure 3.13: A sorter based on the idea that the periodic balanced merger network sorts two interleaved sorted sequences. It consists of two half-sized sorters, one working on the odd elements of the input and one on the even, followed by the balanced merger. The diagram indicates using dotted lines the balanced merger that is the final (rightmost) part of one of the half-size sorters.

```
          (a -> a -> b) -> (a -> a -> b) ->
          Array a -> ArrayP b
ilvVee2 i j f g (Array ixf n)
  = ArrayP (\k -> app a5 k *>* app a6 k) n
  where
    n2 = n `div` 2
    a1 = Array (ixf . left) (n-n2)
    a2 = Array (ixf . right) n2
    a3 = zipWith f a1 a2
    a4 = zipWith g a1 a2
    a5 = ixMap left (push a3)
    a6 = ixMap right (push a4)
    left = insertZero (i+j)
    right = flipBitsFrom i j . left
    app (ArrayP f _) a = f a
```

For both variants of the combinator, we simply add to the `ilv` definitions a new `Int` parameter, `j`, and replace `flipBit i` by `flipBitsFrom i j`. We also insert the zero bit (when calculating the left index) at position `i + j` rather than just at position `i`. `ilvVee` is a generalisation of both `ilv` and `vee`. `ilvVee i 0` has the same behaviour as `ilv i`, and `ilvVee 0 (j-1)` is the same as `vee j`. The `i` parameter controls the degree of interleaving, and the `j` parameter controls the size of the vee-shaped blocks.

```
unsigned int arrayLength = 1 << LOG_L_SIZE;
unsigned int diff = LOG_L_SIZE - LOG_S_SIZE;
unsigned int blocks = arrayLength / S_SIZE;
unsigned int threads = S_SIZE / 2;

sortSmall<<<blocks, threads,4096>>>(din,din);

for(int i = 0 ; i < diff ; i += 1){
  vSwap<<<blocks/2,threads*2,0>>>(din,din,(1<<i)*S_SIZE);

  for(int j = i-1; j >= 0; j -= 1)
    iSwap<<<blocks/2,threads*2,0>>>(din,din,(1<<j)*S_SIZE);

  bmergeSmall<<<blocks,threads,4096>>>(din,din);}
```

Figure 3.14: CUDA code for our large sorter. `sortSmall` and `bmergeSmall` are replaced by Obsidian-generated kernels in the experiments. `vSwap` and `iSwap` are handwritten CUDA kernels that perform one column of compare and swap operations in the vee and ilv shapes respectively, and are parameterised on the stride. (Our generated kernels have fixed input and output size.)

For 16 inputs, the parameters to `ilvVee` that describe the periodic merger on the right of the construction are $i = 0$ (for no interleaving) paired with 3, 2, 1 and 0 for the decreasing size of the vee-shaped blocks. Next, to the left, the mergers are interleaved ($i = 1$) and there are three stages with vee-shaped blocks of decreasing size ($j = 2, 1, 0$), see Figure 3.13. The following code gives an iterative description of the construction for 2^n inputs:

```
vsort :: Int -> Array IntE -> Kernel (Array IntE)
vsort n = composeS . map pure $ [istage (n-i) (i-j)
                                 | i <- [1..n], j <- [1..i]]
  where istage i j = ilvVee2 i j min max
```

The resulting generated code uses one thread per 2 indices.

Measuring performance of the generated kernels

We have measured the performance of generated 512-input sorting and merging kernels by plugging them into a larger sorter written in CUDA. The sorter has exactly the structure shown in Figure 3.11. Figure 3.15 shows the location of smaller sorters and mergers, and of vSwap and iSwap kernels for 16 inputs. Larger sorters simply have more columns of mergers, each preceded by a vSwap and a number of iSwaps. The overall structure of the resulting CUDA code is shown in Figure 3.14. Because `iSwap` is used repeatedly, we also wrote kernels corresponding to compositions of two and three of them (avoiding memory accesses between the columns). That is, we replaced the loop containing `iSwap` above by

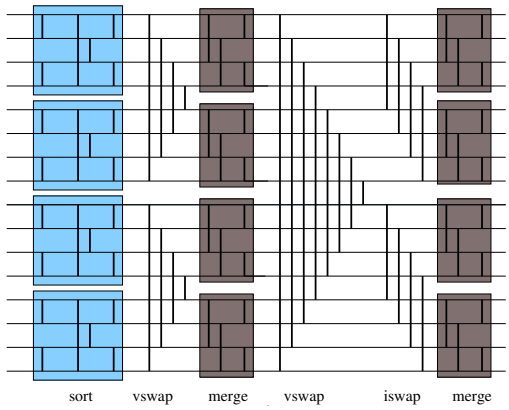

Figure 3.15: This diagram shows the tree-shaped sorting network that we saw earlier, but with 4 input sorting and merging kernels indicated. This is the structure of the network that we have used to implement large sorters, in which the small kernels having $2^9 = 512$ inputs in all cases. For 2^{24} inputs overall, the resulting sorter has one column of small sorters and $24 - 9 = 15$ of small mergers.

```
for(int j = i-1; j >= 0; j -= 3){
  if (j==0)
    iSwap<<<blocks/2,threads*2,0>>>(din,din,(1<<j)*S_SIZE);
    else
    {if (j==1)
       iSwap2<<<blocks/4,threads*2,0>>>(din,din,(1<<j)*S_SIZE);
       else
       iSwap3<<<blocks/8,threads*2,0>>>(din,din,(1<<j)*S_SIZE);}}
```

The Obsidian implementation and all code for examples in the paper are available at `http://www.cse.chalmers.se/~joels/expressive.html`.

Table 3.1 shows the performance figures for the large sorter for 5 different $512-$input small sorter kernels. `tsort1` is defined above, and is a variant of bitonic sort that does not require `if` statements to determine the direction of sorting of pairs, as all comparison operations place the minimum at the same index as the lower input. `tsort2` is the same construction, but built with the combinators `ilv2` and `vee2` that result in the use of one thread per two indices. `vsort`, the fastest kernel, uses a generalisation of those two combinators, and again uses one thread per two indices. `vsort1` is the same construction as `vsort`, but with `ilvVee2` replaced by `ilvVee1`, and so using one thread per index. As a reference small kernel implementation, we include a simple hand-coded bitonic sort that uses one thread per index (see code in Appendix). None of the kernels has been subject to optimisations

k	20	21	22	23	24	24(CPU)
bitonic(CUDA)(512)	5	12	25	54	117	2741
tsort1(512)	4	10	22	47	102	2742
tsort2(256)	4	9	23	45	98	2741
vsort1(512)	4	10	21	47	102	2747
vsort(256)	4	9	20	44	96	2783

Table 3.1: Sorting time (ms) for 2^k inputs, $20 \leq k \leq 24$. The GPU used is an NVIDIA GTX480. Each line shows the result using a particular small 512 input sort kernel with the indicated number of threads as `sortSmall` in the CUDA code above. The small merge kernel used is our generated `bmerge2`, with two indices per thread and 512 inputs. Memory transfer time (GPU-CPU) is not shown. In the 2^{24} input case using `vsort`, the total sorting time, including memory transfers was 141 ms. The rightmost column shows the time taken for the C quicksort function `qsort`, compiled with `gcc -O3`, to sort the same inputs as those sorted on the GPU on an i7-920 2.8GHz CPU.

related to warp size.

We also recorded the GPU time spent in the calls to the small sorters alone (using the NVIDIA CUDA Visual profiler), see Table 3.2. The `tsort1` and `vsort1` kernels, which are built using only pull arrays, (and which have one index per thread in the generated code) are noticeably slower than those that have two indices per thread. The generation of the latter kernels is made possible by the introduction of push arrays.

Although none of our generated kernels is optimised (for example with respect to warp size), their performance is, nevertheless, very good. We are working on automated warp size-related optimisations. It would also be interesting to explore the fusion of adjacent columns of comparators in the small kernels; omitting a `sync` would cause this fusion to happen but we also need to modify the threading behaviour (doubling the number of indices per thread).

3.3.4 Discussion

Push arrays form a new approach to array representation in DSELs. We do not know of similar approaches in the literature, despite the fact that the notions of demand and data flow may feel familiar to the reader who considers Pull and Push arrays. The addition of Push arrays to Obsidian seems highly beneficial. With this new feature, the user gains finer grained control over the code generated and the resulting CUDA

	time	percent
bitonic(CUDA)	30203	32.07
tsort1	15823	19.72
vsort1	14955	18.76
tsort2	11562	15.37
vsort	9228	12.67
bitonic (NVIDIA SDK)	23815	6.55

Table 3.2: time: GPU time (in μs) spent in calls to small sorter kernels in the initial phase of the sorter for 2^{24} inputs. percent: percentage of total GPU time spent in the small sort kernel. The final line shows the numbers for the bitonic code (for large sorts) that is distributed with the the NVIDIA SDK. Its structure also starts with a phase of small sorting kernels. It takes 274 ms to sort 2^{24} key-value pairs on an NVIDIA GTX480 GPU. It could be sped up by using our `vsort` kernel, and also by (hand) fusing single "column" kernels as we did with `iSwap`, although the need to calculate directions of comparators in the bitonic network would complicate this. Our CUDA code is simpler because all comparators point in the same direction in the sorter construction.

kernels perform considerably better than before. This was illustrated in the series of sorters explored in section 3.3.3. The performance of `vsort` is sufficiently good that it can be used as a first phase in a larger sorter (written in CUDA) that can sort 16M elements in 96 ms, while an i7-920 CPU takes around 2740 ms. Further speed improvements look possible, both in the coordination code and in the kernels. An obvious next step would be to investigate the generation of the `iSwap` and `vSwap` kernels from Obsidian. (This is not currently possible because of assumptions that we made about the interfaces to kernels and about how *thread ids* are used. We will look into ways to relax our assumptions.)

The series of kernels also illustrates how the use of combinators brings a form of reuse, and makes design exploration easier. Our experience of using similar combinators in the Lava hardware description language [CSS01] was that a relatively small set of combinators went a long way. So, although we introduced three combinators here, `ilv`, `vee` and `ilvVee`, which includes the other two, we do not believe that every new kernel development exercise would demand a completely new set of combinators. We expect to provide the user with a well-documented set of combinators, so that users can get access to this style of programming without having to develop their own combinators, and without having to think too much about bit-hacking. The bit-manipulation approach chosen to define our combinators automatically created

functions that apply to sub-sequences of the input that are of an appropriate length.

In this paper, we made combinators for the special case of two input, two output operations (built from two two-input funtions that we typically called `f` and `g`). This approach should be generalised to deal with blocks that have 2^k inputs and outputs. Also, we made a compound combinator from `ilv` and `vee`, but generalising to more than two input components would allow for composing combinators, and indeed for recursive descriptions that could be unrolled. Then, ignoring `syncs`, a recursive description of `vsort` could be something like

```
vsortR 0 = id
vsortR n = bpmergeR n . ilv2 1 (vsortR (n-1))
```

It would then be necessary to optimise the code generated from multiple applications of `ilv2 1`, for example, whereas here we have forced the user to figure out both the unrolling and the combinations. Moving to more general combinators would also give the opportunity to provide predefined combinators that capture more of the commonly used threading patterns (for instance k indices per thread rather than the 1 and 2 shown here).

The integration of Push arrays into Obsidian raises some new questions. Previously, there was a direct correspondence between the length of an array and the number of threads used to compute it, which allowed the user to write an initial program without worrying about threads at all, and then to tweak the Obsidian program if he was not satisfied with the threading behaviour of the resulting kernel. Now, as can be seen in the `catArrayPs` example and in the sorters, this correspondence can be broken using Push arrays. The `catArrayPs` example and two of the sorters use half as many threads as the number of elements. For users who are very concerned about the speed of the generated kernels, getting this control through using Push arrays in a particular pattern is clearly a good thing. But adding a second, different way to control thread use in the generated code certainly complicates matters, and further case studies are needed to confirm that the complication pays off.

The addition of Push arrays also adds the possibility to include potentially unsafe operations in Obsidian, for example by writing multiple array elements to the same index, or by discarding elements. This new expressiveness will have to be carefully controlled. On the positive side, it offers the possibility to encode functions like `filter` from Haskell that are simply not expressible using only Pull arrays. Being able to implement `filter` would make programming kernels in obsidian feel much more like programming in Haskell – a welcome loosening of the strait-jacket. Once that is done, it will be time to develop a very simple coordination language to allow programming of entire GPU applications that make use of the kind of small kernel building blocks developed here.

3.3.5 Appendix

```
__device__ inline void swap(int & a, int & b)
{
   int tmp = a;
   a = b;
   b = tmp;
}

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

   const unsigned int tid = threadIdx.x;
   const unsigned int bid = blockIdx.x;

   // Copy input to shared mem.
   shared[tid] = values[(bid*NUM) + tid];

   __syncthreads();

   // Parallel bitonic sort.
   for (unsigned int k = 2; k <= NUM; k *= 2)
   {
     // bitonic merge
     for (unsigned int j = k / 2; j>0; j /= 2)
       {
           unsigned 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.
   results[(bid*NUM) + tid] = shared[tid];
}
```

Acknowledgments

This research has been funded by the Swedish Foundation for Strategic Research (which funds the RAW FP Project) and by the Swedish Research Council.

Bibliography

[ACS+11] Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. The Design and Implementation of Feldspar an Embedded Language for Digital Signal Processing. In *Proceedings of the 22nd international conference on Implementation*

and application of functional languages, IFL'10, pages 121–136, Berlin, Heidelberg, 2011. Springer-Verlag.

[Bat68] K. E. Batcher. Sorting networks and their applications. In *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, New York, NY, USA, 1968. ACM.

[BOA09] Markus Billeter, Ola Olsson, and Ulf Assarsson. Efficient stream compaction on wide SIMD many-core architectures. In *Proc. Conf. on High Performance Graphics*, HPG '09, pages 159–166. ACM, 2009.

[CKL+11] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell Array Codes with Multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, DAMP '11, pages 3–14, New York, NY, USA, 2011. ACM.

[Cla99] Koen Claessen. A poor man's concurrency monad. *J. Funct. Program.*, 9(3):313–323, 1999.

[CSS01] Koen Claessen, Mary Sheeran, and Satnam Singh. The Design and Verification of a Sorter Core. In *Proc. Int. Conf. on Correct Hardware Design and Verification Methods (CHARME)*, volume 2144 of *Springer LNCS*, pages 355–369, 2001.

[DPLR89] Martin Dowd, Yehoshua Perl, and Michael Saks Larry Rudolph. The Periodic Balanced Sorting Network. *Journal of the ACM*, 36:738–757, 1989.

[EFdM03] Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(2), 2003.

[Ell03] Conal Elliott. Functional Images. In *The Fun of Programming*, "Cornerstones of Computing" series. Palgrave, March 2003.

[JS11] Joel Svensson. Obsidian: GPU Kernel Programming in Haskell. Technical Report 77L, Computer Science and Enginering, Chalmers University of Technology, Gothenburg, 2011. Thesis for the degree of Licentiate of Philosophy.

[KCL+10] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel

arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 261–272, New York, NY, USA, 2010. ACM.

[Lar11] Bradford Larsen. Simple optimizations for an applicative array language for graphics processors. In *Proc. sixth workshop on Declarative Aspects of Multicore Programming*, DAMP '11, pages 25–34. ACM, 2011.

[MM10] Geoffrey Mainland and Greg Morrisett. Nikola: Embedding Compiled GPU Functions in Haskell. In *Proceedings of the third ACM Haskell symposium*, pages 67–78. ACM, 2010.

[She89] Mary Sheeran. Describing Butterfly Networks in Ruby. In *Functional Programming*, pages 182–205. Springer Workshops in Computing, 1989.

3.4 Paper E: Counting and Occurrence Sort for GPUs using an Embedded Language

Josef Svenningsson, Bo Joel Svensson, Mary Sheeran

Abstract

This paper investigates two sorting algorithms: counting sort and a variation, occurrence sort, which also removes duplicate elements, and examines their suitability for running on the GPU. The duplicate removing variation turns out to have a natural functional, data-parallel implementation which makes it particularly interesting for GPUs.

The algorithms are implemented in Obsidian, a high-level domain specific language for GPU programming.

Measurements show that our implementations in many cases outperform the sorting algorithm provided by the library Thrust. Furthermore, occurrence sort is another factor of two faster than ordinary counting sort. We conclude that counting sort is an important contender when considering sorting algorithms for the GPU, and that occurrence sort is highly preferable when applicable. We also show that Obsidian can produce very competitive code.

3.4.1 Introduction

Sorting is an ever important, ever fascinating field of computer science. The introduction of GPUs has introduced new challenges in designing fast sorting algorithms.

This paper focuses on counting sort together with a variation which removes duplicate elements. We call this variation *occurrence sort*. Counting sort is a non-comparing sort suitable for parallel implementation. The occurrence sort variation presented here is interesting because it seems to be a particularly good fit for executing on the GPU. It has a very natural functional, data-parallel implementation. We believe we are the first to study this variation in the literature.

These algorithms are explored in Obsidian [JS11, CSS12], a domain specific language targeting GPU programming. The goal of Obsidian is to strike a balance between high-level constructs and low-level control. We want to provide the programmer with a set of tools for low-level experimentation with the details that influence performance when implementing GPU kernels. The counting sort case study shows that Obsidian generates competitive kernels with relatively little programmer effort. That Obsidian is an embedded language allows us to rapidly experiment with

the addition of features and with varying programming idioms. The version used in this case study adds global arrays and atomic instructions to Obsidian, see section 3.4.4.

The contributions of this paper are:

- We provide measurements (section 3.4.7) showing that counting sort is a competitive algorithm for sorting keys on the GPU, outperforming the sorting implementation in the library Thrust[NVId] in many cases.
- Occurrence sort is shown to be particularly suitable for implementing on the GPU (section 3.4.6) and performs well (section 3.4.7).
- The Obsidian implementation of the two sorting algorithms is detailed along with the generated CUDA (sections 3.4.4 and 3.4.5).

Related work

Sorting has applications in the computer graphics field [SA08]. Example instances of sorting and duplicate element removal in computer graphics can be identified in references [OBA12] and [Kar12]. Another example of where sorting and the removal of duplicates have applications is in databases [KGJ+11].

The paper [KVGH11] implements counting sort in CUDA, and also optimisations that overlap computations with transferring data to and from the GPU. Our work differs by being implemented in a high-level embedded language and also by implementation of the occurrence sort variant of counting sort. We have not been concerned with trying to overlap computations and data transfer, but have instead focused solely on computations within the GPU.

Obsidian is a high-level embedded language for GPU programming. Other such approaches include Accelerate [CKL+11] and Nikola [MM10]. Accelerate and Nikola both take an even higher level approach compared to Obsidian and abstract more from GPU details. Another example of providing a high-level interface to programming the GPU is the C++ library Thrust [NVId].

3.4.2 Counting Sort

Counting sort, like radix sort, relies on array indexing rather than comparison to order elements. The key idea of counting sort is to count the number of occurrences of each element, and then perform a prefix sum over all the counts. The prefix sum generates an array which, for each element in the original array, will point to its position in the final sorted array. In order for this to work, counting sort needs a lower and upper

```
countingSort :: (Int,Int)
                -> Array Int Int
                -> Array Int Int
countingSort range input =
  reconstruct $
  scanlArray (+) 0 $
  histogram range input

histogram :: (Int,Int)
             -> Array Int Int
             -> Array Int Int
histogram range =
  accumArray (\i _ -> i+1) 0 range . elems . fmap dup

reconstruct :: Array Int Int -> Array Int Int
reconstruct arr =
  array (0,arr!l-1)
    [ (a,i)
    | (i,e) <- init (assocs arr)
    , a <- [ e .. arr!(i+1) - 1] ]
  where (_,l) = bounds arr

dup a = (a,a)
```

Figure 3.16: A Haskell specification of Counting Sort. The function `scanlArray` is not standard but works much like `scanl` for lists.

bound on the values of the elements in the input array. Counting sort is typically explained in terms of sorting integers and we will do the same here.

The array `{5,2,5,7,1}` will be used as a running example. In addition, the range `(1,10)` will be the approximation of the range of the keys in the input. A diagrammatic illustration of our running example is presented in figure 3.17.

Counting sort consists of three steps: histogram creation, prefix sum and reconstruction.

The histogram creation step computes a new array which records how many times an integer occurs in the input array, i.e. the new array is a histogram of the input array. The size and indices of the histogram array are determined by the range which approximates the lower and upper bounds of the keys.

Computing the histogram of `{5,2,5,7,1}` with the range `(1,10)`, results in `{1,1,0,0,2,0,1,0,0,0}`. Indexing in this array starts at `1` because that's the

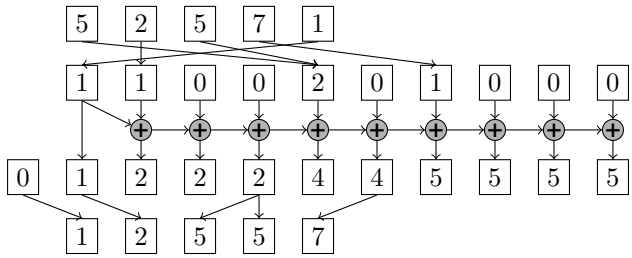

Figure 3.17: An example run of counting sort in diagrammatic form

lower bound given in the range. Each element in the histogram indicates how many times the corresponding value occurs in the input array. For example, `5` occurs twice and `8` occurs zero times.

The second step of the algorithm computes the prefix sum (or inclusive scan with plus) of the histogram array, and prepends a zero. For input array `{5,2,5,7,1}` and range `(1,10)`, the histogram array is `{1,1,0,0,2,0,1,0,0,0}` and its prefix sum is `{0,1,2,2,2,4,4,5,5,5,5}`.

The array computed by the prefix sum is called the position array. It indicates where in the final sorted array each value is placed. The indices of the position array correspond to values in the final array and the elements correspond to positions in the final array. The reconstruct step takes care of distributing the values into the final sorted array. For each index in the position array, the difference between the element at that index and the index plus one is computed. This difference indicates how many times the value occurs in the final array.

The position array in our running example is `{0,1,2,2,2,4,4,5,5,5,5}`. Indexing starts at `1` just as with the histogram. As an example, element `1` occurs once in the final array because the difference between the element at position `1` and `2` in the position array is precisely one. Also, `1` is placed on index `0`. Furthermore, element `5` occurs twice in the final array (the difference between `2` and `4` in the position array), on indices `2` and `3`. Continuing this process with all elements will result in the following final array: `{1,2,5,5,7}`.

A functional description of counting sort can be found in figure 3.16. The main function is `countingSort` which takes two arguments: a range with a safe approximation of the lower and upper bounds of the values in the input array, and the input array itself. The histogram step is implemented by the function `histogram`, the prefix sum with `scanlArray` and the reconstruct step with the function `reconstruct`.

```
countingSort :: (Int,Int)
                -> Array Int Int
                -> Array Int Int
countingSort range input =
  reconstruct $
  scanlArray (+) 0 $
  occurs range input

occurs :: (Int,Int)
          -> Array Int Int
          -> Array Int Int
occurs range =
  accumArray (\_ _ -> 1) 0 range . elems . fmap dup
```

Figure 3.18: A Haskell specification of occurrence sort

Occurrence sort: Removing duplicates

There is an interesting variation of counting sort which removes duplicate elements. It seems to be folklore; it is mentioned on Wikipedia [Wik]. However, we haven't found any description of it in the literature. It is particularly interesting because it allows for an efficient data-parallel implementation.

Figure 3.18 shows the changes we make to counting sort in order to obtain occurrence sort. Only the `histogram` function needs changing so that instead of computing an actual histogram which counts the number of occurrences of an element, it only records if an element occurs or not with a `1` or a `0`. Since the function no longer computes a histogram it has been renamed to `occurs`.

The prefix sum and the reconstruction step can be kept intact. They will still compute the correct result.

Compared to the full counting sort, occurrence sort has some interesting properties which make it more suited for parallel execution. In particular, the histogram construction phase and the reconstruction phase allow for less synchronisation.

When constructing the histogram in the counting sort algorithm, an index has to be incremented each time the corresponding value is found in the input array. When parallelising this operation, it is important that the increments are done atomically, which incurs a cost. The occurs phase presented above doesn't need to employ any special mechanism to ensure atomicity, since it writes a single word with the value 1 to memory.

Given the array `{5,2,5,7,1}` as input and `(1,10)` as range, the occurs step will now compute `{1,1,0,0,1,0,1,0,0,0}`. The prefix sum will in turn pro-

duce the following position array: `{0,1,2,2,2,3,3,4,4,4,4}`. The reconstruct step computes `{1,2,5,7}`. In particular, there is only one occurrence of `5` in the final array.

3.4.3 GPUs and CUDA

The GPUs we target in this paper and with Obsidian are NVIDIA GPUs which support CUDA. Here we provide some background information on GPUs that we hope will aid in the understanding of example programs throughout this paper.

GPUs supporting CUDA are based on a scalable design. The GPU consists of a number of so-called multiprocessors (MPs) each consisting of a number of processing elements (PEs); sometimes these are referred to as cores or as stream processors. An MP also contains a local shared memory through which threads running on the PEs can communicate. The key that makes the architecture scalable is that a GPU consists of one or more of these MPs. This also influences the programming model significantly; since threads can only communicate in a synchronised manner using the local memory (or in a very limited way using atomic operations) the programmer must partition the computation in such a way that its communication patterns follow this architectural constraint. The GPU also has access to a larger memory called the *global* or *device* memory. The sizes of these memories and the number of MPs and PEs per MP vary slightly between generations of GPUs. For example, the GPU used in the benchmarks (section 3.4.7) has a total of 1344 single-precision floating point CUDA cores and has 2 GB global memory. The very latest GPU architectures from NVIDIA slightly digress from what is outlined here but mostly still adhere to the same programming model. For exact details refer to [NVIc].

The programming model that CUDA exposes is called single program multiple threads (SPMT) and is a slight variation of the SPMD concept. This programming model reflects the GPU architecture. In CUDA, the programmer writes a single program that is executed by many threads. Because of the scalable architecture, with its one or more MP, these threads are grouped into *blocks* (now up to 1024 threads per block). A block of threads is executed on an MP; thus only threads within that block can communicate using the shared memory. The same CUDA program can be executed on any GPU along the scale, from the smallest with only one MP to the largest. The only difference is in the number of blocks of threads that can be executed in parallel. This constraint implies that all blocks must be free of any sequential dependencies. The collection of all blocks is called a *Grid*.

A CUDA program has two parts; there is a coordination program that runs on the CPU and there are kernels that execute on the GPU MPs. The program describing a kernel is parameterised over a blockId and a threadId so that decisions can be

```
__global__ void addv(float *i0,
                     float *i1,
                     float *r){
  unsigned int ix =
     blockIdx.x * blockDim.x + threadIdx.x;

  extern __shared__ float sm[];
  sm[threadIdx.x] = i0[ix] + i1[ix];
  r[ix] = sm[threadIdx.x];
}
```

Figure 3.19: The code illustrates elementwise addition of vectors. Shared memory (the sm array) is used to allow the `addv` program to be run with the result `r` pointing to the same memory area as either of the inputs.

made from that information during execution. A typical CUDA program starts out by loading data into local memory using some function of blockId and threadId. It then computes on the local memory and uses a syncthreads primitive when exchanging values between threads. When the local computation is done the final results are written back into global memory again using some function of blockId and threadId.

For more CUDA and GPU specifics see references [NVIc, NVIb].

3.4.4 Obsidian

Obsidian is a programming language for expressing general purpose GPU kernels, compute kernels, in a high level and functional style. Obsidian is embedded in Haskell. Specifically, Obsidian is a library of functions for generating and combining abstract syntax trees. By using features of Haskell such as overloading and higher-order-functions, the library/language border is blurred and we call the result an embedded language. From the abstract syntax trees generated while running an Obsidian program, NVIDIA CUDA code is generated.

A brief history of Obsidian

In the Obsidian project we experiment with combinators and language features for GPU kernel implementation. We seek to strike a balance between low-level control and useful high-level abstractions. Over time, this has led to a number of different versions of Obsidian. In reference [JS11], two versions of Obsidian differing by having a *monadic* or a limited *arrow* like interface are described. We have settled on using the monadic style.

```
#define BLOCK_SIZE 32
#define BLOCKS 4
#define N (BLOCKS * BLOCK_SIZE)

int main(int argc, char **argv){
  float *v1, *v2, *r;
  float *dv1, *dv2, *dr;

  v1 = (float*)malloc(N*sizeof(float));
  v2 = (float*)malloc(N*sizeof(float));
  r = (float*)malloc(N*sizeof(float));

  //Generate or read input data.
  ...

  //Allocate arrays in Global memory
  cudaMalloc((void**)&dv1, sizeof(float) * N );
  cudaMalloc((void**)&dv2, sizeof(float) * N );
  cudaMalloc((void**)&dr, sizeof(float) * N );

  //Copy data into Global memory
  cudaMemcpy(dv1, v1, sizeof(float) * N,
                      cudaMemcpyHostToDevice);
  cudaMemcpy(dv2, v2, sizeof(float) * N,
                      cudaMemcpyHostToDevice);

  //Launch the vector add kernel.
  addv<<<BLOCKS, BLOCK_SIZE, BLOCK_SIZE * sizeof(float)>>>
      (dv1,dv2,dr);

  //Launch further kernels on the data.
  ...

  cudaMemcpy(r, dr, sizeof(float) * N ,
                    cudaMemcpyDeviceToHost);

  cudaFree(dv1);
  cudaFree(dv2);
  cudaFree(dr);

  //Show or further process results on the CPU.
  ...
}
```

Figure 3.20: This code starts a CUDA kernel on the GPU. The syntax `<<<nb,nt,sm>>>` is used to set up a kernel launch configuration. The `nb`, `nt` and `sm` quantities refer to the number of blocks, the number of threads/block and the amount of shared memory.

One of the array representations that Obsidian uses, the Pull array, was present already in the earlier work but went under a different name. It was not until discovering a complementary array representation, that we call push arrays, that the traditional array was renamed to pull.

The Obsidian implementation of pull arrays was influenced by Pan's representation of images as functions from coordinates to colour values [Ell03]. A pull array is a function from index to value and an associated length.

Push arrays were added to Obsidian first in reference [CSS12]. A push array specifies where elements are to end up in a result array, rather than where they come from (as in a pull array). In essence, a push array is a computation which writes an array to memory, but is parameterised on the *write function* which writes a single element into memory. The push array function can then use the write function zero, one, or several times.

Push arrays were added to Obsidian with the purpose of solving some performance issues that we had been struggling with for a long time. These problems were mainly concerned with concatenating or combining arrays in an efficient way. Push arrays have also been adopted by others and are used in the implementation of filters in reference [KN13] and in the new Nikola[5].

Both push and pull arrays are virtual, in the sense that they do not directly correspond to any data in a region of memory. This virtual nature of the array representations gives us fusion of operations for free. The programmer has to explicitly force an array (using a `force` function) to actually compute the values and store them in memory and to prevent operations from fusing.

In the version of Obsidian used in this paper[6], new features were added. We introduce two new variants of push and pull arrays called `GlobPush` and `GlobPull`. This introduces a distinction between local arrays, short enough to be computed by a block of the GPU, and global arrays that represent computations spread over blocks of the GPU. This is an important step when it comes to Obsidian's capabilities; we can now express in what order a kernel fetches elements from global memory. Earlier versions of Obsidian were limited to straight-line block indexing. A kernel generated by older Obsidian accessed blocks of elements in a fixed way (thread block i accessed elements using `(i * blocksize) + f tid`, where `f` can permute indices within a block but there is no similar way to permute the actual blocks).

In order to implement histograms we also need to add atomic operations to the language. The addition of atomic operations here is done in a rather ad hoc way and pushes us to program in a very low-level and imperative style. We strive to incorporate much of CUDA's low-level functionality into Obsidian and adding atomic

[5] github.com/mainland/nikola/blob/master/src/Data/Array/Nikola/Repr/Push.hs
[6] github.com/svenssonjoel/Obsidian/branches/February2013

operations is an attempt in that direction.

The version of Obsidian used in this paper also introduces a new monadic representation of GPU programs. We call the monad `Program`. The `Program` type has a parameter that represents at what level in the GPU hierarchy it executes. There are sequential *Thread* programs and parallel *Block* and *Grid* programs. Type synonyms are available, `TProgram`, `BProgram` and `GProgram`.

Many of the new features we mention above have been further refined and are present in versions of Obsidian following the one we use in this paper. For example, in the master branch of Obsidian[7] we build on the ideas in this paper. There, global and local arrays are represented by the same data type. That work also goes further by implementing hierarchy generic functions that are applicable at different levels of the Thread, Block and Grid hierarchy. We are also experimenting with the addition of mutable arrays in order to better integrate the atomic operations[8].

Obsidian programming example

In the different variants of counting sort that are developed in this paper, the prefix sum operation is the same. It is therefore natural to use as an introductory example of how Obsidian programs are written.

We need to operate on arrays that are too long to fit in a single block. The resulting prefix sum is implemented as shown in reference [HS07], see figure 3.21. There are three steps involved. First local prefix sums are computed; the rightmost elements elements are also stored in a separate, auxiliary, array. Following this, the prefix sum of the auxiliary array is computed. The prefix sums used in phase A and B of the figure can both be generated from the same Obsidian program. Finally the local prefix sums and the result of the auxiliary prefix sum are combined, forming a large prefix sum over the entire array.

The figure also shows what kernels we need to produce. First, something that performs a number of local prefix sums in parallel over the full length of an array is needed. The prefix sum algorithm that we implement here is based on divide and conquer and originates from Sklansky [Skl60].

The code below can be thought of as a program generator. Given an integer, n, it gives a prefix computation for 2^n elements. Here we go directly to generating the prefix sums operations (with the operator + hard coded) rather than a higher order function that can be used to more generically.

[7] github.com/svenssonjoel/Obsidian
[8] github.com/svenssonjoel/Obsidian/tree/mutable

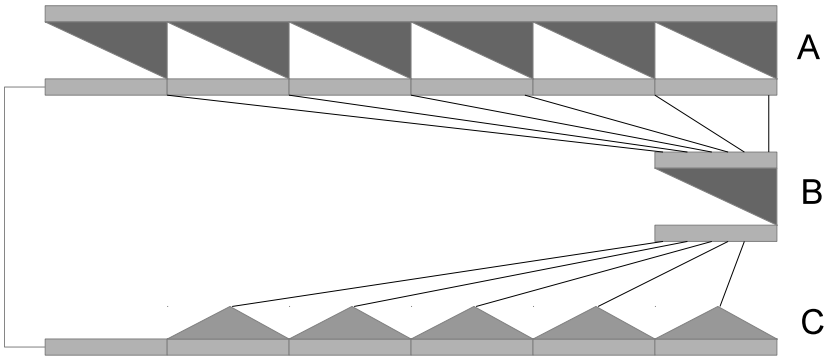

Figure 3.21: Illustration of how to combine many local prefix sum computations into a large one. The algorithm has three steps: A compute local prefix sums, B recursively compute prefix sums on the maximums of the local prefix sum calculations, C distribute summed maximums over the local results.

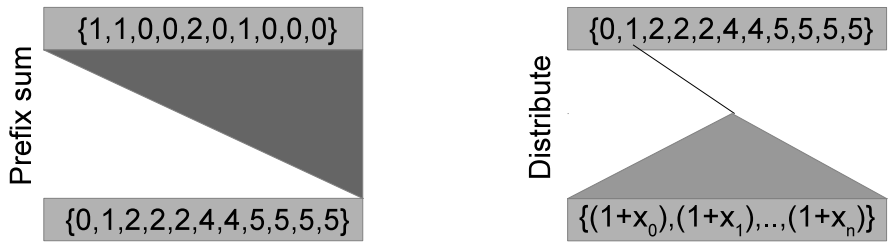

Figure 3.22: Close-up on the prefix sum and distribute operations used in Figure 3.21.

```
sklansky :: Int
            -> Pull EWord32
            -> BProgram (Pull EWord32)
sklansky 0 arr = return arr
sklansky n arr = do
    let arr1 = binSplit (n-1) fan arr
    arr2 <- force arr1
    sklansky (n-1) arr2
```

The `sklansky` code above is short but still contains much that needs explanation. First, the type specifies that the function takes a normal Haskell Int and an input array (`Pull EWord32`). The result is a `BProgram` that produces a result array. The function `binSplit` is in the Obsidian library and is used to implement divide and conquer algorithms. This function splits an array recursively a given number of times and then applies some computation on all parts. In this case the `fan` operation implemented below is used.

```
fan arr =  a1 `conc`  fmap (+ (last a1)) a2
    where
      (a1,a2) = halve arr
```

The `fan` operation splits an array down the middle and then adds the element at the highest index of the first half to all elements of the second. Then the two parts are concatenated back together.

In order to obtain both the local prefix sum results and the maximum element of the block, a small wrapper function is created.

```
sklanskyMax :: Int
               -> Pull EWord32
               -> BProgram (Pull EWord32,
                            Pull EWord32)
sklanskyMax n arr = do
  res <- sklansky n arr
  return (res,singleton (last res))
```

All this wrapper does is compute the prefix sum followed by returning the result and a singleton array containing the maximum, `last`, element.

This local prefix sum is run on sub-parts of the global array. The function `mapDist` splits up the array, runs a local computation on each part using the MPs of the GPU; when it finishes, the result of that computation is an array that remains distributed across the local memories of the GPU MPs.

```
sklanskyG :: Int
             -> GlobPull EWord32
             -> GProgram (GlobPush EWord32,
                          GlobPush EWord32)
sklanskyG logbsize input =
  toGProgram (mapDist (sklanskyMax logbsize)
                      (2^logbsize)
                      input)
```

In the `sklanskyG` function the local prefix combinator implemented so far is turned into a global prefix sum computation. The `toGProgram` function takes a function from blockId to a `BProgram` and gives back a `GProgram`, that is a program running on the full GPU (across many MPs).

The CUDA code generated from `sklanskyG` is shown in figure 3.23. Writing that type of CUDA code by hand is both tedious and error prone. The Obsidian code is both shorter, more compositional and easier to read.

One kernel remains to be implemented, `distribute`.

```
distribute :: EWord32
              -> GlobPull EWord32
              -> GlobPull EWord32
distribute bs maxs inputs = zipWithG (+) maxs' inputs
  where
    maxs' = repeat bs maxs
    repeat n = ixMap (\ix -> ix `div` n)
```

The `distribute` kernel takes three inputs, a number, n, and two global pull arrays. The elements of the first array are repeated n times and then the result is element wise added to the second input array. The code generated from this kernel is shown in figure 3.24. The `distribute` kernel is used in the implementation of a large prefix sum (over more elements then what can fit in shared memory) from local prefix sum computations. This is indicated in figure 3.21.

A note about generated code

All generated code shown in this paper has been edited slightly by hand. This is partly to increase readability but also to conserve space. The changes that have been made are entirely cosmetic. Line breaks have been inserted in too long lines. All occurrences of `blockIdx.x`, `blockDim.x` and `threadIdx.x` have been replaced with `bid`, `bDim` and `tid`. Some decimal constants have been changed to hexadecimal. No other changes have been made.

```
__global__ void sklansky(uint32_t *input0,
                         uint32_t *output0,
                         uint32_t *output1){
  extern __shared__ unsigned char sbase[];
  ((uint32_t *)sbase)[tid] =
    (((tid&0x1)<0x1)
      ? input0[((bid*32)+((tid&0xFFFFFFFE)|(tid&0x1)))]
      : (input0[((bid*32)+((tid&0xFFFFFFFE)|0x0))]+
         input0[((bid*32)+((tid&0xFFFFFFFE)|(tid&0x1)))]));
  __syncthreads();
  ((uint32_t *)(sbase + 128))[tid] =
    (((tid&0x3)<0x2)
      ? ((uint32_t *)sbase)[((tid&0xFFFFFFFC)|(tid&0x3))]
      : (((uint32_t *)sbase)[((tid&0xFFFFFFFC)|0x1)]+
         ((uint32_t *)sbase)[((tid&0xFFFFFFFC)|(tid&0x3))]));
  __syncthreads();
  ((uint32_t *)sbase)[tid] =
    (((tid&0x7)<0x4)
      ? ((uint32_t *)(sbase+128))[((tid&0xFFFFFFF8)|(tid&0x7))]
      : (((uint32_t *)(sbase+128))[((tid&0xFFFFFFF8)|0x3)]+
        ((uint32_t *)(sbase+128))[((tid&0xFFFFFFF8)|(tid&0x7))]));
  __syncthreads();
  ((uint32_t *)(sbase + 128))[tid] =
    (((tid&0xF)<0x8)
      ? ((uint32_t *)sbase)[((tid&0xFFFFFFF0)|(tid&0xF))]
      : (((uint32_t *)sbase)[((tid&0xFFFFFFF0)|0x7)]+
         ((uint32_t *)sbase)[((tid&0xFFFFFFF0)|(tid&0xF))]));
  __syncthreads();
  ((uint32_t *)sbase)[tid] =
    ((tid<0x10)
      ? ((uint32_t *)(sbase+128))[tid]
      : (((uint32_t *)(sbase+128))[0xF]+
         ((uint32_t *)(sbase+128))[tid]));
  __syncthreads();
  ((uint32_t *)(sbase + 128))[tid] = ((uint32_t *)sbase)[tid];
  if (tid<1){
    ((uint32_t *)(sbase + 256))[tid] = ((uint32_t *)sbase)[31];
  }
  __syncthreads();
  output0[((bid*32)+tid)] = ((uint32_t *)(sbase+128))[tid];
  if (tid<1){
    output1[(bid+tid)] = ((uint32_t *)(sbase+256))[tid];
  }
}
```

Figure 3.23: CUDA code generated from the Obsidian `sklanskyG` prefix sum program. This example shows the 32 element version of the prefix sum computation

```
__global__ void distribute(uint32_t s0,
                           uint32_t *input1,
                           uint32_t *input2,
                           uint32_t *output0){
  output0[((bid*bDim)+tid)] =
      (input1[(((bid*bDim)+tid)/s0)]+
       input2[((bid*bDim)+tid)]);

}
```

Figure 3.24: CUDA code generated from the Obsidian `distribute` function.

3.4.5 Implementing counting sort in Obsidian

Histogram

When computing the histogram, atomic operations are needed. Index i in the histogram array is incremented for each occurrence of the value i in the input array, and if there are multiple occurrences of i then multiple threads will try to increment, leading to a possible race. CUDA atomic operations must be applied to data stored at an actual memory location. This does not fit very well into our setting with virtual arrays (arrays that do not necessarily represent data in memory). But in Obsidian it is possible to drop down to a low enough level of abstraction to still implement this function. However, we are searching for suitable higher level abstractions to apply here.

```
histogram :: GlobPull EInt32
             -> GProgram ()
histogram gpull = do
  global <- Output $ Pointer Word32
  forAllT $ \gix ->
     atomicOp global
               (int32ToWord32 (gpull ! gix))
               AtomicInc
```

Below, the code generated from this Obsidian program is shown.

```
__global__ void histogram(int32_t *input0,
                          uint32_t *output0){

  atomicInc(output0+(uint32_t)(input0[((bid*bDim)+tid)]),
            0xFFFFFFFF);

}
```

The generated code uses `atomicInc` to increment a value in an array. The function is conditional, incrementing only if the value in memory is larger than the second argument. Since we always want to increment, no matter the value in memory, we've supplied `0xFFFFFFFF`, which is the largest possible value.

The Obsidian code and the CUDA code are very similar in size and complexity. However, an important advantage of the Obsidian code is that it composes better than the CUDA code. If the input to `histogram` was a `GlobPull` array produced by some other Obsidian function, then the two functions would be fused, typically generating much faster code and not allocating memory for the fused array.

Reconstruct

The kernel for the reconstruct step uses one thread per element in the input array. Each thread is implemented as a loop to write its corresponding element as many times as it should occur in the output.

```
reconstruct :: GlobPull EWord32
               -> GlobPush EInt32
reconstruct (GlobPull ixf) = GlobPush f
  where f k =
    do forAllT $ \gix ->
         let startIx = ixf gix
         in  SeqFor (ixf (gix + 1) - startIx) $ \ix ->
               k (word32ToInt32 gix) (ix + startIx)
```

The generated CUDA code for `reconstruct` can be seen below. Modulo syntax and some index manipulation, the code is very similar to the Obsidian code.

```
__global__ void reconstruct(uint32_t *input0,
                            int32_t *output0){

  for (int i1 =  0;
       i1 < (input0[(((bid*bDim)+tid)+1)]-
             input0[((bid*bDim)+tid)]);
       i1++)
  {
    output0[(i1+input0[((bid*bDim)+tid)])] =
      (int32_t)(((bid*bDim)+tid));

  }

}
```

Again, the difference between Obsidian and CUDA might seem small, but just as above, the code in Obsidian composes better.

3.4.6 Implementing occurrence sort in Obsidian

Occurrence sort uses an `occurs` kernel instead of a histogram. The result of the occurs computation is an array with a one at indices corresponding to values occurring in the input array. The reconstruction of the sorted (and duplicate free) array is also done slightly differently.

Occurs

The occurs kernel is implemented by using the `scatterGlobal` function from the Obsidian library. This function takes two arrays, one of indices to write to and a second of elements to write. In this case, it is applied to the input array and an array containing all ones (`replicateG 1`).

```
occurs :: GlobPull EInt32 -> GlobPush EWord32
occurs elems =
  scatterGlobal (fmap int32ToWord32 elems) (replicateG 1)
```

The CUDA code generated from this function is shown below.

```
__global__ void occurs(int32_t *input0,
                       uint32_t *output0){

  output0[(uint32_t)(input0[((bid*bDim)+tid)])] = 1;

}
```

When running this code, it is possible that many threads write to the same target element. Since all are writing a one, no atomic operations are needed; the result will still be one.

It is worth comparing the Obsidian code for the `histogram` kernel and the `occurs` kernel. The `histogram` kernel is highly imperative in nature, and requires atomic operations to manage synchronisation between threads. The `occurs` kernel, on the other hand, has a very straightforward data-parallel implementation. Not only is it simpler, but as we will see in section 3.4.7, it is also significantly faster.

Reconstruct

The reconstruct kernel for the occurrence sort algorithm is almost identical to the kernel used for standard counting sort. The only difference is that a conditional can be used instead of a loop. This can be seen as a slight optimisation; the previous version of reconstruct could still be used in its place.

```
reconstruct :: GlobPull EWord32
               -> GlobPush EInt32
reconstruct (GlobPull ixf) = GlobPush f
  where f k =
    do forAllT $ \gix ->
         let startIx = ixf gix
         in  Cond ((ixf (gix + 1) - startIx) ==* 1) $
               k (word32ToInt32 gix) startIx
```

This variant of reconstruct results in the generated code below.

```
__global__ void reconstruct(uint32_t *input0,
                            int32_t *output0){

  if ((input0[(((bid*bDim)+tid)+1)]-
       input0[((bid*bDim)+tid)])==1){

  output0[input0[((bid*bDim)+tid)]] =
      (int32_t)(((bid*bDim)+tid));

  }

}
```

3.4.7 Performance evaluation

This section presents the experimental results. Our implementations of counting sort are compared to the implementation of sorting from the Thrust library [NVId].

Thrust is a C++ library, developed by NVIDIA, which provides abstractions similar to the standard template library, but targets the GPU. The reason we have chosen to compare with Thrust is that it has similar goals to Obsidian: to provide high-level constructs while generating efficient GPU code.

All the data being sorted in the measurements are 32 bit unsigned integer valued keys. The convention used in the charts is that the y-axis represents time in ms. The different experiments are called *sorted* for sorting of an already sorted array, and *unique* for sorting of randomised arrays where each element occurs exactly once. There are also a number of experiments called Tx for a number x; the arrays used in these experiments contain elements from an interval of numbers (0 to $2^x - 1$).

Figures 3.25 and 3.26 show the results of comparing our full counting sort to Thrust sort and occurrence sort to Thrust's `sort` and `unique` functions. Our implementation of counting sort consistently outperforms Thrust, except for small ranges of values. The reason counting sort performance degrades is that the number of threads used in the reconstruct step is proportional to the range, and therefore at small ranges the execution becomes more sequential.

We compare occurrence sort to sort followed by unique from the Thrust library because that seems to be the current best practice [KGJ+11]. Our implementation is always at least twice as fast but in many cases it outperforms Thrust by more than a factor of four. Our method is clearly beneficial for removing duplicates.

We also compare occurrence sort and counting sort to each other. The result of this is shown in figure 3.27. We can see that the occurrence sort is typically a factor of two faster than the regular sorting algorithm. The speedup comes from the fact that the `occurs` kernel doesn't need to perform any kind of synchronisation, whereas the `histogram` kernel needs to use atomic operations.

The charts included in this paper show the performance when sorting eight respectively 32 million (2^{23} resp. 2^{25}) elements. However, we ran all benchmarks with a number of elements, N, equal to 2^n for n between 20 and 25. We found that the interesting comparison is not in what happens as N grows larger but rather what happens as we vary the ranges of the elements contained in the arrays.

All timing was performed on a system with an NVIDIA GTX670 GPU.

CUDA framework for the benchmarks

To obtain the timing measurements above, a framework[9] written in CUDA was used. This framework looks very similar to the CUDA code in figure 3.20. The main difference is that a number of kernels are run in sequence, as shown in figure 3.29.

[9] Source code needed to repeat our experiments is available at `www.cse.chalmers.se/~joels/csort.html`

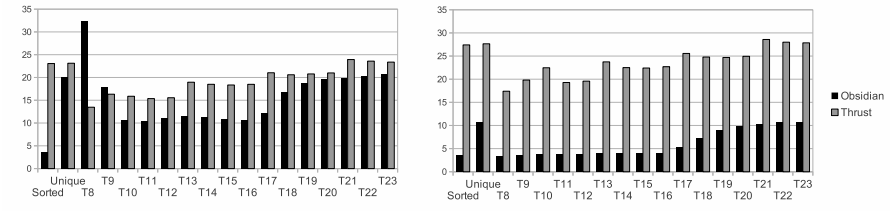

Figure 3.25: On the left, counting sort is compared to the Thrust library. On the right, occurrence sort is compared to the `sort` followed by `unique` function in Thrust. Both charts show running time for 8 Million elements.

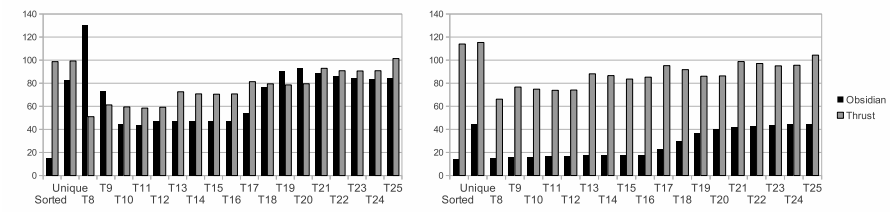

Figure 3.26: This chart shows same comparison as figure 3.25, but for 32 Million elements.

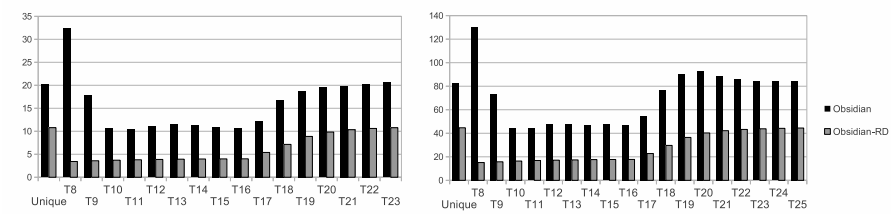

Figure 3.27: These two charts compares occurrence sort to counting sort. The left chart is for 8 Million elements and the right chart is for 32 Million elements.

```
float total_time;

cudaEventRecord(start,0);
for (int i = 0; i < 1000; ++i) {
    // perform counting sort.
}
cudaEventRecord(stop,0);
cudaEventSynchronize(stop);

cudaEventElapsedTime(&total_time,start,stop);
```

Figure 3.28: The timing methodology used in performance experiments. A CUDA event is recorded before and after the execution on the GPU. Their respective recorded times are then compared.

The timing is performed on the GPU computation only, as illustrated by figure 3.28.

3.4.8 Discussion

It is instructive to compare the code of the two variants of counting sort. The standard algorithm requires mutations and its parallel implementation is rather complicated. The occurrence sort variation avoids key synchronisations, making the algorithm data-parallel and this gives it a performance advantage. However, in applying the trick to remove synchronisation we have changed the semantics of the function. In this case, it is still useful, albeit less generally applicable. The idea of removing synchronisation is a general and important idea for speeding up algorithms, but it is not clear how general the method used in this paper is. We currently do not know of any other algorithms which could benefit from a similar trick.

If the desired effect is to sort and remove duplicates (for an example of this see [KGJ+11]) then there is a clear benefit to using occurrence sort over the `sort` combined with `unique` method.

In this paper, we are entirely focused on the implementation and performance of kernels. Reference [KVGH11] goes into the details of performing data transfers in parallel with ongoing computations. We have not addressed this issue at all in our implementations. This is also the reason why we haven't performed any benchmarks against the algorithm in [KVGH11], because it would compare very different things. As a piece of future work, it would be interesting to add the capability of streaming data from the main memory to the GPU to our algorithms.

In counting sort, an auxiliary array holding the count of how many times each el-

```
histogram<<<NB,BS,0>>>(dinput,dhistoutput);

scan256<<<NB,BS,2052>>>
    (dhistoutput,dscanoutput+1,dmaxs+1);

//dmaxs needs to be scaned (dmaxs2 is ignored)
scan256<<<SMALL,BS,2052>>>
    (dmaxs+1,dmaxs+1,dmaxs1+1);

scan16<<<1,SMALL,132>>>
    (dmaxs1+1,dmaxs1+1,dmaxs2);

// distribute can be in-place regarding dresult
// since it reads and writes in the exact
// same location (per thread)
distribute<<<SMALL,BS,0>>>
    (BS,dmaxs1,dmaxs+1,dmaxs+1);
distribute<<<NB,BS,0>>>
    (BS,dmaxs,dscanoutput+1,dscanoutput+1);

reconstruct<<<NB,BS,0>>>
    (dscanoutput,dresult);
```

Figure 3.29: The kernel launch sequence used in the timing of counting sort for 1 Million elements. Any other input data size would vary only some of the prefix sum (scan in the code) sizes.

ement occurs in the input array, the histogram array, is created. The size of this array is based on the range of elements occurring in the input. This means that more memory is needed for larger ranges and may imply that counting sort is suitable only up to some point. A related algorithm, known as radix sort [Knu98], addresses this issue by using multiple counting sort like passes, one for each digit position and thus limits this auxiliary memory usage to an array of size related to the radix (number of unique digits) used in the representation of the values being sorted. During our exploration of the counting sort algorithm on a GPU, we were not much concerned by memory usage; all data and auxiliary arrays fit easily in the device memory. The charts (3.25, 3.26 and 3.27) indicate that our implementation offers the most benefit in the ranges labelled as $T10$ to $T17$; these are in fact relatively small ranges (1024 to 131072 unique values). Our current hypothesis is that the decrease in benefit of counting sort from range $T18$ and onward is due to more and more complicated memory access patterns in the reconstruction phase. However, more in-depth profiling is needed to understand where the decrease in benefit comes from.

Obsidian is work in progress but this paper shows a step forward in its capabilities. New kinds of kernels that operate on global arrays can be implemented. We also added atomic operations to the language. However, when dealing with atomic operations, we express the Obsidian program at a very low level. One task for future work is to find some way to raise the level of abstraction in that area. Another future direction in exploring the counting sort algorithm is to consider further optimisations. One possibility is to work with sequentiality (more work per thread). Continuing exploration in that direction could lead to further improvements of Obsidian when it comes to low-level control.

In the current version of Obsidian, we need to write some CUDA code by hand in order to implement the algorithm we describe. In the future we want to be able to do all coding without leaving Haskell.

Acknowledgments

This research has been funded by the Swedish Foundation for Strategic Research (which funds the Resource Aware Functional Programming (RAW FP) Project) and by the Swedish Research Council.

Bibliography

[CKL+11] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell Array Codes with Multicore GPUs. In *Proceedings of the sixth workshop on Declarative as-*

pects of multicore programming, DAMP '11, pages 3–14, New York, NY, USA, 2011. ACM.

[CSS12] Koen Claessen, Mary Sheeran, and Bo Joel Svensson. Expressive Array Constructs in an Embedded GPU Kernel Programming Language. In *Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming*, DAMP '12, pages 21–30, New York, USA, 2012. ACM.

[Ell03] Conal Elliott. Functional Images. In *The Fun of Programming*, "Cornerstones of Computing" series. Palgrave, March 2003.

[HS07] Mark Harris and Shubhabrata Sengupta. Parallel Prefix Sum (Scan) with CUDA. In *GPU Gems 3*, 2007.

[JS11] Joel Svensson. Obsidian: GPU Kernel Programming in Haskell. Technical Report 77L, Computer Science and Enginering, Chalmers University of Technology, Gothenburg, 2011. Thesis for the degree of Licentiate of Philosophy.

[Kar12] Tero Karras. Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees. In *Proc. of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics*, EGGH-HPG'12. Eurographics Association, 2012.

[KGJ+11] Jens Krueger, Martin Grund, Ingo Jaeckel, Alexander Zeier, and Hasso Plattner. Applicability of GPU Computing for Efficient Merge in In-Memory Databases. In *The Second International Workshop on Accelerating Data Management Systems using Modern Processor and Storage Architectures (ADMS 11)*, 2011. `http://www.adms-conf.org/p19-KRUEGER.pdf`.

[KN13] Abhishek Kulkarni and Ryan R. Newton. Embrace, Defend, Extend: A Methodology for Embedding Preexisting DSLs, 2013. Functional Programming Concepts in Domain-Specific Languages (FPCDSL'13).

[Knu98] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[KVGH11] Vasileios Kolonias, Artemios G. Voyiatzis, George Goulas, and Efthymios Housos. Design and implementation of an efficient integer count sort in CUDA GPUs. *Concurrency And Computing: Practice And Experiance*, 23(18), December 2011.

[MM10] Geoffrey Mainland and Greg Morrisett. Nikola: Embedding Compiled GPU Functions in Haskell. In *Proceedings of the third ACM Haskell symposium*, pages 67–78. ACM, 2010.

[NVIa] NVIDIA. CUDA C Programming Guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[NVIb] NVIDIA. NVIDIA Thrust Library. `https://developer.nvidia.com/thrust`.

[NVIc] NVIDIA. NVIDIAs Next Generation CUDA Compute Architecture: Kepler GK110.

[OBA12] Ola Olsson, Markus Billeter, and Ulf Assarsson. Clustered deferred and forward shading. In *Proc. of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics*, EGGH-HPG'12. Eurographics Association, 2012.

[SA08] Erik Sintorn and Ulf Assarsson. Fast parallel GPU-sorting using a hybrid algorithm. *J. Parallel Distrib. Comput.*, 68(10):1381–1388, 2008.

[Skl60] J. Sklansky. Conditional Sum Addition Logic. *Trans. IRE*, EC-9(2), June 1960.

[Wik] Wikipedia. Wikipedia article: Counting sort. `http://en.wikipedia.org/wiki/Counting_sort`.

3.5 Paper F: A High-Level Embedded Language for Low-Level GPU Kernel Programming

Bo Joel Svensson, Mary Sheeran

abstract

Graphics Processing Units (GPUs) offer potential for very high performance; they are also rapidly evolving. Obsidian is an embedded language (in Haskell) for implementing high performance kernels to be run on GPUs. We would like to have our cake and eat it too; we want to raise the level of abstraction at which the programmer works from that of CUDA code (which we generate) and still to give the programmer fine control over low level details.

We present the user's view of Obsidian by showing case studies and a detailed optimisation effort applied to reduction kernels. Our chosen array representations (pull and push arrays) provide fine control over the use of memory, but are also sufficiently abstract to give guaranteed fusion of composed array operations. The case studies also show that Obsidian helps to make exploration of the design space for kernels easy. Code reuse and composition is easier in Obsidian than in CUDA.

Obsidian uses types to model the hierarchical nature of the GPU, to help the programmer to write suitable (and easy to compile) GPU programs. Type directed compilation provides the user with language constructs that can be used at different levels of the GPU hierarchy; the types constrain the allowed behaviour of programs at different levels, while remaining reasonably unobtrusive. The way in which the hierarchical structure of the GPU architecture has influenced the design of Obsidian is a major theme of this paper. Thus, the paper may function as an introduction to GPUs and GPU programming for functional programmers.

Benchmarks show that the low-level, detailed control offered to the programmer does have an associated pay-off in running time.

3.5.1 Introduction

Graphics Processing Units (GPUs) offer potential for high performance implementations of data parallel computations. However, GPUs are a bit quirky to program; often, the programmer needs to consider very low-level concepts in order to make the best use of the GPU. GPU programming models mirror the hierarchy of the GPU, with threads, groups of threads with special properties and then groups of such groups. There is a matching memory hierarchy, ranging from slow global memory

that is accessible to all threads to fast local memory that allows threads within a single group of threads to communicate with each other. For NVIDIA GPUs, both the architecture and the programming model are called CUDA (for Compute Unified Device Architecture). GPU *kernels* are compiled to run on the GPU, while serial *host code* controls how they are launched and supplied with data. Here, the word kernel denotes an SPMD (Single Program Multiple Data) program that is run on the GPU. Section 3.5.1 covers GPU architecture.

Obsidian is an embedded language for GPU programming that aims to raise the level of abstraction for the GPU programmer, while maintaining control over the low-level details that are needed for performance.

So far, we have chosen not to try to apply automatic transformations or compiler optimisation techniques to a GPU architecture agnostic program and hope to get performance. In Obsidian, we want to put the tools to make the correct decisions into the hands of the programmer. From Obsidian programs, CUDA kernels are generated; host code can be written in Obsidian, but is still preferably written in CUDA.

For key building blocks like scan and reduce, we see a trend towards the use of fewer and much more complicated kernels [MG09], so that the host code is often relatively simple. We wish to support the construction of these sophisticated kernels.

Motivation and contributions

Obsidian aims to provide a higher-level language with a more concise programming style than that of CUDA. We want to achieve this while keeping programmer control over the details that influence performance. The programmer should be able to experiment with global memory access patterns, make tradeoffs between sequential computation per thread and parallel computations across threads (using shared memory) and avoid unnecessary synchronisation.

We envision the user of Obsidian as someone who is familiar with and interested in the details of GPU hardware. For this category of users, the value of Obsidian can be in the rapid prototyping of programming ideas. Obsidian is also useful for trying to understand the actual cost model of a particular GPU, since it is so easy to generate programs that are systematically varied.

Another possible use of Obsidian could be as the code generating backend of a domain specific language that could benefit from GPU acceleration. Reference [CMM12] is an example of this.

Our work on Obsidian makes the following contributions:

- A high-level programming interface that encourages design space exploration through experimentation with details of GPU kernels (section 3.5.4).

- Compositional abstractions for implementation of efficient GPU Kernels (section 3.5.4).

- We use types to model the hierarchical nature of GPUs in our embedded language and to rule out programs that we cannot easily compile to a GPU (section 3.5.3).

- We use type directed compilation of language constructs and implement some GPU hierarchy generic functions (section 3.5.3).

The GPU and CUDA

This section provides some background information related to GPU architecture. Focus is placed on those aspects of GPUs that have influenced our work with Obsidian the most.

The GPUs we target with Obsidian are NVIDIA GPUs that support CUDA [NVIb]. CUDA is the NVIDIA C-dialect for data-parallel programming on their GPUs. We try to stay within the subset of functionality that is common between CUDA and OpenCL [TKG], as we may in future want to generate OpenCL programs.

The NVIDIA CUDA capable GPUs are built on a scalable architecture. A GPU consists of a number of *multiprocessors*; each multiprocessor has a number of processing elements (cores) and a local memory that is shared between threads running on the cores. Figure 3.30 illustrates these concepts. A GPU can come with as few as one of these multiprocessors. The GPUs used in our measurements are the GTX 680, which has eight multiprocessors, with a total of 1536 processing cores, the GTX 670 with 1344 cores, and the GT 650M, a laptop GPU, which has 384 such cores. On these cores, groups of 32 threads called *warps* are scheduled. There are a number of warp scheduling units per multiprocessor. Within a warp, threads execute in lockstep (SIMD); diverging branches, that is those that take different paths on different threads within a warp, are serialised, leading to performance penalties.

The scalable architecture design also influences the programming model. CUDA programs must be able to run on all GPUs from the smallest to the largest. Hence a CUDA program must work for any number of multiprocessors. The CUDA programming model exposes abstractions that fit the underlying architecture; there are *threads* (executing on the cores), *blocks* of threads (groups of threads run by a multiprocessor) and finally the collection of all blocks, which is called the *grid*.

The threads within a block can use the shared memory of the multiprocessor to communicate with each other. A synchronisation primitive, `__syncthreads()`, gives all the threads within a block a coherent view of the shared memory. There is no similar synchronisation primitive between threads of different blocks.

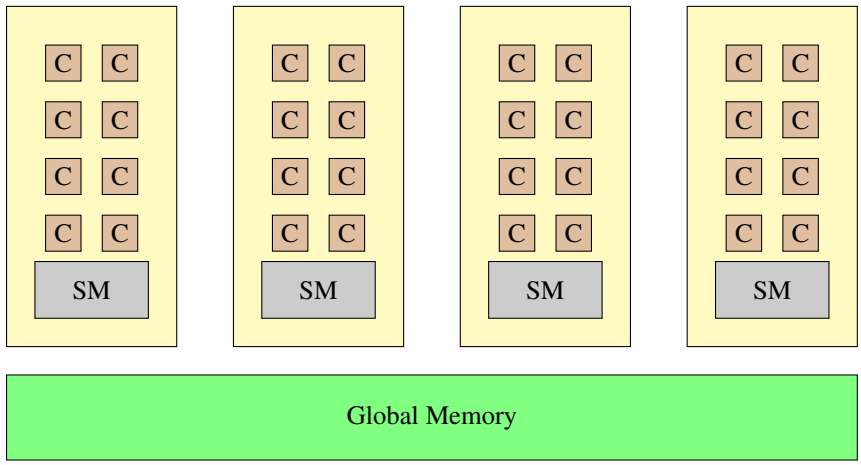

Figure 3.30: A GPU. Each multiprocessor has its own shared memory (SM).

The prototypical CUDA kernel starts out by loading data from global memory. The indices into global memory for an individual thread are expressed in terms of the unique identifier for that block and thread. Some access patterns allow memory reads to be *coalesced*, while others do not, giving very poor performance. The patterns that lead to good performance vary somewhat between different GPU generations, but regular, consecutive accesses by consecutive threads within a warp are best.

A CUDA program is expressed at two levels. Kernels are data-parallel programs that run on the GPU. They are launched by the controlling program, which runs on the CPU of the host machine.

3.5.2 A taste of Obsidian programming

Obsidian, like CUDA, differentiates between *Thread*, *Block* and *Grid* computations. The programmer specifies a sequential computation to be carried out by each thread. Many instances of a sequential computation run in parallel over threads, to form a block. A number of such blocks run in parallel, to form a grid.

Obsidian is a monadic embedded language. Local computations have the form `a0 -> .. -> an -> BProgram b`, where `BProgram` is a representation of programs that can be performed by a block of threads.

There are two different array representations, *Pull* and *Push* arrays. A pull array naturally represents gather operations, while a push array captures scattering operations. Reference [CSS12] describes the introduction of push arrays into Obsidian.

As an example, the code below implements a program that reverses the elements of an short local array.

```
reverseL :: SPull a
                -> BProgram (SPush Block a)
reverseL = liftM push . return . reverse
```

This program does not make use of any specific details of the BProgram monad; instead it just composes `return` and `push` with the Obsidian library function `reverse`. The `push` function converts a pull array to a push array. `SPull` and `SPush` arrays have static lengths, this is explained in more detail on page 160.

We can change the `reverseL` function slightly to have the result stored in local (shared) memory. Although this storing in shared memory serves little purpose in this case, it illustrates the programmer's ability to use shared memory. The `force` function takes either a push or a pull array , computes it and places the result in shared memory. Using `force` requires a `MemoryOps` constraint on the element type.

```
reverseL' :: MemoryOps a
                 => SPull a
                 -> BProgram (SPush Block a)
reverseL' = liftM push . force . reverse
```

The result of `force` is always a program yielding a pull array, and hence the use of `liftM push`.

A CUDA program describes what one thread does and the mapping of that program over many threads is done implicitly. In Obsidian, the programmer specifies the entire kernel program; the whole computation is specified, including how that computation is spread out over blocks, to form a *Grid* program.

The following Obsidian code shows how to split a large array up into chunks of 256 elements (`splitUp 256 arr`) and how the local `reverseL` and `reverseL'` kernels are mapped, in parallel, over blocks using `pConcatMap`.

```
reverses :: DPull a -> DPush Grid a
reverses arr = pConcatMap reverseLocal
                          (splitUp 256 arr)

reverses' :: MemoryOps a => DPull a -> DPush Grid a
reverses' arr = pConcatMap reverseLocal'
                           (splitUp 256 arr)
```

```
__global__ void reverse(int32_t* input0,
                        uint32_t n0,
                        int32_t* output0){

    output0[((blockIdx.x*256)+threadIdx.x)] =
      input0[((blockIdx.x*256)+(255-threadIdx.x))];

}

__global__ void reverse'(int32_t* input0,
                        uint32_t n0,
                        int32_t* output0){

    extern __shared__ uint8_t sbase[];
    ((int32_t*)sbase)[threadIdx.x] =
      input0[((blockIdx.x*256)+(255-threadIdx.x))];
    __syncthreads();
    output0[((blockIdx.x*256)+threadIdx.x)] =
      ((int32_t*)sbase)[threadIdx.x];


}
```

Figure 3.31: Code generated from the `reverses` and `reverses'` programs. `reverses'` stores the array in shared memory before copying it to the global result array.

The `pConcatMap` combinator is overloaded; here it is applied at the following type:

```
pConcatMap :: ASize l
           => (a1 -> Program t (SPush t a))
           -> Pull l a1
           -> Push (Step t) l a
```

`pConcatMap` takes a computation resulting in a push array at a specific level of the GPU hierarchy, `t`, and a pull array. The computation is applied to each element of the pull array and the results are concatenated into a push array at the level above `t`. Here, this is used to apply a block level computation, giving a grid computation.

The CUDA code generated by the above `reverses` function is shown in Figure 3.31. Note how it uses the indices of the block and thread (`BlockIdx.x` and `ThreadId.x`) to ensure that each thread accesses the correct part of the global input array.

To reverse a large array (one that cannot fit in one block), the elements in each block-sized chunk are reversed locally, and the order of those chunks is reversed.

```
largeReverse :: DPull b
             -> Push Grid EWord32 b
largeReverse arr =
  pConcat $ reverse $ fmap reverseLocal
                      (splitUp 256 arr)
```

The kernels that can be generated using Obsidian take arrays in global memory as inputs and output an array to global memory. Kernels with this behaviour have a type of the form:

```
myKernel :: a0 -> ... -> aN -> DPush Grid b
```

where `a0` to `aN` can be pull arrays or scalars. This choice limits the kinds of kernels can be expressed, but makes compilation of those kernels a lot easier. For example, a kernel that outputs data to two different global arrays cannot currently be expressed. Lifting this restriction is work in progress.

3.5.3 Obsidian internals

Many array languages supply a fixed set of parallel primitives like *Reduce*, *Scan* and *Permute*, but do not give the user control over how they are implemented. Obsidian programmers are, instead, encouraged to experiment with the implementation of such primitives, to maximise performance, for example by trading off sequential and parallel work or by varying memory access patterns. Obsidian's two array representations (pull and push arrays) are central to providing this fine control.

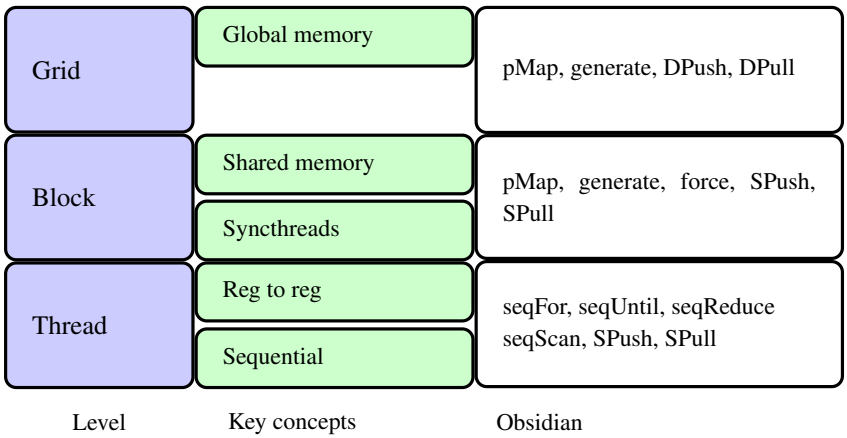

Figure 3.32: The CUDA hierarchy, showing important Obsidian features at each level.

Scalars

An *expression* data type defines the language available at the scalar level of Obsidian. This is implemented as a GADT (Generalised Algebraic Data Type) to provide a type safe programmer's interface:

```
data Exp a where
  Literal :: Scalar a
             => a
             -> Exp a

  BlockIdx :: DimSpec
              -> Exp Word32
  ThreadIdx :: DimSpec
               -> Exp Word32

  Index   :: Scalar a =>
             (Name,[Exp Word32])
             -> Exp a
```

```
  If      :: Scalar a
             => Exp Bool
             -> Exp a
             -> Exp a
             -> Exp a

  BinOp   :: (Scalar a,
              Scalar b,
              Scalar c)
             => Op ((a,b) -> c)
             -> Exp a
             -> Exp b
             -> Exp c

  UnOp    :: (Scalar a,
              Scalar b)
             => Op (a -> b)
             -> Exp a
             -> Exp b
```

Most constructors are standard, implementing arithmetic and conditionals via the `BinOp`, `UnOp` and `If` constructors. There are also two GPU specific variables, `ThreadIdx` and `BlockIdx` referring to the thread identity.

We provide type synonyms for commonly used element types:

```
...
type EInt32  = Exp Int32
type EInt64  = Exp Int64
...
type EWord32  = Exp Word32
type EWord64  = Exp Word64
type EFloat  = Exp Float
type EDouble = Exp Double
type EBool   = Exp Bool
```

Pull arrays

Pull arrays are perfectly suited for so-called gather operations (read many values, compute one value), as well as directly suitable for parallelisation on a GPU. A pull array uses the well known representation of arrays as functions from index to element.

```
data Pull s a = Pull s (EWord32 -> a)
```

Very similar representations are used in many other embedded languages. The embedded language Pan [EFdM03] used a similar representation for images and it has been used since in Obsidian, Feldspar [ACS+11] and Repa [KCL+10]. In Obsidian, there is a `force` function that, when applied to a pull array, writes the elements of that array into shared memory using as many threads as needed from those available. If the array is larger than the number of threads made available, the force operation is split into two or more passes. This is a change compared to earlier versions of Obsidian, where the length of arrays we could force were limited to the maximum number of threads per block (1024 threads). In this version, the threads are, in a sense, virtual. The mapping from virtual threads to real threads is done as a step in the compilation.

The length of an array, the `s` parameter, can be either static (a Haskell known-at-compile-time value) or dynamic (a runtime value). Static lengths are used for local (or block) computations, with those lengths determining shared memory consumption. Dynamic lengths are used for global computations (across many blocks). After specifying a local computation, it can often be applied over a variable number of blocks. The dynamic lengths capture this common use case.

There are type synonyms for the dynamic and static versions of pull arrays:

```
type SPull = Pull Word32
type DPull = Pull EWord32
```

Thread, Block and Grid Computations

The programs that can be specified in Obsidian are represented in a `Program` data type. This data type also represents the lowest level programming interface exposed by Obsidian. In reference [SSS13] we make use of the low-level capabilities provided by that interface to implement histograms using atomic operations (not shown here).

The `Program` data type has a parameter, `t`, that can be either `Thread`, `Block` or `Grid`, corresponding to the GPU hierarchy, as shown in Figure 3.32.

```
data Step a -- A step in the hierarchy
data Zero

type Thread = Zero
type Block  = Step Thread
type Grid   = Step Block
```

Below are some selected parts of the `Program` data type. There is a sequential, `SeqFor`, loop that runs a sequential loop body a given number of times. In this

case, both the loop body and the loop itself are `Thread` Programs. More interesting is the `ForAll` case that introduces different kinds of parallelism depending on the `t` parameter. `ForAll` executes a program body at the level below several times in parallel. If the body of the `ForAll` loop is a `Thread` computation, the `ForAll` loop itself is a `Block` computations. Likewise, if the body is a `Block` computation, the loop itself is a `Grid` computation.

```
data Program t a where
  Identifier :: Program t Identifier
  Assign :: Scalar a
            => Name
            -> [EWord32]
            -> (Exp a)
            -> Program Thread ()
  Allocate :: Name
              -> Word32
              -> Type
              -> Program Block ()

  ...
  SeqFor :: EWord32
            -> (EWord32 -> Program Thread ())
            -> Program Thread ()
  ForAll :: EWord32
            -> (EWord32 -> Program t ())
            -> Program (Step t) ()

  ...
  Sync :: Program Block ()
```

The `Assign` and `Allocate` constructors represent assignment into shared memory or a variable, and allocation of shared memory. The `Identifier` constructor is used to obtain fresh Identifiers and is not meant to be made available to the programmer. The `Sync` constructor corresponds to the CUDA `__syncthreads()` function.

To allow the programmer to use the monadic `do` notation when constructing Obsidian programs, the `Program` data type is made an instance of `Monad`. The details of how this is done are outlined in section 3.5.3.

Push arrays

A push array captures the idea of scattering, focussing on where elements end up rather than where they come from. In Obsidian, a push array is defined as follows:

```
data Push p s a =
  Push s ((a -> EWord32 -> TProgram ()) -> Program p ())
```

A push array has two parts, a length and a higher-order function. The idea is that this function is provided with a method to write an element into memory (a *write-function*), and returns a program that uses that write-function a number of times. Note that the write-function produces a `TProgram` (short for `Program Thread`). As a result, push arrays cannot represent nested data parallelism. This concept is best illustrated by showing how to turn a pull array into a push array.

```
convToPush :: SPull a -> SPush Block a
convToPush arr =
  Push n $ \wf ->
   ForAll (fromIntegral n) $ \tid -> wf (arr ! tid) tid
  where
    n = len arr
```

This is actually one case of the definition of the `push` function, which we will see shortly. In the push array case, the length represents the maximum index the push array might write to.

Push and pull array interplay

Both push and pull arrays represent methods of computing an array. They do not directly correspond to actual data stored in memory, but rather are virtual. This enables fusion of operations on both push and pull arrays. One such example is map fusion: `map f . map g = map (f . g)`. It is only by using `force` that the programmer can explicitly demand that the operations should not be fused but that the actual intermediate result should be stored in memory. In Obsidian, `force` is also the only way to convert a push array into a pull array. With some class constraints removed for simplicity, the type of `force` is:

```
force :: ( ... )
       => arr Word32 a
       -> Program Block (Pull Word32 a)
```

where `arr` is an array (push or pull) with elements that can be stored into memory. Note that the return type of `force` is a program yielding a pull array. This pull array represents reading from a named area of memory.

Converting a pull array to a push array is cheap and is done using a function called `push` that behaves differently (sequentially or in parallel) at different levels of the GPU hierarchy. This is implemented as a type class:

```
class Pushable t where
  push :: ASize s => Pull s e -> Push t s e
```

The `ASize` class has instances for both the static and dynamic lengths and is used to obtain conversion to a representation used internally; the function `sizeConv` converts both static and dynamic lengths to a `Word32` expression.

There are two instances of the Pushable class, one for thread programs and one for block programs:

```
instance Pushable Thread where
  push (Pull n ixf) =
    Push n $
      \wf -> SeqFor (sizeConv n) $ \i -> wf (ixf i) i

instance Pushable Block where
  push (Pull n ixf) =
    Push n $
      \wf -> ForAll (sizeConv n) $ \i -> wf (ixf i) i
```

Now, the `push` function captures just one possible way to convert a pull array into push array – with one write per thread. In the block case, it is possible to vary the amount of work per thread using the `pushN` function provided by the library. Indeed, the conversion of pull arrays into push arrays can be done in many ways. For example, when pushing more than one element per thread one could make choices between placing them consecutively or far apart. We intend to provide a variety of such special purpose push functions, but would also expect Obsidian users to tailor push functions to their own particular needs.

Compilation to CUDA

Obsidian is a monadic language. The `Program` data type is made a monad by adding the following constructors:

```
data Program t a where

   ...

  Return :: a -> Program t a
  Bind   :: Program t a
            -> (a -> Program t b)
            -> Program t b
```

and adding a very simple `Monad` instance:

```
instance Monad (Program t) where
  return = Return
  (>>=) = Bind
```

The choices above influence the compilation procedure. Implementing a compiler for the `Program` data type is not entirely straightforward. Our approach is described in further detail in reference [SS13]; the presentation here focuses instead on the type directed compilation of the `ForAll` primitives.

The first step in the compilation process is to turn the monadic `Program` into a first-order representation. This first-order representation also makes explicit what level of parallelism the `ForAll` constructor refers to. It does this by having separate constructors for parallel loops over all threads of a block `SForAll` and parallel loops over blocks `SForAllBlocks`. The first-order representation is a list of statements. The extra value of type `a` that is carried around with each statement is used to keep track of meta information in later compilation passes.

```
type IMList a = [(Statement a,a)]

type IM = IMList ()

data Statement t =
  forall a. (Show a, Scalar a)
     => SAssign Name [EWord32] (Exp a)

  ...
  | SSeqFor String (EWord32) (IMList t)

  ...
  | SForAll (EWord32) (IMList t)
  | SForAllBlocks (EWord32) (IMList t)

  ...
  | SAllocate Name Word32 Type

  ...
  | SSynchronize
```

The type directed part of the compilation is implemented using a type class called `Compile`. The `Compile` class houses just one function, called `compile`. The

`compile` function takes a splittable value supply and a `Program t a` and outputs intermediate code, of type `IM`, and a value of type `a`. The significance of the value supply and of the value of type `a` will be made clear after considering the instances of `Compile`.

```
class Compile t where
  compile :: Supply Int -> Program t a -> (a,IM)
```

For each of the levels of the GPU hierarchy, `Grid`, `Block` and `Thread`, there is an instance of the `Compile` class. The simplest case is `Thread`, which does not at all support parallel loops; it just delegates the compilation process to a function that takes care of all non type directed cases (called `cs`).

```
instance Compile Thread where
  compile s p = cs s p
```

The cases for blocks takes care of compilation of the `ForAll` constructor into a use of the `SForAll` statement. Likewise, the grid case compiles into an `SForAllBLocks` statement.

```
instance Compile Block where
  compile s (ForAll n f) = (a,out (SForAll n im))
    where
      p = f (ThreadIdx X)
      (a,im) = compile s p
  compile s p = cs s p

instance Compile Grid where
  compile s (ForAll n f) = (a, out (SForAllBlocks n im))
    where
      p = f (BlockIdx X)
      (a,im) = compile s p
  compile s p = cs s p
```

The `out` function used in the compilation function has type `Statement () -> IM` and generates a one element `IM` list with unit attached as meta-data.

Currently, only the `ForAll` constructor is given the type directed treatment. All other language features are tied to a specific level of the GPU or are completely independent of any such concerns. Taking care of these operations as well as the monadic `Return` and `Bind` cases is in the hands of the `cs` function. This is also where the value supply and the `a` parameter play a role.

```
cs :: Compile t => Supply Int -> Program t a -> (a,IM)
cs i Identifier = (supplyValue i, [])
...
```

The `Identifier` constructor does not correspond to any code executed on the GPU. It just helps in reifying the `Program` monad. Now it can be seen that the value of type `a` that is being returned is the monadic value of the `Program`. This follows the same monad reification technique as described in reference [SS13].

The constructors mentioned above are directly compiled into the corresponding statements.

```
cs i (Assign name ix e) = ((),out (SAssign name ix e))
cs i (Allocate id n t) = ((),out (SAllocate id n t))
cs i (Sync)            = ((),out (SSynchronize))
```

Turning to the monadic `Bind` and `Return` constructors, it becomes clear why we need that extra value of type `a` in the return type of the compile functions. We need a value to pass to the function in the `Bind` constructor.

```
cs i (Bind p f) = (b,im1 ++ im2)
  where
    (s1,s2) = split2 i
    (a,im1) = compile s1 p
    (b,im2) = compile s2 (f a)

cs i (Return a) = (a,[])
```

In the `IM`, all arrays in shared memory will have unique names. These names are obtained via the value supply during the reification of the `Program` monad. These names will be used in a memory mapping phase. A *liveness* analysis of the arrays collects information needed to lay out the arrays in the shared memory of the GPU. Currently no spilling of arrays that do not fit into shared memory is performed, so it is very possible to design an Obsidian program that will not work because of shared memory restrictions. After the arrays are assigned addresses in the shared memory, a renaming phase replaces array names with these locations.

The last step in the compilation process uses the Language.C.Quote library to turn the `IM` into CUDA code. This compilation stage takes a *configuration* as parameter. The configuration tells how many actual threads per block the CUDA code should be generated for. At this stage parallel for loops (`SForAll`) are turned into more than one parallel for loop in sequence, if needed.

3.5.4 Case studies

The following case studies show first a typical application, the Mandelbrot fractal, in which the required work is naturally highly parallel. The remaining case studies consider key building blocks, reduce and scan, that have data-flow graphs involving much more communication. Then, the programmer must find ways to decompose the algorithm in order to fit within the constraints imposed by the GPU architecture. These two case studies illustrate the use of Obsidian for exploring the design space when implementing a particular algorithm.

Fractals

The Mandelbrot fractal is generated by iterating a function:

$$z_{n+1} = z_n^2 + c$$

where z and c are complex numbers. The method to generate the fractal presented here is based on a sequential C program from reference [Ste89]. However, the problem is embarrassingly parallel.

In order to get the Mandelbrot image, one lets z_0 be zero and maps the x and y coordinates of the image being generated to the real and imaginary component of the c variable.

```
xmax =  1.2 :: EFloat
xmin = -2.0 :: EFloat
ymax =  1.2 :: EFloat
ymin = -1.2 :: EFloat
```

To obtain the well known and classical image of the set, we let the real part of c range over -2.0 to 1.2 as the x coordinate range from 0 to 512 and similarly for the y coordinate and the imaginary component.

```
-- For generating a 512x512 image
deltaP = (xmax - xmin) / 512.0
deltaQ = (ymax - ymin) / 512.0
```

The image is generated by iterating the function presented above. We have chosen to map the height of the image onto blocks of executing threads. Each row of the image is computed by one block of threads. This means that for a 512×512 pixel image, 512 blocks each of 512 threads are needed.

The function to be iterated is defined below and called `f`. This function will be iterated until a condition holds (defined in the function `cond`). We count the number of iterations and if they reach 512 we break out of the iteration.

```
f bid tid (x,y,iter) =
  (xsq - ysq + (xmin + (w32ToF tid) * deltaP),
   2*x*y + (ymax - (w32ToF bid) * deltaQ),
   iter+1)
  where
    xsq = x*x
    ysq = y*y

cond (x,y,iter) = ((xsq + ysq) <* 4) &&* iter <* 512
  where
    xsq = x*x
    ysq = y*y
```

The number of iterations that are executed is used to decide which colour to assign to the corresponding pixel. In the function below, `seqUntil` iterates `f` until the condition `cond` holds. Then the number of iterations is extracted and used to compute a colour value (out of 16 possible values).

```
iters :: EWord32 -> EWord32 -> SPush Thread EWord8
iters bid tid = fmap extract $
                seqUntil (f bid tid) cond (0,0,1)
  where
    extract (_,_,c) = ((w32ToW8 c) `mod` 16) * 16
```

The final step is to run the iterations for each pixel location. This is done by using the `generate` function:

```
generate :: ASize l
         => l
         -> (EWord32 -> SPush t b)
         -> Push (Step t) l b
```

`generate` is a function that can be used at the thread or block level. It takes a function from index to computation for that index and executes a number of such computations at the level above. Here, it is used to compute an element per thread, a block, and then to execute a number of such blocks over a grid.

```
genRect :: EWord32
         -> Word32
         -> (EWord32 -> EWord32 -> SPush Thread b)
         -> DPush Grid b
genRect bs ts p = generate bs $
                  \bid -> generate ts $ p bid
```

Image	xmax	xmin	ymax	ymin	ms
mandel	1.2	-2.0	1.2	-1.2	2.65
mandel1	-0.690906	-0691060	0.387228	0.387103	1.78
mandel2	-0.723005	-0.793114	0.140974	0.037822	6.79
mandel3	-0.745388	-0.745464	0.113030	0.112967	5.7

Figure 3.33: Running times for the Mandelbrot program. The parameters (`xmax`, `xmin`, `ymax`, `ymin`) correspond to zooming in on different areas of the generated images.

Figure 3.34: *Left:* `evenOdds` - `zipWith` reduction, leads to uncoalesced memory accesses.
Right: `halve` - `zipWith` reduction, leads to coalesced memory accesses. This coalescing is most important during the very first phase, when data is read from global memory.

The largest possible image that can be obtained using this particular method is 1024×1024. This is because of the maximum threads per block limitation of current GPUs. The code below generates a 512×512 element image.

```
mandel = genRect 512 512 iters
```

This case study illustrates the use of the `generate` function at two different levels of the GPU hierarchy. We also make use of iterations within each thread.

Reduction

In this section, we implement a series of reduction kernels. The Obsidian reductions are implemented to take an associative operator as a parameter. In the benchmarking, the operation used will be addition and the elements will be 32 bit integers.

Some of the reduction kernels will also require that the operation is commutative. To illustrate the kind of low level control that an Obsidian programmer has over expressing details of a kernel, we show a series of reduction kernels, each with different

Figure 3.35: *Left:* **BAD** Adding sequential reductions like this, reintroduces memory coalescing issues. Consecutive threads nolonger access consecutive memory locations.
Right: **GOOD** Using sequential reduction but maintaining coalescing

optimisations applied. Many of the optimisations that are applied to the kernels are influenced by a tutorial presentation from NVIDIA [Har].

The kernels implemented in the following subsections use a variant of `force`, called `unsafeForce` that attempts to be clever about the insertion of synchronisation into the generated code. If the array that is being forced is shorter than the warp width, a synchronisation is not needed. It is called unsafe because when using push arrays there is still no absolute guarantee that the threads do not later write to indices that are used by threads that belong to another warp. Programmers should use `unsafeForce` with care; it is however safe together with pull arrays.

This section focuses on local reduction kernels. The construction of large reduction algorithms from these kernels will be illustrated in section 3.5.4.

Reduction 1: Our first attempt at reduction combines adjacent elements repeatedly. This approach is illustrated on the left of Figure 3.34. In Obsidian, this entails splitting the array into its even and its odd elements and using `zipWith` to combine these. This procedure is then repeated until there is only one element left. This kernel will work for arrays whose length is a power of two.

```
red1 :: MemoryOps a
      => (a -> a -> a)
      -> SPull a
      -> BProgram (SPush Block a)
red1 f arr
  | len arr == 1 = return $ push arr
  | otherwise    =
    do
      let (a1,a2) = evenOdds arr
      arr' <- unsafeForce $ zipWith f a1 a2
      red1 f arr'
```

The above code describes what one block of threads does. To spread this computation out over many blocks and thus perform many simultaneous reductions, `pConcatMap` is used (as before):

```
mapRed1 :: MemoryOps a
              => (a -> a -> a)
              -> DPull (SPull a)
              -> DPush Grid a
mapRed1 f = pConcatMap $ red1 f
```

This kernel does not perform well (as can be seen in Figure 3.36). This may be attributed to the memory access pattern used. Remember that one gets better performance on memory access when consecutive threads access consecutive elements, which happens if each thread accesses elements that are some stride apart.

Reduction 2: `red2` lets each thread access elements that are further apart. It does this by halving the input array and then using `zipWith` on the halves (see Figure 3.34). This choice can only be made if the operator is commutative.

```
red2 :: MemoryOps a
           => (a -> a -> a)
           -> SPull a
           -> BProgram (SPush Block a)
red2 f arr
  | len arr == 1 = return $ push arr
  | otherwise    =
    do
      let (a1,a2) = halve arr
      arr' <- unsafeForce $ zipWith f a1 a2
      red2 f arr'
```

Reduction 3: The two previous implementations of reduce write the final value into shared memory (as there is a `force` in the very last stage). This means that the last element is stored into shared memory and then directly copied into global memory. This can be avoided by cutting the recursion off at length 2 instead of 1, and performing the last operation without issuing a `force`.

```
red3 :: MemoryOps a
           => (a -> a -> a)
           -> SPull a
           -> BProgram (SPush Block a)
red3 f arr
  | len arr == 2 =
      return $
      push $
      singleton $ f (arr ! 0) (arr ! 1)
  | otherwise    =
    do
      let (a1,a2) = halve arr
      arr' <- unsafeForce $ zipWith f a1 a2
      red3 f arr'
```

Reduction 4: Now we have a set of three basic ways to implement reduction and can start experimenting with adding sequential, per thread, computation. `red4` uses `seqReduce`, which is provided by the Obsidian library and implements a sequential reduction that turns into a for loop in the generated CUDA code. The input array is split into chunks of 8 that are reduced sequentialy. The partial results are reduced using the previously implemented (`red3`).

```
red4 :: MemoryOps a
           => (a -> a -> a)
           -> SPull a
           -> BProgram (SPush Block a)
red4 f arr =
  do
    arr' <- force $
            pConcatMap (seqReduce f)
                       (splitUp 8 arr)
    red3 f arr'
```

As can be seen by the running times in Figure 3.36, this optimisation did not come out well. The problem is that it reintroduces memory coalescing issues (see Figure 3.35).

Reduction 5: With `red5`, the coalescing problem is dealt with by defining a new function to split up the array into sub arrays. The idea is that the elements in the inner arrays should be drawn from the original array in a strided fashion.

```
coalesce :: Word32 -> SPull a -> SPull (SPull a)
coalesce n arr =
  mkPull s $ \i ->
   mkPull n $ \j -> arr ! (i + fromIntegral s * j)
  where
    s = (len arr) `div` n
```

With `coalesce` in place of `splitUp`, `red5` can be defined as:

```
red5 :: MemoryOps a
           => (a -> a -> a)
           -> SPull a
           -> BProgram (SPush Block a)
red5 f arr =
  do
    arr' <- force $
             pConcatMap (seqReduce f)
                        (coalesce 8 arr)
    red3 f arr'
```

Reductions 6 and 7: Lastly, we try to push the tradeoff between number of threads and sequential work per thread further. `red6` and `red7` represent changing `red5` to reduce 16 and 32 elements in the sequential phase. The performance of the fastest of these kernels is very satisfactory, at a level where the kernel is *memory bound*, that is constrained by memory bandwidth.

Larger local reductions: Figure 3.37 shows running time results for kernels that are very similar to the ones described above, but with a larger number of elements per local reduction (that is per block). This means that all the kernels need to introduce a small amount of sequentiality at the beginning. This is done by introducing a new force function:

```
unsafeForce' arr | len arr > 1024 = return arr
                 | otherwise      = force  arr
```

`unsafeForce'` exploits the fact that `force` doesn't actually change the elements of the array. Skipping the write into shared memory should still give the desired effect, so replacing it with `return` is safe. Two parallel phases are fused into one parallel phase that performs sequential work.

This is a new way of introducing sequential work into kernels; unlike `seqReduce`, it does not result in a for loop, but rather in an unrolled loop in the kernel.

Kernel	Threads	Shared mem.	ms
red1	1024	16384	2.085
red2	1024	16384	1.748
red3	1024	16384	1.741
red4	256	2048	2.113
red5	256	2048	0.455
red6	128	1024	0.441
red7	64	512	0.441

Figure 3.36: Execution time of a grid of 8192 reductions , each of 2048 elements and on a GTX680. The timing figures were obtained using the NVIDIA Profiler.

Kernel	Threads	Shared mem.	ms
red1l	1024	16384	1.567
red2l	1024	16384	1.073
red3l	1024	16384	0.976
red4l	256	2048	2.240
red5l	256	2048	0.443
red6l	128	1024	0.440
red7l	64	512	0.440

Figure 3.37: Execution time of a grid of 4096 reductions, each of 4096 elements on a GTX680. The timing figures were obtained using the NVIDIA Profiler.

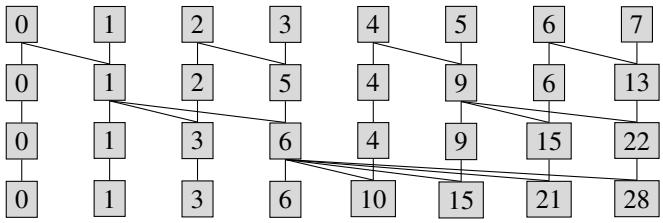

Figure 3.38: Sklansky parallel prefix network

Scan

Scan computes all the prefix sums of a sequence of values using a binary associative operator (and is familiar to Haskell programmers as the `scanl1` function).

Given an array of values $a_0, a_1, \ldots, a_n$ and associative operator $\oplus$, the scan operation computes a new array:

$$\begin{aligned} s_0 &= a_0 \\ s_1 &= a_0 \oplus a_1 \\ &\ldots \\ s_n &= a_0 \oplus a_1 \oplus \ldots \oplus a_n \end{aligned}$$

Figure 3.38 shows a standard divide and conquer decomposition of scan. Data flows from top to bottom and boxes with two inputs are operators. At each level, exactly half of the boxes are operators and in an imperative language the algorithm would naturally be implemented in-place. Since we cannot express in-place algorithms currently in Obsidian, this means that we need to copy unchanged values into a new array during each phase. The memory mapping gives us a ping-ponging implementation, but an in-place implementation would likely be faster.

Also, the threads now do two different things (copy, or perform operation). One can have as many threads as elements, but then each must have a conditional to decide whether to be a copy or operation thread. Or we can launch half as many threads and have each of them perform both a copy and an operation. We will show code for both of these options; the first is easier to implement.

The Obsidian code below implements the scan network from Figure 3.38, using as many threads as there are elements.

```
sklansky :: (Choice a, MemoryOps a)
            => Int
            -> (a -> a -> a)
            -> SPull a
            -> BProgram (SPush Block a)
sklansky 0 op arr = return $ push arr
sklansky n op arr =
  do
    let arr1 = binSplit (n-1) (fan op) arr
    arr2 <- force arr1
    sklansky (n-1) op arr2
```

This is a kernel generator; the (Haskell) `Int` parameter can be used to generate kernels of various sizes by setting it to the log base two of the desired array size.

The `binSplit` combinator used in `sklansky` is part of the Obsidian library and used to implement divide and conquer algorithms. It divides an array recursively in half a number of times (first parameter) and applies a computation to each part (second parameter). The operation applied in this case is `fan`:

```
fan :: Choice a
       => (a -> a -> a)
       -> SPull a
       -> SPull a
fan op arr =  a1 `conc`  fmap (op c) a2
    where
      (a1,a2) = halve arr
      c = a1 ! fromIntegral (len a1 - 1)
```

It is the array concatenation (`conc`) used in this function that introduces conditionals into the generated code.

Two elements per thread: Both to avoid conditionals and to allow for larger scans per block, we move to two elements per thread. Each phase of the algorithm is a parallel for loop that is executed by half as many threads as there are elements to scan. The body of the loop performs one operation and one copy, using bit-twiddling to compute indices. Note the use of two *write functions* in sequence. Similar patterns were used in our implementations of sorting networks [CSS12], for similar reasons.

```
phase :: Int
         -> (a -> a -> a)
         -> SPull a
         -> SPush Block a
phase i f arr =
  Push l $ \wf -> ForAll sl2 $ \tid ->
  do
    let ix1 = insertZero i tid
        ix2 = flipBit i ix1
        ix3 = zeroBits i ix2 - 1
    wf (arr ! ix1) ix1
    wf (f (arr ! ix3) (arr ! ix2) ) ix2
  where
    l = len arr
    l2 = l `div` 2
    sl2 = sizeConv l2
```

For an input of length 2^n, n phases are composed as follows:

```
sklansky2 :: MemoryOps a
             => Int
             -> (a -> a -> a)
             -> SPull a
             -> BProgram (SPush Block a)
sklansky2 l f = compose [phase i f | i <- [0..(l-1)]]
```

`compose` takes a Haskell list of programs and composes them in sequence, forcing intermediate arrays between each step.

```
compose :: MemoryOps a
           => [SPull a -> SPush Block  a]
           -> SPull  a
           -> BProgram (SPush Block  a)
compose [f] arr = return $ f arr
compose (f:fs) arr =
  do
    let arr1 = f arr
    arr2 <- force arr1
    compose fs arr2
```

Comparing the two kernels `sklansky` and `sklansky2` in the NVIDIA profiler indicates that `sklansky2`, while being faster than `sklansky` in many cases, has a worse memory loading behaviour. This indicates that tweaking the way data is loaded into shared memory may be beneficial in that kernel.

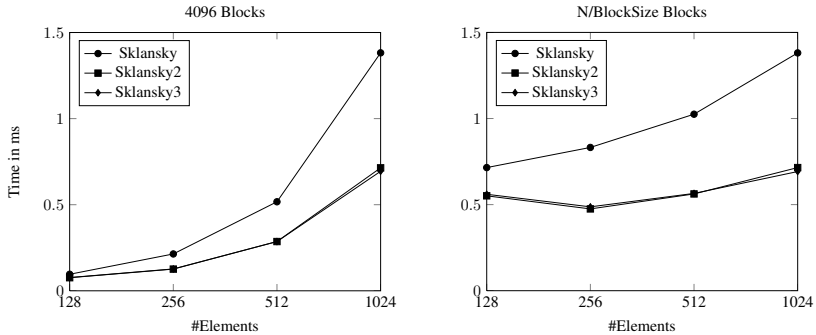

Figure 3.39: *left:*The time it takes to run 4096 blocks of varying sizes using our different implementations of the Sklansky kernel.
Right: varies the number of blocks so that the number of elements is the same in each experiment. These results where obtained on an NVIDIA GTX 670.

```
sklansky3 :: MemoryOps a
             => Int
             -> (a -> a -> a)
             -> SPull a
             -> BProgram (SPush Block a)
sklansky3 l f arr =
  do
    im <- force$ load 2 arr
    compose [phase i f | i <- [0..(l-1)]] im
```

Here we use `load 2` to realise loading of 2 elements per thread but in a strided way that is more likely to lead to a good memory access pattern. This function is an example of one of the custom ways to create a push array from a pull array mentioned in section 3.5.3. The results of these optimisations are shown in Figure 3.39. To make a seriously fast local and global scan, we would need to add more sequential work, as we did in the reductions. We do not yet know if our inability to do in-place updates prevents us from competing with hand-tuned CUDA scan code.

Combining kernels to solve large problems

With Obsidian, we can experiment with details during the implementation of local computations. In section 3.5.4, we saw that the description of a local kernel involves its behavior when spread out over many blocks. However, solving large problems must sometimes make use of many different local kernels or the same local kernel

used repeatedly. For now, we turn to CUDA for describing these algorithms. Here the procedure of making use of combinations of kernels is explained using large reduction as an example.

Large reductions

We implement reduction of large arrays by running local kernels on blocks of the input array. If the local kernel reduces ELT_PER_BLOCK elements to 1 then this first step reduces $BLOCKS*ELT\_PER\_BLOCK$ elements into $BLOCKS$ partial results. The procedure is then repeated on the $BLOCKS$ elements until there is one value.

Chosing `red5l` from section 3.5.4 and a block-size and local size of 4096 gives a 16 Million element reduction in only two stages.

The CUDA code we need to write by hand is very simple and only describes the actual order of the kernels we wish to launch. We make no changes to the kernels themselves. The listing in figure 3.40 shows an outline of the CUDA code for launching `red5l` to give 2^{24} (= 16777216) element reduction.

This is just a very simple example of what CUDA code can look like. Part of the motivation for showing it is to make the presentation complete. We currently write this kind of CUDA skeleton by hand; section 3.5.5 describes future work to improve this situation.

Table 3.41 shows the running time for the Obsidian together with CUDA 2^{24} element reduction compared to Data.Array.Accelerate. Again, the kernel generated from Obsidian is memory bound. These values are interesting since they indicate that the Accelerate reduction skeleton is very well optimised, while maintaining generality. We expected this, since reduction corresponds directly to the Accelerate Fold skeleton, which we believe to have been manually optimised as a one-time effort. It is also interesting that adding the second phase of reduction, in this case one instance of 4096 elements, only negligibly influences the running time.

Related work

There are many languages and libraries for GPU programming. Starting at the low-level end of the spectrum we have CUDA [NVIb]. CUDA is NVIDIA's name for the programming model and extended C language for their GPUs. It is the capabilities of CUDA that we seek to match with Obsidian, while giving the programmer the benefits of having Haskell as a meta programming language.

While remaining in the imperative world, but going all the way to the other end of the high-level - low-level spectrum, we have the NVIDIA Thrust Library [NVId].

```
/* Define sizes to work on */
#define BLOCKS 4096
#define ELT_PER_BLOCK 4096
#define N (BLOCKS * ELT_PER_BLOCK)

int main(void) {

  /* Allocate host and device arrays */
  float *input, result;
  float *dinput, *dresult;

  input  = (float*)malloc(N*sizeof(float));

  cudaMalloc((void**)&dinput,N*sizeof(float));
  cudaMalloc((void**)&dresult,BLOCKS*sizeof(float));

  /* Generate input data */
  ...

  /* Copy data to device */
  cudaMemcpy(dinput,
             input,
             N*sizeof(float),
             cudaMemcpyHostToDevice);


  /* Launch kernels */
  red5l<<<BLOCKS,256,512*sizeof(float)>>>(dinput,
                                          N,
                                          dresult);

  red5l<<<1,256,512*sizeof(float)>>>(dresult,
                                     4096,
                                     dresult);

  /* Copy result from device */
  cudaMemcpy(&result,
             dresult,
             1*sizeof(float),
             cudaMemcpyDeviceToHost);


  /* Process data on host side, if needed */
  ...

  return SUCCESS;
```

Figure 3.40: CUDA code that combines reduction kernels into reduction of a large array.

System	Elements	ms
Obsidian and CUDA	2^{24}	0.449
Accelerate	2^{24}	0.478

Figure 3.41: Running times of 2^{24} element reduction using Obsidian and CUDA or Accelerate. The results were obtained on a NVIDIA GTX680 GPU.

Thrust offers a programming model where details of GPU architecture are completely abstracted away. Here, the programmer expresses algorithms using building blocks like: *Sort*, *Scan* and *Reduce*.

Data.Array.Accelerate is a language embedded in Haskell for GPU programming [CKL+11]. The abstraction level is comparable to that of Thrust. In other words, Accelerate hides most GPU details from the programmer. Accelerate provides a set of operations (that are parallel and suitable for GPU execution, much like in Thrust) implemented as skeletons. Recent work has permitted the optimisation of Accelerate programs using fusion techniques to decrease the number of kernel invocations needed (see reference [MCKL13]). It seems to us that when using Accelerate the programmer has no control over how to decompose his computation onto the GPU or how to make use of shared memory resources. For many users, remaining entirely within Haskell will be a big attraction of Accelerate.

Nikola [MM10] is another language embedded in Haskell that occupies the same place as Accelerate and Thrust on the abstraction level spectrum.

The systems above are all for flat data-parallelism, Bergstrom and Reppy are attempting nested data-parallelism by implementing a compiler for the NESL language for GPUs [Ble96].

The Copperhead [CGK11] system compiles a subset of Python to run on GPUs. Much like other languages mentioned here, Copperhead identifies usages of certain parallel primitives that can be executed in parallel on the GPU (such as reduce, scan and map). But Copperhead also allows the expression of nested data-parallelism and is in that way different from both Accelerate and Obsidian.

In reference [OAB+12], Oancea et al. use manual transformations to study a set of compiler optimisations for generating efficient GPU code from high-level and functional programs based on `map`, `reduce` and `scan`. They tackle performance problems related to GPU programming, such as bad memory access patterns and diverging branches. Obsidian enables easy exploration of decisions related to these issues.

In the implementation of Obsidian we use a very simple and direct approach to monad reification. Related to this work is the approach based on a continuation monad by Persson et al [PAS12]. Recent work by Sculthorpe et al. in reference

[SBGG13] is also related. Our approach to monad reification is simpler but less general.

3.5.5 Future Work

Increase low level control

The capabilities of the GPU are changing and evolving. For example, it is now possible to do warp local computation that exchange values between threads using a set of `shuffle` instructions. These kernels do not need to use shared memory to the same extent as the ones we generate. It would be interesting to try to incorporate these capabilities Obsidian. However, this would mean that plans to target both OpenCL and CUDA would have to be retired.

In reference [SSS13], we experiment with atomic operations and are forced to write our code in a very imperative style. The addition of mutable arrays and a set of operations on them could improve this situation while giving up purity. We will explore what the addition of mutable arrays to Obsidian would mean as future work.

Being able to do in-place operations on arrays is beneficial to some algorithms. There are cases where in-place operation both simplifies implementation and improves performance. Feldspar, an EDSL for Digital Signal Processing being developed in our group at Chalmers, has mutable arrays. In that case, it was Fast Fourier Transform that pushed us towards this choice. Scan, in which elements must be copied unchanged between arrays, while others are operated upon, may also push us towards mutable arrays. We need to do more experiments before deciding whether or not to take this step, particularly as we value the simpilicity of the current version of Obsidian.

To automate or not to automate

Obsidian makes use of types that model the GPU hierarchy. This limits the kind of programs that can be expressed, ensuring that the resulting programs are suitable for the GPU. An alternative is to remove these types and at the same time make the compiler more clever. This would move some of the required GPU detail knowledge from the programmer into compiler transformations. We have recently started a project to investigate this change to Obsidian, together with a Masters student.

To optimise or not to optimise

The version of Obsidian described here does not try to use any compiler optimisation techniques. Instead, we are expecting that the CUDA compiler will apply a good

set of techniques, from common subexpression elimination to more GPU specific transformations. As future work, we will investigate to what extent this reliance on the CUDA compiler is justified. Also, we want to see if there are high-level optimisations that we can perform more easily on our intermediate representations of GPU kernels than what is possible on low-level CUDA code.

Kernel Coordination

In section 3.5.4, we implemented large reduction using CUDA and kernels generated using Obsidian. This is something we want to be able to avoid. We envision a kernel coordination language embedded in Haskell where the programmer can express the kind of sequential and parallel compositions of kernels; note that this is different from composing two local computations into one, which we already have the capability to do. Having an embedded language for expressing full algorithms that make use of many different kernels would further improve the usability of Obsidian. We have started a project with the goal of implementing this language, together with a Masters student.

What should the combinators be?

In this latest version of Obsidian, we have at last got a system that permits the construction of high performance kernels. But it is clear that we need to think hard about the set of combinators that we use to describe common patterns of parallel and sequential computation. Our current choice is a bit ad hoc and needs to be generalised and made more systematic. We will also explore the use of search to find algorithmic decompositions that are well matched to the GPU, exploiting the fact that Obsidian is embedded in a powerful host language.

3.5.6 Conclusion

Obsidian lends itself well to the kind of experimentation with low level GPU details that allow for the implementation of efficient kernels. This is illustrated in section 3.5.4, where we manage to apply techniques used in the NVIDIA tutorial [Har]. The case study also show hows we can compose kernels and thus reuse prior effort.

The use of GPU hierarchy generic functions makes the kernel code concise. The `pMap` and `pZipWith` functions are applicable both at block and grid level and are compiled into suitable parallel loops. The types that are used to model the GPU hierarchy also rule out any programs that we cannot very easily compile to the GPU. An alternative to this was discussed in the Future work section 3.5.5.

Our monad reification method certainly simplified the implementation of Obsidian, and showed that the method works well in practice (see reference [SS13] for more details of the method).

While other approaches to GPU programming in higher level languages tend to deliberately abstract away from the details of the GPU, we persist in our aim of exposing architectural details of the machine and giving the programmer fine control. This is partly because trying to provide simple but effective programming idioms is an interesting challenge. More importantly, though, we are fascinated by the problem of how to assist programmers in making the subtle algorithmic decisions needed to program parallel machines with strange, programmer controlled memory hierarchies, and strange constraints on memory access patterns. This problem is by no means confined to GPUs, and it is both difficult and pressing. We hope that Obsidian can form the basis for work in this area.

Acknowledgments

The work on monad reification presented in this paper is joint work between Josef Svenningsson and Joel Svensson.

Push arrays were invented by Koen Claessen. The implementation of push arrays in Obsidian is targeted at GPUs and restricted compared to Koen's more general idea. Koen has also been a source of important insights and tips that have improved this work greatly.

This research has been funded by the Swedish Foundation for Strategic Research (which funds the Resource Aware Functional Programming (RAW FP) Project) and by the Swedish Research Council.

Bibliography

[ACS+11] Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. The Design and Implementation of Feldspar an Embedded Language for Digital Signal Processing. In *Proceedings of the 22nd international conference on Implementation and application of functional languages*, IFL'10, pages 121–136, Berlin, Heidelberg, 2011. Springer-Verlag.

[Ble96] Guy Blelloch. Programming Parallel Algorithms. *Communications of the ACM*, 39(3), 1996.

[CGK11] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. In *Proc. of Principles*

and practice of parallel programming, PPoPP '11, pages 47–56, New York, NY, USA, 2011. ACM.

[CKL+11] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell Array Codes with Multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, DAMP '11, pages 3–14, New York, NY, USA, 2011. ACM.

[CMM12] Alex Cole, Alistair A. McEwan, and Geoff Mainland. Beauty And The Beast: Exploiting GPUs In Haskell. In Peter H. Welch, Frederick R. M. Barnes, Kevin Chalmers, Jan Baekgaard Pedersen, and Adam T. Sampson, editors, *Communicating Process Architectures 2012*, pages 121–134, aug 2012.

[CSS12] Koen Claessen, Mary Sheeran, and Bo Joel Svensson. Expressive Array Constructs in an Embedded GPU Kernel Programming Language. In *Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming*, DAMP '12, pages 21–30, New York, USA, 2012. ACM.

[EFdM03] Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(2), 2003.

[Har] Mark Harris. Optimizing parallel reduction in CUDA. "`http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf`".

[KCL+10] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 261–272, New York, NY, USA, 2010. ACM.

[MCKL13] Trevor L. McDonell, Manuel M.T. Chakravarty, Gabrielle Keller, and Ben Lippmeier. Optimising Purely Functional GPU Programs, 2013. 18th ACM SIGPLAN International Conference on Functional Programming, ICFP 2013.

[MG09] Duane Merrill and Andrew Grimshaw. Parallel Scan for Stream Architectures. *University of Virginia, Department of Computer Science, Charlottesville, VA, USA, Technical Report CS2009-14*, 2009.

[MM10] Geoffrey Mainland and Greg Morrisett. Nikola: Embedding Compiled GPU Functions in Haskell. In *Proceedings of the third ACM Haskell symposium*, pages 67–78. ACM, 2010.

[NVIa] NVIDIA. CUDA C Programming Guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[NVIb] NVIDIA. NVIDIA Thrust Library. `https://developer.nvidia.com/thrust`.

[OAB+12] Cosmin E. Oancea, Christian Andreetta, Jost Berthold, Alain Frisch, and Fritz Henglein. Financial software on gpus: between haskell and fortran. In *Proceedings of the 1st ACM SIGPLAN workshop on Functional high-performance computing*, FHPC '12, pages 61–72, New York, USA, 2012. ACM.

[PAS12] Anders Persson, Emil Axelsson, and Josef Svenningsson. Generic monadic constructs for embedded languages. In *Proceedings of the 23rd international conference on Implementation and Application of Functional Languages*, IFL'11, pages 85–99, Berlin, Heidelberg, 2012. Springer-Verlag.

[SBGG13] Neil Sculthorpe, Jan Bracker, George Giorgidze, and Andy Gill. The Constrained-Monad Problem, 2013. 18th ACM SIGPLAN international conference on Functional programming, ICFP 2013.

[SS13] Josef Svenningsson and Bo Joel Svensson. Simple and Compositional Reification of Monadic Embedded Languages, 2013. 18th ACM SIGPLAN International Conference on Functional Programming, ICFP 2013.

[SSS13] Josef David Svenningsson, Bo Joel Svensson, and Mary Sheeran. Counting and occurrence sort for GPUs using an embedded language. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*, FHPC '13, pages 37–46, New York, USA, 2013. ACM.

[Ste89] Roger T. Stevens. Fractal Programming in C, 1989. M&T Books.

[TKG] The Khronos Group. OpenCL - The open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`.

Chapter 4

Retargetable Parallel Programming Papers

4.1 Paper G: Programming Future Parallel Architectures with Haskell and Intel ArBB

Bo Joel Svensson, Ryan R. Newton

Abstract

New parallel architectures, such as Cell, Intel MIC, GPUs, and tiled architectures, enable high performance but are often hard to program. What is needed is a bridge between high-level programming models where programmers are most productive and modern parallel architectures. We propose that that bridge is Embedded Domain Specific Languages (EDSLs).

One attractive target for EDSLs is Intel ArBB, a virtual machine for parallel, vectorized computations. We propose to wed ArBB with the functional programming language Haskell, using an EDSL that generates code for the ArBB VM. This Haskell integration provides added safety guarantees compared to other ArBB interfaces. Further, our prototype, Harbb, is one of the first EDSL implementations with optimized backends for multiple parallel architectures (CPU, NVIDIA GPU, and others), allowing portability of source code over devices and their accelerators.

4.1.1 Introduction

Are radical new parallel architectures market-feasible if they require significant changes for programmers? The jury is out. In recent years we have seen difficult-to-program chips suffer (e.g. Cell) and GPU vendors strive to enable more traditional programming features [NVI09] (e.g. C++). There is an increasing tension between ease of programming and efficiency.

The tension shows across diverse chip markets. For example, small embedded devices are most power efficient if their processors and operating systems omit programming features such as virtual memory and threads [HSW+00]. At the other end of the power spectrum, GPU's graphics performance may suffer due to inclusion of hardware to ease GPGPU programming. In short, there is an opportunity cost to including extra hardware for programmability.

In this paper, we argue that a specific technique holds the greatest promise of solving the programmability dilemma. Domain-specific languages, *embedded* within general purpose languages (EDSLs) enable familiar programming models *and* flexible mapping onto new hardware. The key to having this cake and eating it too is *metaprogramming*. Familiar programming features are present, but are eliminated

at an intermediate (metaprogram evaluation) phase and therefore do not reach the parallel hardware itself.

In pursuit of this vision, we offer a new EDSL implementation, called Harbb, that combines existing systems, Accelerate [CKL+11] and ArBB [Int], to produce a unified high-level programming environment equally suited to multicore, vectorized CPUs, as to GPUs and other accelerators (such as Intel MIC chips [SCS+08]). Harbb is a single EDSL implementation with independently optimized backends by different teams; namely, the ArBB backend for CPU/MIC, and a CUDA backend for NVIDIA GPU. This makes Harbb an appealing platform for fair CPU/GPU comparisons, as well as a compelling programming model for single-source portable performance across a range of parallel architectures, present and future.

4.1.2 Embedded Domain-Specific Languages

Domain-specific languages—from Makefiles and LaTeX to Matlab—are almost too ubiquitous to notice. Most relevant to our purposes, domain-specific languages (DSLs) that target a narrow domain and expose communication patterns to the compiler have achieved performance-portability across a wide range of parallel architectures. StreamIt[G+06] is a good example.

DSLs may start out simple and focused, but if they gain popularity they quickly grow in complexity to rival full-blown languages. Feature creep can make DSL implementations complex and expensive to maintain. Further, non-standard DSL syntax and features present a learning curve for users. In the last ten years an attractive solution to this dilemma has emerged: *embed* each DSL into a general-purpose host language that can provide common functionality with familiar syntax and semantics.

When embedding, host language programs generate DSL programs; the *deeper* the embedding, the more integrated the DSL into the syntax and type-system of the host language. A key host-language feature for embedding is that language constructs can be overloaded to operate over *abstract syntax trees* (ASTs)[1]. For example, the following simple function operates on scalars:

```
float f(float x) { return (2*x - 1); }
```

But simply by changing the types, `f` might be lifted to operate on *expressions* (which, when evaluated, will yield `floats`):

```
exp<float> f(exp<float> x) {
  return (2*x - 1);
}
```

[1]This ad-hoc polymorphism is accomplished, for example, through operator-overloading in C++ or type classes in Haskell.

A common arrangement is for the host language program to execute at runtime but to generate ASTs that are executed by a just-in-time (DSL) compiler. This use of metaprogramming (program generation) differs from the more common usage of preprocessors and macros, which typically add extra phases of computation *before* compile time—increasing the number of compile-time rather than runtime phases.

One reason that EDSLs are good for productivity is that the programmer gains the software engineering benefits of the host language (object-orientation, higher-order-functions, modules, etc), while not paying the cost at runtime for additional layers of abstraction or indirection. Indeed, the embedded languages for performance-oriented EDSLs are often simple, first-order languages without pointers [NGC+08, CKL+11].

As a research area, EDSLs and two-stage DSLs have been actively pursued for at least a decade [Ell04, NGC+08] but are gaining steam recently [CGK11, Sta] and are beginning to appear in commercial products [Int]. Further, EDSL techniques have spread beyond their origin in the programming languages community. For example, both Stanford's Parallel Programming Laboratory (PPL) and the Berkeley Parlab are creating EDSLs as their flagship parallel programming solutions for domains such as machine learning and rendering [Sta]. Moreover, EDSLs need not be hosted by esoteric research languages—Intel's ArBB embeds an array language in C++ and Berkeley's Copperhead [CGK11] generates CUDA from simple Python code.

For the remainder of this paper we will focus our discussion on the Intel ArBB VM, a virtual machine for just-in-time generation of vector codes, which implements a restricted domain-specific language of array computations. In this paper we introduce *High-level ArBB*, (Harbb), an EDSL that internally uses the ArBB VM. While Intel's ArBB package already includes an EDSL targeting the VM (for C++), Harbb offers additional advantages, including more succinct programs and additionally safety guarantees—namely, complete deterministic-by-construction parallel programs (across both host and VM languages).

4.1.3 Harbb = ArBB + Accelerate

Our first Harbb prototype adapts an existing EDSL called *Accelerate*. Accelerate targets high-level data-parallel programming in Haskell. Previous work on Accelerate has focused on developing a CUDA-backend for GPU programming. In this paper we describe our effort to retarget Accelerate to ArBB.

With respect to determinism guarantees, the existing Intel ArBB product represents an integration of the safe (ArBB VM) with the unsafe (C++). In the Haskell context, because purely functional computations are guaranteed deterministic (even when executed in parallel), and because ArBB computations invoked by Haskell

functions are themselves free of side-effects, Harbb achieves a guarantee of determinism for complete programs that combine both Haskell computation and ArBB VM computation.

The Accelerate programming model consists of collective operations that can be performed over arrays, together with a simple language of scalar expressions—in the current release, the Haskell type system enforces that parallelism not be nested. Accelerate's collective operations include *Map* (akin to parallel for loops) *ZipWith* (a generalization of elementwise vector addition) and *Fold* (sum generalized)—familiar operations for programmers versed in the functional paradigm. All of these collective operations are easily parallelizable.

```
dotProd (xs :: Vector Float)
        (ys :: Vector Float) =
  let xs_ = use xs
      ys_ = use ys
  in  fold (+) 0 (zipWith (*) xs_ ys_)
```

The Accelerate code listing above specifies a function that takes two 1-dimensional arrays as inputs (of type `Vector Float`, e.g. a vector of floats). The result of the function is a single scalar. The `use` function is applied to an array to convert it for use in the collective operations provided by Accelerate; `use` may result in copying the array to, for example, the GPU in the case of Accelerate's CUDA backend.

After applying `use` to bring in input data, the programmer then constructs a data-parallel program from collective operations. Above, `fold` is a function that takes three arguments, here those arguments are `(+)`, 0 and `zipWith (*) xs_ ys_`. `zipWith`, in turn, is a function taking three arguments, `(*)`, `xs_` and `ys_`. The `zipWith` operation here applies pairwise multiplication to the two arrays and the `fold` sums up all the elements into a single scalar.

Accelerate's CUDA backend implements collective operations using a hand-tuned "skeleton" for each operation (and possibly for different hardware versions). The kernel—for example, the function to mapped over the dataset—is instantiated into the body of the skeleton code. The resulting program is compiled using the NVIDIA CUDA compiler and dynamically linked into the running Haskell program.

4.1.4 Intel Array Building Blocks (ArBB)

The operations exposed by ArBB are similar to those of Accelerate. In ArBB, parallel computations are expressed using a set of built-in primitives. All vectorization and threading is managed internally by ArBB. The programmer uses collective operations with a clear semantics such as `add_reduce` that computes the sum of the

elements in a given array. ArBB also has language constructions for control flow, conditionals and loops. These operations have their usual sequential semantics and are not parallelized by the system, rather, only specific collective operations are executed in parallel.

Today's existing ArBB product is embedded in C++ and provides special types for scalars and arrays (e.g. `dense<f32>` rather than `vector<float>`). Using ArBB/C++ to express the dot product computation can be done as follows:

```
void dot_product(const dense<f32>& a,
                 const dense<f32>& b,
                 f32& c)
  c = add_reduce(a * b);
```

Note that arithmetic operators such as (*) are overloaded for to operate on arrays as well as scalars (e.g. `a * b` above). Thus, the above C++ function multiplies two arrays before summing them with `add_reduce`:

The code listing below is indicative to the amount of glue code needed to invoke an ArBB computation[2]. It shows how the dot product code is launched using `call` and how data is bound, `bind`, for use in ArBB.

```
int main()
{
  double a[SIZE];
  double b[SIZE];

  for ( int i = 0; i < SIZE; ++i) {
    a[i] = ...; b[i] = ...;
  }
  dense<f32> va, vb;
  f32 vc;
  bind(va, a, SIZE);
  bind(vb, b, SIZE);
  call(dot_product)(va, vb, &vc);
  ...
}
```

The model provided by Accelerate is slightly richer than that of ArBB. Even the two very simple `dot_product` examples above manage to illustrate this. In Accelerate there is a more general reduction primitive called `fold` where in ArBB there are

[2]In OpenCL and CUDA the glue code situation is even worse

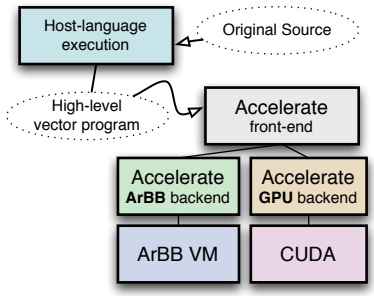

Figure 4.1: Architecture of the Harbb system.

specific reductions, `add_reduce`, `mul_reduce` and so on. Not visible in these small examples is another difference, Accelerate operations are generalized to arbitrary dimensions while ArBB operations are limited to 1, 2, and 3 dimensions (and 0, i.e. scalar). These differences aside, the ArBB and Accelerate programming models are very similar.

4.1.5 Implementation of Harbb

Using Accelerate and ArBB together, we propose a layered architecture for Harbb, pictured in Figure 4.1. The host language execution (via Haskell in this case) executes the programmer's source code, generating a vector program in the restricted language of the Accelerate front-end. Then either Accelerate backend—ArBB or CUDA—may be used. The bottom layer of the ArBB backend consists of a direct mapping of the ArBB virtual machine API (VMAPI) into Haskell including one-to-one bindings for each C function. To [partially] automate the creation of these bindings we used the C2HS system [Cha99]. The ArBB/Haskell bindings are very low-level. The idea is that the ArBB/Haskell bindings should be used to implement backends for higher-level data-parallel EDSLs.

To address the functionality mismatch between Accelerate and ArBB, when possible we reencode the users Accelerate program using existing ArBB mechanisms. Our prototype does not support 100% of Accelerate's compute model (for example, only supporting up to three-dimensional arrays), but the remaining functionality can be mapped onto ArBB in time using known methods—for example, our compiler could map higher dimensional arrays onto a specific lower-dimensional data-layout.

One example of a functionality discrepancy bridged by our implementation is reductions. As mentioned in 4.1.4, Accelerate allows the programmer to reduce using an arbitrary associative function, but ArBB has only built-in reductions with fixed

operations (add, multiply, xor, etc). We plan to provide general reductions in the ArBB backend by a two-fold strategy:

1. Attempt to map an Accelerate reduction directly onto an ArBB primitive such as `add_reduce`.

2. Apply a general reduction technique based on $log(N)$ map operations over successively halving array sizes. Essentially, cut the array in half, combine the halves, repeat. In [Har] this approach is explained in the context of CUDA.

In our current experiments, approach (2) is significantly slower. Therefore, the ideal would be that ArBB exposed general reduction directly in its programming model. We expect this functionality to be added in future releases. In the meantime we plan to explore a technique that would allow us to maximize the number of situations in which (1) above applies. Namely, a reduce operation can often be *factored* into a map followed by a reduce. For example, a reduction that multiplies each input number by a coefficient and sums the results can be split into a map phase for the multiplication followed by the built-in `add_reduce` operator.

Another choice faced by our implementation is the granularity at which the ArBB JIT is invoked. Specifically, should each collective operation result in its own call to the ArBB JIT (in ArBB terminology, *immediate-mode*, akin to the pre-OpenGL 3.0 immediate mode), or should multiple collective operations be placed together inside an ArBB function and passed to the JIT? We will call the latter approach *retained-mode*.

Retained-mode generally offers performance benefits; a bigger chunk is given to the JIT compiler, enabling cross-optimization between collective operations. Our prototype Accelerate backend uses ArBB in a combination of immediate- and retained-mode. The main collective operations are compiled using the retained-mode. For example, in the case of a `map f` operation first the function to be mapped is created and compiled using retained-mode then a small *mapper* function is also created and compiled using retained-mode. Between the collective operations the backend needs to perform data management and copying, which are performed in immediate-mode. It is our belief that Harbb would benefit from using retained-mode exclusively but we leave that as future work.

4.1.6 Preliminary Results

Black-Scholes option pricing is a finance-related benchmark that has been used in similar DSLs targeting GPUs [CKL$^+$11, MM10]. Since we are re-using the Accelerate front-end, we can directly use the Black-Scholes benchmark that is shipped

with that system. Figure 4.2 shows the complete code listing for an Accelerate Black-Scholes function which can be executed on GPUs or any processor targeted by ArBB.

The kernel of this algorithm performs arithmetic on triples of floating point numbers, creating pairs of floats as results. The problem is embarrassingly parallel, consisting of independent computations for every element of an array (a map).

Figures 4.3 and 4.4 show preliminary results obtained on the Black-Scholes benchmark. The figures where obtained on a system with a 4-core Intel Core I7 975 machine with HyperThreading. The GPU used was a NVIDIA GTX480.

Figure 4.3 shows running times obtained when JIT-time is included. JIT compilation time is much larger in the CUDA backend than the ArBB one. Part of this difference can be attributed to the fact that the CUDA backend calls an external compiler (`nvcc`), which takes its input in a file and runs in a separate process. ArBB, on the other hand, has a library interface to its JIT-compiler. Figure 4.4 shows results obtained when pre-compiling the CUDA functions eliminating JIT overhead. In Accelerate, this happens automatically when the same kernel is invoked repeatedly, because the Accelerate CUDA backed uses a caching scheme to avoid unnecessary JIT invocations. (The caching functionality has not yet been duplicated in the ArBB backend.)

These early results demonstrates the principle that even when a kernel executes with higher throughput on a GPU, in a particular program it is difficult to decide whether a computation is worth moving to a GPU, incurring extra data-movement and possibly extra JIT compilation [GH11]. Specifically, we see that the Accelerate Black-Scholes program from Figure 4.2 performs better on the CPU if it executes once (even on a large window of data) whereas the GPU would yield better performance in a sustained series of executions. Because both CPU and GPU execution may be desirable—and the selection may be dynamic—it is beneficial to have a single source code that is portable across both.

4.1.7 Related Work

OpenCL [Khr08] is a programming model very similar to CUDA but with the aspiration to offer both acceleration of computations on GPUs or to multicore CPUs. OpenCL JIT compiles the kernels for the particular hardware available and is in that sense similar to ArBB. OpenCL programs are relatively low-level and require a large amount of boilerplate to create and invoke. In this sense they occupy a very different niche than Accelerate.

Microsoft Accelerator [TPO06] is an embedded language with similar aspirations as ArBB, that is, to target a diverse range of architectures using the same source code. Accelerator can be used from the C# language or the functional F# language

```
blackscholes (xs :: Vector (Float,Float,Float)) =
  map kernel (use xs)

kernel x =
  let (price, strike, years) = unlift x
      r     = 0.02 -- riskfree constant
      v     = 0.30 -- volatility constant
      sqrtT = sqrt years
      d1    = (log (price / strike) +
                    (r + 0.5 * v * v) * years) /
              (v * sqrtT)
      d2    = d1 - v * sqrtT
      cnd d = d >* 0 ? (1.0 - cndfn d, cndfn d)
      cndD1 = cnd d1
      cndD2 = cnd d2
      expRT = exp (-r * years)
  in lift ( price * cndD1 -
            strike * expRT * cndD2
          , strike * expRT * (1.0 - cndD2) -
            price * (1.0 - cndD1))

cndfn d =
  let poly  = horner coeff
      coeff = [0, 0.31, -0.35, 1.78, -1.82, 1.33]
      rsqrt = 0.39894228040143267793994
      k     = 1.0 / (1.0 + 0.2316419 * abs d)
  in rsqrt * exp (-0.5*d*d) * poly k

horner coeff x =
  let madd a b = b*x + a
  in  foldr1 madd coeff
```

Figure 4.2: Complete code listing for a Black-Scholes function expressed in Haskell syntax using the Accelerate and Harbb libraries. Invocations of the functions `use`, `lift` and `unlift` represent additional boilerplate added for conversion in and out of Accelerate types. Specifically, `lift` and `unlift` convert tuples and handle the fact that Accelerate arrays of tuples are really implemented as tuples of arrays. Otherwise, the program is identical to a plain Haskell implementation.

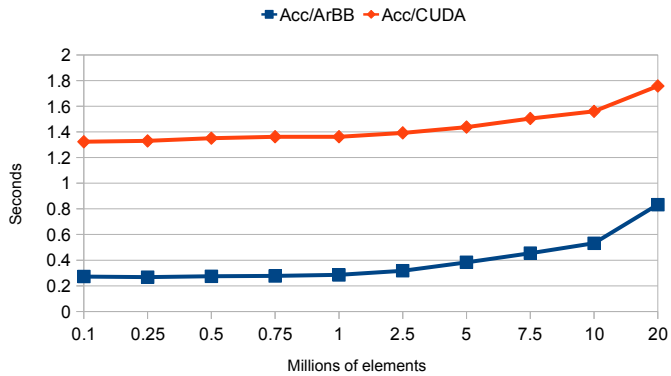

Figure 4.3: Running time experiments including JIT-time.

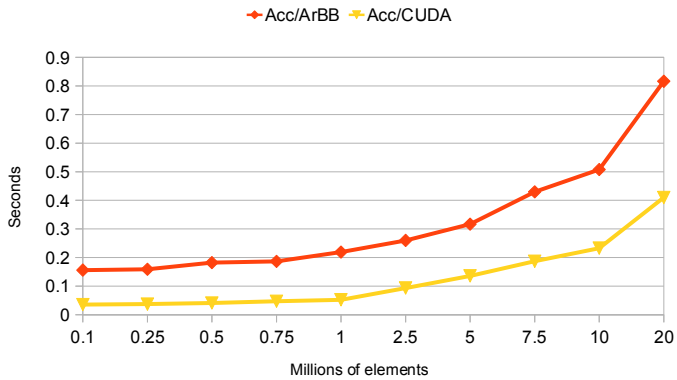

Figure 4.4: Running time experiments JIT-time excluded.

and targets GPUs or of CPUs and their vector units.

Many CPU and GPU comparisons, and some CPU/GPU workload partitioners [LCWM08], rely on redundant hand-written versions of all kernels (though some systems like Qilin [LHK09] allow a single source code). It is difficult in this kind of scenario to make fair comparisons, controlling for the amount of effort put into the respective implementations. For example, comparing unoptimized serial CPU implementations vs. GPU ones is not informative [LKC+10]. In Harbb controlling for effort need happen only once—both CUDA and ArBB are independently optimized by their respective teams of engineers—not for each benchmark.

4.1.8 Discussion and Conclusions

We have demonstrated that an EDSL such as Accelerate is sufficiently platform-independent to break free of its original hardware target (CUDA/GPU) and create efficient programs on other architectures. This gives us hope that Harbb/Accelerate programs will be forward-portable to future parallel architectures and instruction sets.

The EDSL approach changes the playing field for the designer of compiler backends. Rather than contending with full blown languages and their complexities (e.g. pointers, aliasing, inheritance, virtual functions, etc), compiler backends can focus on simple value-oriented compute languages.

But the EDSL method solves only part of the performance-portability problem. Simple as EDSL target languages may be, there remains a substantial challenge in mapping them efficiently to the diversity of parallel architectures available now and in the near future. For example, the idiosyncrasies of memory bank access on NVIDIA GPUs must be taken into account to generate efficient implementations of the high-level collective operations that we have discussed.

This is a compiler backend research challenge. The *skeletons* method mentioned in Section 4.1.3 is one approach to this problem, as are the optimizations studied in the Copperhead [CGK11] and Obsidian [JS11] projects. On the other hand, systems that rely on advanced optimizations typically suffer to some extent from performance-predictability problems. Thus achieving portable, predictable performance on a wide range of architectures—even for the simplest target languages—will be the subject of much future work.

Bibliography

[CGK11] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. In *Proc. of Principles*

and practice of parallel programming, PPoPP '11, pages 47–56, New York, NY, USA, 2011. ACM.

[Cha99] Manuel M.T. Chakravarty. C → Haskell, or Yet Another Interfacing Tool. In *In Koopman and Clack [23*, pages 131–148. Springer-Verlag, 1999.

[CKL+11] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell Array Codes with Multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, DAMP '11, pages 3–14, New York, NY, USA, 2011. ACM.

[Ell04] Conal Elliott. Programming graphics processors functionally. In *Proceedings of the 2004 Haskell Workshop*. ACM Press, 2004.

[G+06] M. I. Gordon et al. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 151–162, New York, NY, USA, 2006. ACM.

[GH11] Chris Gregg and Kim Hazelwood. Where is the data? why you cannot debate gpu vs. cpu performance without the answer. In *Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2011.

[Har] Mark Harris. Optimizing parallel reduction in CUDA. "`http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf`".

[HSW+00] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Architectural support for programming languages and operating systems*, ASPLOS-IX, pages 93–104, New York, NY, USA, 2000. ACM.

[Int] Intel. Intel array building blocks.

[JS11] Joel Svensson. Obsidian: GPU Kernel Programming in Haskell. Technical Report 77L, Computer Science and Enginering, Chalmers University of Technology, Gothenburg, 2011. Thesis for the degree of Licentiate of Philosophy.

[Khr08] Khronos OpenCL Working Group. The opencl specification, version 1.0.29. `http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf`, 2008.

[LCWM08] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Meng. Merge: a programming model for heterogeneous multi-core systems. *SIGPLAN Not.*, 43:287–296, March 2008.

[LHK09] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proc. of IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, New York, NY, USA, 2009. ACM.

[LKC+10] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. *SIGARCH Comput. Archit. News*, 38:451–460, June 2010.

[MM10] Geoffrey Mainland and Greg Morrisett. Nikola: Embedding Compiled GPU Functions in Haskell. In *Proceedings of the third ACM Haskell symposium*, pages 67–78. ACM, 2010.

[NGC+08] Ryan R. Newton, Lewis D. Girod, Michael B. Craig, Samuel R. Madden, and John Gregory Morrisett. Design and evaluation of a compiler for embedded stream programs. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '08, pages 131–140, New York, NY, USA, 2008. ACM.

[NVI09] NVIDIA. NVIDIA's next generation cuda compute architecture: Fermi, 2009.

[SCS+08] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27:18:1–18:15, August 2008.

[Sta] Stanford. Stanford pervasive parallelism laboratory.

[TPO06] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. volume 34, pages 325–335, New York, NY, USA, October 2006. ACM.

4.2 Paper H: Parallel Programming in Haskell Almost for Free

Bo Joel Svensson, Mary Sheeran

Abstract

Nowadays, performance in processors is increased by adding more cores or wider vector units, or by combining accelerators like GPUs and traditional cores on a chip. Programming for these diverse architectures is a challenge. We would like to exploit all the resources at hand without putting too much burden on the programmer. Ideally, the programmer should be presented with a machine model abstracted from the specific number of cores, SIMD width or the existence of a GPU or not. Intel's Array Building Blocks (ArBB) is a system that takes on these challenges. ArBB is a language for data parallel and nested data parallel programming, embedded in C++. By offering a retargetable dynamic compilation framework, it provides vectorisation and threading to programmers without the need to write highly architecture specific code. We aim to bring the same benefits to the Haskell programmer by implementing a Haskell frontend (embedding) of the ArBB system. We call this embedding EmbArBB. We use standard Haskell embedded language procedures to provide an interface to the ArBB functionality in Haskell. EmbArBB is work in progress and does not currently support all of the ArBB functionality. Some small programming examples illustrate how the Haskell embedding is used to write programs. ArBB code is short and to the point in both C++ and Haskell. Matrix multiplication has been benchmarked in sequential C++, ArBB in C++, EmbArBB and the Repa library. The C++ and the Haskell embeddings have almost identical performance, showing that the Haskell embedding does not impose any large extra overheads. Two image processing algorithms have also been benchmarked against Repa. In these benchmarks at least, EmbArBB performance is much better than that of the Repa library, indicating that building on ArBB may be a cheap and easy approach to exploiting data parallelism in Haskell.

4.2.1 Introduction

Modern hardware architectures provide much parallelism in the form of multi- and many-core processors and heterogeneous combinations of the two. We would like to exploit this parallelism without over-burdening the programmer, and without sacrificing too much performance. Ideally, the programmer should be encouraged to write the code in a way that is not too closely linked to the precise combination of cores,

SIMD vector processing power and GPU that is currently targeted. It should also be possible to easily retarget the code to new architectures. As processor architectures develop and are combined at an ever increasing pace, these requirements place considerable demands on programming languages and libraries for parallel programming.

Intel's Array Building Blocks (ArBB) is one approach to the problem of how to increase productivity for programmers who need to exploit hardware parallelism, but for whom very low level parallel programming (as typically found in High Performance Computing) is too difficult or time consuming [NSL+11]. ArBB is a retargettable dynamic compilation framework that is provided as an embedded language in C++. It aims to give the programmer access to data and thread parallelism, while protecting him from common pitfalls such as race conditions and deadlock.

Many of the ideas in ArBB feel familiar to functional programmers. For example, when referring to collection data types, reference [NSL+11] states that "Expressions over collections always act as if a completely new collection were created and assignment between collections always behaves as if the entire input collection were copied." This rather functional view of data is accompanied by extensive optimisation that (as the paper states) removes almost all copying. A key to the approach is the use of standard patterns of computation such as scan and reduce – familiar as higher order functions in a functional setting. The system tries to fuse away intermediate data-structures when these patterns are composed. The patterns are deterministic, so that problems with race conditions are avoided by design. Programs are stated to be short and close to the mathematical specification, and that, again, is a familiar mantra. The system allows C++ users to construct code objects called closures. The first time the *call* operation is applied to a particular function, ArBB captures the sequence of operations over ArBB types and creates an intermediate representation. This representation is then compiled to vector machine language, which is cached for later use. The machine code is also invoked (in parallel on multiple cores, if appropriate). The next time the function is called, ArBB reuses the cached machine code rather than regenerating it. Although dynamically typed closures are available, ArBB is generally statically typed.

ArBB is an embedded language, implemented as a library in the host language C++. The embedding makes heavy use of the C++ template machinery. There is also a lower level interface to ArBB, the ArBB Virtual Machine C API [Int]. The low level C API is not intended for use directly by applications programmers, but rather as a base for implementations of alternative ArBB frontends (embeddings) [NSL+11]. The ArBB VM implements the compilation and parallelisation. Reference [SN11] presents Haskell bindings to the ArBB C API as well as initial steps towards using ArBB as a backend to the embedded language Accelerate [CKL+11].

The design of ArBB seems particularly well suited to a functional setting. In this paper, we present work in progress on embedding ArBB in Haskell. We call the embedding EmbArBB. It is our hope that using Haskell as a host language will be just as user friendly and efficient as the C++ version. EmbArBB does not yet include all of the functionality of ArBB, and this is discussed further in section 4.2.8. However, our first benchmarks show that the performance of EmbArBB is very similar to that of ArBB in C++. This bodes well.

EmbArBB is available at GitHub as
`github.com/svenssonjoel/EmbArBB`.

4.2.2 Related Work

Data.Array.Accelerate

Accelerate [CKL+11] is an embedded language for general purpose, flat data parallel computations on GPUs.

The Accelerate programmer expresses algorithms using collective operations over vectors. These collective operations are similar to the parallel patterns of ArBB, but are generally higher order. That is, where ArBB and EmbArBB have `addReduce`, `mulReduce` and so on, Accelerate has a single higher-order fold function. This means that the Accelerate library gets away with a much smaller set of operations, while maintaining a higher level of expressivity in the language.

Accelerate arrays have their dimensionality (shape) encoded in the type and this was the inspiration for the similar approach taken in EmbArBB. In Accelerate, a one dimensional array of integers has type `Array DIM1 Int`. However, Accelerate does not limit the dimensionality to one, two or three, as ArBB does, but rather supports arbitrary dimensionality.

Data Parallel Haskell

Data Parallel Haskell (DPH) is an extension to GHC for nested data parallelism. This is one thing that DPH and ArBB have in common.

In DPH, the programmer can use parallel arrays of type `[:e:]` that look similar to normal Haskell lists, but give access to data parallel execution. The similarity to Haskell list processing doesn't end there. The operations on these parallel arrays are also reminiscent of Haskell's normal list processing functions. For example, `mapP`, `unzipP` and `sumP` are parallel versions of these well known functions.

DPH relies on a technique called flattening to transform its nested data parallelism into flat data parallelism. A recent article about DPH is reference [JLKC08].

Feldspar

Feldspar is a language for Digital Signal Processing (DSP) programming developed at Chalmers, ELTE University and Ericsson (the telecom company) [ACD+10]. The functional `while` loop of EmbArBB is inspired by the same concept in Feldspar.

Feldspar is based on a deeply embedded core language and implements a vector library as a shallow embedding on top of that. This means that there are no vector specific constructs in the core abstract syntax.

Nikola

Nikola is an embedded language for GPU programming[MM10], also in Haskell. During the early phases of implementation of EmbArBB, the source code of Nikola was often studied for inspiration.

The embedding used in Nikola makes use of an untyped expression data structure wrapped up in phantom types, in the style of Pan [EFdM03]. The EmbArBB embedding works in the same way. One of the listed strengths of Nikola is the ability to generate GPU functions from Haskell functions, which enables function reuse and the ability to amortise the cost of code generation over several launches in an easy way. The ability the generate target language function from Haskell functions is a present in EmbArBB as well.

Nikola, like Accelerate, provides a set of higher-order functions with general `map`, `reduce` and `zipWith` functionality.

Repa

Repa is a library for regular, shape polymorphic parallel arrays [KCL+10]. Repa uses a concept of `delayed` arrays to obtain fusion of operations, such as the typical map fusion:

```
map f . map g  = map (f . g)
```

A delayed array has a representation that is quite direct to parallelise; it is represented as a function from an index-space to an element and the extents of that same index-space. Concretely:

```
data DArray sh e = DArray sh (sh -> e)
```

Parallelising the computation of such an array is done by splitting up the index-space over the available parallel resources of the system. Repa does not use SIMD (vector) instructions; this is something that ArBB does and that EmbArBB gets for free.

Repa is compared to EmbArBB in the benchmarks in section 4.2.7.

4.2.3 Motivation

We have two main motivations for implementing EmbArBB.

Firstly, we are interested in exploring programming idioms for data parallel programming in a functional setting. Embedding ArBB gives a quick route to a platform for such exploration, without too much implementation effort. This is what the "almost for free" in the title is intended to refer to. We wish to explore ways to make use of the fact that we have an array programming library embedded in a sophisticated, strongly typed host language. In our earlier work on the embedded hardware description language Lava, we investigated various approaches to exploiting the host language during netlist synthesis [She04], and we have also experimented with the use of search and dynamic programming in generating parallel prefix (or scan) networks [She11]. We intend to apply similar methods to the development of data parallel programs once the EmbArBB implementation is more complete and stable.

A second motivation is the desire to teach NESL-style data parallel programming in a Masters course on parallel functional programming that has recently been introduced at Chalmers [HS12]. The first instance of the course (in Spring 2012) covered Blelloch's NESL language, with its associated cost model [Ble96], but did not provide any satisfactory way for the students to experience real, nested data parallel programming. This was due not only to a lack of suitable hardware, but also to deficiencies in the available tools. We used the Repa library to get flat data parallelism, but then suffered from the lack of a built-in scan primitive (as many of Blelloch's NESL examples use scan). Perhaps as a result of needing scan, we did not get good performance. We have not done enough examples or experiments yet, but it seems to us that EmbArBB could be used to give students an experience of real parallel programming in a NESL-like functional language. It remains to be seen whether the limited degree of nesting allowed in ArBB can be offset, at least to some extent, by clever use of the host programming language during generation of the desired abstract syntax tree.

4.2.4 Programming in ArBB

What the authors of ArBB call *latent parallelism* is expressed both by using operations like scan and reduce (or fold) over vectors and by mapping functions over collection data-types. ArBB provides both dense and nested vectors. Dense vectors can be one, two or three dimensional; they correspond to ordinary arrays.

Matrix-vector multiplication can be expressed in ArBB and C++ like this:

```
void arbb_matrix_vector(const dense<f32, 2>& a,
                        const dense<f32>& x,
                        dense<f32>& b) {
  b = add_reduce(a * repeat_row(x, a.num_rows()));
}
```

This function takes a dense matrix (a two dimensional array), `a`, and a dense vector, `x`, and returns a dense vector `b`. It uses `add_reduce` and `repeat_row`, which are built in ArBB operations. Note that we are not provided with a general reduction operator that takes a function as parameter, but rather with specific instances.

The matrix-vector multiplication code above is very concise, but some further steps are necessary before it can be run on real data. The code in figure 4.5 shows how to set up a 4x4 scaling matrix and a vector to multiply by that matrix using ArBB. This entails allocating memory on the ArBB heap and copying data into it, reminiscent of the copying of data from host to device that is common in general purpose GPU (GPGPU) programming .

After setting up and copying the data into ArBB, `arbb_matrix_vector` is *called* using the `call` command. At this point, a number of things happen. If the function is called for the first time, it is *captured*. This means that the function is run as a C++ function, which produces an intermediate representation (IR) of the computation (which is standard in such embeddings). For example, if the function being captured contains a normal C++ *for loop*, it will be unrolled, much in the same way as recursion in a Haskell embedded DSL results in unrolled code. After the function is captured, the IR is further optimised and finally executed. Compilation and capture only occur the first time a function is called. The compilation procedure is outlined in [NSL+11].

Nested vectors in ArBB allow the creation of an array of dense vectors of varying length – giving the ability to express less regular parallelism. There is a limit, though, in that only one level of such nesting is allowed. Section 4.2.5 shows a small example that uses this nestedness in EmbArBB to implement sparse matrix vector multiplication. The program in ArBB itself is very similar. We note, though, that one of the samples distributed with ArBB is also sparse matrix vector multiplication, but implemented using dense vectors and a function that can take sections of a vector. We will need to experiment further with nestedness and when it should be used.

4.2.5 Programming in EmbArBB

In EmbArBB, ArBB functions are expressed as Haskell functions on expressions (abstract syntax trees). This is standard Haskell embedded language procedure. A function that adds a constant to all elements of a vector is

```
int main(void)
{
  float matrix[4][4] = {{2.0,0.0,0.0,0.0},
                        {0.0,2.0,0.0,0.0},
                        {0.0,0.0,2.0,0.0},
                        {0.0,0.0,0.0,2.0}};
  float vector[4] = {1.0,2.0,3.0,4.0};

  // set up the matrix in ArBB
  dense<f32, 2> a(4, 4);
  range<f32> write_a = a.write_only_range();
  float* a_ = &write_a[0];
  memcpy(a_,matrix,16*sizeof(float));

  // set up the vector in ArBB
  dense<f32> x(4);
  range<f32> write_x = x.write_only_range();
  float* x_ = &write_x[0];
  memcpy(x_, vector,4*sizeof(float));

  // Room for the result
  dense<f32> arbb_b(4);
  const_range<f32> read_arbb_b;

  call(arbb_matrix_vector)(a, x, arbb_b);
  read_arbb_b = arbb_b.read_only_range();

  for (int i = 0; i < 4; ++i) {
    printf("%f ", read_arbb_b[i]);
  }

  return 0;
}
```

Figure 4.5: ArBB glue code.

```
addconst :: Num a
          => Exp a
          -> Exp (DVector Dim1 a)
          -> Exp (DVector Dim1 a)
addconst s v = v + ss
    where
      ss = constVector (length v) s
```

In this function `(+)` is used for elementwise addition of two vectors. The function `constVector` is part of the EmbArBB library and creates a vector of a given length containing the same value at each index.

EmbArBB supports one, two and three dimensional dense vectors. These vectors are represented by a datatype called `DVector` (for Dense Vector). `DVector` takes two arguments, the first specifying its dimensionality and the second its payload type. Hence, an array of 32 bit words has type `DVector Dim1 Word32`. The one dimensional `DVector` also has the alias `Vector`. The type of the `addconst` function above could also have been written:

```
addconst :: Num a
          => Exp a
          -> Exp (Vector a)
          -> Exp (Vector a)
```

The nested vectors of ArBB correspond to `NVectors` in EmbArBB.

The `addconst` function needs to be *captured* before it can be executed. The term *capture* is borrowed from ArBB nomenclature and corresponds to compiling the embedded language function into an ArBB function. In EmbArBB, a function is captured using the function `capture`; in the C++ embedding, a function is captured when it is called for the first time. The same could have been done in the Haskell embedding of course, but we chose to make capture explicit for simplicity. The `capture` function takes a Haskell function as input and produces an opaque typed function identifier as output. The identifier points out the function in an environment that is managed by a monad called `ArBB`. To make this concrete, Figure 4.6 shows the complete procedure from capturing to execution.

The `withArBB` function (line two) is the "run"-function of the `ArBB` monad. It turns something of type `(ArBB a)` into `(IO a)`. This is how ArBB computations are interfaced with Haskell. A `withArBB` session also sets the scope for any captured functions or vectors created in the ArBB monad.

The result of `capture addconst` on line four in the code has type

```
Function (BEScalar Float :- BEDVector Dim1 Float)
         (BEDVector Dim1 Float)
```

representing a function that takes a float and a vector of floats as input and gives a vector of floats as result. The "BE" prefixes in those names (`BEScalar`,`BEDvector`) refers to this being vectors and scalar that reside in the ArBB heap (Backend vectors). BEVectors and Scalars are mutable and the vector used to store the result needs to be allocated by the programmer. This interface seems rather imperative but ArBB requires output data to be placed into an output vector of the correct size. Because of this, the options were either to try to infer result size (which can be dependent on the value of input data) or to have the programmer supply storage for the result. We chose the latter, and thus also avoid additional runtime overhead due to size inference.

Continuing with the example: the function, once captured, can be executed on data. On line five, a `DVector` is copied into ArBB using the `copyIn` function. A vector to hold the result is created using the `new` function on line eight. The `new` function takes a shape description (`(Z :. 10)`, meaning a one dimensional vector of length 10) and an element to fill the vector with initially. A scalar `c` is created using `mkScalar` on line nine. Next, on line ten, the function is executed by issuing

```
execute f (c :- x) r1
```

The `(c :- x)` is a heterogeneous list of inputs. The `:-` operator is similar to normal cons (`:`) on Haskell lists. The output is stored in `r1`. Had there been more outputs, the output list would have also had a heterogeneous list type of the form `(r1 :- ... :- rn)`. If the result is a scalar, a storage location can be created using a function called `mkScalar` that takes its initial value as input.

All that remains is to copy the output vector out of ArBB and show the result; this is done on lines eleven and twelve in the figure.

In this example, we have seen all of the important parts of the Haskell to EmbArBB interface: how embedded language functions are compiled and executed, and how to transfer data from Haskell into the ArBB world. The following subsections show EmbArBB versions of some common data parallel computations.

In the examples one can assume that following imports have been made:

```
import Data.Vector as V
import Prelude as P
```

In this way, wherever there is potential for mixup between EmbArBB functions and Prelude or Data.Vector functions, these are prefixed by either "`V.`" or "`P.`".

```
1  main =
2    withArBB $
3    do
4       f <- capture addconst
5       x <- copyIn $ mkDVector
6                       (V.fromList [1..10 :: Float])
7                       (Z:.10)
8       r1 <- new (Z:.10) 0
9       c <- mkScalar 1
10      execute f (c :- x)  r1
11      r <- copyOut r1
12      liftIO$ putStrLn$ show r
```

Figure 4.6: The code shows how to capture a function, upload data to ArBB and how to execute the captured function.

```
-- Scatter values into a vector
-- resolves collisions by adding elements
addMerge :: Exp (DVector (t:.Int) USize) -- indices
         -> Exp USize                    -- res length
         -> Exp (DVector (t:.Int) a)     -- src
         -> Exp (DVector (t:.Int) a)

-- Reduce a vector using addition
addReduce :: Num a
          => Exp USize   -- rows, cols or pages
          -> Exp (DVector (t:.Int) a)
          -> Exp (DVector t a)

-- Segmented version of addReduce
addReduceSeg :: Num a
             => Exp (NVector a)  -- nested input
             -> Exp (DVector Dim1 a)

-- Create a nested vector from a dense
applyNesting :: Exp USize  -- lengths or offsets
             -> Exp (DVector Dim1 USize) -- nesting
             -> Exp (DVector Dim1 a)
             -> Exp (NVector a)


-- Create a vector with a constant value
constVector :: Exp USize  -- length
            -> Exp a      -- value
            -> Exp (DVector Dim1 a)
```

Figure 4.7: A list of EmbArBB functions that are used in the examples with short descriptions, part 1.

```
-- Extract a column from a 2D vector
extractCol :: Exp USize              -- col. index
           -> Exp (DVector Dim2 a)
           -> Exp (Vector a)

-- Fill a portion of a vector with a constant
fill :: Exp a                -- fill value
     -> Exp USize            -- start
     -> Exp USize            -- end
     -> Exp (DVector Dim1 a) -- dst
     -> Exp (DVector Dim1 a)

-- Gather elements from a vectors
gather1D :: Exp (DVector Dim1 USize)  -- indices
         -> Exp a                     -- default
         -> Exp (DVector Dim1 a)      -- values
         -> Exp (DVector Dim1 a)

-- Get the number of rows
getNRows :: Exp (DVector (t:.Int:.Int) a)
         -> Exp USize

-- Turn a zero dimensional vector into a scalar
index0 :: Exp (DVector Z a) -> Exp a

-- Create a 2D vector by repeating a 1D vector
repeatRow :: Exp USize            -- #repetitions
          -> Exp (DVector Dim1 a) -- row
          -> Exp (DVector Dim2 a)

-- Replace one column in a 2D vector
replaceCol :: Exp USize            -- col
           -> Exp (DVector Dim1 a) -- new values
           -> Exp (DVector Dim2 a)
           -> Exp (DVector Dim2 a)

-- Convert every element of a vector to USize
vecToUSize :: Exp (Vector a)
           -> Exp (Vector USize)
```

Figure 4.8: A list of EmbArBB functions that are used in the examples with short descriptions, part 2.

- `ISize` is an integer type used to specify for example offsets.
- `USize` is an unsigned integer type used for lengths or indices.
- `Boolean` replaces the `Bool` type for EmbArBB programs. The reason for this is that ArBB internally represents booleans as 8bit words. while the `Storable` instance for `Bool` does not.
- `DVector` is the type of regular shaped vectors.
- `NVector` is the type of irregularly shaped vectors.

Figure 4.9: A list of EmbArBB types with short descriptions

Saxpy

Saxpy is a vector operation that gets its name from the description of what it does "*S*ingle-precision *A*lpha *X P*lus *Y*". There is also a double-precision version called *daxpy*. Here we can use a single source to obtain both versions:

```
saxpy :: Num a
      => Exp a
      -> Exp (DVector Dim1 a)
      -> Exp (DVector Dim1 a)
      -> Exp (DVector Dim1 a)
saxpy s x y = (ss*x) + y
    where
      ss = constVector (length x) s
```

Now, `capture` can be used to instantiate the function at either float or double types.

```
main = withArBB $
  do
     f <- capture saxpy

     let v1 = V.fromList [1,3..10::Float]
         v2 = V.fromList [2,4..10::Float]

     x <- copyIn $ mkDVector v1 (Z:.5)
     y <- copyIn $ mkDVector v2 (Z:.5)

     r1 <- new (Z:.5) 0

     c <- mkScalar 1

     execute f (c :- x :- y)  r1

     r <- copyOut r1

     liftIO$ putStrLn$ show r
```

Changing `Float` to `Double` in the definitions of `v1` and `v2` gives the double precision version of the function. A function needs to be captured at every type at which it is to be used. This is because the resulting ArBB function is not polymorphic

over something corresponding to `Num` types (while the embedded language function is).

Matrix-vector multiplication

In section 4.2.4, matrix-vector multiplication was shown using the C++ ArBB interface. Here, EmbArBB is used to implement the same function. The function definition itself is very similar to the C++ one:

```
matVec :: Exp (DVector Dim2 Float)
        -> Exp (DVector Dim1 Float)
        -> Exp (DVector Dim1 Float)
matVec m v = addReduce rows
           $ m * (repeatRow (getNRows m) v)
```

The `addReduce` function takes a parameter that specifies whether to reduce rows or columns.

The following Haskell `main` function completes the comparison to the C++ version shown in the Introduction:

```
main = withArBB $
  do
     f <- capture matVec
     let m1 = V.fromList [2,0,0,0,
                          0,2,0,0,
                          0,0,2,0,
                          0,0,0,2]
         v1 = V.fromList [1,2,3,4]

     m <- copyIn $ mkDVector m1 (Z:.4:.4)
     v <- copyIn $ mkDVector v1 (Z:.4)

     r1 <- new (Z:.4) 0

     execute f (m :- v)  r1

     r <- copyOut r1

     liftIO$ putStrLn$ show r
```

The complete Haskell version of this program is slightly shorter than the corresponding C++ version. However, the C++ version could probably be made shorter as well; the Haskell and C++ versions must be considered similar in implementation complexity.

Matrix-matrix multiplication

The following EmbArBB implementation of matrix-matrix multiplication is used as a benchmark in section 4.2.7:

```
matmul :: Exp (DVector Dim2 Float)
        -> Exp (DVector Dim2 Float)
        -> Exp (DVector Dim2 Float)
matmul a b = fst $ while cond body (a,0)
  where
    m = getNRows a
    n = getNCols b
    cond (c,i) = i <* n
    body (c,i) =
      let mult = a * repeatRow m (extractCol i b)
          col  = addReduce rows mult
      in (replaceCol i col c, i+1)
```

This example uses a `while` loop. In the C++ embedding of ArBB, the programmer has the option of using a C++ `while` loop or a special `while_` loop. The normal C++ while loop unrolls at capture time; the `while_` is kept by ArBB and corresponds to an ArBB loop. The two loops differ in the kinds of values on which they can depend. The C++ loop can only depend on C++ values, so the number of iterations in a normal `while` loop must be known at capture time. All of these concepts have Haskell counterparts. The `while` loop used in the example corresponds to the `while_` loop and will remain in the generated ArBB code; it can depend on ArBB values at runtime. Haskell recursion is used to get the kind of unrolling that one gets by using a C++ `while` loop.

The EmbArBB `while` loop takes three parameters, a condition, a body and an initial state. In every iteration of the loop, the body is applied to the state and the condition checked.

Histogram

The histogram of a grayscale image contains the frequency with which each shade occurs in the image. In this example, the image used represents the shade of each

pixel with a single byte, so that 256 different shades are possible. An array of length 256 is created to hold at index `i` the number of occurrences of the shade `i` in the image. In essence, this is an array of buckets.

```
histogram :: Exp (DVector Dim2 Word8)
          -> Exp (Vector Word32)
histogram input = addMerge (vecToUSize flat) 256 cv
    where
      flat = flatten input
      cv = constVector (r * c) 1
      r  = getNRows input
      c  = getNCols input
```

The `addMerge` operation takes a vector of inputs, a vector of indices and finally a size (specifying the result size). Elements from the input vector (`cv` in the example) are placed into the result at the index specified at the same location in the indices vector. Elements placed at the same index are added together. Here, the input vector contains all ones, while the image is *cast* into a vector of indices. The result is that index `i` of the output vector contains the number of times shade `i` appears in the image. The histogram computation becomes very concise through the use of the `addMerge` operation.

The code below creates the image shown in figure 4.10. It takes as input a vector on frequencies and outputs a two dimensional vector, a grayscale image.

Figure 4.10: The righ-hand image visualises the frequency with which different shades of gray occur in the left-hand one.

```
histImage :: Exp (Vector Word32)
          -> Exp (DVector Dim2 Word8)
histImage input = fst $ while cond body (cvn,0)
    where
      cond (img,i) = i <* n
      body (img,i) = (replaceCol i col' img,i+1)
          where
            val = input ! i
            col = extractCol i img
            col' = fill black 0 n col
            n = 255 - scale 255 m val

      n = length input
      cv = constVector (n*n) white
      cvn = setRegularNesting2D n n cv
      m   = index0 (maxReduce rows input)
      black = 0
      white = 255
```

Sobel edge detection filter

Sobel edge detection is an example of a stencil computation over an array. At each element of the array, a computation is performed that depends on that element and on elements close by. Which of the nearby elements to use, and how much they influence the result is typically described using a matrix (in the two dimensional case) called the stencil. Below are two stencils used in the sobel edge detection filter:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

In ArBB, the *map* operation gives the programmer a way to implement stencil computation. It takes a function and a vector on which to apply the function at each element. Inside the function being mapped, the programmer may use a set of *getNeighbor* functions to access nearby elements.

The elements used by the stencils are kept (in the form of coordinates relative to the centre point of the matrix) in two Haskell lists. This is an example of how host language (Haskell) features are used as a kind of scripting aid.

```
s1, s2 :: [(Exp ISize,Exp ISize)]
s1 = [(1,1),(0,1),(-1,1),(1,-1),(0,-1),(-1,-1)]
s2 = [(1,1),(1,0),(1,-1),(-1,1),(-1,0),(-1,-1)]
```

The actual weights are placed in a separate list.

```
coeffs :: [Exp Float]
coeffs = [-1,-2,-1,1,2,1]
```

The following functions implement the stencil computation part of the program using the elements pointed out by `s1` and `s2` together with the weights in `coeffs`. Each of the six neighbours is multiplied by the appropriate weight. Haskell's `foldl` function is used to sum up the values, to give the result for the location in question. Using a Haskell fold means that the summation is unrolled.

```
gx :: Exp Word8 -> Exp Float
gx x = P.foldl (+) 0
     $ P.zipWith (*) [toFloat (getNeighbor2D x a b)
                      / 255
                     | (a,b) <- s1] coeffs
```

```
gy :: Exp Word8 -> Exp Float
gy x = P.foldl (+) 0
     $ P.zipWith (*) [toFloat (getNeighbor2D x a b)
                      / 255
                     | (a,b) <- s2] coeffs
```

The helper functions `convertToWord8` and `clamp` take care of converting floats between 0 and 1 to bytes between 0 and 255, and of clamping floating point values into the 0 to 1 range.

```
convertToWord8 :: Exp Float -> Exp Word8
convertToWord8 x = toWord8 $ (clamp x)  * 255

clamp :: Exp Float -> Exp Float
clamp x = max 0 (min x 1)
```

The complete kernel, the program that is to be executed at every index of the image, is given below. The earlier parts are brought together into a function that transforms one `Word8` into another.

```
kernel :: Exp Word8 -> Exp Word8
kernel x = convertToWord8 $ body x
  where
    body x = sqrt (x' * x' + y' * y')
     where
       x' = gx x
       y' = gy x
```

In order to execute the kernel, an ArBB `map` is used.

```
sobel :: Exp (DVector Dim2 Word8)
         -> Exp (DVector Dim2 Word8)
sobel image = map kernel image
```

The implementation of `sobel` is pleasingly simple and concise in this setting. Despite the fact that this is a small example, the value of being able to use Haskell lists and operations is already becoming obvious.

Figure 4.11: The image on the right shows the result of applying the sobel edge detection filter to that on the left.

Image filtering

The blur operation, which is a kind of spatial linear filter, shows another way to implement a stencil operation. The `mapStencil` function used is a composite function in the EmbArBB library; it is not provided as a primitive from the ArBB VM.

```
blur :: Exp (DVector Dim2 Word8)
        -> Exp (DVector Dim2 Word8)
blur image = vec2DToWord8 (res `div`  all16)
  where
    all16 = constVector2D (getNRows image)
                          (getNCols image)
                          16
    res = mapStencil (Stencil [1,2,1
                              ,2,4,2
                              ,1,2,1] (Z:.3:.3))
                     image'

    image' = vec2DToWord32 image
```

In Repa, similar `mapStencil` functionality is present but using Template Haskell to give a safer way to specify the stencil. In our version there is no protection agains mischievous stencil specification. This guarantees only that if the programmer says the stencil is two dimensional then it can only be applied to two dimensional vectors. Applying similar Template Haskell based extension here as well is probably quite easy now that Repa has shown how.

Sparse matrix multiplication

A sparse matrix can be represented by three vectors, one (cidx) containing column indices, one (offsets) indicating indices in the vector of values at which the first non-zero element of each row appears, and one containing the values themselves. This is known as the Compressed Sparse Row (CSR) format. Given such a matrix and a dense vector, the relevant elements of the dense vector can be gathered into a new vector using the column indices. The resuting vector is multiplied pointwise by the vector of non-zero values from the sparse matrix. All that remains, then, is to divide the result into rows and sum each one. The offsets vector points to where the division should happen. The code in EmbArBB is as follows:

```
smvm :: Exp (Vector USize)
     -> Exp (Vector USize)
     -> Exp (Vector Float)
     -> Exp (Vector Float)
smvm mval cidx os vec = addReduceSeg nps
    where
      ps = mval * gather1D cidx 0 vec
      nps = applyNesting offsets os ps
```

The multiplication acts pointwise on the elements of `vals` and the vector (of the same length) containing relevant elements of the dense vector (produced using `gather1D`). The function `applyNesting` produces a nested vector of rows. The function `addReduceSeg` sums each of the rows (or segments) in that array, producing the non-zero elements of the result vector. Note that there is irregular data parallelism here because the rows may have different lengths. We would hope to get parallelism both in the individual summations and between summations. This is a well known algorithm that can be traced back to Blelloch [Ble96].

4.2.6 Implementation

EmbArBB is a deeply embedded language. There is a constructor for every operation that ArBB can perform. A deep embedding is useful when there is need to apply

optimisations or transformations to the AST before executing the operations. However, EmbArBB currently relies entirely on ArBB to perform optimisations to the program. The only optimisation performed on the EmbArBB side is sharing detection. Detecting sharing reduces the number of calls into the ArBB C library. Should we add a GPU backend to EmbArBB, further GPU specific optimisations of the AST will likely become necessary.

Vectors

EmbArBB has support for one, two and three dimensional dense vectors, represented by a datatype called `DVector`.

```
import qualified Data.Vector.Storable as V

data DVector d a
  = Vector {vectorData  :: V.Vector a,
            vectorShape :: d}
```

The payload data in a `DVector` is stored in a vector from the `Data.Vector` library. There is also a `d` parameter that specifies the shape of the DVector.

The `d` parameter is used for a type level representation of the dimensionality of a vector, as in the Repa library [KCL+10]. The dimensionality is encoded using the following types together with the `Int` type.

```
data a :. b = a :. b
infixl :.

data Z = Z
```

The dimensions zero to three are represented as follows:

```
type Dim0 = Z
type Dim1 = Dim0 :. Int
type Dim2 = Dim1 :. Int
type Dim3 = Dim2 :. Int
```

For example in the reduction functions provided by EmbArBB the output vector is of a dimensionality one less than the input vector used. Below is the type of `addReduce` to illustrate this.

```
addReduce :: Num a
          => Exp USize
          -> Exp (DVector (t:.Int) a)
          -> Exp (DVector t a)
```

A somewhat unfortunate side effect of this is that currently EmbArBB has two kinds of scalars, `Exp (DVector Dim0 a)` and `Exp a`. The result of reducing a one-dimensional vector is a zero-dimensional vector. An operation called `index0` converts from zero-dimensional vectors to scalars.

Irregular container, in ArBB called Nested vectors, are represented in EmbArBB by the type `NVector`. Currently there are no versions of the `copyIn`, `copyOut` or `new` functions for nested vectors. The programmer must transfer dense vectors into the ArBB heap and then apply nesting to them as part of the computation to perform thereupon. This means that having a concrete representation for a NVector is currently not useful. In the hope of being able to implement some of the transfer functions, even without direct support from the ArBB VM, we represent NVectors as a vector of dense data together with a vector containing segment lengths.

```
data NVector a =
       NVector { nVectorData     :: V.Vector a
               , nVectorNesting  :: V.Vector USize}
```

As part of the interface between Haskell and EmbArBB there are also mutable vectors, of a type called `BEDVector`. These are vectors that reside in the ArBB heap and are represented only by an integer identifier with which the actual data can be accessed from ArBB. These are used to store the inputs and outputs of calls to `execute` on a captured function.

```
data BEDVector d a =
       BEDVector { beDVectorID :: Integer
                 , beDVectorShape :: d}
```

New `BEVector`s are created using the function `new`. The function `copyIn` copies a `DVector` from Haskell to the ArBB heap and there is a function `copyOut` to retrieve data from ArBB.

The language

The EmbArBB language is implemented as a set of library functions, operating on an expression datatype:

```
data Expr = Lit Literal
          | Var Variable

          | Index0 Expr
          | ResIndex Expr Int

          | Call (R GenRecord) [Expr]
          | Map  (R GenRecord) [Expr]

          | While ([Expr] -> Expr)
                  ([Expr] -> [Expr])
                  [Expr]

          | If Expr Expr Expr
          | Op Op [Expr]
```

Most of the ArBB functionality is taken care of by the `Op` constructor. The datatype `Op` used to represent operations has over 120 constructors, so only a selection is shown in figure 4.12. Having all these 120+ operations taken care of by one constructor in the expression datatype simplifies the implementation of the backend, since all of these operations are handled in a very similar way. The few remaining special ArBB capabilities such as loops, and function mapping and calling are handled by their own cases, discussed below. There are also some implementation specific details that result in their own constructors in the `Expr` type, namely `Index0` for turning a zero dimensional DVector into a scalar, and `resIndex` that helps with implementing operations that have more than one result. An example of an such an operation is `SortRank`, which returns both the sorted result of an input vector and a vector of indices that specifies a permutation that would have sorted the input vector.

The `Expr` type also has constructors for the *call* and *map* functionality. To call a function means to apply it to input data. The `map` operation specifies elementwise application of a function over vectors, like NESL's *apply-to-each* [Ble96], so it corresponds to Haskell's `map` and `zipWith`. Both `Call` and `Map` take a `(R GenRecord)`. This is inspired by the delayed expressions that enable the implementation of `vapply` in Nikola [MM10]. The `R` is the reification monad used to create DAGs (directed acyclic graphs) from embedded language functions. These DAGs are part of a `GenRecord` that contains all information that the ArBB code generator needs to generate the ArBB function.

The `While` loop is represented using higher order abstract syntax, that is using functions to represent the condition and body. The state is represented by a list of expressions; this needs to be generalised somewhat in order to support loops with

```
data Op =
   -- elementwise and scalar
     Add | Sub | Mul | Div | Max | Min
   | Sin | Cos | Exp
   ...

   -- operations on vectors
   | Gather | Scatter | Shuffle | Unshuffle
   | RepeatRow | RepeatCol | RepeatPage
   | Rotate | Reverse | Length | Sort
   | AddReduce | AddScan | AddMerge
   ...
```

Figure 4.12: ArBB scalar, elementwise and vector operations, which are handled by the `Op` constructor in the `Expr` datatype. This is just a selection from the more than 120 different operations ArBB provides.

general tuples in the state. Something more structured than a list is needed for this.

The expression data type used in EmbArBB is untyped (not using a GADT). A typed interface to the language is supplied using the same phantom types method as used in *pan* [EFdM03] and many other Haskell embedded languages since then. The choice to use an untyped `Expr` datatype was based on the wish to keep the backend (ArBB code generation) as simple as possible.

```
-- Phantom types
type Exp a = E Expr
```

As an example, the operation `addReduce`, which reduces a vector across a specified dimension, is implemented as follows in the EmbArBB library:

```
addReduce :: Num a
           => Exp USize
           -> Exp (DVector (t:.Int) a)
           -> Exp (DVector t a)
addReduce (E lev) (E vec) =
  E $ Op AddReduce [vec,lev]
```

Interfacing with Haskell and Code generation

The interface between ArBB and Haskell consists of the `DVector`, `NVector`, `BEDVector` and `BEScalar` types, the `capture`, and `execute` functions, and the ArBB monad with its `withArBB` "run"-function. This section describes what happens when the programmer captures an embedded language function, and when `execute` is called on a captured function.

Capture and execution of functions takes place in the `ArBB` monad, which manages state of type `ArBBState`:

```
data ArBBState =
  ArBBState
    { arbbFunMap :: Map.Map Integer
                            ArBBFun
    , arbbVarMap :: Map.Map Integer
                            VM.Variable
    , arbbUnique :: Integer }

type ArBBFun = (VM.ConvFunction, [Type], [Type])
```

This state contains a map from function names to ArBB functions and their input and output types. The `VM.ConvFunction` is how ArBB functions are represented by the ArBB-VM bindings. There is a map from vector and scalar IDs to their corresponding ArBB variables. The last item in `ArBBState` is an Integer that is used to generate new function names and variable IDs as the programmer captures more functions or creates new arrays on the ArBB heap. Now, the `ArBB` monad is defined as:

```
type ArBB a = StateT ArBBState VM.EmitArbb a
```

`VM.EmitArbb` is also a concept from the virtual machine bindings. It manages low-level functions (the `VM.ConvFunction` functions) and implements an interface to the low level ArBB-VM API. `EmitArBB` is the ArBB code generating monad from the arbb-vm bindings. For more or less everything that can be done with the arbb-vm bindings, there is a function of the form

```
f :: arg1 -> ... argn -> EmitArBB out
```

For example, for generating an operation node (such as +) in the ArBB IR, there is a function of type

```
op_ :: Opcode
     -> [Variable]
     -> [Variable]
     -> EmitArbb ()
```

while the function for generating a while loop in the ArBB IR has type

```
while_ :: (EmitArbb Variable)
        -> EmitArbb a
        -> EmitArbb a
```

The details of the translation are omitted for brevity, as the approach is standard.

When a function `f` of type

```
Exp tin1 -> ... -> Exp tinN -> Exp tout
```

is captured, it is first applied to expressions that represent variable names. For each of the inputs, `(tin1,..,tinN)`, a variable is created. The result is an expression (or expressions) representing the function `f`. On this expression, sharing detection is performed, and a directed acyclic graph (DAG) is created. The method of sharing detection used is based on the StableNames method [Gil09].

Then, the code generation is implemented using a very direct approach; no extra optimisations or transformations are applied. This is a reasonable choice, since the whole point of ArBB is that the built-in JIT compiler knows and performs architecture specific optimisations. Sharing detection on the Haskell side makes sense because code generation for each node in the DAG results in at least two calls into the ArBB-VM API, which means going through the FFI and incurring the associated cost. Detecting the sharing already in the host language should give a smaller workload for the ArBB JIT compiler, thus reducing the time spent on JITing. It remains to be seen how important JIT cost will be in practice, however, as we expect it to be amortised over a large number of executions of the JITed code. We will need to conduct experiments with a suite of larger examples in order to decide if sharing detection on the Haskell side is worthwhile.

Applying `capture` to a function gives an object of type

```
type FunctionID = Integer

data Function i o = Function FunctionID
```

The `i` and `o` parameters to `Function` represent the input and output types of the captured function. As an example, capturing

```
f :: Exp (DVector Dim1 Word32)
   -> Exp (DVector Dim1 Word32)
```

results in an object of type

```
Function (BEDVector Dim1 Word32)
         (BEDVector Dim1 Word32)
```

This is just phantom types placed over a function name that is just a `String`, but it does offer a typed interface for the `capture` and `execute` functions.

The `execute` function that launches a captured function takes a `Function i o` object, inputs of type `i` and outputs of type `o`. The function name is looked up in the `ArBB` environment (the monad). The inputs and outputs are also looked up in the environment and then the function is executed.

```
execute :: (VariableList a, VariableList b)
         => Function a b -> a -> b -> ArBB ()
execute (Function fid) a b =
  do
    (ArBBState mf mv _) <- S.get
    case Map.lookup fid  mf of
      Nothing -> error "execute: Invalid function"
      (Just (f,tins,touts)) ->
        do
          ins <- vlist a
          outs <- vlist b

          liftVM$ VM.execute_ f outs ins
```

The `vlist` function goes through the heterogeneous list of inputs or outputs and looks up each of the elements in the `arbbVarMap`; the result is a Haskell list of `VM.Variable`. The function `VM.execute_` is part of the virtual machine API bindings, and corresponds directly to a C function in that library.

4.2.7 Benchmarks

In this section, matrix multiply, Sobel edge detection and an image blur algorithm are used as benchmarks in comparing sequential C++ code, ArBB, EmbArBB, and a Haskell library called Repa. Repa provides regular shape polymorphic arrays; it permits parallel execution of the resulting code, making use of multiple cores, but not of SIMD parallelism [KCL$^+$10]. The Repa versions of the benchmarks used in the comparison come from the `repa-examples-3.2.1.1` package on Hackage.

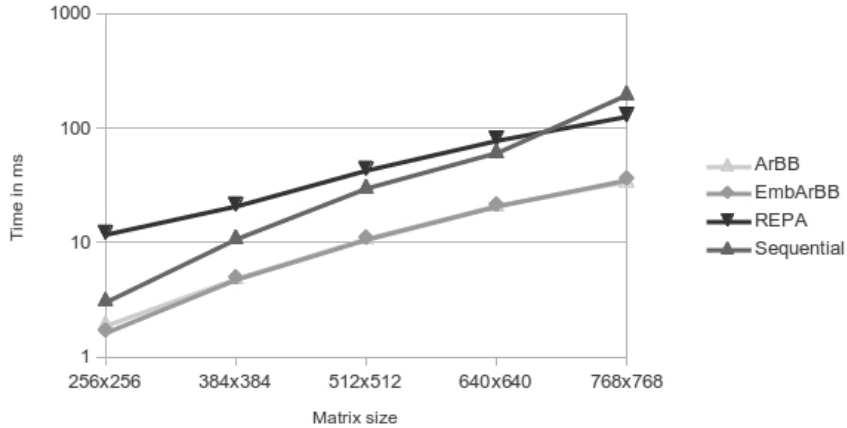

Figure 4.13: Shows in a log-scale the execution time of matrix-matrix multiplication comparing ArBB, EmbArBB, Repa and sequential C++. The ArBB and EmbArBB lines are indistinguishable.

The processor used for all measurements is a four core Intel Core-I7 930 at 2.80Ghz.

matrix-matrix multiplication

The matrix multiplication benchmarks consist of square matrices of sizes 256x256, 384x384, 512x512, 640x640 and 768x768. Figure 4.13 shows the runtimes for the four implementations being compared, using the best settings in numbers of cores or threads found by previous experiments. The sequential C++ and ArBB versions come directly from the ArBB distribution.

Sobel edge detection

In the Repa Sobel code, only the applications of the stencils are timed. This part is defined as follows

```
gradientX :: Monad m => Image -> m Image
gradientX img
        = computeP
        $ forStencil2 (BoundConst 0) img
              [stencil2| -1  0  1
                         -2  0  2
                         -1  0  1 |]

gradientY :: Monad m => Image -> m Image
gradientY img
        = computeP
        $ forStencil2 (BoundConst 0) img
             [stencil2|  1  2  1
                         0  0  0
                        -1 -2 -1 |]
```

A corresponding EmbArBB code to use in this comparison was implemented.

```
gx :: Exp (DVector Dim2 Float)
    -> Exp (DVector Dim2 Float)
gx = mapStencil
         (Stencil [-1,0,1
                  ,-2,0,2
                  ,-1,0,1] (Z:.3:.3))

gy :: Exp (DVector Dim2 Float)
    -> Exp (DVector Dim2 Float)
gy = mapStencil
         (Stencil [ 1, 2, 1
                  , 0, 0, 0
                  ,-1,-2,-1] (Z:.3:.3))
```

We compare these two examples, in which only the applications of the stencils are timed; we also time the complete Sobel implementation in EmbArBB (shown in section 4.2.5). The Sobel benchmark is run om images of sizes 256x256, 512x512, 1024x1024, 2048x2048 and 4096x4096. Again, Figure 4.14 shows that EmbArBB performs well. As we had expected, the Haskell embedding, once a function is captured, seems to impose little or no overhead compared to the C++ implementation.

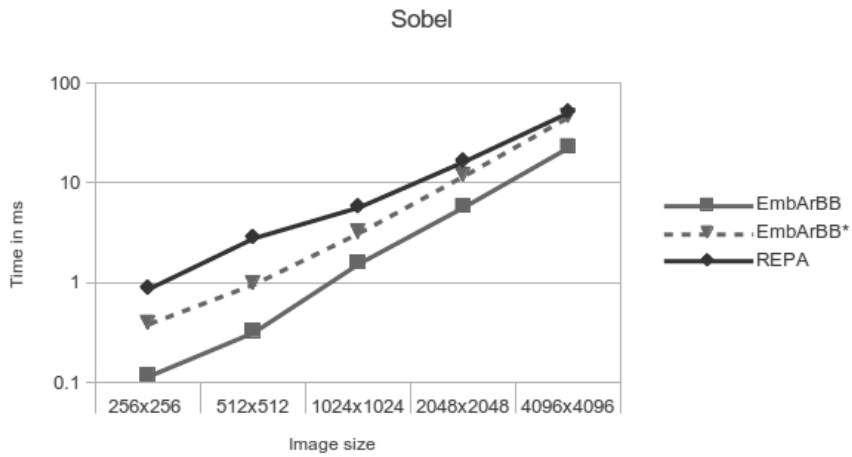

Figure 4.14: Shows in a log-scale the execution time of a key part of the sobel edge detection program. The chart compares EmbArBB to REPA and also displays for reference the execution times of the full sobel program as implemented in section 4.2.5, called *EmbArBB** in the chart.

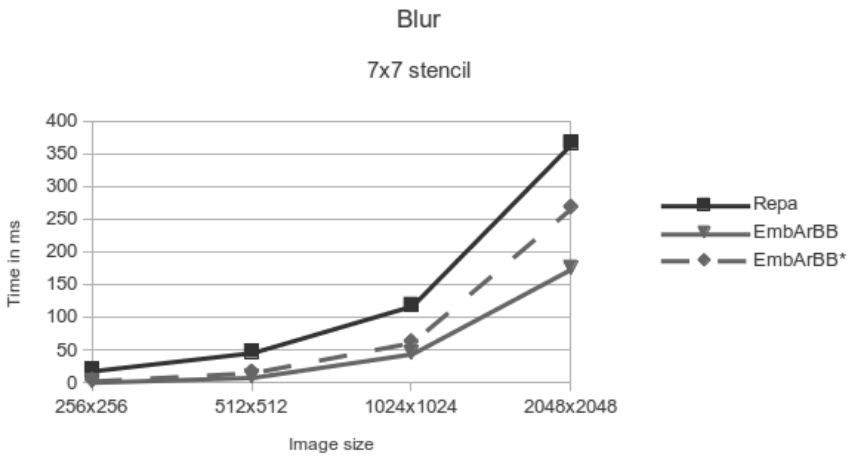

Figure 4.15: Shows the execution time of a 7x7 blur filter applied to images of various sizes.

Blur

The `blur` benchmark is performed using an algorithm similar to the example code in section 4.2.5 but with the following changes. The image used is in RGB color. This means that the stencil needs to be applied three times, once to each color plane. Also the image is converted into a form where each color intensity is represented by a `Double`.

Here as well as in the sobel case the Repa code from `repa-examples-3.2.1.1` times only the actual computational kernel (the application of the stencils) including conversion to Doubles. The corresponding EmbArBB part was broken out and timed separately as well. The chart 4.15 shows comparison of runtime for the key part of the computation but also adds the full execution time of the EmbArBB version (called *EmbArBB** in the chart). The full EmbArBB implementation of the blur filter including conversion into Doubles, image decomposition into R,G and B planes, application of the stencil and reconstructing a planar RGB image in the end.

	256x256	512x512	1024x1024	2048x2048
Repa 3x3	12	27	72	190
EmbArBB 3x3	1	2	20	77
EmbArBB* 3x3	2	9	33	153
Repa 5x5	13	32	87	254
EmbArBB 5x5	1	5	28	105
EmbArBB* 5x5	3	12	46	204
Repa 7x7	20	48	119	368
EmbArBB 7x7	2	10	46	176
EmbArBB* 7x7	4	17	63	270

Table 4.1: The table shows execution times (rounded to ms) for various image and stencil size combinations in both Repa and EmbArBB.

About the numbers

This section presents three benchmarks, two of which compare to Repa only and one that compares to Repa, ArBB in C++ and sequential C++ code. In all of these comparisons, JIT compilation time is excluded.

The comparison to ArBB in C++ shows that the Haskell embedding does not impose any extra overhead (at least not in this benchmark); this matched our expectations. More comparisons to the C++ version of ArBB are needed to confirm these first impressions about overhead in the Haskell embedding. If future directions of EmbArBB development develop techniques (such as size inference) that impose a runtime overhead, then the comparison in execution to the C++ version becomes more important.

The comparisons to Repa all show that the performance of EmbArBB compares favourably. This can be attributed to the way in which ArBB's developers at Intel have incorporated vectorisation and threading.

4.2.8 Future Work

The C++ embedding of ArBB allows for dense containers of structs in some cases. The operations on vectors supplied by the ArBB virtual machine are exclusively over vectors of scalar types. So the C++ embedding must be performing a AOS to SOA (Array of Struct to Struct of Array) transformation. The Haskell embedding does not implement any similar transformation. This is an important addition that would for example make implementing functions on complex numbers easier.

We stress that this paper presents first steps in the implementation of EmbArBB. Our benchmarks, while promising, are very limited. We must devote effort to devel-

oping a suite of interesting, larger data parallel programs for use in benchmarking EmbArBB. It is particularly important to explore the nested vectors, and the effects of the limitation to one level of nesting in ArBB. Those parts of ArBB that support nested vectors seem to be less well developed than those supporting dense vectors, as evidenced by the sample applications distributed with ArBB, none of which uses nesting. We expect ArBB to become more complete, and perhaps we will be able to contribute interesting examples both in the C++ and Haskell embeddings.

Having exercised EmbArBB more thoroughly, we will assess the results of the benchmarking and experiments with programming idioms, and decide on future research directions. We expect to focus on ways to provide users with an interface that is more functional in style than the current C++ oriented one.

4.2.9 Conclusion

We have shown that Intel's Array Building Blocks (ArBB) provides an interface that is well suited to functional programming. The programs are quite close to mathematical specifications, in the style of NESL [Ble96]. We have only just completed the embedding of the part of ArBB that deals with nested vectors, and we need to tackle many more case studies. In our case studies using dense vectors, ArBB seems to do a good job of efficiently using parallel hardware resources – both cores and vector units. By embedding ArBB, we can, with little implementation effort, provide quite an attractive data parallel programming language in Haskell. Our benchmarks are preliminary and small, but they show very good performance. Once we have completed the ArBB embedding, we will have an interesting platform on which to experiment with and develop new programming idioms that exploit the fact that we have a data parallel programming language embedded in an expressive, strongly typed host language. We feel that work in this area (as distinct from implementation methods) is overdue. We would be happy to receive suggestions for interesting case studies or collaborations.

Acknowledgments

Svensson was first introduced to ArBB while on a three month internship at Intel during 2011. Thanks go to Ryan R. Newton for inspiring supervision during that internship.

This research has been funded by the Swedish Foundation for Strategic Research (which funds the Resource Aware Functional Programming (RAW FP) Project) and by the Swedish Research Council.

Bibliography

[ACD+10] Emil Axelsson, Koen Claessen, Gergely Dévai, Zoltán Horváth, Karin Keijzer, Bo Lyckegård, Anders Persson, Mary Sheeran, Josef Svenningsson, and Andras Vajda. Feldspar: A domain specific language for digital signal processing algorithms. In *8th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, pages 169–178. IEEE Computer Society, 2010.

[Ble96] Guy Blelloch. Programming Parallel Algorithms. *Communications of the ACM*, 39(3), 1996.

[CKL+11] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell Array Codes with Multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, DAMP '11, pages 3–14, New York, NY, USA, 2011. ACM.

[EFdM03] Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(2), 2003.

[Gil09] Andy Gill. Type-Safe Observable Sharing in Haskell. In *Proceedings of the 2009 ACM SIGPLAN Haskell Symposium*, 09/2009 2009.

[HS12] John Hughes and Mary Sheeran. Teaching parallel functional programming at Chalmers. In *presentation at Trends in Functional Programming in Education, Workshop associated with Conf. on Trends in Functional Programming, St. Andrews*, 2012.

[Int] Intel. Intel(r) Array Building Blocks Virtual Machine Specification. `http://software.intel.com/sites/whatif/arbb/arbb_vm.pdf`.

[JLKC08] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M T Chakravarty. Harnessing the multicores: Nested data parallelism in haskell. In Ramesh Hariharan, Madhavan Mukund, and V Vinay, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2008)*, volume 2 of *Leibniz International Proceedings in Informatics*, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[KCL+10] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel

arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 261–272, New York, NY, USA, 2010. ACM.

[MM10] Geoffrey Mainland and Greg Morrisett. Nikola: Embedding Compiled GPU Functions in Haskell. In *Proceedings of the third ACM Haskell symposium*, pages 67–78. ACM, 2010.

[NSL+11] Chris J. Newburn, Byoungro So, Zhenying Liu, Michael McCool, Anwar Ghuloum, Stefanus Du Toit, Zhi Gang Wang, Zhao Hui Du, Yongjian Chen, Gansha Wu, Peng Guo, Zhanglin Liu, and Dan Zhang. Intel's array building blocks: A retargetable, dynamic compiler and embedded language. In *Proceedings of the 2011 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 224–235, Washington, DC, USA, 2011. IEEE Computer Society.

[She04] Mary Sheeran. Generating fast multipliers using clever circuits. In *Int. Conf. on Formal Methods in Computer Aided Design (FMCAD)*, volume 3312 of *LNCS*, pages 6–20, 2004.

[She11] Mary Sheeran. Functional and dynamic programming in the design of parallel prefix networks. *J. Funct. Program.*, 21(1):59–114, 2011.

[SN11] Bo Joel Svensson and Ryan Newton. Programming Future Parallel Architectures with Haskell and ArBB. `http://faspp.ac.upc.edu/faspp11/pdf/faspp11-final12.pdf`, 2011. Presented at the workshop: Future Architectural Support for Parallel Programming (FASPP), in conjuction with ISCA '11.