Thesis for the degree of Licentiate of Engineering

Obsidian: GPU Kernel Programming in Haskell

Joel Svensson

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University Of Technology and Göteborg University
Göteborg, Sweden 2011

Obsidian: GPU Kernel Programming in Haskell
Joel Svensson

Technical Report 77L
ISSN 1652-876X
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 2011

Abstract

Graphics Processing Units (GPUs) are evolving into powerful general purpose computing platforms. At first, GPU performance was driven by the requirements of 3D graphics computer games. To fit this workload, a GPU is a many-core processor suitable for the data-parallel programming paradigm. Today, GPUs come with hundreds of processing elements and a theoretical single precision floating point performance in the teraflop range.

Because of the computing power of modern GPUs, programmers are increasingly interested in making use of them for non-graphics applications. This desire has given rise to the research field that studies General Purpose Computations on GPUs (GPGPU). The manufacturers of GPUs are also acknowledging this trend and are tailoring their GPUs to meet both the desires of those playing games and the GPGPU community.

CUDA is NVIDIA's tool-set for GPGPU programming on their GPUs. CUDA is a big improvement for the GPGPU programmer compared to what was available before. In the early days, the GPGPU programmer was forced to express the algorithm being implemented as a computer graphics computation. CUDA provides a C compiler and a set of libraries for general purpose programming on the GPU, freeing the programmer from graphics APIs. In CUDA, the programmer decomposes the problem into a set of kernels. A kernel is an isolated data-parallel program executed by a number of threads on the GPU. CUDA has some problems. For example, CUDA is a very low level interface to the GPU capabilities and there is also the issue that CUDA kernels are not easily composable.

Obsidian is an embedded language for implementing kernels in the functional programming language Haskell. From higher level descriptions of algorithms based on combinators, CUDA code is generated. Using this approach, Obsidian kernels are more compositional and also relieve the programmer from inventing the typically complex index arithmetic expressions that are used to load and store data in data-parallel algorithms. The indexing arithmetic is hidden away from the programmer in the set of combinators provided as a library.

The performance obtained from the kernels generated using Obsidian is decent. It does not compare to optimized handwritten code but if the implementation effort is taken into consideration performance is good. Obsidian allows the programmer to think about the problem at hand, rather than being weighed down by the lower level details.

In this thesis, two different implementations of Obsidian are shown. The first of these implementations is based on monads and the second on arrows, two concepts familiar to functional programmers. A number of applications are presented, expressed using the arrow based version.

Acknowledgments

I extend thanks to my supervisor Mary Sheeran, for her support, insights and motivational abilities. I also thank her very much for reading this thesis many times and pointing out all those typically Swedish mistakes I have made while writing. I would also like to thank my co-supervisor Koen Claessen for all the technical help he has offered during the work leading up to this thesis. Without Koen's help and input, this thesis would not have been possible.

Thanks also go to the "graphics guys", Erik Sintorn, Markus Billeter and Ola Olsson. We do not meet often but I always enjoy our encounters very much. Their insights into graphics processor architecture and programming is a valuable source of information. I learn something new each time we meet. I also thank their supervisor, Ulf Assarsson, for giving awesome graphics courses and also being very helpful with graphics processor related questions.

Thanks to my room-mates Nicholas Smallbone and Michal Palka and to colleagues Anders Persson and Emil Axelsson. They are all intriguing persons with large wells of knowledge that I wish I could tap into more often.

Sean Lee and Trevor McDonell, both at the University of New South Wales when I first met them, had many interesting suggestions and comments about Obsidian. Some of their suggestions found their way into the implementation. Sean and Trevor are people I have often turned to with technical questions.

Thanks to everyone at the computer science and engineering department at Chalmers for creating a pleasant work atmosphere.

Finally, and most importantly thanks to my wife for being so supportive of my endeavors. She is very patient with my occasionally awkward working hours.

This research is funded by the Swedish Research Council.

Publications

This thesis is based on the work previously published as:

- GPGPU kernel implementation and refinement using Obsidian [36]
  Published 2010 in Procedia Computer Science.
- Obsidian: A Domain Specific Embedded Language for General-Purpose Parallel Programming of Graphics Processors [38]
  To appear in Springer Lecture Notes in Computer Science (nr: 5836).
- GPGPU Kernel Implementation using an Embedded Language: a Status Report [37]. Published 2010, technical report 2010:1
  Dept. of Computer Science and Engineering,
  Chalmers University of Technology.

Contents

List of Figures

Chapter 1

Introduction

1.1 Graphics Processing Unit: Supercomputer on a PCIe card

Graphics Processing Units (GPUs) today are many-core processors with floating point capabilities in the teraflop range. The performance development of GPUs is driven by the requirements of the graphics used by modern games. Many-core denotes a chip with a high number of small simple processing elements suitable for the fine grained data-parallelism typical of graphics applications. Current GPUs have several hundreds of processing elements on a single chip. For example, The NVIDIA Fermi based GPUs come in models of up to 512 processing elements [27]. The number of processing elements per chip will most likely keep increasing at a high rate. Lately there has been a shift in focus of processor design from single threaded performance to multithreaded performance. For more information on this development see [14, 23, 6].

The computing power available in GPUs makes them interesting in areas other than graphics as well. Even before there were proper tools for general purpose programming on GPUs, people used the graphics APIs in clever ways to exploit the GPU for non-graphics algorithms [28]. In 2006 NVIDIA created CUDA (Compute Unified Device Architecture) that provided a proper tool set for general purpose programming on their GPUs [24]. Using GPUs as data-parallel computing devices is a popular trend which can be seen by the number of languages and tools that have appeared in order to make use of them. Following CUDA, OpenCL arrived. OpenCL is an open API for general purpose programming that can be used with both NVIDIA and AMD GPUs. OpenCL can also be implemented independently of operating system [19]. DirectCompute is also an API for general purpose programming on GPUs DirectCompute is part of the Microsoft DirectX API.

1.2 Obsidian: Embedded language for GPU computing

This thesis describes the implementation of Obsidian. Obsidian is an domain specific language for general purpose programming on GPUs. Obsidian is implemented as an embedded language using Haskell as host. The goal of Obsidian is to raise the level of abstraction of GPU programming and offer the programmer a tool that encourages experimentation. From the higher level descriptions, CUDA code is generated.

1.3 Research questions

- CUDA is a huge improvement for the GPU programmer. However, there are some quirks that limit the compositionality of CUDA programs. Can an embedded language improve upon this situation while maintaining some of the low level control needed for performance?
- What are the useful combinators (higher order functions) for GPGPU programming and is it possible to generate efficient code for these?
- Can combinators and a "connection pattern" mode of thought borrowed from hardware description be used to alleviate the kind of indexing computations that are ubiquitous in CUDA and similar data-parallel programming models?
- Details of a kernel's implementation, such as number of threads used, should be easily configurable by the programmer with minimum rewriting of code. Can an embedded language approach provide help here?

1.4 Structure of this thesis

Chapter 1 contains an informal introduction to the area and a description of what is to come in the following chapters. In this chapter a number of research questions are posed.

Chapter 2 is an introduction to the GPGPU area with a section developing a small program in CUDA. This chapter also gives reference to other sources of information on GPGPU programming and tools.

Chapter 3 contains an introduction to the embedded language approach to solving problems.

Chapter 4 explains programming in Obsidian. In a series of examples the language features are introduced and used.

Chapter 5 shows two different implementations of Obsidian.

Chapter 6 contains a few larger examples developed in Obsidian together with some performance measurements.

Chapter 7 presents related work. Important influences as well as competing approaches to GPGPU programming are treated in this chapter.

Chapter 8 is the closing chapter where conclusions and reflections upon this work are presented.

Chapter 2

General purpose computations on GPUs

2.1 Introduction

As graphics processors (GPUs) became more programmable, the desire to utilize their parallel processing power for non-graphics applications arose. At first people wanting to use GPUs for general purpose computations (GPGPU) had to express their problem as a computer graphics computation. This was both limiting in what was expressible and hard. The data to use as input was stored in textures (two dimensional arrays optimized for image storage) and the algorithm to compute on that data expressed using OpenGL or DirectX (the two leading Graphics APIs). For a more in depth description of early (pre 2006) GPGPU methods see [28].

2.2 NVIDIA CUDA

The programmability and performance of GPUs kept rising and in 2006 NVIDIA released the 8800GTX graphics card. The 8800GTX graphics card was the first card with a unified GPU architecture released by NVIDIA [26]. Being a unified architecture means that there is only a single kind of processing core on the chip. This is different from earlier GPUs where there where different kinds of processing elements to process *vertex* and *fragment* data. Vertex and fragment refer to the computer graphics concepts, see for example [1] for more details.

In order to simplify making use of the GPU for general purpose computations, NVIDIA released CUDA at the introduction of their unified GPU architecture. CUDA (Compute Unified Device Architecture) is the name of their programming model for general purpose computations on NVIDIA GPUs [25]. CUDA is a big improvement compared to using the graphics APIs for general purpose programming. The programmer is no longer required to translate the problem into a graphics context. CUDA delivers a C compiler and a set of of tools and libraries for general purpose parallel programming on NVIDIA GPUs.

The NVIDIA 8800GTX card came with 128 processing elements, 768MB of memory and had a theoretical peak single precision floating point capability of 518.4 gigaflops [26].Today, in 2010, NVIDIA's top graphics card has 4 times the processing elements (in a single chip) and double the amount of memory.

2.2.1 CUDA architecture

A CUDA enabled GPU comes with a number of multiprocessors. Each multiprocessor contains a number of SIMD (Single Instruction Multiple Data) cores, load-store units, special function units and a local, *shared*, memory. The local memory per multiprocessor is called shared memory because it is used to share data between threads running on a multiprocessor. The number of SIMD cores per multiprocessor is today either 8 or 32 depending on GPU series. The latest version of the CUDA architecture is called Fermi and GPUs based on this architecture have 32 SIMD cores per multiprocessor [27]. The number of multiprocessors in a GPU chip ranges from 1 to 16 today. Thus the number of SIMD processing elements is in the range 8 to 512.

The shared memory per multiprocessor is configurable up to 48KB on Fermi and a fixed 16KB on previous generations. Each multiprocessor also has a number of registers, 32768x32bit registers on Fermi and half that amount in the previous generations.

On the graphics card there is also a *global* memory accessible by all multiprocessors. This global memory, which is often referred to as "device memory", is today in the area of 1GB in size. There are also computing platforms built around the CUDA enabled GPU specifically for general purpose computations. These devices usually come with a larger global memory (4 GB).

The resources available in a multiprocessor determine the maximum number of threads that can be maintained at any one time by that multiprocessor. On Fermi that number is 1024; for earlier generations it is 512. Now, 512 and 1024 are both numbers greater than the number of SIMD cores per multiprocessor. The threads are scheduled on those SIMD cores, maintained by a per multiprocessor scheduler. Groups of 32 threads are executed simultaneously on the the SIMD cores. These groups of 32 threads are called *warps*. Now, 32 is also a number larger than the 8 processing units available. These 32 threads are executed interleaved. threads 0 through 7 execute instruction 1 then threads 8 through 15 execute instruction 1 and so on.

2.2.2 CUDA programming model

The CUDA programming language is based on C but with a small set of language extensions for parallelism and synchronization. In CUDA, problems are solved by specifying a hierarchy of threads. This hierarchy of threads mirrors the architecture. The hierarchy consists of threads that belong to *Blocks* that are part of a *Grid*.

Blocks

Since a CUDA enabled GPU can come with varying numbers of multiprocessors, programs needs to be specified in a way that scales with the number of multiprocessors [25]. This leads to a concept exposed to the programmer called *blocks*. A block is a group of threads, at most 512 (1024 on Fermi), that cooperate in solving some subproblem. A block of threads is executed in one multiprocessor and the threads of the block may communicate using shared memory. Each block executes completely independently from any other block, thus giving the desired scalability effect. If there are more multiprocessors available the system simply launches more blocks in parallel. If there is only a single multiprocessor, all blocks will be executed in sequence. The program that is executed by a block is called a *kernel*. The kernel programming model is based in the Single Program Multiple Data (SPMD) paradigm. The kernel is a single program parameterized by a thread identity.

A block of threads can be specified to have one, two or three dimensions. This influences how threads are identified within that block. Threads are given an identity called `threadIdx` that is a three dimensional vector. The three dimensions of the identity are accessed as `threadIdx.x`, `threadIdx.y` and `threadIdx.z`. The maximum sizes of a block in x, y and z is 512, 512 and 64. However, the value $x*y*z$ must not be larger than 512 (1024 on Fermi). The kernel program may use the thread identity and conditionally chose one path or another. If this happens in such a way that threads within the same warp chooses different paths these execution paths will execute in sequence, not in parallel. This is a result of the SIMD style execution of the threads within a warp.

There is also a vector called `blockDim` that informs a thread of the size of the block it is part of.

The warp concept mentioned in section 2.2.1 is related to the notion of blocks in the sense that all blocks consist of a number of warps. The warp concept is not something the programmer has influence over. The threads of a block are divided into warps so that thread 0 to 31 is one warp, thread 32 to 63 is another and so on. However, knowledge of the number of threads in a warp and how the threads are divided into warps gives opportunity for performance optimizations. For example threads within a warp may communicate using shared memory without using synchronization primitives. This is because the threads of a warp will execute in SIMD style. That is, each thread is at the same point of execution. Threads that belong to different warps need to use a barrier synchronization primitive called `__syncthreads()` to ensure a uniform view of the shared memory.

Grid

A grid is a collection of blocks that all execute the same kernel. A block is identified within a grid by a `blockIdx` vector. One or two dimensional grids may be specified. The maximum number of blocks in any dimension is 65535. So at most 65535*65535 blocks may be launched simultaneously on the GPU.

A thread can be globally identified within a grid by using the values of `threadIdx`, `blockDim` and `blockIdx`. This will be shown in section 2.2.3.

The host

The host system is the computer that controls the CUDA enabled device. On this system, a controlling CPU thread is in charge of launching grids of blocks on the GPU available to the host. Grids are specified using special CUDA syntax that looks as follows:

```
kernel<<<blocks,threads,sharedmem>>>(arg1, arg2, ..., argn);
```

This launches the kernel called `kernel` on the GPU in a number of blocks specified by the vector `blocks`, a number of threads specified by `threads` and a shared memory size of `sharedmem`. The syntax also allows the `blocks` and `threads` to be specified using scalars if the grid or block is one dimensional. The `arg1`, `arg2` up to `argn` are the arguments passed to the kernel. If these arguments are arrays of data, these should reside in the device memory.

The host is also responsible for allocating memory on the device and uploading the data needed to the GPU before launching kernels. Allocating storage in the device memory is done using a `cudaMalloc` operation and memory transfer to and from the device is done using `cudaMemcpy`.

Programmer view

The CUDA device is a computer system capable of managing a very high number of threads. If it is possible to decompose a problem into pieces that are compatible with the hierarchical thread capabilities of the GPU, the gains can be big. There are reports of speed-ups in the range 10x to 1000x compared to CPU solutions, see for example [24] where a number of success stories are posted.

When a programmer aims to solve a data-parallel programming problem using CUDA, she must decompose the problem into subproblems that can be solved by a grid of blocks. These subproblems must have the characteristic that the data can be divided into chunks that can be computed on completely independently from any other chunk. A kernel is designed that performs the desired computation per chunk. Now, hopefully there is an efficient way to combine the results that this provides per chunk into a solution of that particular sub-problem.

2.2.3 CUDA programming example

In this section, a CUDA program for computing the dot product of two large arrays will be implemented. Computing the dot product has been chosen because it is a simple problem to describe and yet provides room for quite a bit of experimentation in the CUDA implementation. This gives the opportunity to show some of the problems and opportunities presented to the CUDA programmer.

Given two sequences of numbers X and Y of length n, the dot product is given by summing up the all the products $X_i * Y_i$ for $i \in 0..(n-1)$.

A possible implementation of this operations is sequential C code is the following:

```
float dotProduct(float *x, float *y, unsigned int n) {
  float r = 0.0f;
  for (int i = 0; i < n; ++i) {
    r += x[i] * y[i];
  }
  return r;
}
```

As the dot product computation is presented above, it is an entirely sequential operation. By splitting this algorithm up into a summing part and a multiplying part, opportunities for parallelization present themselves. Computing each of the products, $X_i * Y_i$, can be done completely independently and is easily parallelizable. Computing the products is very easily transformed into CUDA. The naive kernel that performs this operation is completely block size independent. This means that the multiplication kernel can be applied on varying input data sizes without change.

```
__global__ void multKernel(float *result, float *x, float *y) {
  unsigned int gtid = blockIdx.x * blockDim.x + threadIdx.x;
  result[gtid] = x[gtid] * y[gtid];
}
```

The `__global__` declaration of the `multKernel` specifies for the CUDA compiler that this function is a GPU kernel. This kernel takes three pointers as arguments, `result`, `x` and `y`. These pointers point to equal length arrays that reside in the device memory.

The next line computes the global thread id that is used to index into the result and input arrays. If this kernel is used to compute the element wise products of two arrays of length 2000 then 2000 threads are needed to compute all the results. 2000 threads cannot be maintained in a single block, so a multitude of smaller sized blocks must be launched. In this case, the programmer is free to launch for example ten 200-thread blocks to compute the products. This call to `multKernel` launches ten 200-thread blocks:

```
multKernel<<<10,200,0>>>(r,x,y);
```

Another option would be to launch 100 20-thread blocks to compute the same result. But it is generally not recommended to choose a block size that is smaller than warp size. Even in the 200-threads per block case, the situation is non-optimal. A multiple

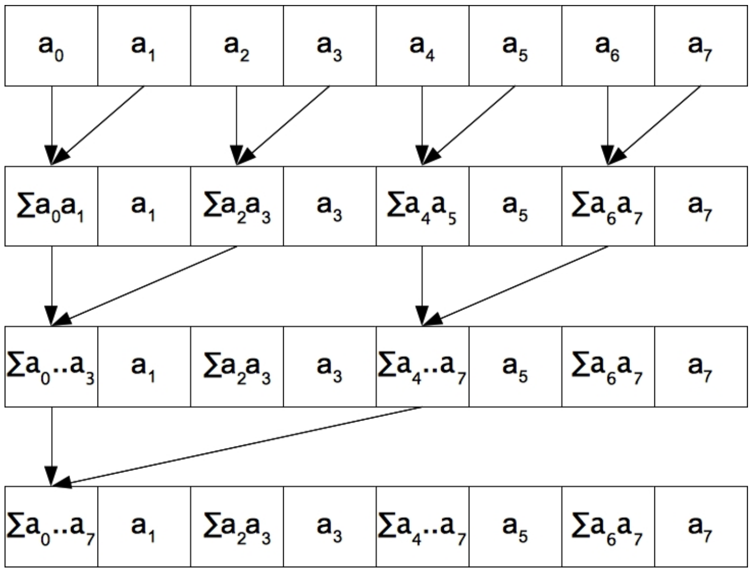
a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7
$\Sigma a_0 a_1$ a_1 $\Sigma a_2 a_3$ a_3 $\Sigma a_4 a_5$ a_5 $\Sigma a_6 a_7$ a_7
$\Sigma a_0..a_3$ a_1 $\Sigma a_2 a_3$ a_3 $\Sigma a_4..a_7$ a_5 $\Sigma a_6 a_7$ a_7
$\Sigma a_0..a_7$ a_1 $\Sigma a_2 a_3$ a_3 $\Sigma a_4..a_7$ a_5 $\Sigma a_6 a_7$ a_7

Figure 2.1: A binary tree summation kernel

of warp size would have been better. But forcing the block size to a multiple of warp size would in this case mean the input data would need to have been padded or extra bounds checking code inserted into the kernel. Inserting bounds checking would be impractical because every instance of the kernel would need to execute some extra conditional but only one of them is on the boundary.

Even when implementing this very simple operation as a CUDA kernel there are a number of choices that could have been made differently. The implementation above uses one thread to compute each result. Instead, each thread could compute a number of results but this limits the generality of this kernel. For example, a kernel that computes two results per thread could only be applied to even length arrays.

The second part of the dot product algorithm is to compute the sum of all the elements in the array of results from the multiplication step. This step is not as direct as the previous one and the parallel solution imposes more limitations on input data size than the `multKernel` did. The summation kernel will use the binary tree shaped approach, as illustrated in figure 2.1. This summation kernel will be applicable to arrays of length a power of two.

The summation kernel will be implemented using one thread per element and the blocks will need to be of size a power of two for proper operation. In order to sum up a large array, larger than the block size, the host must launch further grids to sum up the partial results. This imposes further limitations; for example if an array of size 256*512 is to be summed up this can be done by launching 256 512-thread blocks followed by a single 256-thread block to get the total sum. But if the array to sum up is 100*512 elements it is harder to use this kernel to sum up the 100-element partial result. Either this array needs to be padded with zeroes to 128 elements or a specialized kernel that can handle 100 elements would need to be written to take care of that case.

Below is the CUDA code that implements this kernel; it will be explained in detail following the listing:

```
__global__ void sumKernel(float *result, float *x) {

  unsigned int tid = threadIdx.x;

  extern __shared__ float sm[];

  sm[tid] = x[blockIdx.x * blockDim.x + tid];

  for (int i = 1; i < blockDim.x; i *= 2) {
    __syncthreads();
    if (tid % (2*i) == 0) {
      sm[tid] += sm[tid + i];
    }
  }

  if (tid == 0) {
    result[blockIdx.x] = sm[0];
  }
}
```

The first line specifies that this kernel takes two pointers as arguments, a `result` array and an input array called `x`. Then a short name for the local thread identity is defined. This kernel will compute sums of block sized arrays in shared memory and during these computations the local thread id will be used to index into that memory. The global thread identity is only used to read in data from the large input array.

The line `extern __shared__ float sm[];` names the shared memory array. The size of the shared memory array is set from the host when launching the grid.

Now, data can be read from global memory to local memory using global thread id, `sm[tid] = x[blockIdx.x * blockDim.x + tid];`.

The `for` loop runs log base 2 of the block size times. Each time through the loop a number of threads will perform an addition. In the first iteration block size divided by 2 threads will be active and add up elements, as shown in figure 2.1. The `__syncthreads()` call is placed in the beginning of the loop so that it acts both as a barrier synchronization between the iterations of the loop and as a barrier between the initial reading of global memory and the first iteration of the loop. The conditional in the loop essentially shuts a number of threads off.

The very last thing this kernel does is writing the sum to global memory. This is done by thread 0. No extra barrier is needed before writing the result to global memory since thread zero will be the one to compute the final iteration through the loop.

Now the `multkernel` and `sumKernel` can be used to implement a dot product algorithm. The host code below implements dot product for 128*512 elements. The block size in use will be 512 and in the multiplication step a 128 block grid is launched. In the summation step two grids will be launched – first a grid of 128 elements and then one containing a single block of 128 threads.

After allocating all device memory and uploading the arrays to GPU memory, the host code would launch the following grids:

```
multKernel<<<128,512,0>>>(device_x,device_x,device_y);
sumKernel<<<128,512,512*sizeof(float)>>>(result, device_x);
sumKernel<<<1,128,128*sizeof(float)>>>(result,result);
```

After executing these grids, the result of the entire computation is stored at index zero in the result array.

2.2.4 Conclusions

CUDA is a huge improvement compared to using graphics APIs for GPGPU programming. This increases programmer productivity and has effect on what can be expressed.

In the previous section a multiplication kernel and a summation kernel were implemented separately. It is, however, possible to implement a fused multiply-sum kernel, which may be desirable for performance purposes.

```
__global__ void mult_sumKernel(float *result, float *x, float *y) {

  unsigned int tid = threadIdx.x;

  extern __shared__ float sm[];

  sm[tid] = x[blockIdx.x * blockDim.x + tid] *
            y[blockIdx.x * blockDim.x + tid];

  for (int i = 1; i < blockDim.x; i *= 2) {
    __syncthreads();
    if (tid % (2*i) == 0) {
      sm[tid] += sm[tid + i];
    }
  }

  if (tid == 0) {
    result[blockIdx.x] = sm[0];
  }
}
```

Another transformation of the `sumKernel` that may be desirable is to unroll the loops. Unrolling the loops however will impose further input size restrictions.

Section 6.7 shows some running time measurements of the `mult_sumKernel`.

There are limitations to how the `__syncthreads()` barrier can be applied. These limitations are not enforced by the compiler which will gladly compile programs that have unspecified behavior according to the programming manual. For example, `syncthreads()` can only be used within a conditional if all threads follow the same execution path through it [25].

2.3 Other GPGPU programming languages

2.3.1 OpenCL

OpenCL is an open standard for parallel programming designed by the Khronos group [19]. OpenCL offers a programming model very similar to that of CUDA. This section will list the similarities between OpenCL and CUDA and also point out some differences..

In OpenCL, just as in CUDA, the programmer specifies a hierarchy of threads (in OpenCL terminology called work-items). Each work-item belongs to a group, a work-group; corresponding to the CUDA block. The work-group belongs to an NDRange, that corresponds to CUDA's grid. OpenCL also makes the distinction between host and device. The device is a processing unit capable of running OpenCL kernels and the host runs a controlling thread, just as in CUDA.

OpenCL takes things one step further by being less platform bound. An OpenCL application can be executed on a wider range of hardware by utilizing just-in-time compilation. The kernels that make up an application will at runtime be specialized for the kind of hardware that is present in the computer system. Amongst OpenCL enabled devices are GPUs (both NVIDIA and AMD) and multi-core CPUs.

Chapter 3

Embedded domain specific languages

3.1 Introduction

Embedding domain specific languages within other programming languages is becoming popular. There are many examples of embedded languages in several areas, see chapter 7. An embedded language is implemented as a library within a host language. As a host language, Haskell is a well tested candidate. With Haskell's type class system, the embedded language blends in almost seamlessly.

Domain specific languages (DSLs) are languages that are written to solve problems within a certain domain. Implementing a DSL in the traditional way, as a standalone compiler, may be too expensive in time and effort. Implementing a DSL as an embedded language spares the programmer from implementing the so called frontend part of compiler. The frontend takes care of parsing and lexing, and building the abstract syntax trees that the back-end of the compiler works on [29]. In an embedded language there is a collection of library functions that create the same abstract representations of programs that the frontend usually produces from a source code listing. The embedded language implementor has the choice of implementing an interpreter that computes the result of the program or to embed a compiler. An embedded compiler could, for example, output assembly, virtual machine bytecode or C code.

The presentation of embedded language design and implementation techniques given here borrows extensively from Conal Elliot et al. *Compililing embedded languages* [12].

That article describes the implementation of an embedded compiler for an image manipulation program called *Pan*.

3.2 Abstract syntax

An embedded language is a set of program generating functions in some other language, here Haskell. These programs that are generated need to be represented in some way. One such way to represent programs is as actual source code listings, strings [18]. It is also possible to represent the programs with abstract syntax trees. Below is an example of an expression data type for floating point expressions:

```
data FloatE
     = Var String
     | LitFloat Float
     | FloatE ‘Add‘ FloatE
     | FloatE ‘Sub‘ FloatE
     | FloatE ‘Mul‘ FloatE
     | FLoatE ‘Div‘ FloatE
     | Sin FloatE
     | Sqrt FloatE
```

Functions in the embedded language can be represented by Haskell functions over the expression type. For example the function `cos` can be defined as follows:

```
cos :: FloatE -> FloatE
cos theta = Sin (LitFloat 1.5708 'Sub' theta)
```

3.3 Syntax

If the programmer using the embedded language were forced to express herself using the constructors of the expression data type, the embedded approach would probably not be popular. Haskell has a system for overloading that becomes very useful in the implementation of embedded languages. This feature of Haskell makes the embedded language blend in elegantly. This is done using the type class system. For example there is a type class called *Num* for basic numeric types. An instance declaration for the expression type looks like this:

```
instance Num FloatE where
  (+) a b = a `Add` b
  (-) a b = a `Sub` b
  (*) a b = a `Mul` b
  fromInteger n = LitFloat (fromInteger n)
```

With this *Num* instance and one for *Floating*, that contains amongst other things `pi`, cosine could be expressed using sine in the following way:

```
cos theta = sin (pi / 2 - theta)
```

This is an improvement compared to the first implementation of cosine. In some sense the syntax of the host language is being hijacked and used as syntax for the embedded language.

3.4 Inlining

The function, `square`, in the embedded language, can be thought of as taking a value and giving back its square. As a Haskell function, it generates an expression that contains the `Mul` constructor and two instances of the input. The definition of square is.

```
square x = x * x
```

When applying `square` to `cos t` where `cos` is defined as above and `t` is a constant, would result in a expression like the one expressed here in mathematical notation: $\sin(\pi/2 - t) * \sin(\pi/2 - t)$. The definition of `cos` has been inlined at both the occurrences of `x` in the `square` function. This replication of computations leads to inefficient generated code if not properly taken care of in an optimization phase.

3.5 Optimization and smart constructors

The *Compiling Embedded Languages* [12] article suggests Common Subexpression Elimination (CSE) to improve on the situation with code replication. However, it also explains the use of *Smart Constructors* to apply some optimizations on the generated code as it is being generated. For example, when implementing the `(+)` operation for the embedded language, the inputs to it can be inspected and optimizations applied. If both inputs to `(+)` are literals of the form `LitFloat x`, then the generated code does not need to contain any add operation at all.

```
(+) (LitFloat x) (litFloat y) = LitFloat (x + y)
...
(+) x y  = x ‘Add‘ y
```

In the last case of `(+)`, where `x` and `y` are of a form that cannot be processed further, the `Add` constructor is applied. The `...` indicates that there may be many more cases to capture here and some of them may be target architecture specific.

3.6 Conclusion

This section introduced the embedded language concept by drawing information from the formative Compiling Embedded Languages paper [12]. The main topic not considered here is the representation and manipulation of typed expressions. An embedded language would most likely have more types than just floating point. This is of course also addressed in [12] where Floats, Ints and Booleans all can be represented in the same expression type. The programmer is given a typed interface though by the use of phantom types [15].

In Pan, images are represented by functions from points (pairs of `int` expressions) to colors. In order to produce a viewable image, this function needs to be applied at every point in a grid (pixels). C code can be generated that does this by applying the image function to a pair of symbolic points (variable expressions, `Var "x"` and `Var "y"`) and using the result as the body of a nested `for` loop that traverses the grid.

The main benefit of the embedded language approach is the effort saved in implementation. There is no need to implement a compiler frontend. A large part of DSL implementation is experimentation and trial and error concerning what features to include and in what way. Being able to perform this cycle of feature implementation and rejection in an embedded language is preferable from a productivity point of view.

Chapter 4

Obsidian

4.1 Introduction

Obsidian is an embedded language for implementation of GPGPU kernels. The goals of Obsidian is to give the programmer a higher level and more compositional language in which to express GPU kernels. From the higher level descriptions of GPGPU kernels, CUDA code is generated.

Obsidian consists of three parts. First of all there is a collection of operations that can be performed on scalar types. In this category you find, as usual, operations such as , `+` , `-`, `*` and `/`. On top of that Obsidian supplies *arrays*. The decision was made to let the length of these arrays specify the degree of parallelism. When generating CUDA code from an Obsidian description, each element of the resulting array is computed by one thread. In the case of nested arrays, the top level array in the nesting specifies the degree of parallelism and the inner arrays are computed sequentially. With the arrays, Obsidian also supplies a library of functions on these arrays. Lastly, there is the third part, a set of operations (combinators) used to construct kernels and guide the code generation.

4.2 Array language

This section contains examples of functions from the array language part of Obsidian. The array language consists of a an Array type `Arr a` and a collection of functions on arrays. The Array language programs cannot by themselves be executed on the

GPU. Instead, the array language programs need to be turned into GPU kernels using the tools presented in section 4.4.

In Obsidian, an array is represented by an indexing function and a length.

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

The length of the array is held in a Haskell `Int` and is static (not subject to change during runtime). The length being static enables some optimizations that will be explained later, see section 5.4.4. The `IndexE` type represents 32bit unsigned integers. This will be explained in more detail in section 5.

Obsidian arrays can contain elements that are booleans, 32bit floating point numbers or 32bit integers. Since Obsidian in the end is compiled into CUDA, booleans will be represented by 32bit integers. The arrays can also contain tuples or arrays and nesting thereof.

One of the most basic operation that can be performed on an array is to map a function over it. This operation is data-parallel and is of course the first one added into the library:

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

The `fmap` function can be used to write a program that increments every element of an array:

```
incr = fmap (+1)
```

Another operation on arrays that springs to mind is `foldr`:

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

An example use of `foldr` is to sum up an array. Given an array with the elements {1,2,3,4,5,6}, `foldr` with the operation + and the the value 0 is (1+(2+(3+(4+(5+(6+0)))))). The standard Haskell `foldr` on lists can be defined like this:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ z []     =  z
foldr f z (x:xs) =  f x (foldr f z xs)
```

As can be seen from the example above, `foldr` is a sequential computation. The array language implementation of `foldr` is also entirely sequential but this is not a problem. Being able to use both sequential and parallel operations in combination is useful when implementing algorithms for GPUs.

In the array language there are many functions that Obsidian inherits from Lava. Lava is a domain specific embedded language for hardware description [4]. There is more information about the Lava hardware description language in the related work section 7.3.1. The first two Lava inspired functions are `pair` and `unpair`:

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

The function `pair` takes an array of even length and returns an array of pairs where the first element is paired up with the second, the third with the forth and so on. The `unpair` function does the opposite, see figure 4.1.

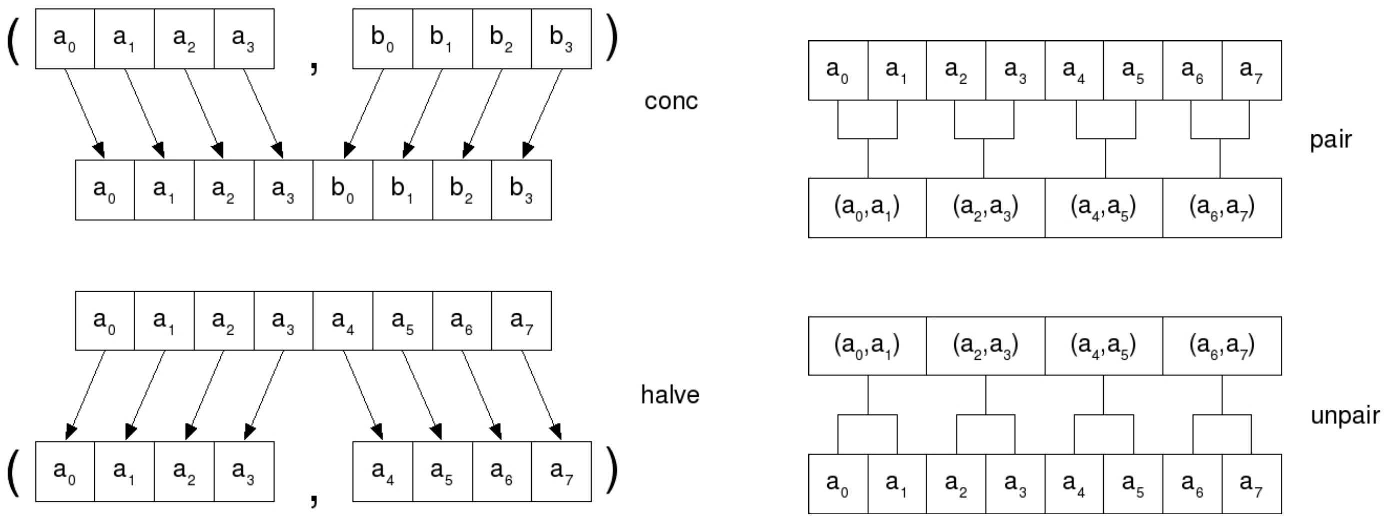

Figure 4.1: The functions `conc`, `halve`, `pair` and `unpair`

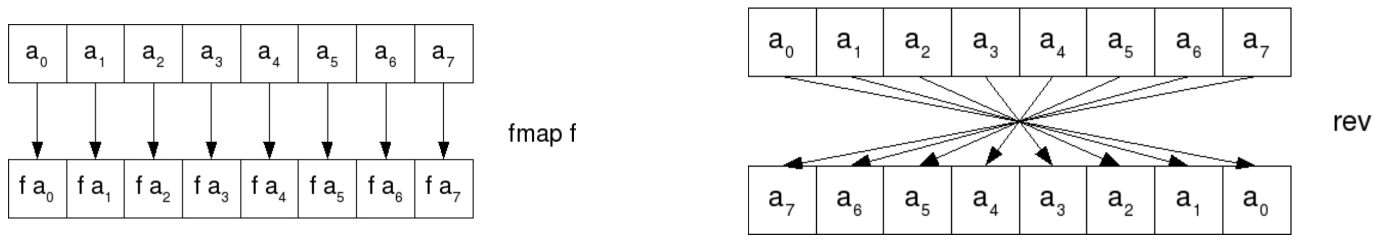

Figure 4.2: The functions `fmap` and `rev`

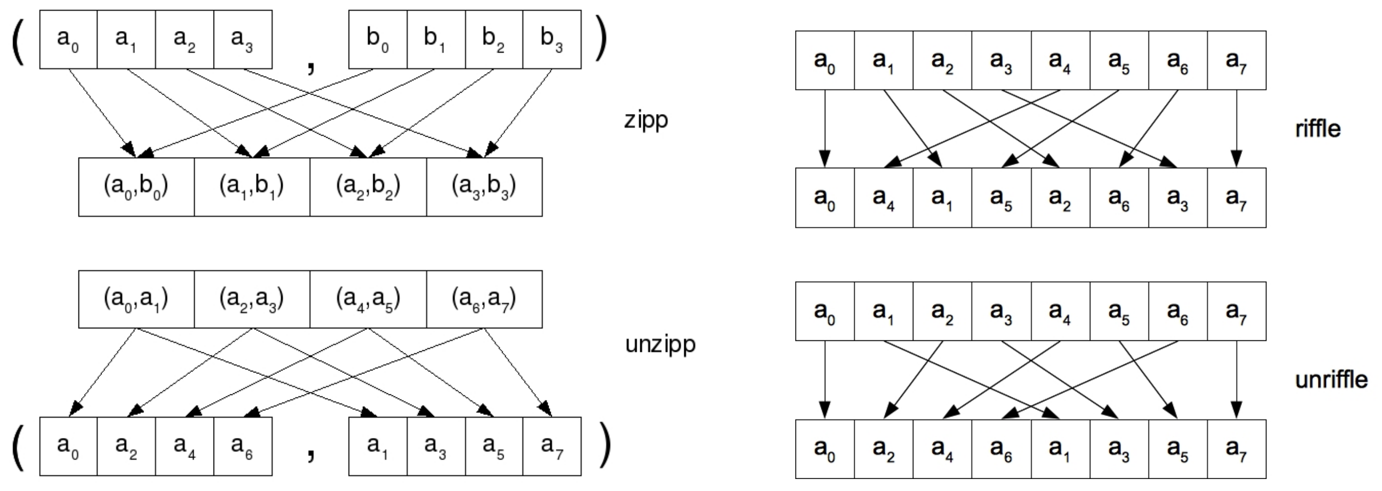

Figure 4.3: The functions `zipp`, `unzipp`, `riffle` and `unriffle`

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

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

There is also a function `??` defined that chooses either the first or the second element of a tuple depending on a condition. This can be used to implement a program that replaces every occurrence of a specific element in an array by some given constant:

```
condReplace e c = fmap (\x -> (x ==* e ?? (c,x)))
```

Using the functions `pair`, `fmap` and `unpair`, two higher order functions, `pairwise` and `evens`, can be defined. `pairwise` is takes a two input function and applies it to pairs of an array:

```
pairwise :: (a -> a -> b) -> Arr a -> Arr b
pairwise f = fmap (uncurry f) . pair
```

The function `uncurry` used in the implementation of `pairwise` is a higher order function that takes a function of type `a -> b -> c` and the result is a function that takes a pair `(a,b)` as input and produces a `c`.

`evens` is similar to `pairwise` but requires the function to be a two input/two output function:

```
evens :: Choice a => ((a,a) -> (a,a)) -> Arr a -> Arr a
evens f = unpair . fmap f . pair
```

In the implementation of `evens` above, Haskell function composition (`.`) is used.

There is also a function called `odds` that is very similar to `evens` except that it lets the first and the last element of the array pass through unchanged. The functions `odds` behaves just like `evens` on the remaining elements of an array. Figure 4.4 illustrates `evens` and `odds` on six inputs. The `odds` function is a bit more complicated to implement. A way to pass the first and the last element of an array through unchanged is needed, and doing so using as few conditionals as possible. The implementation of the `odds` function is shown in chapter 5.

In chapter 6, Obsidian will be used to implement sorting networks. One building block that is useful when constructing sorting networks is `sort2`. Given an array of

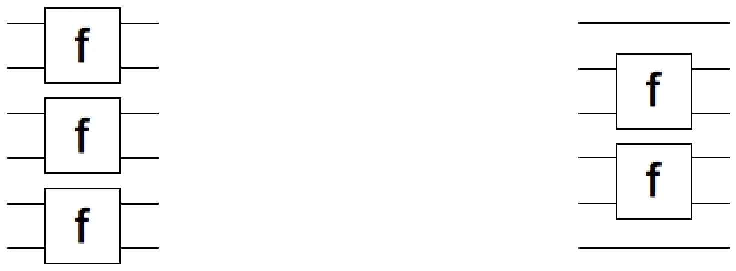

Figure 4.4: The functions evens (left) and odds (right)

even length, `sort2` gives an array where the elements at index 0 and 1 are sorted, the elements at index 2 and 3 are sorted and so on. For example if the input array is $\{1, 0, 4, 5, 3, 2\}$ the result is $\{0, 1, 4, 5, 2, 3\}$.

The `sort2` function can easily be implemented using `evens` given that we have a *compare and swap* operation, `cmp`:

```
cmp :: (Choice a , Ordered a) => (a, a) -> (a, a)
cmp (a, b) = (min a b, max a b)
```

Now `sort2` can be implemented as:

```
sort2 = evens cmp
```

The `Ordered` class provides a set of comparison functions:

```
(<*) :: Ordered a => a -> a -> BoolE
(<=*) :: Ordered a => a -> a -> BoolE
(>*) :: Ordered a => a -> a -> BoolE
(>=*) :: Ordered a => a -> a -> BoolE
```

Two other examples of functions from the array library are `zipp` and `unzipp`, see figure 4.3. These two functions are very similar to the Haskell `zip` and `unzip` functions on lists:

```
zipp :: (Arr a, Arr b) -> Arr (a, b)
unzipp :: Arr (a, b) -> (Arr a, Arr b)
```

As an example, `zipp` and `fmap` can be used to implement the well known `zipWith` function:

```
zipWith :: ((a,b) -> c) -> (Arr a,Arr b) -> Arr c
zipWith f = fmap f . zipp
```

The functions `zipp` and `zipWith` here are uncurried. The Haskell counterparts are curried. The reason for this is related to the choice to have array lengths define the number of threads. Working with pairs is a common operation when using Obsidian.

Some functions in the array library only rearrange the elements of an array. These functions could be called permutation functions. An example of a permutation is `rev` that takes an array and reverses the order of the elements, see figure 4.2:

```
rev :: Arr a -> Arr a
```

There are also two functions called `riffle` and `unriffle`, figure 4.3. These are also justified by their usefulness in sorting networks:

```
riffle :: Arr a -> Arr a
unriffle :: Arr a -> Arr a
```

The function `riffle` can be implemented using `halve` that splits an array down the middle into two arrays, `zipp` and `unpair`:

```
riffle :: Arr a -> Arr a
riffle = unpair . zipp . halve
```

Chapter 5 also shows an alternative implementation of `riffle` that is more efficient on a GPU.

The functions described so far are part of the array library of Obsidian. These functions are in some sense the building blocks of which programs are constructed. Most of the functions shown have been motivated by their usefulness in sorting network implementation. As we explore more applications, there will surely be more functions added to the library. In this section, there has been very little talk about parallelism and how these programs are supposed to be executed on a GPU, but this is deliberate. Section 4.4 will go into how GPU kernels are constructed using these building blocks, and some new ones.

```
rev :: Arr a -> Arr a
fmap :: (a -> b) -> Arr a -> Arr b
foldr :: (a -> b -> b) -> b -> Arr a -> b
pair :: Arr a -> Arr (a,a)
unpair :: Choice a => Arr (a, a) -> Arr a
halve :: Arr a -> (Arr a, Arr a)
split :: Int -> Arr a -> (Arr a,Arr a)
conc :: Choice a => (Arr a, Arr a) -> Arr a
zipp :: (Arr a, Arr b) -> Arr (a, b)
unzipp :: Arr (a, b) -> (Arr a, Arr b)
riffle :: Arr a -> Arr a
unriffle :: Arr a -> Arr a
singleton :: a -> Arr a
chopN :: Int -> Arr a -> Arr (Arr a)
```

Figure 4.5: A selection of functions from the array library

```
(??) :: Choice a => BoolE -> (a, a) -> a
(==*) :: Equal a => a -> a -> BoolE
(<*) :: Ordered a => a -> a -> BoolE
min :: Ord a => a -> a -> a
max :: Ord a => a -> a -> a
```

Figure 4.6: Example functions on elements

4.3 Notes on recursion

Defining functions by means of recursion is very useful. However, when implementing Obsidian array functions or kernels the programmer should be aware that it might lead to code size explosion. This is a typical problem in embedded languages, and is related to how definitions are inlined [12]. Recursion and function application in general take place entirely in the Haskell world and will be completely gone in the resulting generated code. What is left in the generated code is a completely unrolled and inlined version of the computation.

The Haskell recursion should be viewed as a tool for program generation for the Obsidian programmer. Recursion is a very useful tool and should be applied, but with some care. The programmer should be aware that a recursive call to a function in Obsidian will result in CUDA code with the body of that function inlined. There will be many examples of this use of recursion as a code generation tool in the coming sections.

4.4 GPU kernels

Section 4.2 showed how to express functions on arrays in Obsidian. This section goes into how these array functions can be mapped onto the GPU in different ways. It is in the language described in this section that parallelism and whether to share computed values or recompute them becomes expressible.

A GPU kernel in Obsidian is represented by a data type `a :-> b`. This type can be thought of as a program taking an `a` as input and producing a `b`. The kinds of kernels that can be represented in this type are illustrated in figure 4.7. This figure shows a program that performs some computation using a number of threads followed by a barrier synchronization, and so on. The boxes marked with *Pure* can be thought of as containing an array program like those in section 4.2.

4.4.1 Pure

One basic way to create a new GPU kernel is to take a given array language program, such as `incr = fmap (+1)`, from the previous section. This program increments every element of an array. This array language program can be turned into a GPU kernel by using the function `pure`:

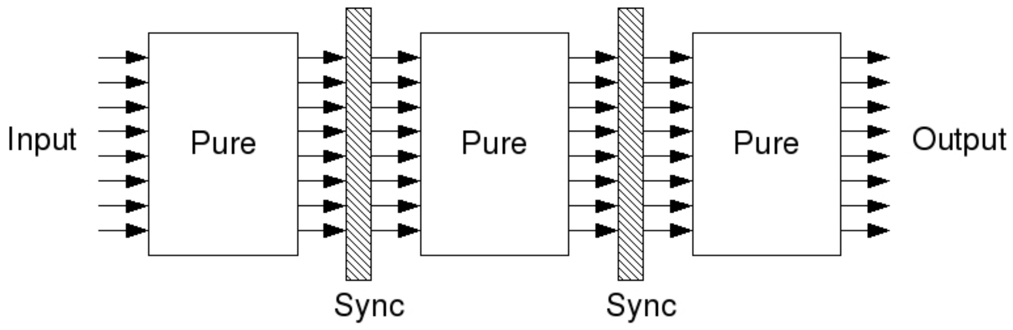

Figure 4.7: A GPU Kernel, an object of type `a :-> b`, can be thought of as some pure computations interspersed by synchronization

```
incrKernel :: Num a => Arr a :-> Arr a
incrKernel = pure $ incr
```

4.4.2 Execute

A kernel such as `incrKernel` can be executed on the GPU from within a *GHCI* (the Glasgow Haskell interpreter) session using a function called `execute`:

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

The class `Flatten` will be explained in detail in section 5.4, but 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 to run the kernel `incrKernel` on the GPU:

```
*Obsidian> execute incrKernel [0..9 :: IntE]
[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 again.

The code generated from the `incrKernel` program is presented below. The details of this source code listing are explain below.

```
__global__ void generated(word* input0,word* result0){
  const unsigned int tid = (unsigned int)threadIdx.x;
  const unsigned int bid = (unsigned int)blockIdx.x;
  ix_int(result0,(tid + (10 * bid))) =
    (ix_int(input0,(tid + (10 * bid))) + 1);
}
```

The generated CUDA kernel takes two arguments, an array of input words and an array of output words. Words are 32-bit quantities that can be either floating point or integer valued. This kernel uses no shared memory. The result is computed and written directly back into global memory. This generated CUDA code differs from what a CUDA programmer would write by hand mainly in the use of the `ix_int` macro for indexing into an array. There is also a macro called `ix_float` for accessing elements as floating point data. The generated CUDA code also uses `blockIdx.x` to calculate the location of the input and output data. This means that many instances of the generated kernel can be run in parallel over blocks of a large input array, as indicated in figure 4.8. The code generation procedure is described in chapter 5. Since this is the first example of generated code, it will be described line by line.

The first line specifies the name and inputs to the kernel. This is no different from what a CUDA programmer might write by hand except for the type `word` used for the input and output pointers:

```
__global__ void generated(word* input0,word* result0){
```

Generated kernels take one or more array as input. These are named `input0`, `input1` and so on. The outputs are named `result0`, `result1` and so on in the same way.

The second and third lines just set up aliases for the two integers `threadIdx.x` and `blockIdx.x`. This of course makes no difference for the meaning of the program but makes the program text a bit more readable:

```
const unsigned int tid = (unsigned int)threadIdx.x;
const unsigned int bid = (unsigned int)blockIdx.x;
```

The third line of the code is the interesting one in this case:

```
ix_int(result0,(tid + (10 * bid))) =
  (ix_int(input0,(tid + (10 * bid))) + 1);
```

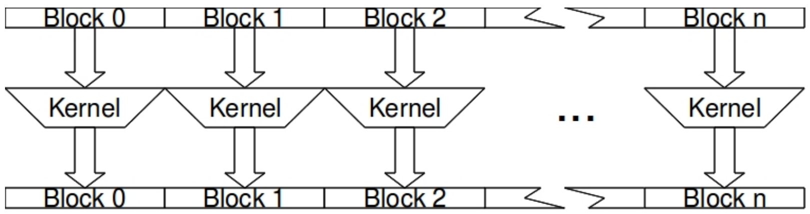

Figure 4.8: Many instances of a kernel executed in parallel over blocks of an array.

Here, each thread assigns a value to position `(tid + (10 * bid))` in array `result0`. The value assigned is the value found at position `(tid + (10 * bid))` in array `input` incremented by one. The number, 10, that appears in the generated code is the length of the array. If many instances of this kernels are launched as a grid, each instance will produce a 10 element subarray into a larger array in global memory.

4.4.3 Sequential composition

Given two kernels `f` and `g` of the following types:

```
f :: a :-> b
g :: b :-> c
```

a new kernel `f ->- g` can be created. The `->-` operator is sequential composition. This means that `f ->- g` is a kernel that takes an `a` as input. It lets `f` compute a `b` that is in turn passed to `g`. The `->-` operator has the following type:

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

The following example illustrates the use of `->-` by implementing a kernel that both adds one to each element of an array and reverses the array:

```
incrRev :: Num a => Arr a -> Arr a
incrRev = pure incr ->- pure rev
```

It is also possible to specify the `incrRev` kernel as follows:

```
incrRev :: Num a => Arr a -> Arr a
incrRev = incrKernel ->- pure rev
```

Executing either of these two versions of `incrRev` on the GPU gives the following result:

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

The CUDA code generated from `incrRev` is very similar to that of `incrKernel` but the indexing is reversed:

```
__global__ void generated(word* input0,word* result0){
  const unsigned int tid = (unsigned int)threadIdx.x;
  const unsigned int bid = (unsigned int)blockIdx.x;
  ix_int(result0,(tid + (10 * bid))) =
    (ix_int(input0,((9 - tid) + (10 * bid))) + 1);
}
```

The CUDA code for `incr` and `increv` are both executed using 10 threads in order to compute the desired result. However, there is nothing in the actual kernel code that specifies the number of threads that are executing it. Instead, that responsibility falls on a controlling process running on the CPU. The controlling process invokes the kernel using special CUDA syntax that looks like this:

```
generated<<<1,10,0>>>(input, output);
```

This call sets up the kernel `generated` to be executed on the GPU using 1 block, 10 threads and 0 bytes of shared memory. For more information about programming directly in CUDA see section 2.2

The kernel `incrRev` could also have been expressed in this slightly different way:

```
incrRev :: Num a => Arr a -> Arr a
incrRev = pure $ rev . incr
```

4.4.4 Sync

When both arguments to `->-` are implemented using `pure` alone, `->-` is defined using Haskell functional composition. However, `incrRev` can also be specified with an explicit storing of intermediate values between the `rev` and `incr`. This is accomplished using a primitive kernel called `sync`:

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

Adding a sync to the incrRev kernel does not influence the values computed at all. It does, however, affect how the results are computed. More information about sync can be found in section 4.4.5.

```
incrRevs :: (Num a, Flatten a) => Arr a :-> Arr a
incrRevs = pure incr ->- sync ->- pure rev
```

The incrRevs kernel computes the same result as incrRev did. However, it does so by computing incr on the array, storing the intermediate results in shared memory followed by computing the reverse. The CUDA C code for this version of incrRev looks like this. Notice how the shared memory is used:

```
__global__ void generated(word* input0,word* result0){
  const unsigned int tid = (unsigned int)threadIdx.x;
  const unsigned int bid = (unsigned int)blockIdx.x;
  extern __shared__ unsigned int s_data[];
  word __attribute__((unused)) *sm1 = &s_data[0];
  word __attribute__((unused)) *sm2 = &s_data[10];
  ix_int(sm1,tid) =
    (ix_int(input0,(tid + (10 * bid))) + 1);
  ix_int(result0,(tid + (10 * bid))) =
    ix_int(sm1,(9 - tid));
}
```

Using sync in Obsidian code does not always result in a call to CUDA __syncthreads(). Only if the number of threads needed to compute the elements of the array is larger than the warp size is a __syncthreads() call generated. See section 2.2 for more information about the warp concept. The approach taken here, to add a call to synchtreads as soon as the number of threads writing is larger then warp size, is a simplification. A more thorough approach would be to investigate the communication pattern of these writes and detect if all communication is within a single warp or not. If all threads that communicate only do so with threads within the same warp, there is no need to synchronize.

Below is code generated from the same Obsidian incrRev kernel but for arrays of size 100 elements instead of 10; notice the call to __syncthreads():

```
__global__ void generated(word* input0,word* result0){
  const unsigned int tid = (unsigned int)threadIdx.x;
  const unsigned int bid = (unsigned int)blockIdx.x;
  extern __shared__ unsigned int s_data[];
  word __attribute__((unused)) *sm1 = &s_data[0];
  word __attribute__((unused)) *sm2 = &s_data[100];
  ix_int(sm1,tid) =
    (ix_int(input0,(tid + (100 * bid))) + 1);
  __syncthreads();
  ix_int(result0,(tid + (100 * bid))) =
    ix_int(sm1,(99 - tid));
}
```

4.4.5 Sync and parallelism

The previous examples, `incrKernel` and `incrRev`, are very easily parallelizable. In fact, given the choice we made to let the length of the result array decide the number of threads used to compute it, these two examples are by default maximally parallelized. However, there are examples where the result array is of length one but where there is still room for parallelism in the computation of that one result. One example of such a computation is summing up all the elements of an array. As an example, take an array of these eight elements {0,1,2,3,4,5,6,7}. The elements of this array can be summed up in parallel by creating a new array with the sums {0+1,2+3,4+5,6+7} = {1,5,9,13}. All of these additions can be done in parallel. The result of that, the array {1,5,9,13}, is then processed in the exact same way, producing {1+5,9+13}. The procedure is repeated until there is only a single element. Now, the `sync` kernel allows precisely this. The following examples show how `sync` can be used to guide code generation. A number of different implementations of `sum` will be given. The first version, `sumSeq`, sums up the elements of an array using a single thread:

```
sumSeq :: Arr IntE :-> Arr IntE
sumSeq = pure $ singleton . (foldr (+) 0)
```

The GPU kernel `sumSeq` uses `foldr (+)` to add up all the elements of the input array. The result array is created by taking the sum value and creating a one element array using the `singleton` function.

The code generated from `sumSeq` for 8 elements looks like this:

```
__global__ void generated(word* input0,word* result0){
  const unsigned int tid = (unsigned int)threadIdx.x;
  const unsigned int bid = (unsigned int)blockIdx.x;
  ix_int(result0,(tid + bid)) =
    (ix_int(input0,(0 + (8 * bid))) +
     (ix_int(input0,(1 + (8 * bid))) +
      (ix_int(input0,(2 + (8 * bid))) +
       (ix_int(input0,(3 + (8 * bid))) +
        (ix_int(input0,(4 + (8 * bid))) +
         (ix_int(input0,(5 + (8 * bid))) +
          (ix_int(input0,(6 + (8 * bid))) +
           ix_int(input0,(7 + (8 * bid))))))))));
}
```

This kernel sums up the elements of an array of eight elements using `foldr`. That is the elements are added like this $(0 + (1 + (2 + (3 + (4 + (5 + (6 + 7)))))))$. But as we saw, it is also possible to sum the values up by repeatedly adding up pairs of elements from the original array. In Obsidian, this way of adding up the elements can be expressed like this:

```
sum1 :: Int -> Arr IntE :-> Arr IntE
sum1 0 = pure id
sum1 n = pure op ->- sum1 (n-1)
  where
    op  = fmap (uncurry (+)) . pair
```

The `sum1` kernel uses `pair` to pair up the first element of the array with the second, the third with the fourth and so on. On the resulting array, `uncurry (+)` is applied to each pair, giving an array of half the length. This is composed using `->-` with a recursive call of `sum1` that computes the sum on that shorter array. This summation algorithm works for arrays of length a power of two. The `Int` argument to the `sum1` kernel should be the log base two of the length of the array.

Now, the CUDA code generated from `(sum1 3)` and an input size of eight is still single threaded. This is because of the decision that the length of the result array dictates the number of threads needed. What is needed is a way to state that the result of an array computation should be computed and stored as an intermediate array. This is accomplished using `sync`. In terms of computation, `sync` is the identity function on arrays but its use states that the array synced upon should be written to shared memory using a number of threads equal to the length of that array.

As an example to illustrate this concept, `sync` is added to the `sum1` kernel, here called `sum2`:

```
sum2 :: Int -> Arr IntE :-> Arr IntE
sum2 0 = pure id
sum2 n = pure op ->- sync ->- sum2 (n-1)
  where
    op  = fmap (uncurry (+)) . pair
```

The `sum2` kernel uses $n/2$ threads to sum up an array of length n. Below is first the code generated from `sum1` and then the code generated from `sum2` for comparison.

```
__global__ void sum1(word* input0,word* result0){
  const unsigned int tid = (unsigned int)threadIdx.x;
  const unsigned int bid = (unsigned int)blockIdx.x;
  ix_int(result0,(tid + bid)) =
    (((ix_int(input0,((tid << 3) + (8 * bid))) +
       ix_int(input0,(((tid << 3) | 0x1) + (8 * bid)))) +
      (ix_int(input0,(((tid << 3) | 0x2) + (8 * bid))) +
       ix_int(input0,(((tid << 3) | 0x3) + (8 * bid))))) +
     ((ix_int(input0,(((tid << 3) | 0x4) + (8 * bid))) +
       ix_int(input0,(((tid << 3) | 0x5) + (8 * bid)))) +
      (ix_int(input0,(((tid << 3) | 0x6) + (8 * bid))) +
       ix_int(input0,(((tid << 3) | 0x7) + (8 * bid))))));
}
```

```
__global__ void sum2(word* input0,word* result0){
  const unsigned int tid = (unsigned int)threadIdx.x;
  const unsigned int bid = (unsigned int)blockIdx.x;
  extern __shared__ unsigned int s_data[];
  word __attribute__((unused)) *sm1 = &s_data[0];
  word __attribute__((unused)) *sm2 = &s_data[4];
  ix_int(sm1,tid) =
    (ix_int(input0,((tid << 1) + (8 * bid))) +
     ix_int(input0,(((tid << 1) | 0x1) + (8 * bid))));
  if (tid < 2){
    ix_int(sm2,tid) =
      (ix_int(sm1,(tid << 1)) +
       ix_int(sm1,((tid << 1) | 0x1)));
  }
  if (tid < 1){
    ix_int(sm1,tid) =
      (ix_int(sm2,(tid << 1)) +
       ix_int(sm2,((tid << 1) | 0x1)));
  }
  if (tid < 1){
    ix_int(result0,(tid + bid)) =
      ix_int(sm1,tid);
  }
}
```

In the generated code for the `sum2` kernel, four stages can be seen. The first computes the sums of the neighboring pairs of elements at indices 0 and 1, 2 and 3 and so on:

```
 ix_int(sm1,tid) =
   (ix_int(input0,((tid << 1) + (8 * bid))) +
    ix_int(input0,(((tid << 1) | 0x1) + (8 * bid))));
```

The next stage does the same as the first but at this point the array is only 4 elements long. An `if` statement shuts two threads off and then threads 0 and 1 compute sums. This is followed again by a similar stage but where all but one thread are shut off. The very last stage uses a single thread to write the computed sum into the result array.

It is also possible to sum up an array using a combination of sequential and parallel computation. One way to do this is the following:

```
sum3 :: Int -> Arr IntE :-> Arr IntE
sum3 0 = pure id
sum3 n = pure op ->- (if (n <= 4)
                      then sync
                      else pure id) ->- sum3 (n-1)
  where
    op = fmap (uncurry (+)) . pair
```

Here, a normal Haskell conditional is used to decide whether to sync or not. If code is generated from the `sum3` program for 32 elements, chunks of 4 elements would be summed up sequentially per thread giving an array of length 8. The array of length 8 is then summed up using the parallel method.

Another kernel that does the same thing as the previous one is the following:

```
sum4 :: Int -> Arr IntE :-> Arr IntE
sum4 n = pure ((fmap (foldr (+) 0)) . chopN 4) ->- sync ->- sum2 n
```

The kernel chops the array up into an array of arrays where the inner arrays have a length of four. The inner arrays are summed up sequentially using `foldr`; the kernel then proceeds by letting the parallel `sum2` kernel sum up the resulting array. The `Int` argument to `sum4` tells how many stages the parallel summation should consist of. So to sum up an array of 32 elements this should be 3, because 32 divided by 4 is 8 and summing up 8 elements needs 3 stages.

The kernels, `sum3` and `sum4` can be executed on the GPU like this:

```
*Test> execute (sum3 5) [0..31 :: IntE]
[496]
*Test> execute (sum4 3) [0..31 :: IntE]
[496]
```

And the CUDA code generated for the two of them differ only in the way the sequential summation is parenthesized:

```
__global__ void sum3(word* input0,word* result0){
  const unsigned int tid = (unsigned int)threadIdx.x;
  const unsigned int bid = (unsigned int)blockIdx.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(input0,((tid << 2) + (32 * bid))) +
      ix_int(input0,(((tid << 2) | 0x1) + (32 * bid)))) +
     (ix_int(input0,(((tid << 2) | 0x2) + (32 * bid))) +
      ix_int(input0,(((tid << 2) | 0x3) + (32 * bid)))));
  if (tid < 4){
    ix_int(sm2,tid) =
      (ix_int(sm1,(tid << 1)) +
       ix_int(sm1,((tid << 1) | 0x1)));
  }
  if (tid < 2){
    ix_int(sm1,tid) =
      (ix_int(sm2,(tid << 1)) +
       ix_int(sm2,((tid << 1) | 0x1)));
  }
  if (tid < 1){
    ix_int(sm2,tid) =
      (ix_int(sm1,(tid << 1)) +
       ix_int(sm1,((tid << 1) | 0x1)));
  }
  if (tid < 1){
    ix_int(result0,(tid + bid)) = ix_int(sm2,tid);
  }
}


__global__ void sum4(word* input0,word* result0){
  const unsigned int tid = (unsigned int)threadIdx.x;
  const unsigned int bid = (unsigned int)blockIdx.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(input0,((tid << 2) + (32 * bid))) +
     (ix_int(input0,(((tid << 2) + 1) + (32 * bid))) +
      (ix_int(input0,(((tid << 2) + 2) + (32 * bid))) +
       ix_int(input0,(((tid << 2) + 3) + (32 * bid))))));
  if (tid < 4){
    ix_int(sm2,tid) =
      (ix_int(sm1,(tid << 1)) +
       ix_int(sm1,((tid << 1) | 0x1)));
  }
  if (tid < 2){
    ix_int(sm1,tid) =
      (ix_int(sm2,(tid << 1)) +
       ix_int(sm2,((tid << 1) | 0x1)));
  }
  if (tid < 1){
    ix_int(sm2,tid) =
      (ix_int(sm1,(tid << 1)) +
       ix_int(sm1,((tid << 1) | 0x1)));
  }
  if (tid < 1){
    ix_int(result0,(tid + bid)) = ix_int(sm2,tid);
  }
}
```

4.4.6 Sync and sequentiality

The number of threads needed to compute a result can be reduced by computing a number of elements in sequence. In Obsidian, this is done using the `sync` primitive. The `sync` primitive signals that the array supplied as input should be computed and written into memory. The number of threads used to compute the elements of the array is equal to the length of the array. So, one way to halve the number of threads used is to sync on an array of pairs instead of syncing on an array of scalar elements. For example:

```
incrP :: Num a => Arr a :-> Arr (a,a)
incrP = pure (fmap (+1)) ->- pure pair ->- sync
```

The CUDA code generated for the kernel above uses half as many threads as the `incrKernel` kernel implemented earlier:

```
__global__ void generated(word* input0,word* result0){
  const unsigned int tid = (unsigned int)threadIdx.x;
  const unsigned int bid = (unsigned int)blockIdx.x;
  extern __shared__ unsigned int s_data[];
  word __attribute__((unused)) *sm1 = &s_data[0];
  word __attribute__((unused)) *sm2 = &s_data[10];
  ix_int(sm1,(tid << 1)) =
    (ix_int(input0,((tid << 1) + (10 * bid))) + 1);
  ix_int(sm1,((tid << 1) + 1)) =
    (ix_int(input0,(((tid << 1) | 0x1) + (10 * bid))) + 1);
  ix_int(result0,((tid << 1) + (10 * bid))) =
    ix_int(sm1,(tid << 1));
  ix_int(result0,(((tid << 1) + 1) + (10 * bid))) =
    ix_int(sm1,((tid << 1) + 1));
}
```

Of course, this means that the resulting array is an array of pairs:

```
*Obsidian> execute incrP [0..9 :: IntE]
[(1,2),(3,4),(5,6),(7,8),(9,10)]
```

It is now up to the user of the values to unpair them and process them further.

Even though GPUs are capable of managing thousands or threads, there is a need to be able to compute more using fewer threads. For example, a kernel might reach the block limit of 512 or 1024 threads depending on the GPU. Another reason to compute more per thread is to be able to better hide memory latencies. If a kernel performs very little work per fetched data-element, the kernel's execution time will most likely be memory bandwidth bound. It is important to have enough computation going on to hide long memory latencies; otherwise processing elements will stand idle and wait for memory loads to finish.

Another way to introduce sequentiality is to split the array up into an array of arrays. Only the top level array is computed in parallel and the elements of the sub-arrays are computed in sequence. There is a function called `chopN` in the array library that splits an array up into an array of arrays. Using `chopN`, the kernel looks like this:

```
incrC :: (Flatten a, Num a) => Arr a :-> Arr (Arr a)
incrC = pure (fmap (+1)) ->- pure (chopN 5) ->- sync
```

The generated CUDA code for arrays of 10 elements is.

```
__global__ void generated(word* input0,word* result0){
  const unsigned int tid = (unsigned int)threadIdx.x;
  const unsigned int bid = (unsigned int)blockIdx.x;
  extern __shared__ unsigned int s_data[];
  word __attribute__((unused)) *sm1 = &s_data[0];
  word __attribute__((unused)) *sm2 = &s_data[10];
  for (int i0 = 0; i0 < 5; i0 ++){
    ix_int(sm1,((tid * 5) + i0)) =
      (ix_int(input0,(((tid * 5) + i0) + (10 * bid))) + 1);
  }
  for (int i0 = 0; i0 < 5; i0 ++){
    ix_int(result0,(((tid * 5) + i0) + (10 * bid))) =
      ix_int(sm1,((tid * 5) + i0));
  }
}
```

As can be seen from the code, syncing on an array of arrays means that the elements of the inner arrays are computed in sequential for loops.

In the kernel above, `chopN 5` was used. This means that the kernel can only correctly be used with arrays that are a multiple of 5 elements. There are no static checks that enforce this however.

4.4.7 Divide and conquer

General parallel composition of kernels is not efficient. Letting threads 0 to $n-1$ compute one kernel and threads n to $m-1$ another leads to serialization. In CUDA, you would express this using conditionals:

```
if (tid < n)
   A
else
   B
```

There are specific cases where this is not all that bad. If the values of n and $m-n$ are such that entire warps always choose the `then` or `else` branch, the execution takes place in parallel. In general, however, the code above leads to serialization and is then no different from sequential composition of kernels. The combinators described in this section circumvent this problem by implementing a more limited form of parallel composition. CUDA places one more severe restriction on the programs called `A` and `B` above. They must not contain calls to `__syncthreads`. Hence, `A` and `B` cannot be general kernels. The combinators shown in this sections make no such restriction on the kernel taken as input.

In the divide and conquer paradigm, problems are solved by being split up into smaller sub-problems that can be solved independently. The solutions to the sub-problems are then merged into a solution to the original problem. For the purpose of implementing divide and conquer algorithms of a certain kind, Obsidian provides a combinator called `two`. The `two` combinator is a special case of parallel composition but, as we have seen, general parallel composition is bad on the GPU. The `two` combinator is limited to compose in parallel two instances of the same kernel. This limitation enables the generation of efficient GPU code. The type of `two` is the following:

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

The type says that `two` takes a kernel (`Arr a :-> Arr b`) and that the result is again a kernel (`Arr a :-> Arr b`). The resulting kernel splits the input array down the middle into two parts. It then executes the input kernel on both halves and concatenates the two result arrays into a single result array. This combinator can be used to solve divide and conquer problems that have the property that both sub-problems are solved identically.

For example, `two` can be used to find the minimum element of an array. First the array is split in half and the minima of each half are found recursively. This is followed by simply selecting the smaller of the two minima as the result. In Obsidian, using `two`, this program is implemented as follows:

```
minimum :: Int -> Arr IntE :-> Arr IntE
minimum 0 = pure id
minimum n = two (minimum (n-1)) ->- pure min2 ->- sync
```

The array program `min2` computes the minimum element in an array of length 2. It is implemented using indexing, `(!)`, and `singleton` to create a one element result array:

```
min2 arr
     | len arr /= 2 = error "Wrong input size"
     | otherwise = singleton $ min a b
     where
       a = arr ! 0
       b = arr ! 1
```

Below is the code generated from `minimum` for eight inputs. One thing to notice is how the applications of `two` are implemented as indexing computations using bitwise operations. The implementation of `two` will be shown in section 5.4:

```
__global__ void minimum(word* input0,word* result0){
  const unsigned int tid = (unsigned int)threadIdx.x;
  const unsigned int bid = (unsigned int)blockIdx.x;
  extern __shared__ unsigned int s_data[];
  word __attribute__((unused)) *sm1 = &s_data[0];
  word __attribute__((unused)) *sm2 = &s_data[4];
  ix_int(sm1,tid) =
    min (ix_int(input0,(((tid << 1) & 0x6) + (8 * bid))),
         ix_int(input0,((((tid << 1) & 0x6) | 0x1) + (8 * bid))));
  if (tid < 2){
    ix_int(sm2,tid) = min (ix_int(sm1,((tid << 1) & 0x2)),
                           ix_int(sm1,(((tid << 1) & 0x2) | 0x1)));
  }
  if (tid < 1){
    ix_int(sm1,tid) = min (ix_int(sm2,0),ix_int(sm2,1));
```

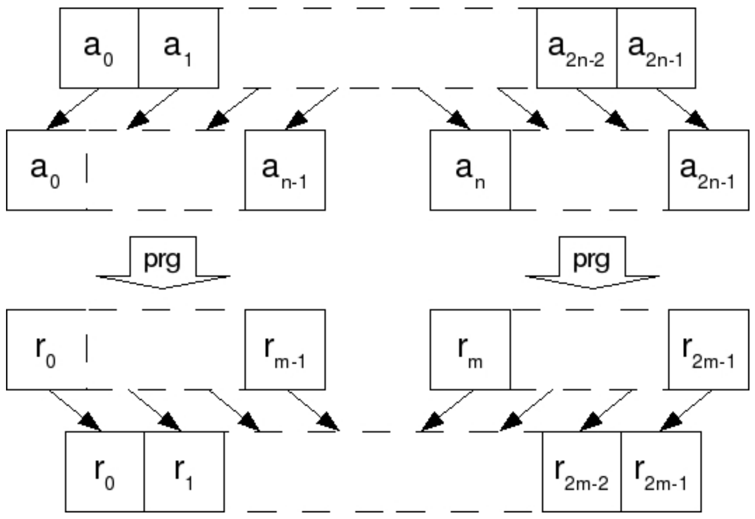

Figure 4.9: `two`, the divide and conquer combinator.

```
  }
  if (tid < 1){
    ix_int(result0,(tid + bid)) = ix_int(sm1,tid);
  }
}
```

Another combinator related to `two` is `ilv`. Instead of splitting the array in the middle, `ilv` splits the array into one array of its even indexed elements and another for the elements at odd indices. Then, just as in `two`, the same kernel is executed on both arrays.

While `two` is given as a primitive, `ilv` can be implemented using `two`, `riffle` and `unriffle`:

```
ilv :: (Arr a :-> Arr b) -> Arr a :-> Arr b
ilv f = pure unriffle ->- two f ->- pure riffle
```

For example, the `ilv` combinator can be used to implement mergers. A merger is a useful building block when implementing sorters. This use of `ilv` will be demonstrated in chapter 6.

Currently, using `ilv` generates quite complicated code. This may partly be because `ilv` is a more complicated operation than two. But there may also be room for improvement in the implementation of `ilv`. To illustrate the differences in the code generated from applications of `two` and `ilv`, the `minimum` kernel is implemented again using `ilv`

```
minimum2 :: Int -> Arr IntE :-> Arr IntE
minimum2 0 = pure id
minimum2 n = ilv (minimum2 (n-1)) ->- pure min2 ->- sync
```

Below is the code generated from this version of `minimun`. Notice how much larger the indexing expressions are compared to the generated code above.

```
__global__ void minimum2(word* input0,word* result0){
  const unsigned int tid = (unsigned int)threadIdx.x;
  const unsigned int bid = (unsigned int)blockIdx.x;
  extern __shared__ unsigned int s_data[];
  word __attribute__((unused)) *sm1 = &s_data[0];
  word __attribute__((unused)) *sm2 = &s_data[4];
  ix_int(sm1,tid) =
    min (ix_int(input0,((((((tid << 1) & 0x4) |
                        (tid & 0x1)) << 1) & 0x7) |
                        ((tid >> 1) & 0x1)) + (8 * bid))),
         ix_int(input0,(((((((tid << 1) & 0x4) | 0x2) |
                        (tid & 0x1)) << 1) & 0x7) |
                        ((((((tid << 1) & 0x4) | 0x2) |
                        (tid & 0x1)) >> 2) & 0x1)) + (8 * bid))));
  if (tid < 2){
    ix_int(sm2,tid) =
      min (ix_int(sm1,((tid << 1) & 0x2)),
           ix_int(sm1,(((tid << 1) & 0x2) | 0x1)));
  }
  if (tid < 1){
    ix_int(sm1,tid) =
      min (ix_int(sm2,0),ix_int(sm2,1));
  }
  if (tid < 1){
    ix_int(result0,(tid + bid)) = ix_int(sm1,tid);
  }
}
```

The generated kernel shown above is a very small example. The generated code is for eight elements. Comparing the performance of the `minimum` and `minimum2` kernel would not likely show much difference on 8 elements. Instead, the problem is that as larger and larger input size kernels are generated from `minimum`, the indexing calculation complexity remains constant. In the `minimum2` case, however, the indexing calculation keeps growing as the number of inputs increases. For larger input sizes, the difference in execution time is visible. The 256 element versions of `minimum` and `minimum2` have the following running time per block. The table also shows how much time it takes to only transfer the dataset on which the test is performed:

Kernel	Threads	ns / block	ns / block - transfer time
minimum	256	879	289
minimum2	256	1412	822
Data transfer	– –	590	– –

The figures above show that the version of minimum based on `two` is almost 3 times faster than that the one implemented using `ilv`.

The Obsidian implementation used in these examples applies some optimizations to the expressions generated. These optimizations are shown in section 5.4.5. The implementations of `two` and `ilv` are shown in section 5.4.4.

4.5 List of contributions

4.5.1 Compositionality

Obsidian makes combining kernels into larger kernels simpler in comparison to CUDA. CUDA, being based in C, has quite cumbersome syntax. To make a kernel perform the work of two given kernels often requires rewriting the two given kernels to make them work together. There is an example of this, although simple, in section 2.2.4.

Another example of how CUDA is limited in compositionality is in section 4.4.7. That section explains how CUDA makes it hard to reuse kernels as building blocks of other kernels because of how the `__syncthreads()` barrier synchronization works. In that section, it is also shown how Obsidian tackles that issue, through the introduction of the `two` combinator.

4.5.2 Combinators for parallel programming

Our intention is to come up with a set of combinators for parallel programming that ensures efficient execution on the GPU. One combinator for parallel programming that immediately springs to mind is `par P Q`. Given the programs `P` and `Q`, `par P Q` means execute `P` and `Q` in parallel. The `par` combinator is not one we would encourage use of when programming a GPU though. The GPU is an SIMD class computer and not suited for task parallelism within a kernel. `par P Q` would, when executed on the GPU, lead to serialization and `P` and `Q` would execute one after the other. For that reason, we include a more limited form of parallel composition that we call `two`. The `two` combinator takes a single program as argument that is intended to be executed in parallel with itself. Executing `P` in parallel with itself is possible to do efficiently on the GPU, since the threads will be executing the same instruction at all times, not giving rise to any serialization.

4.5.3 Data/work partitioning

When programming in CUDA, decisions about the number of threads to use and on how many data elements each thread should operate need to be taken early in the implementation of an algorithm. In Obsidian, these decisions can easily be made and changed all through the implementation process. One benefit of this is that experiments can be performed more rapidly and the most efficient data/work partitioning be found more easily. This ease of experimentation is a big advantage of raising the level abstraction at which programs are written.

4.5.4 Indexing computations

CUDA programs are parameterized over a thread identity. Often, this thread identity is used to compute the index of the element to use as input and output of the program. This summation kernel from section 2.2.3 illustrates the concept:

```
__global__ void sumKernel(float *result, float *x) {

  unsigned int tid = threadIdx.x;

  extern __shared__ float sm[];

  sm[tid] = x[blockIdx.x * blockDim.x + tid];

  for (int i = 1; i < blockDim.x; i *= 2) {
    __syncthreads();
    if (tid % (2*i) == 0) {
      sm[tid] += sm[tid + i];
    }
  }

  if (tid == 0) {
    result[blockIdx.x] = sm[0];
  }
}
```

In Obsidian, a summation kernel can be described as follows:

```
sum :: Int -> Arr IntE :-> Arr IntE
sum 0 = pure id
sum n = pure (pairwise (+)) ->- sync ->- sumUp (n-1)
```

Here, the indexing computations needed to assign elements to threads are taken care of behind the scenes by the pairwise combinator.

Chapter 5

Implementation

This chapter shows the implementation of two different versions of Obsidian. The two versions have a large common part, namely the array language and the operations available on scalar data. On top of the array language is the part of Obsidian that maps the array language computations onto the GPU and it is this part that has been implemented in two different ways.

The two different versions each have their strengths and weaknesses, which will be discussed in section 5.5.

5.1 Expressions

Now it is time to take a closer look at the expression type. We have already seen several kinds of expressions in use, `IndexE`, `IntE`, `FloatE` and `BoolE`. These four types are all actually type synonyms:

```
type IntE = Exp Int
type FloatE = Exp Float
type BoolE = Exp Bool
type IndexE = Exp Word
```

The `Exp a` type is a wrapper around an untyped expression data type called `DExp`:

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

Hence, all expressions in Obsidian are represented by the same `DExp` type. This is an example of *Phantom types* [15]. Phantom types can be used to provide a typed interface to an untyped structure.

The type `DExp` represents expressions that can be `Int`, `Float`, `Bool` or `Word` valued. The `DExp` type models a subset of C's expressions and can be used very directly in the generation of C code as will be shown in later sections.

```
data DExp = LitInt  Int
          | LitUInt Word32
          | LitBool Bool
          | LitFloat Float
          | Op2 Op2 DExp DExp
          | Op1 Op1 DExp
          | If DExp DExp DExp
          | Index Name [DExp] Type
            deriving(Eq,Show,Read)
```

The `Op1` data type represents some of the unary operations for which there is support on the GPU:

```
data Op1 = Not
         | BitwiseComp
         | Exp | Log | Sqrt
         | Log2i
         | Cos | Sin | Tan
         | CosH | SinH | TanH
         | ACos | ASin | ATan
         | ACosH | ASinH | ATanH
         | IntCastFloat
         | IntCastInt
         | IntCastUInt
         | FloatCastFloat
         | FloatCastInt
         | FloatCastUInt
         | UIntCastFloat
         | UIntCastInt
         | UIntCastUInt
```

The `Op2` data type contains a selection of binary operations:

```
data Op2 = Add | Sub | Div | Mul | Mod | Pow
         | And | Or
         | BitwiseAnd | BitwiseOr | BitwiseXor
         | Lt | Leq | Gt | Geq | Eq
         | Shl | Shr
         | Min | Max
         | Atan2
```

To make these expressions more pleasant to work with, suitable instances of `Num`, `Integral`, `Fractional` amongst others are created. It is also here that optimizations are applied. For example, `(+)` inspects its two arguments and performs operations such as constant folding. More details about the optimizations performed are shown in section 5.4.5.

5.2 Array language

In Obsidian, arrays are represented by a function from index to element and an integer representing the length. That the length is a normal Haskell `Int` is sometimes used in order to generate more efficient code.

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

There are a number of functions defined in order to allow the programmer to create and access arrays:

```
(!) (Arr f _) ix = f ix

len (Arr _ n) = n

mkArr f n = Arr f n

singleton = rep 1

rep n x = Arr (\ix -> x) n
```

Indexing into an array (`!`) is accomplished simply by applying the function part of the array to an expression of type `IndexE`. The `len` function returns the length. The three functions that are left create arrays in different ways. The function `mkArr` creates an array from a given indexing function and a length. `singleton` creates an array of length one given an element. It does so by using `rep` that creates an array that at each index contains the same given element.

Section 4.2 showed how to use some of the functions from the array library. This section will go into the details of how the functions in that library are implemented. To begin with, the functions `(!)`, `len` and `mkArr` are used to implement array reversal:

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

An array is reversed by creating a new array whose indexing function is looking up index $n - 1 - ix$ in the original array, where n is the length and ix is the index.

The implementation of the `pair` function from the array library is given below.

```
pair :: Arr a -> Arr (a,a)
pair arr | odd (len arr) = error "Pair: Odd n"
         | otherwise =
             mkArr (\ix -> (arr ! (ix ‘shiftL‘ 1),
                            arr ! ((ix ‘shiftL‘ 1 ) .|. 1))) nhalf
          where
            n = len arr
            nhalf = div n 2
```

The `pair` function requires the input array to be of even length. If the input array is of even length, an array of pairs is created. The element at index ix of the array of pairs is found by indexing in the original array at indices $ix * 2$ and $ix * 2 + 1$. The indexing computations are realized using bit shifts and bitwise `or`. A bitwise shift left one step is equivalent to multiplication by 2. Since the zeroth bit is now cleared, adding 1 and `or`ing with 1 gives the same result. In this case, the multiplication by 2 would most likely have been spotted by the CUDA compiler and automatically been replaced by a shift instruction during compilation. The reason array library functions are implemented using bitwise operations is that hopefully it will give more opportunity for optimizations to be applied by Obsidian. The optimizations Obsidian applies to expressions are shown in section 5.4.5.

Going back from an array of pairs to an array is also useful. In the library, the function `unpair` is implemented like this:

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

As the type shows, the `unpair` function requires the elements of the array to be in the `Choice` class. This is because a conditional is used in the indexing function to locate the sought element. This has some performance implications as well. The function `pair` is efficient; the code generated is simply some indexing arithmetic while the function `unpair` becomes a conditional (a diverging conditional), which is bad for performance.

Other functions that come in pairs like the `pair` and `unpair` function have the same issues, one of them being efficient and the other not. Two functions that have this issue are `halve` and `conc`. These two functions are implemented as:

```
halve :: Arr a -> (Arr a,Arr a)
halve arr = split (len arr `div` 2) arr

split :: Int -> Arr a -> (Arr a,Arr a)
split  m arr =
    let n  = len arr
        h1 = mkArr (\ix -> arr ! ix)  m
        h2 = mkArr (\ix -> arr ! (ix + (fromIntegral m))) (n-m)
    in  (h1,h2)

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

However, zipp and unzipp are both efficient:

```
zipp :: (Arr a, Arr b) -> Arr (a,b)
zipp (arr1,arr2) = mkArr (\ix -> (arr1 ! ix,arr2 ! ix)) n
    where n = min (len arr1) (len arr2)

unzipp :: Arr (a,b) -> (Arr a, Arr b)
unzipp arr =  (mkArr (\ix -> fst (arr ! ix)) n,
               mkArr (\ix -> snd (arr ! ix)) n)
    where n = len arr
```

In section 4.2, evens was implemented using fmap, pair and unpair. The function odds was also mentioned and will now be implemented. odds is a bit more complicated to implement because it has special cases. The first and the last element should be passed through unaffected while the middle elements should be processed using evens. To make this operation as efficient as possible, operations that introduce unnecessary conditionals should be avoided. One way to perform this operation is to split the input array into one array containing the first element and another containing the rest; then splitting the array containing the rest into two parts again, one with the last element removed and one array consisting only of that last element. Then evens could be performed on the middle part and the two one-element arrays concatenated back onto it. This means using conc twice to fuse the two removed elements back on, and giving nested conditionals, see the implementation of conc above. The implementation used here avoids one of the extra levels of conditionals by using array rotation functions (arrRotR and arrRotL) to rearrange the array using only arithmetic on the indices (no conditionals). The elements of the input array are rearranged so that the first and the last element end up next to each other at the end of the array. That array is then split once and evens is applied. Following this only a single conc is needed and a rotation of the elements in the opposite direction.

```
odds f arr = arrRotR 1 (conc (a1',a2))
  where
    arr' = arrRotL 1 arr
    (a1,a2) = split (len arr -2) arr'
    a1' = evens f a1
```

The two array rotation functions are implemented as follows:

```
arrRotL :: Int -> Arr a -> Arr a
arrRotL j arr =
   mkArr (\ix -> arr ! ((ix + (fromIntegral j)) `mod` fromIntegral n)) n
  where
    n = len arr
```

```
arrRotR :: Int -> Arr a -> Arr a
arrRotR j arr = mkArr (\ix -> arr ! ((ix + offs) `mod` fromIntegral n)) n
  where
    offs = fromIntegral (n-j)
    n = len arr
```

The functions `evens` and `odds` were justified by their usefulness in sorting applications. Two other functions that have applications in the same area are `riffle` and `unriffle`. These two functions compute very specific permutations of an array. The functions `pair`, `unpair`, `halve`, `conc`, `zip` and `unzip` can be used to implement the two functions `riffle` and `unriffle`:

```
unriffle = conc . unzip . pair

riffle = unpair . zipp . halve
```

Applying `unriffle` to an array named "input" of length 8 gives the following indexing expression:

```
((tid < 4) ?
  ix_int(input,(tid << 1)) :
  ix_int(input,(((tid - 4) << 1) | 0x1)))
```

If `tid` is below half the length of the array (`tid < 4`), `tid` is multiplied by two (shifted left one step). This means that threads 0 to 3 access the even indices of "input". If `tid` is 4 or higher, 4 is subtracted. For a thread id in the interval 0 to 7, being 4 or higher means that bit 2 is set. In the same interval, being less than 4 means that the same bit is not set. This means that the conditional can be entirely removed from the expression above by rotating the `tid` one bit to the left, moving the bit deciding "above or below" to position 0. Using this approach, the result of `unriffle` on the same input array is.

```
ix_int(input,(((tid << 1) & 0x7) | ((tid & 0x4) >> 2)))
```

The bits in use for the thread id have here been rotated using bitwise operations. It is possible to implement `riffle` in a very similar way but then the bit rotation goes in the other direction. Below are the implementations of `riffle` and `unriffle` using this bit rotation approach:

```
riffle' :: Arr a ->  Arr a
riffle' arr | even (len arr) =
               mkArr (\ix -> arr ! (rotLocalR ix bits) ) n
                  where
                    n = len arr
                    bits = fromIntegral $ intLog n
riffle' _ = error "riffle' demands even length"

unriffle' :: Arr a ->  Arr a
unriffle' arr | even (len arr) =
                 mkArr (\ix -> arr ! (rotLocalL ix bits) ) n
                   where
                    n = len arr
                    bits = fromIntegral $ intLog n
unriffle' _ = error "unriffle' demands even length"
```

These two functions are a bit more limited though. The functions `rotLocalL` and `rotLocalR` rotates the *bits* least significant bits one step. This means that not only does the length need to be even, but to produce correct results it also needs to be a power of two.

This section showed the implementation of a selection of functions that make up the array library. The other functions in this library are all implemented in a very similar way. The functions `riffle` and `unriffle` are examples of functions that can be implemented using existing primitives but are also given a lower level implementation for efficiency reasons. The need for giving more low level implementations of certain functions may be reduced by improved optimization techniques, but exploring this is future work.

5.3 The monadic approach

This section describes a version of Obsidian different from the one used in all the examples previously as well as the case studies in chapter 6. The information found here was previously published in [38] but the treatment here will be more in depth.

In this version of Obsidian, a GPU kernel has type `GArr a -> GPU (GArr b)`. The type `GArr` stands for global array, that is an array that resides in the GPU global memory. This version of Obsidian allows the programmer to choose to place arrays in global or shared memory. The type `GPU a` is a monad that contains some state needed during code generation. As the code is generated, it is stored into a state carried by this monad. More details about the `GPU a` type and code generation can be found in section 5.3.5.

The following kernel illustrates the differences between this version and the one used in the examples:

```
sum :: Int -> SArr IntE -> GPU (SArr IntE)
sum 0 = pure id
sum n = pure op ->- sync ->- sum (n-1)
  where
    op  = fmap (uncurry (+)) . pair
```

This program to sum up an array is almost identical to the `sum2` program in section 4.4.5. Only the type is different. Here, `SArr` means that an array in shared memory. So this program takes an array in shared memory and sums it up and gives the result as an array in shared memory. In this version of Obsidian, the `sum` kernel could also have been specified like this:

```
sum :: SArr IntE -> GPU (SArr IntE)
sum arr | len arr == 1 = return arr
        | otherwise    = (pure op ->- sync ->- sum) arr
  where
    op  = fmap (uncurry (+)) . pair
```

Here, guards can be used and the length of the input array decides whether to proceed recursively or not. This is not possible in the other version due to restrictions imposed by the type of GPU kernels used there.

In order to be able to run this on some data, an array needs to be placed in shared memory. For the purpose of moving arrays to and from shared memory there are two functions:

```
cache :: GArr a -> GPU (SArr a)
wb :: SArr a -> GPU (GArr a)
```

The function cache signals that when this array is synchronized it shall we written to shared memory. wb signals the opposite – that when synced upon this array should be written to global memory.

The two types GArr and SArr are defined like this:

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

type GArr a = Arr Global a
type SArr a = Arr Shared a
```

Apart from the extra type parameter s in the arrays, this array type is completely identical to that described in section 5.2. The only purpose of the extra type parameter is to keep track of which memory the array shall be stored in. It does not influence the implementation of the array language library functions at all.

Now, a kernel can be created by wrapping up the sum program with a cache and a wb:

```
kernel k = cache ->- k ->- wb ->- sync
```

Now this kernel can be executed just like the previous examples:

```
*Obsidian> execute (kernel sum) [0..7 :: IntE]
[28]
```

And the generated code, in this case, looks like this:

```
__global__ static void generated(int *source0,char *gbase){
  extern __shared__ char sbase[] __attribute__ ((aligned(4)));
  const int tid = threadIdx.x;
  const int n0 __attribute__ ((unused)) = 8;
  ((int *)(sbase+0))[tid] =
    (source0[(tid * 2)] + source0[((tid * 2) + 1)]);
  __syncthreads();
  if ((tid < 2)){
    ((int *)(sbase+16))[tid] =
      (((int *)(sbase+0))[(tid * 2)] +
       ((int *)(sbase+0))[((tid * 2) + 1)]);
  }
  __syncthreads();
  if ((tid < 1)){
    ((int *)(sbase+0))[tid] =
      (((int *)(sbase+16))[(tid * 2)] +
       ((int *)(sbase+16))[((tid * 2) + 1)]);
  }
  __syncthreads();
  if ((tid < 1)){
    ((int *)(gbase+0))[tid] =
      ((int *)(sbase+0))[tid];
  }
  __syncthreads();

}
```

This code is a bit crude compared to the other examples of generated code seen so far. For example, there are calls to `__synchthreads` where there need not be (and all of them are actually unnecessary). But the main difference is in the handling of memory. Here the arrays `sbase` and `gbase` are used to store the Obsidian arrays.

In this version of Obsidian, it is possible to synchronize arrays more freely than in the other on,e where a `sync` needs to encompass all data. For example, an array can be split in two halves and each half can be synchronized independently:

```
syncs :: SArr IntE -> GPU (SArr IntE)
syncs arr =
    do
      a1' <- sync a1
      a2' <- sync a2
      pure conc (a1',a2')
    where (a1,a2) = halve arr
```

The code generated from the example above needs to manage, in shared memory, two different arrays. It is not as clear as in the other version when arrays are alive or not. An array is considered alive if it is going to be used again in the future. The generated code for input arrays of length 8 looks like this:

```
__global__ static void generated(int *source0,char *gbase){
  extern __shared__ char sbase[] __attribute__ ((aligned(4)));
  const int tid = threadIdx.x;
  const int n0 __attribute__ ((unused)) = 8;
  if ((tid < 4)){
    ((int *)(sbase+0))[tid] = source0[tid];
  }
  __syncthreads();
  if ((tid < 4)){
    ((int *)(sbase+16))[tid] = source0[(tid + 4)];
  }
  __syncthreads();
  ((int *)(gbase+0))[tid] = ((tid < 4) ?
                             ((int *)(sbase+0))[tid] :
                             ((int *)(sbase+16))[(tid - 4)]);
  __syncthreads();

}
```

Here, two arrays are maintained in shared memory, the first starting at `sbase+0` and the second at `sbase+16` (4 32bit elements away).

The `GPU` Monad is implemented as follows:

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

data S = S { arrayId :: Int,
             symtab  :: SymbolTable }

type GPU a = StateT S (Writer IC) a
```

The state `S` contains a `SymbolTable` that is a mapping from names to types and sizes. The names in the symbol table refer to intermediate arrays created when generating the kernel. The state also holds an integer, `arrayId`, that is used to form the name of the next intermediate array to use. After creating a new intermediate array, `arrayId` is incremented. The monad also has a *Writer* that accumulates an intermediate form of the kernel. The information stored in the `SymbolTable` and the intermediate code (`IC`) are later used to create runnable CUDA code.

5.3.1 Pure

In this version of Obsidian, the `pure` function that makes a kernel out of an array language program is defined as follows:

```
pure :: (a -> b) -> a -> GPU b
pure f a = return $ f a
```

As can be seen, `pure` does not have any effect on the intermediate code being accumulated in the `GPU` monad.

5.3.2 Sync

Calling sync on an array implies that the contents of that array should be computed and stored in memory. This means that some restrictions are needed on what these arrays can contain. Arrays that can be written to memory should be of a type that is in the `Syncable` class:

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

The `syncable` class provides two functions, `sync` and `commit`. Of these, only `sync` is exposed to the programmer. The `commint` function is used in the implementation of `sync`. The most basic entity that can be synchronized is an array of one of the base types, `IntE`, `FloatE` and `BoolE`. The instance below shows how `sync` is implemented for one of these basic arrays.

```
instance TypeOf (Exp a) => Syncable (SArr (Exp a)) where
    sync arr = do
      arr' <- commit arr
      write [Synchronize]
      return arr'
    commit arr = do
      let n = len arr
      var  <- newSharedArray (typeOf (arr ! 0))  n
      write [(var,[unE tid]) ::= unE (arr ! tid)]
      return $ mkArr (\ix -> index var ix) n
```

`commit` gets a new identifier from the `GPU` monad using `newSharedArray`. Then it continues by accumulating a piece of code, an assignment, that computes the values of the input Obsidian array and stores them in the new shared memory array just created. The next step is to create a new Obsidian level array that indexes into the shared memory where the values just computed are stored.

`sync` calls `commit` but also inserts a `Syncronize` statement into the kernel code. The reason `sync` is split up into two functions, `sync` and `commit`, is explained by the following instance of `Syncable` for arrays of pairs:

```
instance (Syncable (Arr s a), Syncable (Arr s b)) =>
          Syncable (Arr s (a,b)) where
    sync arr = do
      arr' <- commit arr
      write $ \ix -> [Synchronize]
      return arr'
    commit arr = do
      (a1,a2) <- pure unzipp arr
      a1' <- commit a1
      a2' <- commit a2
      arr' <- pure zipp (a1',a2')
      return arr'
```

Arrays of pairs are synchronized by unzipping the array into two arrays that are committed separately. This is then followed by a single `Synchronize` in generated code. This means that no call to CUDA `__syncthreads()`is generated between the two assignments.

5.3.3 Sequential composition

In this version of Obsidian,The sequential composition operator is simply implemented using the `>=>` operator from the monad library.

```
(>=>) :: (Monad m) => (a -> m b) -> (b -> m c) -> (a -> m c)
```

In order to give a consistent look a new operator `->-` is defined simply by:

```
(->-) = (>=>)
```

5.3.4 The `two` combinator

One useful combinator is `two`, used in section 4.4.7 for example. In this version of Obsidian, it is hard to implement this combinator efficiently. The `two` combinator takes a kernel and applies it, preferably in parallel, to each half of an array.

In this version of Obsidian, the easy route is chosen and the applications of `f` to each half of the array happens sequentially. This is bad from a performance point of view and one of the main motivations for trying a completely different Obsidian implementation. It is possible that there is a way to make `two` more efficient even in this version, but currently it is implemented as follows:

```
two f arr = do
  (a1,a2) <- pure halve arr
  a1' <- f a1
  a2' <- f a2
  pure conc (a1',a2')
```

The poor performance of `two` in this version is the main motivation for the newer version described in 5.4.

5.3.5 Code generation

The CUDA generation process goes through a number of stages. First the Obsidian program is run. An Obsidian program is just a Haskell program whose output is something that can be turned into CUDA code. Running the Obsidian program produces two things: intermediate code and a symbol table that maps names to types and sizes. The intermediate code is of the following type:

```
data Statement
        = Synchronize
        | (Name,[DExp]) ::= DExp
        | IfThen BoolE [Statement]

type IC = [Statement]
```

The intermediate code is a list of statements that can be assignments to arrays or variables, conditional execution of a list of statements or a `Synchronize` statement (corresponding directly to CUDAs `__syncthreads()`). The conditional statements can only be introduced by the code generator at a later stage. The `IfThen` statement does not correspond to the Obsidian `ifThenElse` conditional.

Running the Obsidian program also results in a symbol table which is a structure of the following type:

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

To illustrate this, here is the `SymbolTable` and `IC` generated from the sum kernel (`kernel sum`) on page 59. First the symbol table in list syntax:

```
[ ("arr0",(Shared_Array Int,4))
, ("arr1",(Shared_Array Int,2))
, ("arr2",(Shared_Array Int,1))
, ("arr3",(Global_Array Int,1))
]
```

This symbol table holds four arrays. The first of these, `arr0`, is an array of four integers residing in shared memory.

Below is the intermediate code. Statements have been truncated to fit on a line:

```
[ ("arr0",[Index "tid" []]) ::= Op2 Add (Index "source0" ...
, Synchronize
, ("arr1",[Index "tid" []]) ::= Op2 Add (Index "arr0" ...
, Synchronize
, ("arr2",[Index "tid" []]) ::= Op2 Add (Index "arr1" ...
, Synchronize
, ("arr3",[Index "tid" []]) ::= Index "arr2" [Index "tid" []]
, Synchronize
]
```

The intermediate code is a list of eight statements in this case. The first of these computes the values that go into `arr0`. The symbol table above indicates that the length of `arr0` is four, so the variable `tid` in that statement must go from 0 to 3. If sequential code were to be generated, a `for` loop could be used to compute `arr0`:

```
for (int tid = 0; tid < 4; ++tid) {
  arr0[tid] = ...
}
```

In the `sum` example, the values in `arr0` will only be used to compute the values of `arr1`. The values of `arr1` will in turn only be used once, to compute the values of `arr2`. This means that the memory used for `arr0` can be freed up and reused as soon as `arr1` is computed. In order to discover when arrays can be freed, a liveness analysis is performed over the `IC`. The result of this analysis associates to each statement in the `IC` a set of arrays that are alive at that point:

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

Again from the sum program, `ICLive` looks like this:

```
[ (("arr0",[Index "tid" []]) ::= Op2 Add (Index "source0" ...),
    fromList ["arr0","source0"])
, (Synchronize,
   fromList ["arr0"])
, (("arr1",[Index "tid" []]) ::= Op2 Add (Index "arr0" ...),
    fromList ["arr0","arr1"])
, (Synchronize,
   fromList ["arr1"])
, (("arr2",[Index "tid" []]) ::= Op2 Add (Index "arr1" ...),
   fromList ["arr1","arr2"])
, (Synchronize,
   fromList ["arr2"])
, (("arr3",[Index "tid" []]) ::= Index "arr2" [Index "tid" []],
   fromList ["arr2","arr3"])
, (Synchronize,
   fromList [])
]
```

The `ICLive` object can now be used to form a new list of statements where the memory is being used more efficiently. This step introduces the arrays `gbase` and `sbase` in which all the data is stored. This step also attaches an integer to each statement that specifies how many threads are needed to compute it. For an assignment statement to an array, this integer is the length of the array and for any other statement (`Synchronize`) it is zero and will not be used. An example of the intermediate code at this stage is.

```
[ (("((int *)(sbase+0))",[Index "tid" []]) ::= ... ),
   4)
, (Synchronize,
   0)
, (("((int *)(sbase+16))",[Index "tid" []]) ::= ... ),
   2)
, (Synchronize,
   0)
, (("((int *)(sbase+0))",[Index "tid" []]) ::= ... ),
   1)
, (Synchronize,
   0)
, (("((int *)(gbase+0))",[Index "tid" []]) ::= ... ,
   1)
, (Synchronize,
   0)
]
```

Only one more step is needed to produce CUDA code: each of the integers above that symbolize the number of threads needed must be turned into a conditional that shuts off those threads not needed in this assignment. The generated CUDA kernel will run using the maximum of the number of threads needed by the statements. Only those assignments needing fewer than the maximum number of threads will be wrapped in a conditional. When this step has been performed, the generated code has reached its final form:

```
((int *)(sbase+0))[tid] =
   (source0[(tid * 2)] + source0[((tid * 2) + 1)]);
__syncthreads();
if ((tid < 2)){
  ((int *)(sbase+16))[tid] =
     (((int *)(sbase+0))[(tid * 2)] +
       ((int *)(sbase+0))[((tid * 2) + 1)]);
}
__syncthreads();
if ((tid < 1)){
  ((int *)(sbase+0))[tid] =
    (((int *)(sbase+16))[(tid * 2)] +
     ((int *)(sbase+16))[((tid * 2) + 1)]);
}
```

```
__syncthreads();
if ((tid < 1)){
  ((int *)(gbase+0))[tid] =
    ((int *)(sbase+0))[tid];
}
 __syncthreads();
```

5.4 The arrow based approach

The version of Obsidian that is described in this section, builds upon the lessons learnt in the implementation of the monadic Obsidian described previously. Here, kernels are more restricted in a way that makes code generation easier. The version described here is also simplified in the sense that it does not allow the programmer to control whether an array should be stored in global or shared memory. Kernels here perform their computations entirely within shared memory.

It is in this version of Obsidian that all the examples in section 4.4 and chapter 6 are implemented.

In this version of Obsidian, a GPU kernel has the type `a :-> b`. This type has two constructors, `Pure` and `Sync`:

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

Here, a kernel is a sequence of computations interspersed with barrier synchronizations, see figure 4.7. Compared to the monadic version described earlier, this makes thinking about memory easier. The kernel data type, `:->`, only allows syncing on data that can be turned into a single array (the `a -> Arr FData`, in the type). Also notice that a kernel is simply a single sequence of such synced computations so all data that is needed must be piped through all the way. This means that the CUDA code can be generated having just two arrays that the data is ping-ponged between at the syncs. This choice of always using two arrays is not entirely ideal. There are cases where the computation could be performed *in place*. In that case it would have been more space efficient to only use a single array worth of storage. Another issue arises when the computation leaves parts of the array unchanged. In that situation, unchanged data will be copied unnecessarily. The current implementation of Obsidian does not try to solve these issues.

5.4.1 Pure

The function pure that creates a GPU kernel corresponds directly to the constructor Pure of the :-> type:

```
pure :: (a -> b) -> a :-> b
pure = Pure
```

5.4.2 Sync

The implementation of sync is not as direct as that of pure. This is what the implementation of sync looks like:

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

The type FData represents things that can be written to the GPU memory. The class Flatten provides two functions, toFData and fromFData, to convert to and from a format storeable in memory. There are instances of Flatten for all the basic types, IntE, FloatE and BoolE, as well as for arrays and pairs of things that can be flattened.

The FData type is defined as follows:

```
data InternalRepr a
  = Nil
  | Unit a
  | Tuple [InternalRepr a]
  | For (Arr (InternalRepr a))

type FData = InternalRepr (DExp, Type)
```

A base type such as an IntE is turned into FData using the Unit constructor:

```
instance Flatten IntE where
  toFData (E a) = Unit (a, Int)
  fromFData (Unit (a,Int)) = E a
```

A pair is turned into `FData` by using the `Tuple` constructor and arrays are turned into `FData` using the `For` constructor. In the generated code, data represented by the `For` constructor is computed in a `for` loop. This means that if the type synced upon is an array of arrays, the inner arrays are computed by sequential for loops:

```
instance Flatten IntE where
  toFData a = For (fmap toFData a)
  fromFData (For a) = fmap fromFData a
```

5.4.3 Sequential composition

The type, `:->`, of GPU kernels is very similar to normal Haskell lists. There is a unit GPU kernel, created by `pure`, corresponding to a list of length one. A GPU kernel can also be a sync of something that produces an array of `FData` followed by a GPU program. This resembles the `:` operator on lists. So, a kernel corresponds to a list of computations separated by barrier synchronizations.

With this idea in mind, it is a good time to look at the implementation of the sequential composition operator (->-):

```
(->-) :: (a :-> b) -> (b :-> c) -> (a :-> c)
Pure f ->- Pure g   = Pure (g . f)
Pure f ->- Sync g h = Sync (g . f) h
Sync f h1 ->- h2    = Sync f (h1 ->- h2)
```

Composing two kernels that are both created with `Pure` is the same as Haskell function composition. The next case, where `pure f` is composed with `Sync f h`, means that some more computation should take place before the sync, that is the kernel should perform `g.f` then sync and continue with `h`. The last case is something constructed using `Sync` composed with anything else. In this case `->-` proceeds recursively.

5.4.4 The `two` combinator

As explained in section 4.4.7, the task of the `two` combinator is to make a special kind of parallel composition efficient on the GPU.

The `two` combinator takes a kernel, `k`, with type `(Arr a :-> Arr b)` as input. The type of `two k` is `(Arr a :-> Arr b)` but operating on arrays twice as long as those `k` is designed for. What `two` does is take a kernel and run it twice in parallel on both halves of an input array. Since the `:->` type only allows synchronization across all threads, the two instances of the kernel must be woven together. This could be done naively like this:

```
two :: (Arr a :-> Arr b) -> Arr a :-> Arr b
two (Pure f)   = Pure $ twof f
two (Sync f g) = Sync (twof f) (two g)

twof :: Choice b =>  (Arr a -> Arr b) -> (Arr a -> Arr b)
twof f arr = conc (a1',a2')
    where
      (a1,a2)   = halve arr
      (a1',a2') = (f a1,f a2)
```

But taking this approach will result in generated code with conditionals checking "is this thread in the low half of the array or not?". The code generated from `two incr` illustrates this:

```
__global__ void generated(word* input0,word* result0){
  const unsigned int tid = (unsigned int)threadIdx.x;
  const unsigned int bid = (unsigned int)blockIdx.x;
  ix_int(result0,(tid + (8 * bid))) =
     ((tid < 4) ?
      (ix_int(input0,(tid + (8 * bid))) + 1) :
      (ix_int(input0,(tid + (8 * bid))) + 1));
}
```

One possible code transformation on the above code is to move the conditional inwards. This can be accomplished using the following alternative implementation of `twof`:

```
twof' :: (Arr a -> Arr b) -> Arr a -> Arr b
twof' f arr =
    mkArr (\i -> f (
            mkArr (\j -> arr ! (ifThenElse (top i)
                                          (j + fromIntegral m)
                                          j)) m) ! (bot i)) nl
    where
```

```
      m = len arr `div` 2
      top i = i >=* (fromIntegral m)
      bot i = i `mod` (fromIntegral m)
      nl = len arr
```

This version of `twof` is a bit limited as it takes no regard to functions `f` that alter the length of the array. This detail will be corrected again in the final version of `twof` shown further down. But first the code generated for the example (`two incr`) using the above version of `twof` looks like this:

```
__global__ void generated(word* input0,word* result0){
  const unsigned int tid = (unsigned int)threadIdx.x;
  const unsigned int bid = (unsigned int)blockIdx.x;
  ix_int(result0,(tid + (8 * bid))) =
    (ix_int(input0,(((tid >= 4) ?
                     ((tid % 4) + 4) :
                     (tid % 4)) + (8 * bid))) + 1);
}
```

Now, this code still contains a conditional. However it is possible to completely optimize this away. In Obsidian, this is done by implementing `two` using bitwise operations and not conditionals:

```
twof :: (Arr a -> Arr b) -> Arr a -> Arr b
twof f arr =
    Arr (\i -> f (
          Arr (\j -> arr ! ((i .&. num2) .|. j)) n2) !
                                   (i .&. mask)) n
    where
      n       = len arr
      n2      = n `div` 2
      bit     = logInt n2

      num2    = fromIntegral $ 2^bit
      mask    = complement num2
```

The bitwise operations in the above code listing, `(i .&. num2) .|. j`, take the bit that tells whether to index in the high part or the low part from `i`. This is done using `(i .&. num2)`. `num2` points out which bit decides the division. The index, `j`, indexes into the half size arrays, created by the division. `j` and the division bit are ored together; creating the index into the original array. All that is left now is to make this work correctly when `two` is used with a function whose input array and output array differ in length. This is done by moving the interesting bit from `i` after obtaining it. If `f` halves the length of the array this means that the interesting bit (the division bit) is located one step to the left compared to earlier and needs to be shifted right before being used.

```
twof :: (Arr a -> Arr b) -> Arr a -> Arr b
twof f arr =
    Arr (\i -> f (
          Arr (\j -> arr ! ((sh bit bit2 (i .&. num2)) .|. j)) n2) !
                                (i .&. mask)) nl
    where
      n2       = len arr `div` 2
      bit      = logInt n2

      bit2     = logInt nl2
      num2     = fromIntegral $ 2^bit2
      mask     = complement num2

      nl      = 2 * nl2
      nl2     = len (f (Arr (\j -> arr ! variable "X") n2 ))

sh :: (Bits a) => Int -> Int -> a -> a
sh b1 b2 a | b1 == b2 = a
           | b1 <  b2 = a `shiftR` (b2 - b1)
           | b1 >  b2 = a `shiftL` (b1 - b2)
```

This version of `twof` also takes care of functions that alter the length of the array. It does this by computing `f` applied to a symbolic array and checking the length of the result. This illustrates how the the fact that array lengths are Haskell integers is used. Now, generating the code for `two incr` gives the code below. This result is due to the optimizations on expressions described in section 5.4.5:

```
__global__ void generated(word* input0,word* result0){
  const unsigned int tid = (unsigned int)threadIdx.x;
  const unsigned int bid = (unsigned int)blockIdx.x;
  ix_int(result0,(tid + (8 * bid))) =
          (ix_int(input0,(tid + (8 * bid))) + 1);
}
```

Repeated applications of two, such as those that occur when recursively using `two` to generate kernels, leads to complicated indexing expressions. The following section considers some transformations on expressions that manage fairly well to reduce the size of the indexing computations in the case of nested applications of `two`.

5.4.5 Optimization of expressions

This version of Obsidian applies a number of transformations on expressions. These transformations have been chosen in a rather ad-hoc way by generating code and investigating that generated code, by hand, for optimization candidates. The transformations are added to the actual implementations of the operators that they operate on, not as a separate pass over the expressions once they are created in their entirety. For example, the implementation of `(+)` has a special case for $a + 0$ resulting in just a. As a result of this, each transformation is only given very local knowledge of the expression it is optimizing. A greedy approach has been chosen; a transformation is applied if it results in fewer operations being used (given its very local knowledge) or if inspection of the generated code shows that the transformation was beneficial. This leads to the problem that a transformation that seems beneficial given some kinds of program will be detrimental in other situations. This problem is acknowledged and placed on the future work list for investigation.

An important aspect here is that operations such as (`+`), `shiftL` and `.&.` (for bitwise `and`) on the Obsidian types (`IntE`, `FloatE`, `BoolE`) build expression trees. This means that the operations will, in the end, find their way into the CUDA code. For example if the Obsidian (`+`) operation is used, which is defined as part of a Haskell `Num` instance for expressions, it will correspond to a CUDA addition in the generated code. It is, however, not always necessary that there is a one to one correspondence between the operations expressed by the programmer in the Obsidian code and the operations that end up in the CUDA code. For example, if the Obsidian (`+`) operation is used with two known constants, it is better to add them directly and let the resulting sum show up the generated CUDA. The operations that can be performed in Haskell and thus not end up in the generation will be called compile time operations and those that end up in the generated code will be referred to as runtime operations.

This approach, applying transformations to expressions as they are created, is called smart constructors. For more details on smart constructors see for example [12].

Bit shift operations on unsigned integers

In Obsidian, thread ids are unsigned integers. The optimizations described here are applied to unsigned integer expressions as they are created in an Obsidian program. The purpose of this is to make the indexing calculations in the resulting generated CUDA code more efficient.

Shifting left or right by zero has no influence:

```
a ‘shiftL‘ 0 = a
a ‘shiftR‘ 0 = a
```

Shifting a constant:

```
(E (LitUInt a)) ‘shiftL‘ n = E (LitUInt (a ‘shiftL‘ n))
(E (LitUInt a)) ‘shiftL‘ n = E (LitUInt (a ‘shiftR‘ n))
```

This is an example of where a value is computed at compile time. That is, the `shiftL` and `shiftR` operations on the right hand side are evaluated instantly and result in a constant in the generated code.

Shifting twice in the same direction:

```
shiftL (E (Op2 Shl a (LitInt n1))) n = (E a) ‘shiftL‘ (n+n1)
shiftR (E (Op2 Shr a (LitInt n1))) n = (E a) ‘shiftR‘ (n+n1)
```

Shifting some bit pattern in the same direction by some distances twice is the same as shifting it once by the combined distance. Here, the Haskell `Bits` type class helps us by specifying that the shift distance argument to the bit shift operations (`shiftL` and `shiftR`) is always a constant.

Shifting left then right or right then left by same amount:

```
shiftL (E (Op2 Shr a (LitInt n1))) n
          | n == n1 = (E a) .&. (complement (fromIntegral (2^n-1)))

shiftR (E (Op2 Shl a (LitInt n1))) n
          | n == n1 = (E a) .&. (fromIntegral (2^(32-n)-1))
```

This allows two shift operations to be replaced by a single `and` operation.

Shifts distribute over bitwise and:

```
shiftL (E (Op2 BitwiseAnd a b)) n
       | eitherLit a b =
       ((E a) `shiftL` n) .&. ((E b) `shiftL` n)

shiftR (E (Op2 BitwiseAnd a b)) n
       | eitherLit a b =
       ((E a) `shiftR` n) .&. ((E b) `shiftR` n)
```

This transformation is not as easy to justify given the limited local knowledge upon which the decision to apply a transformation or not is made. The reason is that the transformation does not give fewer operations on the right hand side. The `eitherLit` condition on a and b is true if either of them is a constant known at compile time. This condition makes sure that at least applying this transformation does not increase the number of operations in the given expression.

shifting and bitwise or:

```
shiftL (E (Op2 BitwiseOr a b)) n
        | eitherLit a b =
        ((E a) `shiftL` n) .|. ((E b) `shiftL` n)
shiftR (E (Op2 BitwiseOr a b)) n
        | eitherLit a b =
        ((E a) `shiftR` n) .|. ((E b) `shiftR` n)
```

Bitwise and on unsigned integers

Bitwise and with zero is zero:

```
(.&.) (E (LitUInt 0)) a  = 0
(.&.) a (E (LitUInt 0))  = 0
```

Bitwise and with 0xFFFFFFFF, identity:

```
(.&.) a (E (LitUInt 0xffffffff)) = a
(.&.) (E (LitUInt 0xffffffff)) a = a
```

Under some circumstances `(a .|. b) .&. c = a .&. c`. This happens when `b`'s and `c`'s binary representations are completely disjoint, meaning that `b .&. c = 0`. This rule is applied as follows:

```
(.&.) (E (Op2 BitwiseOr a (LitUInt b)))
      (E (LitUInt c))
      | b .&. c == 0 = (E a) .&. (E (LitUInt c))
```

Because of commutativity of bitwise `and` there is also a version of the above transformation with the arguments flipped:

```
(.&.) (E (LitUInt c))
      (E (Op2 BitwiseOr a (LitUInt b)))
      | b .&. c == 0 = (E a) .&. (E (LitUInt c))
```

Shifting right introduces zeros at the left end of a binary number. This transformation detects the scenario where a value is shifted right n steps and thereafter bitwise `and`ed with a number that leaves the value unchanged. This happens whenever the number being `and`ed with, consists of all ones except for the n leftmost bits. The n leftmost bits, can be either one or zero. This transformation is applied because this scenario occurs in some examples of generated code.

```
(.&.) a@(E (Op2 Shr _ (LitInt n)))
        (E (LitUInt x)) | lookup n patterns == (Just x) = a
```

The `patterns` list simply lists tuples of shift distances and corresponding bit pattern. If the value `and`ed with is such that according to the list it will influence the result at all, it is discarded.

Most of the transformations described in this section come in a second version as well, the only difference being the order of the arguments.

Bitwise or on unsigned integers

Transformations on expressions containing bitwise `or` have so far been harder to find. Indexing expressions generated by Obsidian when using nested applications of `ilv` result in sequences of `or` interleaved with `and`s that the present set of transformations does very little good on.

There are some simple transformation to apply to expressions containing `or`. The first one is bitwise `or` with zero.

Bitwise or with zero

```
(.|.) a (E (LitUInt 0)) = a
(.|.) (E (LitUInt 0)) a = a
```

The rule above states that bitwise `or`ing something with zero leaves that something unchanged. Using bitwise `or` with a number consisting of all bits set, leads to a result with all bits set.

Bitwise or with `0xffffffff`

```
(.|.) a (E (LitUInt 0xffffffff)) = E (LitUInt 0xffffffff)
(.|.) (E (LitUInt 0xffffffff)) a = E (LitUInt 0xffffffff)
```

The last transformation takes care of the case where both inputs to bitwise `or` are know at compile time. As seen before, this leads to an expression not containing the bitwise `or` operation.

Bitwise or between two constants

```
orOptU (E (LitUInt a)) (E (LitUInt b)) = E (LitUInt (a .|. b))
```

Unexplored

Bitwise `xor` (exclusive or) is interesting but has not been considered during the implementation of the optimizations or the combinators of Obsidian. Bitwise `xor` relates to bitwise `and` and `or` like this:

```
a ‘xor‘ b = (a .&. ~b) .|. (~a .&. b) = ~(a .&. b) .&. (a .|. b)
```

Using these identities may be key to making some cases of nested `and`s and `or`s in the generated code disappear. This is left as work to investigate for the future.

5.4.6 Code generation

In order to generate C code from an Obsidian description, it is first necessary to gather some information about the program. In order to be able to run the kernel correctly, the number of threads needed and the amount of shared memory required needs to be discovered. All intermediate arrays in the computation will be stored in

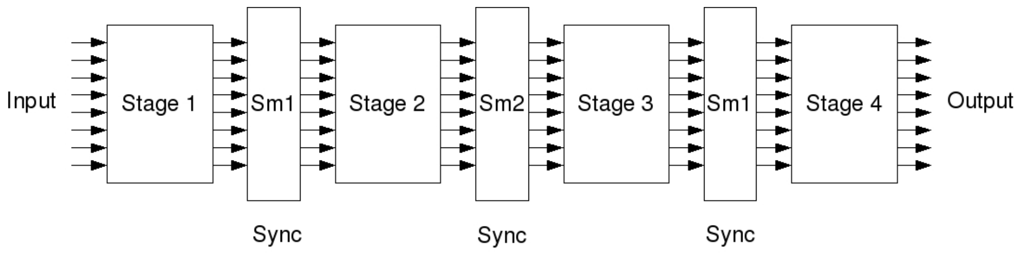

Figure 5.1: A Kernel alternately using the arrays sm1 and sm2 for storage.

the GPU shared memory. A GPU kernel can be seen as a sequence of computations separated by barrier synchronizations. The amount of shared memory that is needed for a kernel in total is the maximum of the memory requirements of the computations separated by syncs times 2. The memory requirement is twice the maximum size because two shared memory arrays will be used alternately, see figure 5.1. So the memory requirement of a kernel is found by going through the entire program and measuring the size of each intermediate array. The number of threads needed to write an intermediate array is the same as the length of that array. So the number of threads needed is given by the maximum length intermediate array. The pseudo code below illustrates how finding the shared memory requirements can be done:

```
sharedMemReq :: (a :-> b) -> a -> Int
sharedMemReq (Pure _) = 0
shaerdMemReq (Sync f g) input =
  maximum (memory_needed_for (f input))
          (sharedMemReq g (f input))
```

A `Pure` computation needs no shared memory. The result is written directly to the output in global memory. In the `Sync` case, enough memory to store the result of `f` applied to `input` is needed.

When the number of threads and the amount of shared memory needed has been computed, it is time to generate the actual code. When generating the code for a particular stage in the kernel, the number of threads needed to compute that stage is needed. If the number of threads needed for the kernel in total is larger than the number of threads needed for a given stage, the code for that stage is enclosed in a conditional that disables a number of threads. Examples of where this happens are some of the summation kernels in section 4.4.5.

Now, the code generation process will be gone through for a hypothetical example, figure 5.2. The description of the code generation process assumes two functions, `f` and `g`, of the following types:

```
f :: Arr IntE -> Arr IntE
g :: Arr IntE -> Arr IntE
```

The kernel for which code is generated is.

```
example :: Arr IntE :-> Arr IntE
example = pure (fmap f) ->- sync ->- pure rev ->-
           sync ->- pure (fmap g)
```

The data structure that represents the `example` kernel has the following form:

```
Sync f1 (Sync f2 (Pure f3))
```

Here `f1`, `f2` and `f3` are the functions:

```
f1 = fmap toFData . fmap f :: Arr IntE -> Arr FData
f2 = fmap toFData . rev . fmap fromFData :: Arr FData -> Arr FData
f3 = fmap g . fmap fromFData :: Arr FData -> Arr IntE
```

Given this data structure, CUDA code is generated by applying the function `f1` to a symbolic, named, input array:

```
input = mkArr (\ix -> index "input" ix Int) 256
```

Notice how the input array used needs to provide a length, an integer. The result of applying `f1` to `input` is an array of `FData`. This array of `FData` is used to create a C assignment statement by examining the shape of the elements of the array. In this case, the element is something created with the `Unit` constructor of the `FData` type constructor and hence a single assignment is needed:

```
sm1[tid] = f(input[tid]);
__syncthreads();
```

Had the array synced upon been an array of pairs, the `Tuple` constructor would have been used in the element and two assignments would have been generated:

```
sm1[tid << 2] = f(input[tid << 2]);
sm1[tid << 2 + 1] = f(input[tid << 2 + 1]);
__syncthreads();
```

In the generated code, `sm1` is used since it is the first time data is written into shared memory. Now, a new symbolic array must be created to pass to the next function, `f2`. This array must represent indexing into `sm1`. Now, the function `f2` is applied to this array. `f2`'s main function is the reversal of the array which will be visible in the indexing:

```
sm2[tid] = sm1[256 - 1 - tid];
__syncthreads();
```

The same procedure is repeated for the last stage, function `f3`, but since we have arrived at an instance of the `Pure` constructor we know this is the last computation of the kernel. The result of this function should be written to the output array of the kernel:

```
output[tid] = g(sm2[tid]);
```

In its entirety the kernel generated by the procedure described above looks like this:

```
sm1[tid] = f(input[tid]);
__syncthreads();
sm2[tid] = sm1[256 - 1 - tid];
__syncthreads();
output[tid] = g(sm2[tid]);
```

The CUDA code above looks slightly different to the examples of real generated code shown in other places in this document. The difference is that the code shown uses a couple of macros for indexing into arrays. There is one macro for indexing into the array, treating it as an array of integers, called `ix_int`. There is also a similar macro for indexing into the array as an array of floating point values, called `ix_float`. The data itself is stored in arrays of 32-bit words (`unsigned int`). The reason for this is that a choice was made to store arrays of pairs as a single array where the elements of the pairs are laid out next to each other. This means that if Obsidian is working on arrays of `(IntE,FloatE)` pairs, integer and floating point values will be interleaved in the CUDA level array.

This section gave an outline of how CUDA code is generated from an Obsidian kernel. Details have been left out, but the key ideas and concepts are present in the description.

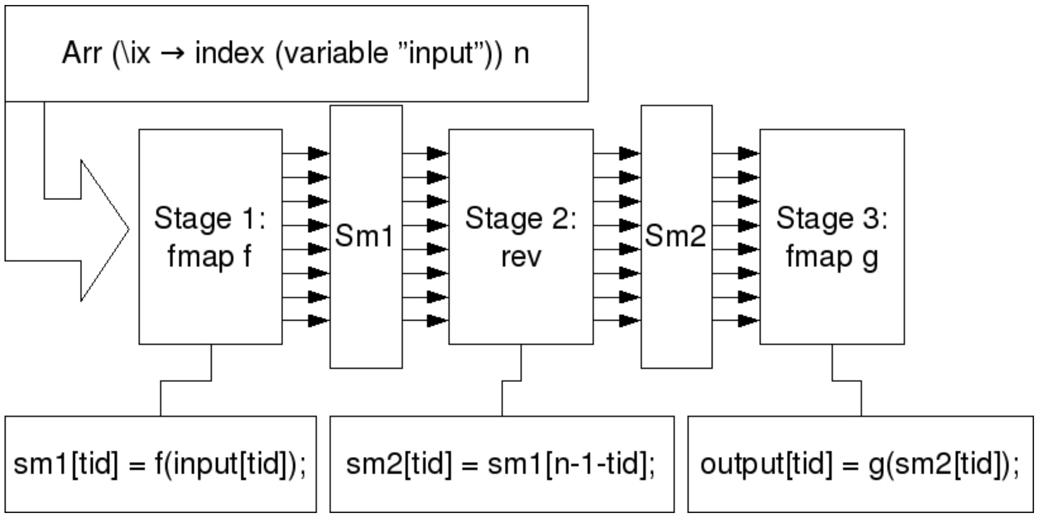
Arr (\ix → index (variable "input")) n
Stage 1: fmap f
Sm1
Stage 2: rev
Sm2
Stage 3: fmap g
sm1[tid] = f(input[tid]);
sm2[tid] = sm1[n-1-tid];
output[tid] = g(sm2[tid]);

Figure 5.2: A sketch of how code is generated for a kernel.

5.4.7 Sidetrack

Sections 4.4.5 and 4.4.6 show how `sync` can be used to guide code generation and how data elements can be grouped for processing in sequence by a single thread. For example, elements can be paired up using `pair` and then one thread can be used to compute on them. Currently, this method of work grouping has an unwanted side effect. Paired up elements are stored next to each other when applying `sync`; this means that in some cases data-elements would be moved in memory needlessly, just for the purpose of work grouping. The needless movement of data also imposes extra overhead by requiring additional indexing calculations.

The example that showed work grouping by pairing looked like this:

```
incrP :: Num a => Arr a :-> Arr (a,a)
incrP = pure (fmap (+1)) ->- pure pair ->- sync
```

This example instructs each thread to operate on two neighboring elements. It is, however, also possible to instruct the threads to operate on the elements at index tid and $tid + n/2$, where n is the length of the array. This could be done with using something like:

```
syncNHalf = pure (pair . riffle) ->- sync ->- pure (unriffle . unpair)
```

What is sought after is a method to specify that thread *tid* should compute the result at index *tid* and $tid + n/2$ without the need to move those two elements next to each other and then move them back. An experiment in this direction was performed but did not lead to fully satisfactory results. A new synchronization primitive called `syncHow` was added to Obsidian. This synchronization primitive took an extra argument of type `How`:

```
syncHow :: How -> Arr a :-> Arr b
```

The `how` argument contains instructions on how the data is supposed to be written to memory. Assume a function `pairNth :: int -> How` that creates a `How` object that instructs thread *tid* to compute value at index *tid* and $tid + n$. With this function, a new version of `incrP` can be implemented:

```
incrP :: Num a => Arr a :-> Arr (a,a)
incrP = pure (fmap (+1)) ->- syncHow (pairNth 4)
```

The code generated by this `incrP` for eight elements uses four threads to compute the eight result values:

```
__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(result,tid) = (ix_int(input,tid) + 1);
  ix_int(result,(tid + 4)) = (ix_int(input,(tid + 4)) + 1);
}
```

The generated code above shows a small discrepancy with the other examples of generated code. It does not make use of the `blockIdx` to compute on several blocks as the other kernels do. This is simply because that was not part of Obsidian at the time this experiment was performed.

Because of the complexity it introduced, this sidetrack is left largely unexplored. It is possible that with a decent set of combinators for creating these `How` objects it could lead to something good. Those combinators would need to contain operations that are array length dependent for example.

At the time of this experiment, the `How` type was implemented as a function from a thread id to a list of thread ids. Each thread was associated with a list of virtual thread ids by the `How` function. An obvious risk with this is that the programmer supplies functions that make no sense here. This should have been hidden away as an abstract data type only allowing certain unproblematic `How` objects to be created. It is also somewhat unclear what should happen with these `How` objects when kernels are composed into larger kernels.

There is one more problem with this approach. One of the goals of Obsidian is to free the programmer from the kind of indexing computations that are so common in data-parallel programming using CUDA. But this approach in some sense puts that indexing computing back into the hands of the programmer again.

5.5 Comparison of the two implementations

In this section, the two implementations shown in section 5.3 and section 5.4 are compared. The version described in section 5.3 is referred to as monad Obsidian and that described in section 5.4 is called arrow Obsidian.

Monad Obsidian was described first. In this version of Obsidian a GPU kernel is just a function, a monadic function, taking an array as input. This lets the programmer implement kernels using guards and conditionally choose different paths depending on array length. This cannot be done in arrow Obsidian. For example, a kernel that computes the maximum value of an array implemented using guards looks like this:

```
maxKernel :: SArr IntE -> GPU (SArr IntE)
maxKernel arr | len arr == 1 = return arr
              | otherwise    =
                  (two maxKernel ->- pure maxOfTwo ->- sync) arr
   where
     maxOfTwo arr = singleton (max (arr ! 0) (arr ! 1))
```

It feels very natural when implementing this kernel to use the length of the input array to make the decision on whether to split and recurse or not. In arrow Obsidian, the version used in the examples, an extra argument that specifies the recursion depth would be required:

```
maxKernel :: Int -> Arr IntE :-> Arr IntE
```

Monad Obsidian also allows more freedom in the use of `sync`. In arrow Obsidian, all data needed in the future has to be streamed through the syncs. At each application of `sync`, all the data needed in the future must be stored and re-read. This makes monad Obsidian more pleasant to work with and also allows for more efficient implementations of some kernels. Kernels that split their input data in parts and leave some parts unused until later are more natural in monad Obsidian. In arrow Obsidian, such a kernel would need to store and read back unchanged elements from memory at all synchronizations until the point where the data is used. However, the cost of this freedom is the extra work that takes place in the monad Obsidian code generation.

Monad Obsidian has efficiency problems when it comes to the divide and conquer combinator. It may of course be possible to improve upon this performance but as it is now this is the main motivation for starting work on the other version.

One strength of arrow Obsidian is the simplified code generation procedure enabled by the restricted view on kernels. In this version, it is clear that at most two memory arrays need to be managed. The first of these arrays contains all the data before the `sync` and the other one contains all the data after the `sync`. There is no need for any liveness analysis during code generation in this version.

Also, the abstract data type used to describe kernels enables the more efficient implementation of the divide and conquer combinator, `two`.

5.6 Future work

There are areas where Obsidian needs improvement. The efficiency of generated code is one such place. One thing that is very apparent when looking at the generated code is that CSE, Common Subexpression Elimination, would be beneficial. Applying a CSE step is future work. When applying CSE in the setting of Data-parallel GPGPU programming, some issues should be considered. The balance to strike between recomputation and register storage space is most likely different from that of single threaded CPU programs. Memory access to local shared memory in the GPU is no more expensive than register access in current generation GPUs.

The optimizations described in section 5.4.5 are selected on a rather ad-hoc basis. A firmer grasp of what kind of optimizations are needed must be obtained. Another important aspect of this is that it is always better to not generate code that needs a lot of optimization in first place. This may be accomplished by thinking more closely about what kind of combinators make the most sense both to the programmer and the

GPU architecture. The optimizations currently implemented are completely context unaware. For example, it may happen that code such as `tid & 0x7` is generated at a place where it is impossible for `tid` to be above 7. Taking care of cases such as this would require some knowledge about what range a variable is in at this particular place of the program. Finally the role that bitwise `xor` might play during optimization of expressions must also be investigated.

The two versions of Obsidian show some interesting symmetries. The `two` combinator results in efficient code in the arrow based version and poor code in the monad based version. Consider instead a combinator, `one`, that splits an array in two parts just like `two` does but only applies the input kernel to one of the halves instead. The `one` combinator is very easily implemented in monad Obsidian and leads to generated code that is very efficient. In the arrow based version, because of the limitation that all data must be streamed through all the uses of `sync`, this leads to code that performs a lot of unnecessary copying back and forth of unchanged data. This seems to suggest that the arrow based approach should be augmented with more operations in the representation of kernels to enable passing unchanged data along efficiently. However, it is probable that generalizing the arrow based version will lead to the need to reintroduce some of the complexities of the monad based version. This may be a necessary step to take though and will be evaluated as future work.

Section 5.4.7 illustrated a problem that arises in both versions of Obsidian, that work grouping and data placement are co-dependent. It would be nice if it were possible to let a single thread operate on more than one data-element without imposing unnecessary data-movement and or superfluous indexing computations. Finding a satisfactory solution to this problem is left for investigation as future work.

The two versions of Obsidian described here take the step to CUDA rather directly. It may be desirable to find a more general intermediate language to compile the Obsidian programs into and then turn that into CUDA or some other parallel language such as OpenCL. Besides being less tied to a single platform, an approach such as this may provide an easier setting for performing code transformations before generating the final CUDA kernels.

Currently, Obsidian is limited in one very fundamental way. The kinds of algorithms that can be expressed must have the property that from the size of the inputs alone the size of the outputs are determined. In some sense this means that the kind of kernels we can express in Obsidian are hardware circuit like. Allowing the programmer to express algorithms that are more general and data-dependent is desired. However, it does introduce extra complexity. Exploring what these complexities are and how to circumvent them is left as future work.

Kernels are not the whole story, when it comes to GPGPU programming; there is also the level above that, which deals with the coordination of kernels. Usually, the program to accelerate on the GPU is split up into many parts that can be solved by launching groups of kernels. This coordination of kernels is not expressible in Obsidian currently. One approach to adding this capability would be to add a second embedded language, a kernel coordination language. For example, this language might be based on Haskell/CUDA bindings.

Chapter 6

Case studies

6.1 Reduction

Reduction is a generalization of the concept seen in section 4.4.5 with the summation kernels. Given a sequence S of length n and an operation, $\oplus$ (that needs to be associative to enable parallelism), the reduction of the array using the operation is defined as $S_0 \oplus S_1 \oplus ... \oplus S_n$.

This kernel highlights one of the strengths of a higher level language compared to CUDA. In Obsidian, it is possible to make the reduction kernel parameterized on the operation to use. From this parameterized description, code for particular reductions, such as sum or max, can be generated.

The reduction kernel can, just like sum, be implemented in many different ways. First the basic parallel way from the `sum2` example is.

```
reduce :: Flatten a => Int -> (a -> a -> a) -> Arr a :-> Arr a
reduce 0 f = pure id
reduce n f = pure op ->- sync ->- reduce (n-1) f
  where
   op = fmap (uncurry f) . pair
```

Now, reduce can be used to compute for example sums, minimum and maximum

```
*Obsidian> execute (reduce 3 (+)) [0..7 :: IntE]
[28]
```

```
*Obsidian> execute (reduce 3 min) [0..7 :: IntE]
[0]

*Obsidian> execute (reduce 3 max) [0..7 :: IntE]
[7]
```

Reduce can also be used to sum up an array of 3D vectors. A 3D vector type can be defined as follows:

```
type Vec3 a = (a,a,a)
```

And addition of 3D vectors can be defined like this:

```
vecPlus :: Num a => Vec3 a -> Vec3 a -> Vec3 a
vecPlus (x1,y1,z1) (x2,y2,z2) = (x1+x2,y1+y2,z1+z2)
```

Now, reduce is applicable to arrays of 3D vectors without change:

```
*Obsidian> let input = replicate 8 (1,1,1) :: [Vec3 IntE]
*Obsidian> execute (reduce 3 vecPlus) input
[(8,8,8)]
```

All the tweaks applied in the summation examples, section 4.4.5, are applicable to reduce as well. Instead, a different change to the kernel is performed here. What reduce does is apply the operation to neighboring elements; it is also possible to split the input array in half and apply the operation between the first elements of both arrays, the second elements and so on. This can be implemented using `zipWith`:

```
reduce' :: Flatten a => Int -> (a -> a -> a) -> Arr a :-> Arr a
reduce' 0 f = pure id
reduce' n f = pure (zipWith (uncurry f) . halve)  ->- sync ->-
                reduce' (n-1) f
```

This change to the reduction kernel affects the memory access patterns of the threads executing the generated code. Below is code generated from (`reduce' 3 (+)`):

```
__global__ void generated(word* input0,word* result0){
  const unsigned int tid = (unsigned int)threadIdx.x;
  const unsigned int bid = (unsigned int)blockIdx.x;
  extern __shared__ unsigned int s_data[];
  word __attribute__((unused)) *sm1 = &s_data[0];
  word __attribute__((unused)) *sm2 = &s_data[4];
  ix_int(sm1,tid) =
    (ix_int(input0,(tid + (8 * bid))) +
     ix_int(input0,((tid + 4) + (8 * bid))));
  if (tid < 2){
    ix_int(sm2,tid) =
      (ix_int(sm1,tid) + ix_int(sm1,(tid + 2)));
  }
  if (tid < 1){
    ix_int(sm1,tid) =
      (ix_int(sm2,tid) + ix_int(sm2,(tid + 1)));
  }
  if (tid < 1){
    ix_int(result0,(tid + bid)) = ix_int(sm1,tid);
  }
}
```

This code is generated for an input array of eight elements. In the first stage, thread i accesses element i and $i + 4$ not $i * 2$ and $i * 2 + 1$, expressed using shifts and or, as seen in the snippet of code below from `sum2`:

```
  ix_int(sm1,tid) =
    (ix_int(input0,((tid << 1) + (8 * bid))) +
     ix_int(input0,(((tid << 1) | 0x1) + (8 * bid))));
```

There are no performance measurements of the reduction kernel alone presented in this thesis. There are however comparisons of dot product kernels differing only in which reduction kernel they use. These comparisons are shown in section 6.7.

6.2 Dot product

Section 2.2.3 showed how to implement a dot product kernel in CUDA. In this section, a number of different ways to implement dot product in Obsidian will be shown. As the CUDA example illustrated, the dot product algorithm consists of a multiplication part and a summation part. For the summation part, the kernels `reduce` or `reduce'` can be used.

The multiplication kernel is defined using `zipWith` from the array library, section 4.2:

```
mult :: Num a => (Arr a,Arr a) :-> Arr a
mult = pure $ zipWith (uncurry (*))
```

The `mult` kernel takes two input arrays. These arrays are then multiplied together element-wise using `zipWith`.

The CUDA code generated from `mult`, shown below, is less general than the hand-written version in section 2.2.3. The generated code has the constant 256 where the handwritten version uses `blockDim.x`. The reason for this is that Obsidian, at the moment, has no means to express constructs dependent on `blockDim`. There is no block concept exposed to the Obsidian programmer.

```
__global__ void mult(word* input0,word* input1,word* result0){
  const unsigned int tid = (unsigned int)threadIdx.x;
  const unsigned int bid = (unsigned int)blockIdx.x;
    ix_float(result0,(tid + (256 * bid))) =
      (ix_float(input0,(tid + (256 * bid))) *
       ix_float(input1,(tid + (256 * bid))));
}
```

As a building block in other kernels the Obsidian `mult` kernel is general. It can be combined with other kernels and it is just at the very end, the code generation, that a fixed size has to be chosen.

The dot product algorithm can now be implemented using either `reduce` or `reduce'`. The Obsidian dot product kernels below correspond to the fused multiply and summation kernel, `mult_sumKernel` in section 2.2.4:

```
dotProd :: (Flatten a, Num a) => Int -> (Arr a, Arr a) :-> Arr a
dotProd n = mult ->- sync ->- reduce n (+)

dotProd' :: (Flatten a, Num a) => Int -> (Arr a, Arr a) :-> Arr a
dotProd' n = mult ->- sync ->- reduce' n (+)
```

The code generated from `dotProd` for 256 elements is shown below. The `dotProd'` code looks very similar but uses another access pattern into memory

```
__global__ void dotProd(word* input0,word* input1,word* result0){
  const unsigned int tid = (unsigned int)threadIdx.x;
  const unsigned int bid = (unsigned int)blockIdx.x;
  extern __shared__ unsigned int s_data[];
  word __attribute__((unused)) *sm1 = &s_data[0];
  word __attribute__((unused)) *sm2 = &s_data[256];
  ix_float(sm1,tid) =
    (ix_float(input0,(tid + (256 * bid))) *
     ix_float(input1,(tid + (256 * bid))));
  __syncthreads();
  if (tid < 128){
    ix_float(sm2,tid) =
      (ix_float(sm1,(tid << 1)) +
       ix_float(sm1,((tid << 1) | 0x1)));
  }
  __syncthreads();
  if (tid < 64){
    ix_float(sm1,tid) =
      (ix_float(sm2,(tid << 1)) +
       ix_float(sm2,((tid << 1) | 0x1)));
  }
  __syncthreads();
  if (tid < 32){
    ix_float(sm2,tid) =
      (ix_float(sm1,(tid << 1)) +
       ix_float(sm1,((tid << 1) | 0x1)));
  }
  if (tid < 16){
    ix_float(sm1,tid) =
      (ix_float(sm2,(tid << 1)) +
       ix_float(sm2,((tid << 1) | 0x1)));
  }
  if (tid < 8){
    ix_float(sm2,tid) =
      (ix_float(sm1,(tid << 1)) +
       ix_float(sm1,((tid << 1) | 0x1)));
  }
  if (tid < 4){
    ix_float(sm1,tid) =
      (ix_float(sm2,(tid << 1)) +
       ix_float(sm2,((tid << 1) | 0x1)));
  }
  if (tid < 2){
    ix_float(sm2,tid) =
      (ix_float(sm1,(tid << 1)) +
       ix_float(sm1,((tid << 1) | 0x1)));
  }
  if (tid < 1){
     ix_float(sm1,tid) =
       (ix_float(sm2,(tid << 1)) +
        ix_float(sm2,((tid << 1) | 0x1)));
  }
  if (tid < 1){
    ix_float(result0,(tid + bid)) = ix_float(sm1,tid);
  }
}
```

In Obsidian, there are a number of transformations that can be applied to the dot product kernel very easily. For example, The multiplications can be *fused* with the first stage of the reduction. This fusing means that the multiplication does not take place as a discrete step followed by a sync. This transformation also leads to the kernel computing its value using half as many threads. All that needs to be done to perform this change to the kernel is to remove the `sync` that separates the `mult` and the `reduce`:

```
dotProd1 :: (Flatten a, Num a) => Int -> (Arr a, Arr a) :-> Arr a
dotProd1 n = mult ->- reduce n (+)

dotProd1' :: (Flatten a, Num a) => Int -> (Arr a, Arr a) :-> Arr a
dotProd1' n = mult ->- reduce' n (+)
```

The effect this transformation has on running time is shown in section 6.7.

6.3 Mergers

A merger has the property that if it is given two sorted sequences as input it merges them into one sorted output sequence. These mergers can then be used in the

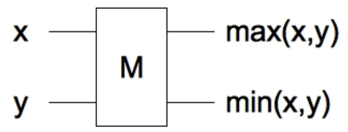

Figure 6.1: Small merging network.

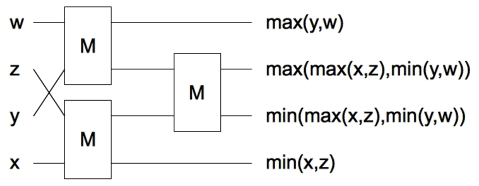

Figure 6.2: A four input merging network.

implementation of sorting kernels. This section shows how two well known merging networks, Odd-even merge and Batcher's Bitonic merger [3], can be implemented in Obsidian.

A four element version of the odd-even merger will be illustrated in pictures. The smallest merging network can be thought of as a component that takes two input values and outputs them in order, as illustrated by figure 6.1.

Larger merging networks can be built by connecting together a number of small ones. Figure 6.2 shows how a merger that merges two sequences of length two can be built using the small two input merger.

The first merger implemented in this section is called the odd-even merger. More information about this merger can be found in [3]. The right-hand picture in figure 6.3 illustrates this merger's access pattern.

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

The function `mergeOE` above describes the network parameterized over a two input merger component. The definition of `mergeOE` states that given two odd-even mergers that take n inputs, a merger that operates on $2n$ elements can be implemented. This is done by interleaving the inputs to the two smaller mergers and then combining the results with a last use of `odds`. What was just described is the recursive decomposition of the odd-even merger. This recursion has its base in the application of the two input/two output merger component. At the base case, the array is 2 elements long and `evens` is used as a kind of type conversion allowing `f` to operate on two element arrays rather than pairs.

The two input/two output merger component that is used in the examples here is shown below:

```
cmp :: (Choice  a, Ordered a) => (a, a) -> (a, a)
cmp (a,b) = (min a b, max a b)
```

If the input array to odd-even merge is an even length array where the first and second halves are individually sorted, then the output is a sorted array:

```
*Obsidian> let input = [0,2,4,6,1,3,5,7 :: IntE]
*Obsidian> execute (mergeOE cmp 3) input
[0,1,2,3,4,5,6,7]
```

The butterfly network can also be used to implement mergers [9]. A merger implemented using the butterfly network merges two sequences where the first one is sorted and the second one is sorted in the reversed order. The picture on the left side in figure 6.3 illustrates the butterfly network.

```
bfly :: (Choice a, Flatten a) =>
        ((a,a) -> (a,a)) -> Int -> (Arr a :-> Arr a)
bfly f 0 = pure id
bfly f n = ilv (bfly f (n-1)) ->- sync ->- pure (evens f)
```

The butterfly code is slightly simpler than the odd-even merge code. Odd-even merge has a special case `evens` at the bottom of the recursion that `bfly` does not have.

```
*Obsidian> let input = [0,2,4,6,7,5,3,1 :: IntE]
*Obsidian> execute (bfly cmp 3) input
[0,1,2,3,4,5,6,7]
```

Both `mergeOE` and `bfly` use `ilv`. As a result of using `ilv`, the CUDA kernels generated from these mergers are inefficient. A merger that does not use `ilv` can be constructed by using the shuffle exchange network. The shuffle exchange network (`shex`) is implemented by sequentially composing a number of stages that all consist of `(evens f . riffle)`. Composing several stages sequentially can be done in Obsidian using a function called `compose`:

```
compose :: [a :-> a] -> a :-> a
compose [] = pure id
compose (x:xs) = x ->- compose xs
```

To implement `shex`, n (log base two of array length) riffle-evens interspersed by syncs are composed. In this implementation of `shex` the more efficient `riffle'` is used:

```
shex f n = compose prg
  where
    stages = replicate n (pure (evens f . riffle'))
    prg    = intersperse sync stages
```

`shex` behaves exactly as `bfly` did. It takes an array where the first half is sorted and the second half is sorted in the reverse order and outputs a sorted array:

```
*Obsidian> let input = [0,2,4,6,7,5,3,1 :: IntE]
*Obsidian> execute (shex cmp 3) input
[0,1,2,3,4,5,6,7]
```

The code generated for an eight input `shex` looks like this:

```
__global__ void shex(word* input0,word* result0){
  const unsigned int tid = (unsigned int)threadIdx.x;
  const unsigned int bid = (unsigned int)blockIdx.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) =
    (((tid & 0x1) == 0) ?
     min (ix_int(input0,((tid >> 1) + (8 * bid))),
          ix_int(input0,(((tid >> 1) | 0x4) + (8 * bid)))) :
     max (ix_int(input0,((tid >> 1) + (8 * bid))),
          ix_int(input0,(((tid >> 1) | 0x4) + (8 * bid)))));
  ix_int(sm2,tid) =
    (((tid & 0x1) == 0) ?
     min (ix_int(sm1,(tid >> 1)),
          ix_int(sm1,((tid >> 1) | 0x4))) :
     max (ix_int(sm1,(tid >> 1)),
          ix_int(sm1,((tid >> 1) | 0x4))));
  ix_int(result0,(tid + (8 * bid))) =
    (((tid & 0x1) == 0) ?
     min (ix_int(sm2,(tid >> 1)),
          ix_int(sm2,((tid >> 1) | 0x4))) :
     max (ix_int(sm2,(tid >> 1)),
          ix_int(sm2,((tid >> 1) | 0x4))));
}
```

This code may not look very nice at first glance. There are a lot of repeated calculations that could be removed using common subexpression elimination. However, it is not that bad. Section 6.7 shows some running time measurements of the kernels generated in this section.

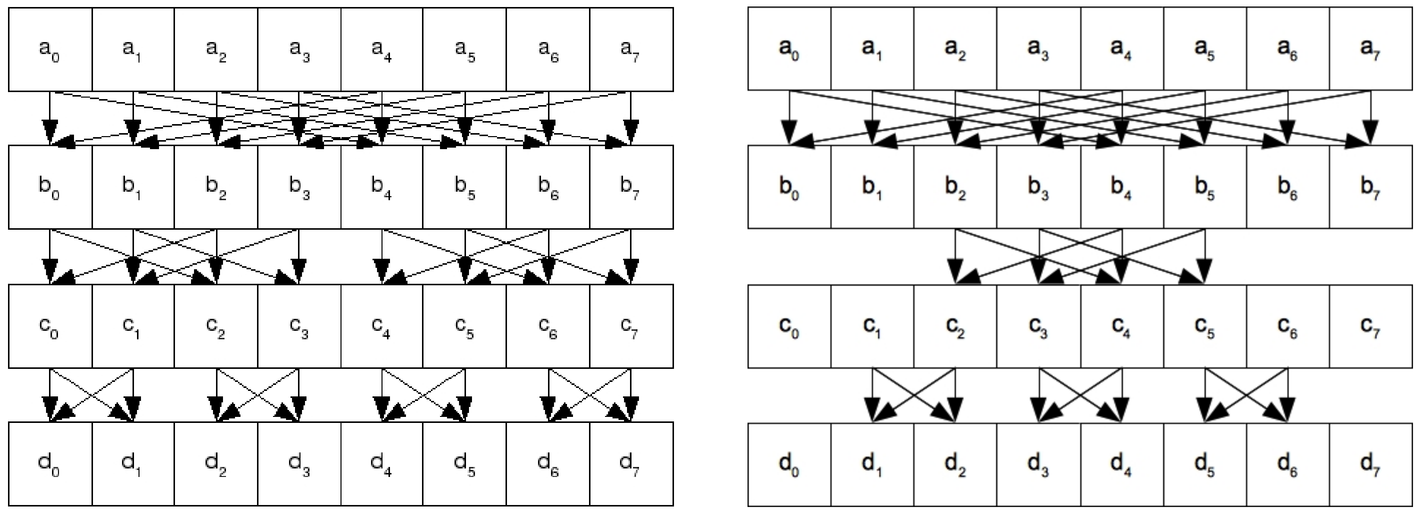
a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7
b_0 b_1 b_2 b_3 b_4 b_5 b_6 b_7
c_0 c_1 c_2 c_3 c_4 c_5 c_6 c_7
d_0 d_1 d_2 d_3 d_4 d_5 d_6 d_7

Figure 6.3: The Butterfly network (left) and the Odd-Even merging network (right).

6.4 Sorting networks

In this section, a number of sorters will be implemented in Obsidian. Sorting on GPUs has been explored using many different approaches, see for example [34, 7, 32].

The sorters presented here are so-called sorting networks. They are algorithms that have a fixed number of compare and swap operations, independent of the actual data set being sorted. See [3] for more details about sorting networks.

The first algorithm is Odd-even transposition sort. This is the "bubble-sort" of sorting networks. It is not the most efficient sorting network but because of its simplicity it is a good place to start.

Odd-even transposition sort is implemented using the array language building blocks `evens` and `odds`. Figure 6.4 shows a four input version of odd-even transposition sort. The sorter is implemented by alternating between using `evens` and `odds`. If the input array is of length $2n$, n stages of `evens` and n stages of `odds` are needed. Another way to look at it is that in total there are n stage of combined `even-odds` stages. This is how the Obsidian version is implemented. In the Obsidian code below a stage is defined to contain one instance of `evens` composed with one of `odds`. This stage is then composed n times using the function `compose` from the previous section:

```
sortOET :: (Flatten a, Ordered a, Choice a) => Int -> Arr a :-> Arr a
sortOET n = compose (replicate n stage)
  where
    stage = pure (evens cmp) ->- sync ->- pure (odds cmp) ->- sync
```

The next sorting network is called odd-even merge sort. This sorter is built around the odd-even merger from section 6.3. This merger can be applied to an array of $2n$ elements where each half (n elements) is sorted. The result is a sorted array of $2n$ elements. The strategy is to recursively build this network so that at the bottom neighboring pairs of elements are merged. In the next step arrays of 4 elements are merged with arrays 4 elements and so on. The approach is: split the array in two halves, sort each half recursively and merge the results. In Obsidian this pattern is accomplished using two:

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

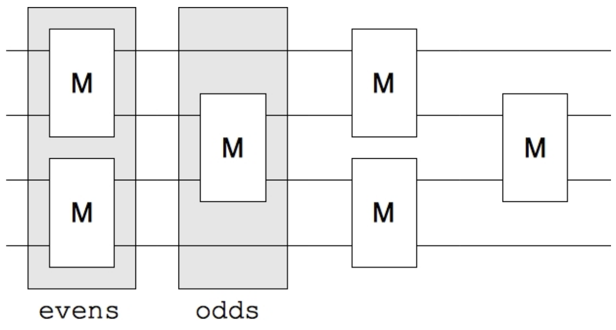

Figure 6.4: A four input version of odd-even transposition sort.

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

Bitonic sort is implemented using a similar recursive approach. This sorter uses the butterfly merger instead. The butterfly merger requires one of the arrays to be merged to be sorted in the reversed order. The implementation needs to make sure that this is the case by reversing the corresponding part of the array before every merge. The resulting Obsidian code is.

```
sortB :: Int -> (Arr IntE :-> Arr IntE)
sortB 0 = pure id
sortB n = two (sortB (n-1)) ->- pure revHalf ->- sync ->- bfly n cmp
   where
     revHalf arr = let (a1,a2) = halve arr
                   in  conc (a1, rev a2)
```

```
*Obsidian> execute (sortB 3) [0,7,1,2,3,6,5,4 :: IntE]
[0,1,2,3,4,5,6,7]
```

Bitonic sort can also be implemented using the shuffle-exchange network defined in section 6.3. Since `bfly` and `shex` have the same input output behavior, `shex` can be plugged in directly.

```
sortBshex :: Int -> (Arr IntE :-> Arr IntE)
sortBshex 0 = pure id
sortBshex n = two (sortBShex (n-1)) ->- pure reverseHalf ->- sync ->- shex cmp n
          where reverseHalf arr = let (a1,a2) = halve arr
                                  in  conc (a1,rev a2)
```

Section 6.7 compares the execution time of the sorting algorithms implemented in this section. In this comparison there is also a handwritten version of bitonic sort for reference.

6.5 Parallel prefix

This section shows the implementation of a parallel prefix (also called scan) kernel known as `sklansky` after J. Sklansky[35].

The prefix sum of a sequence, $s = s_0, s_1, \ldots, s_n$, given an associative binary operator $\oplus$, is a new sequence a such that:

$$\begin{aligned}
&a_0 = s_0\\
&a_1 = s_0 \oplus s_1\\
&a_2 = s_0 \oplus s_1 \oplus s_2\\
&\ldots\\
&a_n = s_0 \oplus ... \oplus s_n
\end{aligned}$$

Since the operator $\oplus$ is associative, the prefix sums can be computed in many different ways. For more information on prefix networks, see for example [5].

Figure 6.5 shows the recursive decomposition of the Sklansky parallel prefix network. The Sklansky algorithm is implemented by splitting the input array in two halves and recursively applying Sklansky to both halves. The two sub-results are then joined by applying the operation between the element at the highest index of the first result with 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))
```

Now, a kernel `sklansky` can be implemented like this:

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

The result of executing `sklansky` on the GPU looks as follows:

```
*Obsidian> execute (sklansky 3 (+)) [0..7 :: IntE]
[0,1,3,6,10,15,21,28]
```

The code generated from `sklansky 7 (+)` is shown below. This is the 128 element version of the code:

```
__global__ void sklansky128(word* input0,word* result0){
  const unsigned int tid = (unsigned int)threadIdx.x;
  const unsigned int bid = (unsigned int)blockIdx.x;
  extern __shared__ unsigned int s_data[];
  word __attribute__((unused)) *sm1 = &s_data[0];
  word __attribute__((unused)) *sm2 = &s_data[128];
  ix_int(sm1,tid) = (((tid & 0xffffff81) < 1) ?
                     ix_int(input0,(tid + (128 * bid))) :
                     (ix_int(input0,((tid & 0x7e) + (128 * bid))) +
                      ix_int(input0,(tid + (128 * bid)))));
  __syncthreads();
  ix_int(sm2,tid) = (((tid & 0xffffff83) < 2) ?
                     ix_int(sm1,tid) :
                     (ix_int(sm1,((tid & 0x7c) | 0x1)) + ix_int(sm1,tid)));
  __syncthreads();
  ix_int(sm1,tid) = (((tid & 0xffffff87) < 4) ?
                     ix_int(sm2,tid) :
                     (ix_int(sm2,((tid & 0x78) | 0x3)) + ix_int(sm2,tid)));
  __syncthreads();
  ix_int(sm2,tid) = (((tid & 0xffffff8f) < 8) ?
                     ix_int(sm1,tid) :
                     (ix_int(sm1,((tid & 0x70) | 0x7)) + ix_int(sm1,tid)));
  __syncthreads();
  ix_int(sm1,tid) = (((tid & 0xffffff9f) < 16) ?
                     ix_int(sm2,tid) :
                     (ix_int(sm2,((tid & 0x60) | 0xf)) + ix_int(sm2,tid)));
  __syncthreads();
  ix_int(sm2,tid) = (((tid & 0xffffffbf) < 32) ?
                     ix_int(sm1,tid) :
                     (ix_int(sm1,((tid & 0x40) | 0x1f)) + ix_int(sm1,tid)));
  __syncthreads();
  ix_int(sm1,tid) = ((tid < 64) ?
                     ix_int(sm2,tid) :
                     (ix_int(sm2,63) + ix_int(sm2,tid)));
  __syncthreads();
  ix_int(result0,(tid + (128 * bid))) = ix_int(sm1,tid);
}
```

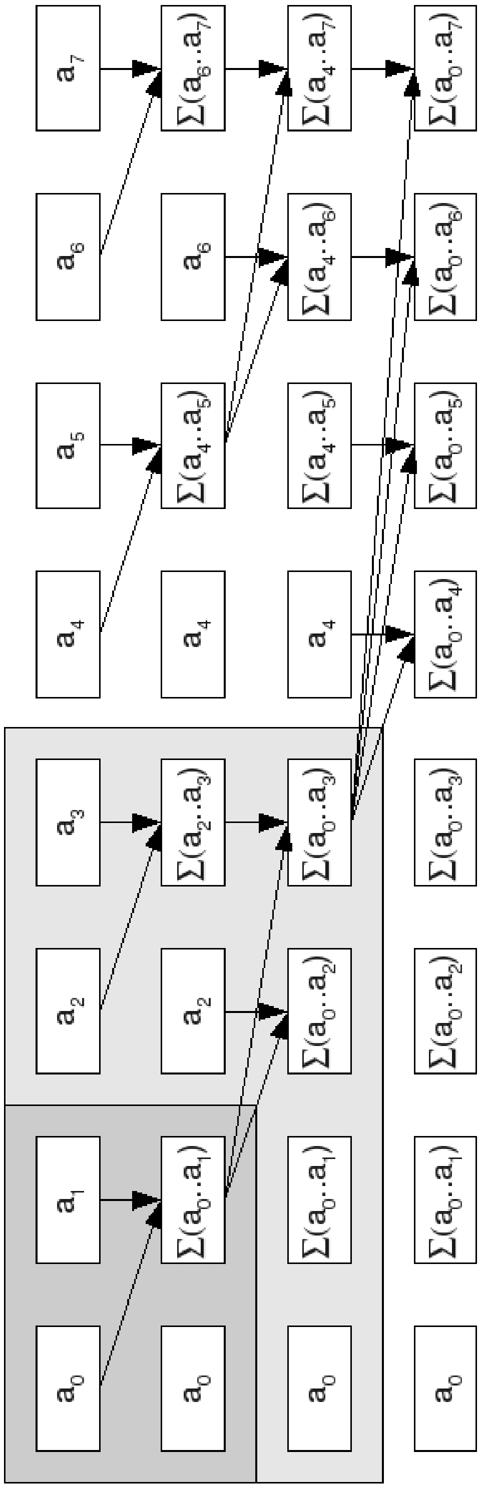

Figure 6.5: The Sklansky parallel prefix network.

6.6 Parallel prefix on large arrays

In this section, an algorithm for computing the parallel prefix sums for large arrays on the GPU will be developed. Currently this cannot be done entirely in Obsidian; a short CUDA program needs to be written also. All the kernels needed in the implementation can be written in Obsidian but the program that coordinates and runs these kernels will be written in CUDA.

The previous section describes how to implement a parallel prefix kernel. This kernel can compute the parallel prefix sums of arrays of short length, such as 128 or 256 elements for example. If the array we want to compute the prefix sums on is of length $n * 128$, we can run n instances of the kernel next to each other over the input array. In order to turn this into a prefix sum the n sub-results need to be merged. This merging can be done in a very similar way to how the Sklansky kernel is implemented. That is the value at the maximum index of sub-result number 1 needs to be combined with all the elements of sub-result number 2. When the sub-results 1 and 2 have been merged the value at the maximum index of that array can be combined with all the elements of sub-results number 3 and so on. Combining the sub-results this way is sequential; however, the task can be parallelized.

Say that n sub-results of length 128 have been computed; call the values at the maximum index of these sub-results m_0 through m_n. In order to get the large prefix, sum all elements of sub-result 1 need to have m_0 added to them, all elements of sub-result 2 need $m_0 \oplus m_1$ added to them. The last sub-result needs $m_0 \oplus \ldots \oplus m_{n-1}$ added to it. Now this is another example of a prefix sum. So, to compute the large prefix sum, compute n small ones saving their maxima in an auxiliary array. The prefix sum of this auxiliary array is then computed and the results thereof distributed over the n sub-results.

Below is a slightly modified Sklansky kernel that not only computes the prefix sum, but also outputs the topmost result to a special array.

```
sklansky2 :: (Flatten a, Choice a) =>
             Int -> (a -> a -> a) -> (Arr a :-> (Arr a,Arr a))
sklansky2 n op = sklansky n op ->- pure out
    where
      out arr = (arr,singleton (arr ! (n-1)))
          where n = fromIntegral $ len arr
```

Below is the second kernel needed. This kernel takes two arrays as input. The second input array is a singleton array whose element will be combined with every element of the first array using the function `op` passed in:

```
distribute :: (a -> a -> a) -> ((Arr a, Arr a) :-> Arr a)
distribute op = pure distr
    where
      distr (a1,a2) = fmap (op (a2 ! 0)) a1
```

As an example, CUDA code for computing the parallel prefix sum of 262144 integers is implemented. This will be done using a kernel for 512 elements generated from the `sklansky2` Obsidian program. The number of elements (262144) has been chosen to simplify the description here. Since 512*512=262144, the first step in this algorithm is to launch 512 instances of the 512 element `sklansky` kernel on the GPU. Following this, the 512 block maxima are scanned and lastly the scanned block maxima are distributed over the array of results from the first step. The block maximum of the second scan is simply discarded. This means that a possible optimization is to use another scan kernel for this stage that does not output block maxima.

```
int* dvalues;
int* blockmaxs, *unused;;

// Allocate GPU memory.
cudaMalloc((void**)&unused, 1);
cudaMalloc((void**)&blockmaxs, sizeof(int) * 512);;
cudaMalloc((void**)&dvalues, sizeof(int) * 262144);
cudaMemcpy(dvalues, values, sizeof(int) * 262144 ,
             cudaMemcpyHostToDevice);



// Compute the prefix sums.
sklansky<<< 512, 512, sizeof(int)*1024>>> ((word*)dvalues,
                                           (word*)dvalues,
                                           (word*)blockmaxs);

sklansky<<< 1,   512, sizeof(int)*1024>>> ((word*)blockmaxs,
                                           (word*)blockmaxs,
                                           (word*)unused);
```

```
distribute<<<511 , 512,0  >>> ((word*)(dvalues + 512),
                                (word*)blockmaxs,
                                (word*)(dvalues + 512));


// Copy result back to host memory.
int *result = (int *)malloc(sizeof(int) * 262144);
cudaMemcpy(result, dvalues, sizeof(int) * 262144,
           cudaMemcpyDeviceToHost);
```

6.7 Notes on performance

6.7.1 Dot product

Section 2.2.3 showed one way to implement dot product in CUDA. In section 6.2, Obsidian versions of the dot product algorithm were implemented. This section compares the performance of these Obsidian kernels to the handwritten one from section 2.2.3.

This table contains the running time per block when launching 32768 blocks of 256 threads on the GPU. The table contains two kernels generated using Obsidian, `dotProd` and `dotProd'` and one handwritten kernel, from section 2.2.3, referred to as `dotProdCuda` in the table:

Kernel	Threads	Elems per block	ms per block
dotProd	256	256	0.0015
dotProd'	256	256	0.0014
dotProdCuda	256	256	0.0030

The two generated kernels show very similar running times. Since the only difference between `dotProd` and `dotProd'` is the memory access pattern, the conclusion is that in this case the access pattern did not have much effect. However, most of the accesses are to shared memory and there the restrictions are not as severe as with global memory accesses. The difference between the handwritten kernel and the generated ones in running time is most likely due to the unrolling of the loops that takes place in the generated code.

The table below shows the running times of the two dot product kernels that use half as many threads:

Kernel	Threads	Elems per block	ms per block
dotProd1	128	256	0.0013
dotProd1'	128	256	0.0013

6.7.2 Mergers

The table below shows the execution time per block of the mergers implemented in section 6.3. 32768 instances of the merger kernel where launched and the average execution time per block found. The number 32768 was chosen since it is clearly enough to keep all the multiprocessors of the current generation GPUs fully loaded and in that way give an indication on how the kernel performs under full load. The running times in the table below where obtained using an NVIDIA GTX320M GPU.

Kernel	Threads	Elems per block	ms per block
Odd-even merge	256	256	0.0036
Butterfly	256	256	0.0023
Shuffle exchange	256	256	0.0015

Odd-even merge and the butterfly merger both use `ilv` in their implementation. This leads to large indexing expressions (many operator applications) and it is most likely thar this is reflected in the execution time.

Figure 6.3 shows that for a given size odd-even merge performs fewer operations compared to the butterfly but still the butterfly network outperforms it in execution speed. This discrepancy most likely comes from the use of `odds` in odd-even merge. The butterfly network only uses `evens`. The implementation of `odds` has an extra level of conditionals compared to that of `evens`. This extra conditional may be the cause of the higher execution time of odd-even merge.

6.7.3 Sorters

Here, the performance of the sorters from section 6.4 is estimated. The amount of time it takes per block to execute on the GPU is measured by launching a large number of blocks on the GPU. In this case 32768 blocks were launched. The time it takes for the GPU to finish all those blocks is then divided by 32768, giving the figures in the *ms* field in the table below. The time measurements presented here were obtained on an NVIDIA GTX320M GPU.

Kernel	Threads	Elems per block	ms per block
Odd-even merge sort	256	256	0.0086
Bitonic sort (bfly)	256	256	0.0061
Bitonic sort (shex)	256	256	0.0038
Odd-even transposition	256	256	0.0238
Handcoded Bitonic	256	256	0.0061

The sorter listed in the table as "Handcoded Bitonic" is implemented directly in CUDA. The source code is given below:

```
__global__ void bitonic(int *input, int *output){

  extern __shared__ int sm[];

  unsigned int tid = threadIdx.x;
  int j,k;

  sm[tid] = input[tid + NUM_ELEMS * blockIdx.x];
  __syncthreads();

  for (k=2;k<=NUM_ELEMS;k*=2) {
    for (j=k>>1;j>0;j=j>>1) {
      int ixj=tid^j;

      if (ixj>tid) {
        int a = sm[tid];
        int b = sm[ixj];
        if ((tid&k)==0 && a>b) { sm[tid] = b; sm[ixj] = a;}
        if ((tid&k)!=0 && a<b) { sm[tid] = b; sm[ixj] = a;}
      }
      __syncthreads();
    }
  }
  output[tid + NUM_ELEMS * blockIdx.x] = sm[tid];
}
```

6.7.4 Parallel prefix

In order to estimate the relative performance of the Sklansky parallel prefix kernel generated using Obsidian, a 1 million element (1048576 elements to be exact) scan algorithm was implemented. Implementing this 1 million element scan is done in almost the same way as described in section 6.6, but with some minor changes to accommodate for increase in number of elements. The large scan algorithm is implemented using three different Sklansky parallel prefix kernels. One of the kernels is handwritten directly in CUDA but no effort has been made on performance optimization. Another kernel is also written directly in CUDA with some performance considerations such a unrolling of loops. The third kernel is the Sklansky kernel generated using Obsidian.

The table below shows the average time it takes to compute the parallel prefix sum of 1 million integers using the three different kernels. These values have been obtained on a NVIDIA GTX320M GPU:

Kernel	Threads	ms
Handwritten no opt.	512	4.36
Handwritten opt.	256	2.93
Obsidian Generated.	512	3.84

The same experiments performed on a NVIDIA GTX480 GPU produces the following values:

Kernel	Threads	ms
Handwritten no opt.	512	0.34
Handwritten opt.	256	0.23
Obsidian Generated.	512	0.32

As can be seen from the figures above, the code generated using Obsidian is in this case comparable to handwritten CUDA. When reading these results, some consideration should also be made to the programmer effort in each kernel implementation. The Obsidian code requires least effort and is also describing the recursive decomposition of the Sklansky kernel directly. In the CUDA version, the programmer needs to apply some effort to express the recursive decomposition using indexing calculations. This is not trivial. The handwritten optimized version of the kernel represents a lot of work in optimization effort and thought. In this light, the performance of the generated Obsidian code must be considered good.

6.7.5 GPU details

The running time measurements have been obtained using two different GPUs. The GPU specifications reported below are as they appear using the the `deviceQuery` program that comes with the NVIDIA CUDA SDK.

GPU	Compute Capability	Multi Processors	Cores	Clock
GTX320M	1.2	6	48	0.95Ghz
GTX480	2.0	15	480	1.40Ghz

The computers housing these GPUs have the following specifications:

System Type	GPU	CPU	System memory
Stationary	GTX480	Intel Core I7 2.80 Ghz	12 Gb
Laptop	GTX320M	Intel Core2Duo 2.66 Ghz	4 Gb

Chapter 7

Related work

7.1 Embedded GPGPU programming languages

The following subsections list three competing approaches to embedded GPGPU programming language implementation. The abstraction level of these languages must be considered higher than that offered by Obsidian. These languages offer as primitives the kind of operations that Obsidian can be used to implement.

7.1.1 Accelerator

Accelerator is an embedded language for parallel programming implemented in C♯ [39]. The programmer is offered a set of operations on data-parallel arrays such as reductions (for example sum and product) and transformations (such as rotate, shift and transpose).

The programmer writes programs in C♯ using standard data structures but has to convert the data into the data-parallel arrays before using any GPU accelerated functions on it. Being part of the *.Net* framework, Accelerator is also accessible for F♯ programmers. F♯ is a functional programming language.

The reference [33] shows the implementation of a convolver (an algorithm often used for image smoothing) in Accelerator. From the Accelerator implementation of the convolver both GPU and multi-core CPU code is generated. The CPU code makes use of SIMD extensions. Accelerator utilizes Just In Time (JIT) compilation on the data-parallel parts of the algorithms. In that way, the available resources are being used properly while the source code remains platform independent.

Further information about Accelerator is available in reference [31].

7.1.2 Accelerate

Data.Array.Accelerate is an embedded language for general purpose computing on GPUs [8]. Just like Obsidian, Accelerate is embedded in Haskell and it also targets NVIDIA CUDA as backend. The approach taken to generating the CUDA code is quite different in Accelerate compared to Obsidian.

In Accelerate, there are scalar and *collective* operations. The collective operations are the keys to exploiting the parallel resources of the GPU. Amongst the collective operations are `map`, `fold` and `zipWith`. The collective and scalar operations are kept separate by the type system to ensure that no nested parallelism can be expressed.

Each of the collective operations corresponds to a *skeleton*, a code template with "holes" that are filled in with the parts specific to the current operation. For example, if the programmer uses `map f`, the map skeleton will be used with the code representing `f` pasted into its body. The Accelerate code skeleton concept does not match up entirely with kernels. A code skeleton may consist of one or more CUDA kernels.

In some sense, Accelerate is solving a problem different from that approached in Obsidian. The Accelerate collective operations that are given as skeletons are very similar to the kernels that can be implemented in Obsidian. In that light, Obsidian is a lower level language than Accelerate and the two languages are not in direct competition.

In Data.Array.Accelerate, the dot product algorithm can be implemented as:

```
dotProd :: Vector Float -> Vector Float -> Acc (Scalar Float)
dotProd xs ys
  = let
      xs' = use xs
      ys' = use ys
    in
    fold (+) 0 (zipWith (*) xs' ys')
```

For comparison, the Obsidian implementation from chapter 6 is.

```
dotProd :: (Flatten a, Num a) => Int -> (Arr a, Arr a) :-> Arr a
dotProd n = pure (zipWith (uncurry (*)))  ->- sync ->- reduce n (+)
```

The definition of `mult` has been inlined in the code above to make the similarities clearer.

The two versions of `dotProd` are not very different. Each consists of a reduction, `fold` in Accelerate and `reduce` in Obsidian, and a `zipWith`. The main difference between the two implementations is the use of `sync` in Obsidian. The real difference between these two programs however is what they are intended to do. The Accelerate `dotProd` computes the dot product of two arrays of any length. For example, it can be used on two arrays of length 1 million elements. This is not what the Obsidian version is intended to do. The code generated from the Obsidian `dotProd` is a kernel intended to be used as a building block for algorithms on large arrays.

Another important difference between the two languages is that `fold` and `zipWith` are built-in operations in Accelerate. The counterparts in Obsidian are implemented using lower level smaller building blocks. The idea behind this is that there are many ways to sum up elements of an array on the GPU and the programmer should be able to choose the most suitable way.

The Obsidian `dotProd` contained a use of `sync` between the summation and the computation of the products. This `sync` could be removed, leading to the fusion of the products computation with the first stage of the reduction. In the Accelerate code, there is an implicit global synchronization barrier between the `fold` and the `zipWith` stage of the algorithm. The `zipWith` part is computed by launching a grid of kernels; the system then waits for all those kernels to finish before launching a grid of `fold` kernels. These two synchronization concepts are not completely analogous, but the comparison is meant to illustrate the difference in programmer control and abstraction level.

The reference [8] shows benchmarks for the applications, dot product, Black-Scholes and sparse-matrix vector multiplications with competitive results. The highly optimized CUDA libraries compared to are faster, but still the results are impressive.

7.1.3 Nikola

Nikola is the latest addition to the collection of embedded GPGPU DSLs. Just like Obsidian and Accelerate, Nikola uses CUDA as its backend [22]. Nikola also has operations such as `map` and `zipwith` in its expression data type. This indicates that Nikola must also be considered a higher level language than Obsidian. Listed as one of the strengths of Nikola in [22] is the ability to generate CUDA functions from functions in Haskell. This enables reuse of generated code.

The reference [22] shows two benchmarks, Black-Scholes and Radix sort. Compared to native Haskell implementations of the same algorithms. In the Black-Scholes case, Nikola surpasses native Haskell for data sets larger than 32kB. Compared to Accelerate, in the previous section, the Nikola version seems to be in the area of 2 orders of magnitude slower, for 1 million data elements, on similar hardware. At least this is the picture that can be extrapolated from [8] and [22].

7.2 Embedded languages for graphics and image manipulation

The embedded languages listed in this section are specialized for the graphics and image manipulation area. As such they are not as directly related to the Obsidian project as those of the previous section. However, these languages have been an important influence and source of information in the implementation of Obsidian.

7.2.1 Pan

Pan is a domain specific embedded language for image manipulation and creation [10]. In Pan, an image is represented by a function from points in two dimensions to colors. As this function is evaluated over a two dimensional region, an image appears. Pan is compiled into C code and performs various optimizations such as algebraic simplification, CSE and code hoisting.

7.2.2 Vertigo

Vertigo is another embedded language for functional graphics [11]. Vertigo can be used to describe textures and surfaces and also to express shaders. A shader is a program that describes the reflexive properties of a surface. Shaders are traditionally expressed in C like languages. The reason for this is to appeal to graphics programmers. Conceptually, a shader fits very nicely in the functional paradigm. The code generated by Vertigo targets DirectX 8.1.

7.2.3 PyGPU

PyGPU is a language for writing image processing algorithms [20]. PyGPU is embedded in Python which seems to be both a benefit and a drawback. Python is a dynamic language and these are hard to compile. The ability to inspect the bytecode of functions at runtime, on the other hand, is noted as a benefit.

PyGPU offers a library of image manipulation functions to the programmer. Some of these are familiar, like reductions, but they are extended to two dimensional arrays to fit better with image manipulation applications. Others are less familiar like the `convolve` operation.

In [21] there are a number performance measurements performed on a NVIDIA 6600 GPU. The benchmarks used are skin detection and convolutions, amongst others. The performance reached is between 0.5 and 4 GPixel operations per second, obtained on a GPU with a 4.8 GPixel operations per second theoretical peak performance. The performance measurements are not compared to any competing approach to image processing.

7.3 Embedded languages

In this section, two other embedded domain specific languages with rock based named are presented, Lava and Feldspar. The rock based names follow a tradition started by Mary Sheeran with her *Ruby* language [16]. Lava is mentioned because it is the main influence on the programming style of Obsidian. Feldspar is closely related to Obsidian and even uses the same array representation as Obsidian.

7.3.1 Lava

Lava is a hardware design language embedded in Haskell [4]. In Lava a circuit can be specified and simulated. Lava also connects to external tools to offer verification of the designed circuits. Lava is the main influence on Obsidian in terms of programming style. In many cases a Lava program and an Obsidian program will look very similar. As an example this is the Odd-even merger taken from[9]:

```
mergeOE 1 cmp = cmp
mergeOE n cmp = ilv (mergeOE (n-1) cmp) >-> mid (evens cmp)
```

Compare this code to the Obsidian merger in section 6.3. They are not very different. However, in Lava, a circuit taking inputs can be represented as a Haskell function taking a list of inputs. In Obsidian, the `Arr` data type serves the same end. This means that Obsidian functions like `conc`, that are given for free in Lava, need a special implementation.

7.3.2 Feldspar

Feldspar is a DSL for Digital Signal Processing (DSP) algorithms embedded in Haskell [2]. Feldspar is built upon a Core language, a small language with features easily compilable into C. The core language, however, is functional. On top of the core language there is a *Vector* library. The vectors are represented as functions from indices to elements; this idea is borrowed from Obsidian.

The version of Feldspar described in reference [2] is compiled into entirely sequential C code but there are plans to make use of the parallel capabilities of DSP processors in the future.

7.4 Data-parallel programming languages

This section contains short descriptions of functional data-parallel programming languages. Both languages mentioned here support nested data-parallelism. Nested data-parallelism means that data-parallel operations can be applied in parallel to nested data structures. For example, this means that every array in an array of arrays can be summed up in parallel. Nested data-parallelism is implemented by a flattening procedure. The programmer can express nested data-parallelism but in the compilation procedure all data is transformed into flat arrays and the operations are transformed into segmented flat data-parallel operations.

7.4.1 NESL

NESL is one example of a functional programming language that allows the programmer to express nested data-parallelism [13]. NESL is compiled into an intermediate language called VCODE for which backends exist for a range of machines such as Cray and the Connection Machine. In NESL, parallelism is exposed to the programmer as a set of operations on sequences. Amongst these operations are `scan`, `sum` and `permute`. The flattened versions of these operations (segmented versions) are implemented as a low level library called CVL for efficiency reasons.

7.4.2 Data parallel Haskell

Data Parallel Haskell (DPH) [17] brings the ideas from NESL to the Haskell programming language. In DPH, the programmer is offered a set of operations on parallel arrays. The operations will be familiar to a Haskell programmer with for example `mapP`, `filterP` and `zipWithP` that are the familiar `map`, `filter` and `zipWith`, but for parallel arrays.

7.5 C/C++ based approaches to GPGPU programming

7.5.1 CUDA

CUDA is NVIDIA's language for general purpose computations on GPUs[25]. CUDA is based on the C programming language but with a small set of extensions for implementing kernels that run on the GPU and some extra syntax for launching such kernels. CUDA was released in 2006 and at that point was a great improvement for people wanting to use the NVIDIA GPU for general purpose.

CUDA is the target language for Accelerator, Nikola and Obsidian.

7.5.2 OpenCL

OpenCL is another language for parallel programming but OpenCL targets both GPUs and multi-core CPUs [19]. Programming in OpenCL is very similar to programming in CUDA.

7.6 Conclusion

There are many languages targeting data-parallel programming and GPUs. Some of these try to hide the details of the underlying architecture completely and rely heavily on optimization techniques to obtain performance on the target platform or platforms. The benefit of such an approach is of course that it gives a degree of platform independence. To target a new platform, a new set of optimizations and code generation might be needed, but the language offered to the programmer remains the same. There have always been some programmers who use a lower level language alternative. The C programming language did not abolish the use of Assembly. And Haskell did not abolish the use of C. Obsidian wants to sit in between the higher level approaches to GPGPU programming and the CUDA/OpenCL level and offer some of the benefits of each of those levels.

Chapter 8

Conclusion

Future computer systems will contain multi- and many-core processors. Methods for expressing fine- and coarse-grained parallelism will be needed to an increasing degree.

Parallel programming is not easy. Easy to use and powerful, expressive, languages are needed that allow programmers to be creative. A suitable level of abstraction must be found that maximizes both platform independence and performance. This goal is probably very hard to achieve but still we must aim for it.

Obsidian is trying to raise the level of abstraction for the GPGPU programmer in a way that allows for easier code reuse and better compositionality of GPU kernels compared to CUDA. The intention is to still allow the programmer to make the kinds of decision that influence performance on the target platform. Choices of work division and memory locations of data should be in the hands of the programmer. These choices should not affect compositionality of kernels. The context in which a kernel is used should specify the parameters by which it operates following the guidelines put in place by the programmer.

By no means are these goals all reached by Obsidian today. There are still many paths left to explore. But the implementations of Obsidian shown in this thesis illustrate that it is possible to express quite complex kernels using small elegant and high level descriptions, while maintaining the kind of compositionality that is desired and obtaining generated code of reasonable quality. The abstraction level of Obsidian also allows the programmer to think about the problem rather then how to express it using indexing computations. The example of the `sklansky` parallel prefix network from chapter 6 illustrates this. In that chapter, the Obsidian Sklansky kernel implementation describes the recursive decomposition of the algorithm to the letter. A direct implementation in CUDA would most likely not be possible without spending quite some time computing all the access patterns on a piece of paper before starting to program. At least, this is my personal view.

The generated code shows some opportunities for further optimization as pointed out in the future work section. Adding Common Subexpression Elimination alone will not solve the performance problem entirely. What is needed is to increase the level of control available to the programmer. Allowing the programmer to specify the work grouping without imposing any data movement is very important and would most likely move the performance of generated code up more than what adding CSE would. Grouping work in different ways amongst the threads is a common technique in CUDA programming for obtaining performance, avoiding inefficient memory access patterns and hiding memory latency. The problem left is to figure out what tools to place in the programmers to hands to accomplish this goal.

In Obsidian, the programmer has some control over the way work is split up into units for processing by a thread. The `sync` operation can be used both to introduce sequentiality and to parallelize an algorithm. This comes from the choice to let the length of target arrays define the number of threads to use. Nested arrays or the pairs in an array of pairs are computed in sequence while each element of top-level arrays is produced in parallel. As a tool offered to the programmer, this method gives the desired degree of control but in the current version, arrow Obsidian, the grouping of work introduces unnecessary data movement. The version of Obsidian called monad Obsidian also experimented with allowing the programmer to place arrays either in shared memory or in global memory. To accomplish this, the Haskell type system was used to differentiate between shared and global arrays. This is a feature that is not available in arrow Obsidian and it has not been much missed. In arrow Obsidian, a kernel assumes it is computing on values stored in shared memory entirely. However, it is clear that from a performance point of view making use of the memory hierarchy of the GPU in the correct way is necessary and both approaches tried seem to do that equally well.

The performance measurements undertaken (described in section 6.7) have a drawback in that they are all comparisons either between kernels all generated in Obsidian or to handwritten code originating from the same research group as the author. When only Obsidian kernels have been used in the comparison, all this shows is the relative performance of using a given Obsidian programming technique compared to another. It is desirable to compare Obsidian generated kernels to code generated from some other embedded language for GPGPU programming. However, The other approaches shown in the related work (chapter 7) all choose a higher level approach making direct comparison hard. One way around this would be to write larger applications using both Obsidian and CUDA. The kernels would be implemented in Obsidian and the glue code written in CUDA. The performance of this Obsidian/CUDA program could then be compared to the same algorithm expressed in some other language. This has not yet been done but it is likely that in that setting an Obsidian application would perform well but for the price of increased implementation effort in writing the CUDA glue code. As the future work states, looking at methods for addressing the higher level problem of kernel management and launching from within Obsidian is required. That would make a future comparison to other approaches much easier.

The performance measurements also show that, compared to handwritten kernels, the generated code perform quite well, at least if the programmer effort that goes in to the kernels is also taken into consideration. Obsidian also encourages experimentation from the programmer by offering a rapid edit, compile and run cycle.

The trend with increasing numbers of processing elements in multi- and many-core processors is likely to continue. In order to feed all these processing elements with data, more complicated and layered memory hierarchies are appearing. Modern GPUs and the Cell processor [30] have programmer managed local memories where traditional CPUs have hardware managed caches. Multi-paradigm parallel programming skills are required from the programmer. GPUs require a mix of data-parallel programming on the kernel level and task parallelism on the kernel coordination level. If we are to rise to the challenge posed by the new generation of computing devices, new languages at different levels of abstraction are needed. Computer scientists throughout the world are together contributing to a toolbox for the programmer to use when facing these new challenges. Obsidian aspires to be part of that toolbox.

Bibliography

[1] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition.* A. K. Peters, Ltd., Natick, MA, USA, 2008.

[2] E. Axelsson, K. Claessen, G. Dvai, Z. Horvth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajda. Feldspar: A domain specific language for Digital Signal Processing algorithms. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 169 –178, July 2010.

[3] 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.

[4] 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.

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

[6] Shekhar Borkar. Thousand Core Chips: A Technology Perspective. In *DAC '07: Proceedings of the 44th annual Design Automation Conference*, pages 746–749, New York, NY, USA, 2007. ACM.

[7] Daniel Cederman and Philippas Tsigas. GPU-Quicksort: A practical Quicksort algorithm for graphics processors. *J. Exp. Algorithmics*, 14:1.4–1.24, 2009.

[8] Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell Array Codes with Multicore GPUs. `http://www.cse.unsw.edu.au/~chak/papers/CKLM+10.html`.

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

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

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

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

[13] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a Portable Nested Data-Parallel Language. *Journal of Parallel and Distributed Computing*, 2(1):4–14, apr 1994.

[14] Maurice Herlihy. The Multicore Revolution. In V. Arvind and Sanjiva Prasad, editors, *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*, volume 4855 of *Lecture Notes in Computer Science*, pages 1–8. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-77050-3_1.

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

[16] G. Jones and M. Sheeran. Circuit design in Ruby. In *Formal Methods for VLSI Design*, pages 13–70. Elsevier Science Publications, North-Holland, 1990.

[17] Simon L. Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *FSTTCS*, pages 383–414, 2008.

[18] Samuel Kamin. Standard ml as a meta-programming language. `http://www-sal.cs.uiuc.edu/~kamin/pubs/ml-meta.ps`, 1996.

[19] Khronos OpenCL Working Group. The opencl specification, version 1.0.29. `http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf`, 2008.

[20] 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.

[21] Calle Lejdfors and Lennart Ohlsson. PyGPU: A high-level language for high-speed image processing. `http://www.cs.lth.se/home/Calle_Lejdfors/publications/pygpu.pdf`, 2006.

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

[23] Wen mei Hwu, Kurt Keutzer, and Timothy G. Mattson. The Concurrency Challenge. *IEEE Design and Test of Computers*, 25:312–320, 2008.

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

[25] NVIDIA. NVIDIA CUDA Programming Guide 3.1. `http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf`.

[26] NVIDIA. Technical Brief: NVIDIA GeForce 8800 GPU Architecture Overview. `http://www.nvidia.com/page/8800_tech_briefs.html`, 2006.

[27] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, 2009.

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

[29] Paul Hudak. Building Domain-Specific Embedded Languages. *ACM Computing Surveys*, 28, 1996.

[30] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The Design and Implementation of a First-Generation CELL Processor. pages 184–592 Vol. 1, 2005.

[31] Microsoft Research. An Introduction to Microsoft Accelerator v2. `http://research.microsoft.com/en-us/projects/accelerator/accelerator_intro.docx`, 2010.

[32] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10, 2009.

[33] Satnam Singh. GPGPU and x64 Multicore Programming with Accelerator from F♯. `http://blogs.msdn.com/b/satnam_singh/archive/2009/12/15/gpgpu-and-x64-multicore-programming-with-accelerator-from-f.aspx`, 2009.

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

[35] J. Sklansky. Conditional sum addition logic. *Trans. IRE*, EC-9(2):226–230, June 1960.

[36] Joel Svensson, Koen Claessen, and Mary Sheeran. GPGPU Kernel Implementation and Refinement using Obsidian. *Procedia Computer Science*, 1(1):2065 – 2074, 2010. ICCS 2010.

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

[38] 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, September 2008. LNCS number: 5836.

[39] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, 2006.