CHALMERS |  UNIVERSITY OF GOTHENBURG

GPGPU Kernel Implementation using an Embedded Language: a Status Report

JOEL SVENSSON
KOEN CLAESSEN
MARY SHEERAN

Department of Computer Science and Engineering
Division of Software engineering and technology
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden, 2010
Technical Report No. 2010:01

GPGPU Kernel Implementation using an Embedded Language: a Status Report
Joel Svensson

Computer Science and Engineering
Technical report no 2010:01
ISSN number 1652-926X

Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 10 00

Department of Computer Science and Engineering
Göteborg, Sweden 2010

GPGPU Kernel Implementation using an Embedded Language: a Status Report

Joel Svensson, Koen Claessen, Mary Sheeran
Chalmers University of Technology
412 96 Gothenburg, Sweden

January 22, 2010

Abstract

Obsidian is a domain specific language for general purpose computations on graphics processing units (GPUs) embedded Haskell. This report present examples of GPU kernels written in Obsidian as well as parts of the current implementation of Obsidian.

The goal with Obsidian is to raise the level of abstraction for the programmer while not scarifying performance. The kind of decisions and tradeoffs considered by a GPU kernel implementer should be easy to make and change in Obsidian.

1 Introduction

Multicore and manycore processors are becoming more and more common. Modern graphics processing units (GPUs) are examples of manycore processors, today GPUs come with hundreds of processing elements capable of managing thousands of threads [17].

Obsidian is a domain specific language for general purpose programming on GPUs (GPGPU), capable of generating code for modern NVIDIA GPUs. The Obsidian project is about exploring ways to program these new multicore and manycore machines.

GPU design is driven by the performance demands of graphics applications. The kind of processing that is common in graphics falls in the data-parallel category [20]. One example of a computation from graphics is the transformation of vertices between different coordinate systems. That is, each vertex is multiplied by the same transformation matrix, fitting the data-parallel paradigm perfectly.

1.1 NVIDIA GPUs and CUDA

Starting with the 8000 series of GPUs (the G80 architecture), NVIDIA's GPUs came with a unified architecture, unified meaning that all the processing elements on the GPU are of the same kind. This was different from the previous generation's GPUs, where there usually were two kinds of processing elements. There was one kind of processor designed to process fragment/pixel programs and another to process vertex data. Now these two kinds of processors are replaced by a single kind with capabilities surpassing both of the old ones. This new unified architecture together with development tools for GPGPU (General Purpose computations using GPUs) programming go under the name CUDA (Compute Unified Device Architecture) [18, 1, 17]. CUDA offers the GPGPU programmer a C compiler and libraries for CUDA enabled GPUs (NVIDIA 8000 series and above).

Figure 1, shows a conceptual picture of a CUDA enabled GPU. The GPU has a number of *Multiprocessors* (MP in the picture). These MPs contain a number of small processing elements called *Streaming Processors* (SP). These SPs operate in an SIMD fashion. All SPs in a given MP execute the same instruction each clock cycle.

Each MP is capable of maintaining a large number of threads in flight at the same time. A group of threads running on an MP is referred to as a *block*. A block can contain more threads then there are processing elements,

today a block can hold up to 1024 threads. There is a scheduler mapping these threads over the SPs available. Threads are scheduled in groups called *warps* (a word borrowed from the textile industry) consisting of 32 threads. A Warp is executed in SIMD fashion on the MPs. There are however, more threads in a warp then there are SPs, so the SPs are switching threads to work on between every instruction. This means that conditionals that take different paths within a warp have a negative effect on performance, the two diverging paths will in fact be executed sequentially [19].

The MPs also have some local memory that is shared between all the SPs of that MP and thus referred to as *Shared Memory* in the figure. The shared memory can be used to exchange information between threads running on the SPs. It can also be used as a software managed cache.

Threads within a warp can safely communicate using the shared memory without the need for any synchronisation. Threads from different warps however, need to use a synchronisation primitive to ensure a coherent view of the shared memory. In CUDA C this primitive is called `__syncthreads()`. The `__syncthreads()` primitive provides a barrier that all threads of a block must reach before any is allowed to proceed.

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

1.1.1 CUDA example program

As an example of CUDA programming, this section outlines the implementation of an algorithm for summing up an array of integers on the GPU. Summing up the elements of an array can be done in many different ways. The first is of course to sequentially add the elements up using for example a single for loop, which is how it may be done in a single processor system. But, since addition is associative, it is possible to choose between many other approaches. On a GPU it is important to split the work of summing up the array in a way that makes use of the GPU's resources in a good way.

In this example, the choice made is to divide the array into parts of length 256. These parts will be summed up potentially in parallel on the MPs. Each MP is capable of managing about a thousand threads in flight but it is recommended to stay below that. This is so that each MP can be in charge of running more than one block, which is useful to hide memory latencies. The 256 element sub-arrays will be added up using 256 threads. This number of threads is chosen to give a direct relationship between thread IDs and indices.

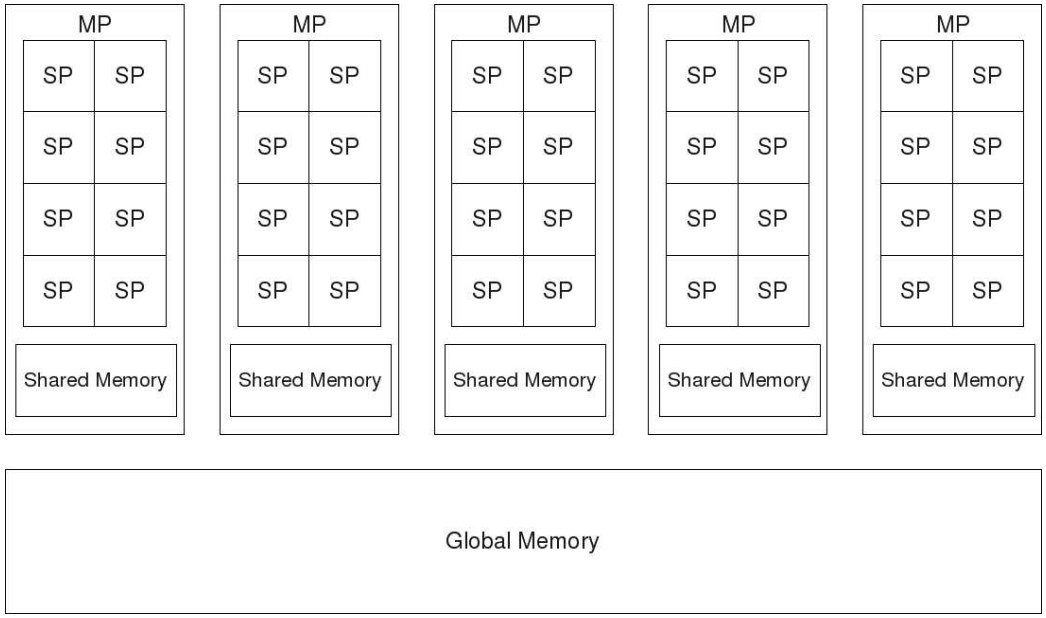

Figure 1: Conceptual image of a CUDA enabled GPU.

The following CUDA C code listing shows the *kernel* used to sum up a 256 element sub array. A kernel is a program that can be executed on an MP by a block of threads. Kernels are the building blocks of larger GPU programs. The algorithm used in this example is an adaptation of an sum algorithm described in [10]. Figure 2 illustrates how the algorithm computes the sum.

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

  extern __shared__ int smem[];
  int tid = threadIdx.x;
  int blockoffset = blockIdx.x * blockDim.x;

  smem[tid] = input[blockoffset + tid];

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

  if (tid == 0) output[blockIdx.x] = smem[blockDim.x - 1];
}
```

The kernel above is also applicable to other array sizes, as long as the size is a power of two.

A number of CUDA concepts are used in the kernel. For example there are a number of, from a thread's point of view, constants:

- `threadIdx`: the identity of a thread within a block. This is a three dimensional vector.
- `blockIdx`: the identity of a block within a grid. This is a two dimensional vector.
- `blockDim`: the dimensions of a block. This is also a three dimensional vector.

The `smem` array used in the kernel resides in shared memory. The size of this array is set when the kernel is started. That is, it is set by the controlling C program running on the computer's CPU. In currently available CUDA enabled GPUs the shared memories offer 16KB of storage.

The kernel computes the sum of its sub-array by reading it into shared memory. The program then goes into a for loop that first of all performs a `__synchtreads()`. In the first iteration this synchronisation ensures that all the data is read into shared memory before any thread attempts to access it. In the following iterations the synchronisation ensures that all the threads have completed their operation and that they will all have access to a correctly updated shared memory. When the summation is done the result is stored in the result array by a single thread.

Kernels are started on the GPU from a controlling program running on the CPU. CUDA adds new syntax to C for instantiating these computations on the GPU. Invoking a computation using a kernel called `myKernel` on the GPU looks like this:

```
myKernel<<<griddim,blockdim,size_smem>>>(arg1, ... , argn);
```

This starts the kernel `myKernel` on the GPU with grid dimensions specified by `griddim`. The number of threads is specified by `blockdim`. The size of shared memory that the kernel is allowed to use is given by `size_smem`.

It is also possible to supply the `griddim` and `blockdim` as scalars in the case where one dimensional grids and blocks are desired. As an example, to execute the `sum` kernel to sum up an array of 256 elements this is what the call would look like:

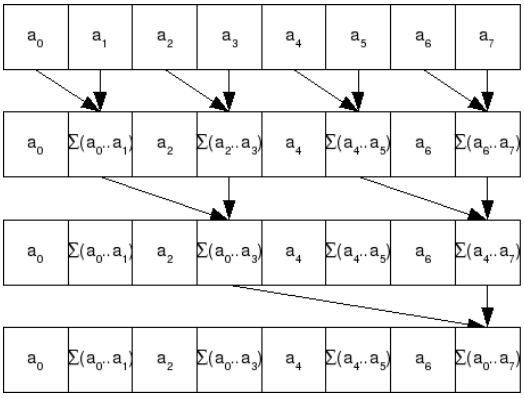

Figure 2: Computing the sum of an array.

```
sum<<<1,256,256*sizeof(int)>>>(input,output);
```

In order to use the kernel to sum up 2^{20}, (roughly one million), elements this approach would work:

```
sum<<<4096,256,256*sizeof(int)>>>(input,output);
sum<<<16,256,256*sizeof(int)>>>(output,output);
sum<<<1,16,16*sizeof(int)>>>(output,output);
```

This requires that the `output` array is large enough to hold 4096 elements. The output array is then reused in the following kernel launches. After completing the execution of these kernels the result will be found in `output[0]`.

The kernel described in this section is not optimised at all. A first step in optimising the given kernel could be to unroll the loop, this also means that a particular array size must be chosen. The kernel uses 256 threads, which gives a simple relation between thread IDs and indices. However, at no point is more than 128 threads really needed to perform the summation. Halving the number of threads gives a more efficient usage of the resources but also results in slightly more arithmetic in the index calculations.

1.2 Aims of Obsidian

The goal of Obsidian is to simplify the development of GPU kernels, the building blocks of larger GPU programs. When implementing an algorithm in CUDA, what to compute and how to compute it becomes codependent.

Choices such as how many elements the kernel operates on and how many threads it uses to do that affect each other greatly. Once a kernel is designed with a particular number of elements per thread these parameters become hard to tweak.

Obsidian provides an environment where it is easier to experiment with different partitionings and choices when implementing an algorithm. In Obsidian it is possible to write a simple running first prototype version of your kernel without thinking about architectural details. The prototype implementation can then be refined into a more efficient implementation.

The aim of Obsidian is to raise the level of abstraction for the GPGPU programmer, to relieve the programmer of details such as laying things out in memory. Performance affecting decisions should be easy to make and change without major rewrites of the code.

2 Programming in Obsidian

Obsidian is a language for GPGPU programming embedded in Haskell. Many of the language features resemble those of Lava, a hardware description and verification language [3]. The justification to use language constructs similar to that of a hardware description language came from the observation that GPGPU algorithms often were explained using circuit-like pictures, see for example [16].

Obsidian can be used to describe hardware like algorithms. These are algorithms where the number of inputs and outputs are fixed, not dependent on the values of inputs. There are a large number of algorithms falling into this category. For example, there are numerous sorting algorithms (sorting networks) and parallel prefix algorithms implementable in this way. Also, there are examples in the literature where GPGPU programmers implement their kernels to operate on very specific array sizes [16]. In a later stage, when the kernels are composed into algorithms on large arrays, support for variable length arrays (with length a multiple of that supported by kernel) is introduced.

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

2.1 Array Language

Arrays in Obsidian do not, like in C, name an area of memory. Instead, an array is represented by the computation that gives its elements. The array type consists of two parts, a function from indices to elements and an integer representing its length:

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

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

Obsidian provides a number of functions on this array type. For example, a function can be mapped over an array using `fmap`, see figure 3:

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

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

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

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

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

The function `pair` takes an array and returns an array of pairs where the first element of the input array is paired up with the second, the third with the forth and so on. the `unpair` function does the opposite, see figure 4.

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

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

These three functions `pair`, `unpair` and `fmap` can be used to define a function `evens` that applies a function over pairs on an array. Normal Haskell function composition is used here:

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

In turn, `evens` together with a *two-sorter*, gives a useful building block for implementing sorting networks, see for example [8]:

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

sort2 = evens cmp
```

Given an array, `sort2` rearranges it so that the elements at index 0 and 1 are sorted, the elements at 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`}.

In section 4 there are examples of sorting algorithms expressed in Obsidian.

The `Ordered` class provides the usual comparison functions:

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

Related to `evens` is the function `odds` that behaves very similarly. The `odds` function passes the first element through unchanged and then behaves like evens on the rest of the elements. That is, it pairs the second element with third and so on. Then the operation is performed on these pairs followed by unpairing them. The `odds` function is also useful to implement sorters, see section 4.

Another example of a function given in the array library is `zipp`:

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

This function performs on arrays what the normal Haskell `zip` does on lists. However, the input to `zipp` is a pair of arrays, see figure 5.

Here, `zipp` is used together with `fmap` to implement a function called `replace`. The `replace` function takes an element and a pair of arrays as input. It replaces each occurrence of the given element in the first array by the element at the corresponding index in the second array. Elements of types that are in the `Equal` class can be tested for equality using `(==*)`:

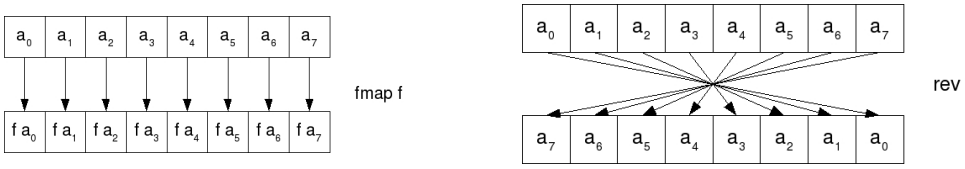

Figure 3: The functions `fmap` and `rev`.

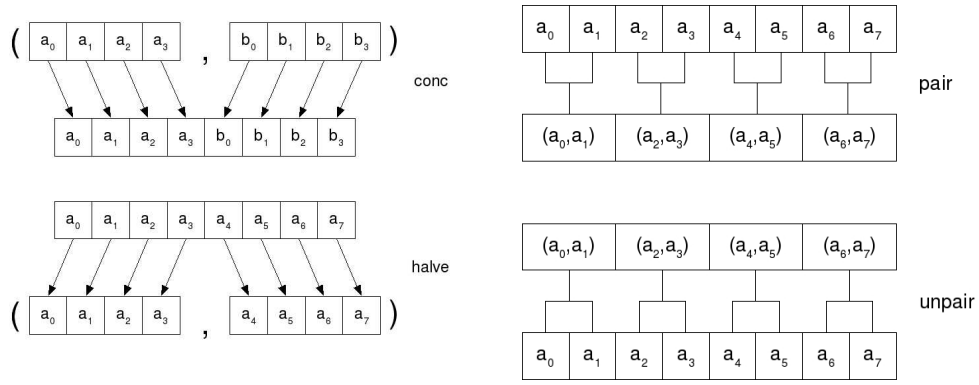

Figure 4: The functions `conc`, `halve`, `pair` and `unpair`

```
replace :: (Equal a, Choice a) => a -> (Arr a, Arr a) -> Arr a
replace i = fmap f . zipp
    where
      f (a,b) = (a ==* i) ?? (b,a)
```

Inspired by the (`?  :`) operator in C, the `??` operator makes a choice between the two elements of a pair depending on a boolean. It is implemented using `ifThenElse`.

Array programs like those described here make up the building blocks used to form larger GPU programs. How to create a GPU program from these is shown in the following section.

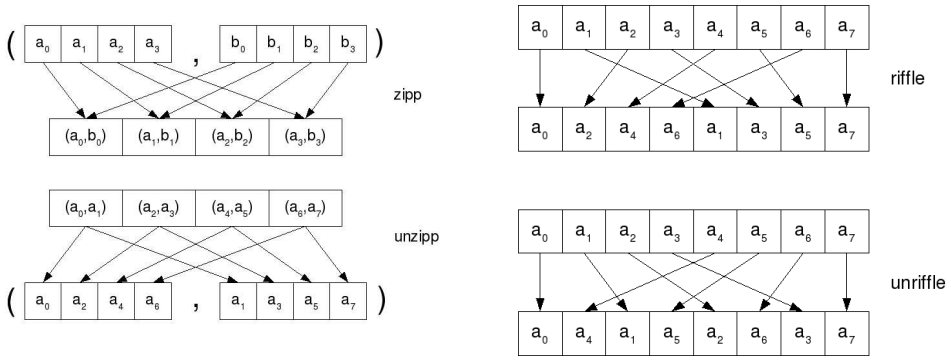

Figure 5: The functions `zipp`, `unzipp`, `riffle` and `unriffle`

```
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)
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 6: A selection of functions from the array API

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

Figure 7: Example functions on elements

2.2 GPU Programs

The second layer of Obsidian offers a data type that represents a GPU program taking an `a` as input and producing a `b`:

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

The details of `a :-> b` are shown in section 3. Informally we can think of `a :-> b` as representing programs that operate as illustrated in figure 8. This figure shows a program that performs some computation using a number of threads followed by a barrier synchronisation, and so on. The contents of the boxes marked with *Pure* can be thought of as containing an array program, such as those in section 2.1.

One way to create a GPU program is by using the function `pure`:

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

For example the array language program, `fmap (+1)`, that increments every element of an array can be lifted to a GPU program like this:

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

GPU programs such as `incr` can be executed on the GPU from a *GHCI* 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 3, but instances of `Flatten` are all the types that we can store in the GPU memory. Examples of types that are in `Flatten` are `IntE`, `FloatE`, `BoolE`. Arrays and pairs of things that are in Flatten are also instances of `Flatten`.

Here, `execute` is used in order to run an instance of the `incr` program on the GPU:

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

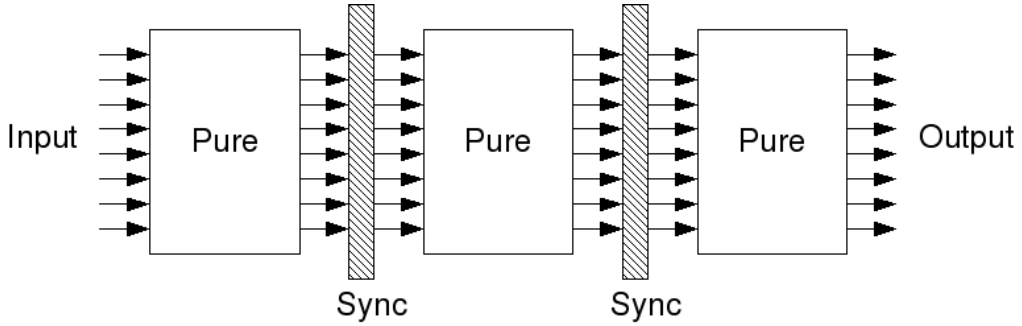

Figure 8: A GPU program, an object of type `a :-> b`, can be thought of as some pure computations interspersed by syncs.

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

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

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

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

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

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

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

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

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

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

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

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

The GPU program `increv` could also have been specified as:

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

In fact, when both arguments to `(->-)` are implemented using `pure` alone, `(->-)` is defined using Haskell functional composition. However, `increv` can also be specified with an explicit storing of intermediate values between the `rev` and the `fmap (+1)`. This is accomplished using a primitive GPU program called `sync`:

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

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

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

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

2.2.1 Sync and parallelism

In the previous examples, `incr` and `increv`, all parallelism is implicit. The following examples show how `sync` can be used to guide the introduction of parallelism.

In order to illustrate how `sync` can be used to guide the code generation, a number of implementations of a sum program will be used. The first version, `mySum1`, sums up the elements of an array using a single thread:

```
mySum1 :: Int -> Arr IntE :-> Arr IntE
mySum1 0 = pure id
mySum1 n = pure op ->- mySum1 (n-1)
    where
      op = fmap (uncurry (+)) . pair
```

The `mySum1` program uses `pair` to pair up the first element of the array with the second, the third with the fourth etc. On the resulting array `uncurry (+)` is applied to each pair, giving an array of half the length. This is composed, (`->-`), with a recursive call to `mySum1` that continues the summation until only a single element is left. Hence, this algorithm for summing up the elements of an array only works for arrays with a length a power of two. The generated code is single threaded because the length of the output array is one.

The direct approach to parallelising the `mySum1` program is to do the following: if given an array of length 2^n, pair the elements up giving an array of length 2^{n-1} then in parallel apply `(+)` to these pairs using 2^{n-1} threads. Then proceed with the recursion, in the next stage using 2^{n-2} threads, until the sum is found. In Obsidian this version of the summation program is obtained by simply adding a `sync` at a suitable place:

```
__global__ void generated(word* input,word* result){
  unsigned int tid = (unsigned int)threadIdx.x;
  extern __shared__ unsigned int s_data[];
  word __attribute__((unused)) *sm1 = &s_data[0];
  word __attribute__((unused)) *sm2 = &s_data[0];
  ix_int(result,tid) =
    (((ix_int(input,(tid << 3)) +
       ix_int(input,((tid << 3) + 1))) +
      (ix_int(input,(((tid << 2) + 1) << 1)) +
       ix_int(input,((((tid << 2) + 1) << 1) + 1)))) +
     ((ix_int(input,(((tid << 1) + 1) << 2)) +
       ix_int(input,((((tid << 1) + 1) << 2) + 1))) +
      (ix_int(input,(((((tid << 1) + 1) << 1) + 1) << 1)) +
       ix_int(input,((((((tid << 1) + 1) << 1) + 1) << 1) + 1)))));
}
```

Figure 9: Code generated from the sequential version of mySum

```
mySum2 :: Int -> Arr IntE :-> Arr IntE
mySum2 0 = pure id
mySum2 n = pure op ->- sync ->- mySum2 (n-1)
    where
      op = fmap (uncurry (+)) . pair
```

The `sync` primitive signals that at this point the array should be computed and stored in memory. The computation of the array is performed using one thread per element. Using `sync` also enables sharing of the computed values between threads in the future. The addition of a `sync` here does however not have any effect on the semantics of the program, only on the performance of the generated code. Figures 9 and 10 show the code that is generated from the two different versions of `mySum`.

This is an example of executing `mySum2` on the GPU.

```
*Obsidian> execute (mySum2 3) [0..7]
[28]
```

```
__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[4];
  ix_int(sm1,tid) = (ix_int(input,(tid << 1)) +
    ix_int(input,((tid << 1) + 1)));
  __syncthreads();
  if (tid < 2){
    ix_int(sm2,tid) = (ix_int(sm1,(tid << 1)) +
      ix_int(sm1,((tid << 1) + 1)));
  }
  __syncthreads();
  if (tid < 1){
    ix_int(result,tid) = (ix_int(sm2,(tid << 1)) +
      ix_int(sm2,((tid << 1) + 1)));
  }
}
```

Figure 10: Code generated from the parallel version of mySum, `mySum2`.

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:

```
mySum3 :: Int -> Arr IntE :-> Arr IntE
mySum3 0 = pure id
mySum3 n = pure op ->- (if (n <= 3 )
                        then sync
                        else pure id) ->- mySum3 (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 this Obsidian program for 32 elements, chunks of 4 elements would be summed up in sequence giving an array of length 8. The array of length 8 is then summed up using the parallel method.

Another program that accomplishes the same thing is the following:

```
mySum4 :: Int -> Arr IntE :-> Arr IntE
mySum4 n = pure ((fmap (foldr (+) 0)) . chopN 4) ->-
               sync  ->- mySum2 n
```

The program chops the array up into an array of arrays where the inner arrays are of length 4. The inner arrays are then folded using `(+)`. The program then proceeds by letting the parallel `mySum2` program sum up the resulting array. The `Int` argument to `mySum4` tells how many stages the parallel summation should consist of. So to sum up an array of 32 elements this argument should be 3, because 32 divided by 4 is 8 and summing up 8 elements needs 3 stages. Running this version of the program on the GPU gives the following result:

```
*Obsidian> execute (mySum4 3) [0..31]
[496]
```

The code generated from this program looks like this. Notice how four elements are summed up sequentially in each thread before proceeding as in `mySum2`:

```
__global__ void generated(word* input,word* result){
  unsigned int tid = (unsigned int)threadIdx.x;
  extern __shared__ unsigned int s_data[];
  word __attribute__((unused)) *sm1 = &s_data[0];
  word __attribute__((unused)) *sm2 = &s_data[8];
  ix_int(sm1,tid) =
     (ix_int(input,(tid << 2)) +
      (ix_int(input,((tid << 2) + 1)) +
       (ix_int(input,((tid << 2) + 2)) +
        ix_int(input,((tid << 2) + 3)))));
  __syncthreads();
  if (tid < 4){
    ix_int(sm2,tid) = (ix_int(sm1,(tid << 1)) +
       ix_int(sm1,((tid << 1) + 1)));
  }
  __syncthreads();
  if (tid < 2){
    ix_int(sm1,tid) = (ix_int(sm2,(tid << 1)) +
       ix_int(sm2,((tid << 1) + 1)));
  }
  __syncthreads();
  if (tid < 1){
    ix_int(result,tid) = (ix_int(sm1,(tid << 1)) +
       ix_int(sm1,((tid << 1) + 1)));
  }
}
```

2.2.2 Grouping of work

The `sync` primitive can also be used to group work together. This can be used to lower the number of threads needed to perform a certain task by having each thread do more work. The `sync` primitive signals that the array supplied as input to `sync` should be computed and stored in memory. The number of threads used to compute the array is equal to the number of elements in the array. So, one way to halve the number threads used is to sync on an array of pairs instead of syncing on an array. For example:

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

The code generated from this Obsidian program increments all elements of an array of length $2n$ using n threads:

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

Of course, this means that the resulting array is an array of pairs and it is left up to the user of the result to transform it into a suitable form.

2.2.3 Divide and conquer

In the divide and conquer paradigm problems are solved by splitting them into smaller sub-problems that can be solved independently. The solutions to the sub-problems are then fused together into a solution to the original problem.

Obsidian provides a combinator called `two`:

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

The `two` combinator is a special case of parallel composition. It takes a single GPU program as input and gives as result a new GPU program. The resulting GPU program splits the input in the middle and applies the original program to both halves, as shown in figure 11. This combinator is useful for divide and conquer algorithms where the same program is used to solve both subproblems.

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 both halves are found recursively. This is followed by simply selecting the smaller of the two minima as the answer. In Obsidian, using `two`, this program is implemented as:

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

The array program `min2` uses indexing, `(!)`, and a function called `singleton :: a -> Arr a` that creates a one element array:

```
min2 :: Arr IntE -> Arr IntE
min2 arr
    | len arr /= 2 = error "wrong input"
    | otherwise =  singleton $ ifThenElse (a <* b) a b
    where
      a = arr ! 0
      b = arr ! 1
```

Below, the `minimum` program is executed on eight inputs:

```
*Obsidian> execute (minimum 3)  [9,9,2,5,7,4,4,3]
[2]
```

The following listing shows code generated from the `minimum` program for eight inputs. One thing to notice in the generated code is how the applications of `two` have been turned into bitwise operations on the indexes. The implementation of `two` will be shown in section 3.

```
__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[4];
  ix_int(sm1,tid) =
     ((ix_int(input,((tid << 1) & 0x6)) <
       ix_int(input,(((tid << 1) & 0x6) | 0x1))) ?
        ix_int(input,((tid << 1) & 0x6)) :
        ix_int(input,(((tid << 1) & 0x6) | 0x1)));
  __syncthreads();
  if (tid < 2){
    ix_int(sm2,tid) =
       ((ix_int(sm1,((tid << 1) & 0x2)) <
         ix_int(sm1,(((tid << 1) & 0x2) | 0x1))) ?
          ix_int(sm1,((tid << 1) & 0x2)) :
```

```
          ix_int(sm1,(((tid << 1) & 0x2) | 0x1)));
  }
  __syncthreads();
  if (tid < 1){
    ix_int(result,tid) =
       ((ix_int(sm2,0) < ix_int(sm2,1)) ?
ix_int(sm2,0) :
        ix_int(sm2,1));
  }
}
```

Another combinator related to `two`, is `ilv`. Instead of splitting the array in the middle, `ilv` splits the array into one array of its odd elements and another of its even elements. Then, just as in `two`, the same program is applied to both of the arrays.

While `two` is a given primitive, `ilv` can be implemented using `two`, `riffle` and `unriffle`, see figure 5:

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

The `ilv` combinator can, for example, be used to implement mergers. A merger is a useful building block in sorting algorithms. This usage of `ilv` will be demonstrated in section 4.

2.2.4 Experimental features

In section 2.2.1, we saw how `sync` can be used to guide the code generation and introduce parallelism and in section 2.2.2, `sync` was used to group work. The example in section 2.2.2 showed only one way to group work. The work was grouped by pairing neigbouring elements up and using one thread to compute each pair. However, there are other ways you may want to be able to group the work. The `pure pair ->- sync` approach lets a single thread compute two neighbouring elements of an array. But what if a better grouping of work is to let each thread compute element tid and $tid + (n/2)$ in an array of length n? This grouping of work is of course possible to achieve, for example by `pure (zipp . halve) ->- sync`, see figure 12. The drawback of this approach is twofold. First, the zipping and halving of the array introduces extra computation in the indexing function that not at all contributes to the computation of the result. Of course, grouping of

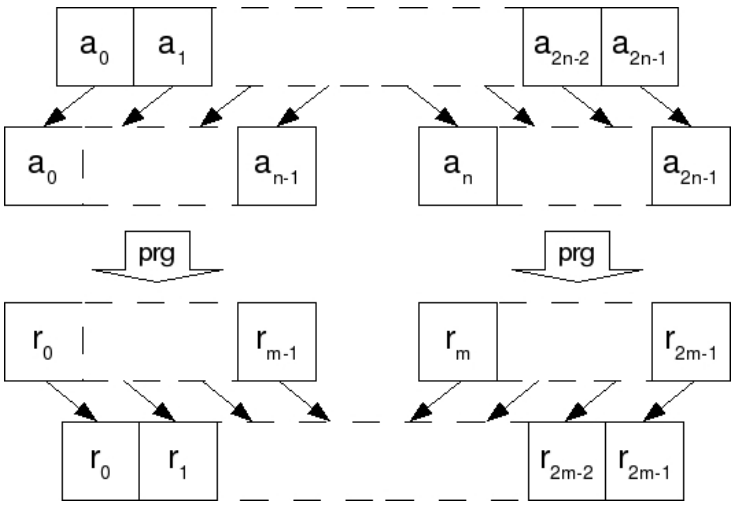

Figure 11: Divide and conquer combinator.

work will result in more arithmetic in the index calculations but it should be kept to the minimum possible. Second, this approach results in movement of data. When syncing on pairs the two parts of the pair are stored next to each other in memory.

What is needed is a method to assign work to threads that does not introduce unnecessary computation in the indexing function and that does not move elements. In the current version of Obsidian there is an experimental solution that has these two properties. Currently this solution is called `syncHow`. `syncHow` is very similar to `sync` but it has an extra argument:

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

The extra argument to `syncHow` holds information on how the array synced upon should be written to memory. The information provided in the how argument contains a mapping between thread IDs and indices.

The details of the `How` type are shown in section 3, but here is an example of its use. For this example, let us assume that there is a function `pairNth :: Int -> How` that takes an `Int`, n, and creates a `How` object that instructs the synchronisation to let each thread tid compute and store element tid and $tid + n$. Using `pairNth` and `syncHow` a new version of `incrP` can be implemented:

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

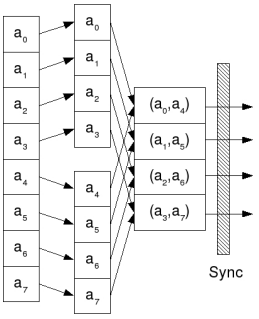

Figure 12: Halve the array, zip the two halves together, then sync.

Now, one immediately apparent drawback is the hardcoded 4 which means that this program is only correctly applicable to arrays of length 8. Future work will focus on removing this limitation. Executing this version of `incrP` on the GPU results in:

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

In the generated code it can be seen how the `How` argument has been used in the code generation. Each thread computes two values of the output array, `result`. The thread with thread ID *tid* computes element *tid* and element $tid + 4$:

```
__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);
}
```

Another limitation of the `sync` primitive is that it always results in a corresponding `__syncthreads()` call in the generated code, even in cases where a barrier synchronization is really not needed. Using the barrier synchronisation is not necessary in the cases where all the communication that takes place is between threads that belong to the same warp. For future versions of Obsidian the decision to generate a `__syncthreads()` or not will most likely

```
pure :: (a -> b) -> a :-> b
(->-) :: (a :-> b) -> (b :-> c) -> (a :-> c)
sync :: Flatten a => Arr a :-> Arr a
two :: (Arr a :-> Arr b) -> Arr a :-> Arr b
ilv :: (Arr a :-> Arr b) -> Arr a :-> Arr b
```

Figure 13: A selection of functions from the GPU Program API

be done automatically, but in the current version the programmer can experiment with this directly by using another sync primitive called `syncWarp`. This kind of sync does not generate a `__syncthreads()` and it is up to the programmer to decide if it is safe to exclude this barrier or not.

There is one last primitive, called `syncIP`, that fits in this category of more experimental features. The generated code shown so far in this document uses two shared memory arrays. One array acts as source and the other as target and then their roles are exchanged. There is however a class of *in-place* algorithms that do not need two memories to work. Instead the result can be computed into the same space that is occupied by the source. The `syncIP` primitive allows the programmer to express that the result should be stored in the same place as the input is gathered from, making the algorithm in-place in regard to shared memory.

The functions `syncIP` and `syncWarp` are also available with a `How` parameter, called `syncIPHow` and `syncWarpHow`.

The experimental features described in this section will be shown in action in section 4, where they are used to optimize the implementation of an algorithm.

This section introduced Obsidian by showing language features using very small examples. Section 4 applies Obsidian to some slightly larger case studies such as parallel prefix and sorting.

3 Implementation of Obsidian

This section describes the implementation of Obsidian starting with the base types used in the language, `IntE`, `FloatE` and `BoolE`. Following this is an exploration of the array language by giving the implementation of some of the functions found there. The rest of the section explains GPU programs and how code is generated from them.

3.1 Expressions

Now its 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 type synonyms:

```
type IntE  = Exp Int

type FloatE = Exp Float

type BoolE = Exp Bool

type IndexE = Exp Word
```

The `Exp a` type is simply a wrapper around an untyped expression, `DExp`:

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

Hence, all expressions in Obsidian are represented by the same `DExp` type. This is an example of usage of *Phantom types* [11]. Phantom types can be used to provide a typed interface to an untyped structure.

The type `DExp` represents expressions that can be `Int`, `Float` or `Bool` valued. The `DExp` type models a subset of C's expressions and can be used very directly to output C code, see section 3.5:

```
data DExp = LitInt  Int
          | LitUInt Word32
          | LitBool Bool
          | LitFloat Float
          | Op2 Op2 DExp DExp
          | Op1 Op1 DExp
          | If DExp DExp DExp
          | Variable Name
          | Index DExp DExp Type
```

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

```
data Op1 = Not
         | BitwiseComp
         | Exp | Log | Sqrt
```

```
         | Cos | Sin | Tan
         | CosH | SinH | TanH
         | ACos | ASin | ATan
         | ACosH | ASinH | ATanH
```

The `Op2` data type contains those binary operations that are supported.

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

To make these expressions more pleasant to work with suitable instances of `Num`, `Integral`, `Fractional` amongst others are created. It is also here that optimisations are applied, for example `(+)` inspects its two arguments and performs operations such as constant folding.

3.2 Array Language

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

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

There are two functions `len` and `(!)` defined on `Arr`:

```
len :: Arr a -> Int
len (Arr _ n) = n

(!) :: Arr a -> IndexE -> a
(!) (Arr ixf _) ix = ixf ix
```

The `len` function simply returns the length. The indexing function `(!)` applies the array's index function to a given index. For example, these two functions are used in the implementation of the `rev` function in the array library:

```
rev :: Arr a -> Arr a
rev arr  = Arr 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, n is the length of the original array.

Let us look at the implementation of a few more array functions, for example the function `pair` is implemented as follows in the Obsidian libraries:

```
pair :: Arr a -> Arr (a,a)
pair arr | odd (len arr) = error "Pair: Odd n"
         | otherwise = Arr (\ix -> (arr ! (ix * 2),
                                    arr ! ((ix * 2) + 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. An element of the array of pairs is found by indexing in the original array at $ix * 2$ and $ix * 2 + 1$.

Going back from an array of pairs to an array is also useful:

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

The `unpair` function requires the elements of the array to be in the `Choice` class. This is because it uses a conditional in the indexing function.

The functions `pair`, `unpair`, `halve`, `conc`, `zipp` and `unzipp` can be used to inplement the two functions `riffle` and `unriffle`:

```
riffle :: Choice a => Arr a -> Arr a
riffle = conc . unzipp . pair

unriffle :: Choice a => Arr a -> Arr a
unriffle = unpair . zipp . halve
```

However, in an attempt to increase the performance of the generated code, `riffle` and `unriffle` are also given a lower level implementation. This implementation of `riffle` and `unriffle` is based on rotation of bits in the index.

```
riffle' :: Arr a ->  Arr a
riffle' arr | even (len arr) =
               Arr (\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) =
                 Arr (\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 `rotLocalL` and `rotLocalR` functions rotate 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 the functions that make up the array language. The other functions in this library are all implemented in a similar way. The functions `riffle` and `unriffle` are examples of functions that can be implemented using existing primitives but are instead also given a primitive implementation for efficiency reasons. The need for giving more low level implementations of certain functions may be reduced by improved optimisation techniques.

3.3 GPU Programs

GPU Programs are represented by the `a :-> b` type. Currently this type has two constructors, `Pure` and `Sync`.

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

The function `pure` that creates a GPU program corresponds directly to the constructor `Pure` of the `(:->)` type:

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

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

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

The type `FData` represents things that can be written to the GPU memory. The class `Flatten` provides the two functions `toFData` and `fromFData` to convert to and from a format storable 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 type `FData` is defined as follows:

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

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

A base type such as an `Int` valued expression, an `IntE`, is turned into `FData` using the `Unit` constructor:

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

the function `unE :: Exp a -> DExp` turns a typed expression into its untyped representation.

A pair is turned into `FData` by using the `Tuple` constructor. Arrays however, are turned into `FData` by using the `For` constructor. This also results in a for loop over that very array in the generated C code. This means that if you sync upon arrays of arrays the elements of each inner array are computed sequentially.

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

The type, `(:->)`, of GPU programs is very similar to normal Haskell lists. There is a unit GPU program, created by `Pure`, corresponding to a list of length one. A GPU program 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 in a sense a GPU program is a list of computations and the idea is that these computations should be performed in sequence with a barrier synchronisation in between.

With this idea in mind it is a good time to look at the implementation of the 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 GPU programs created with `Pure` is the same as Haskell function composition. The next case, where `Pure f` is composed with `Sync g h`, means that some more computation should take place before the sync, ie. it means, perform `g . f` then sync and continue with h. The last case is where something constructed using `Sync` is composed with anything else. In this case `(->-)` proceeds recursively.

3.3.1 The `two` combinator

The `two` combinator is a bit special. The input to it is a single GPU program and the output is a new GPU program. The input program is either `Pure f` or constructed using `Sync`. In the case of `Pure f` a `Pure f'` must be constructed where `f'` performs `f` on both halves of an array. In the `Sync f g` case the scenario is just slightly more complicated. The same transformation as in the `Pure` case must be performed on the function `f` but then it must also be applied recursively to the GPU program g.

In the `Pure f` case, one way to accomplish the desired result is to let `f' = conc . (\(x,y) -> (f x, f y)) . halve`, but the resulting code is not efficient. Especially if two is used in an algorithm recursively (as it most often will be), this ends up being very inefficient. This is mainly because of

the conditionals that `conc` introduces into the indexing function of the array. Instead, a similar approach to that used in `riffle'` and `unriffle'` is used. The conditionals are completely replaced by bitwise logic on the indices. It is then up to a set of optimisations on bitwise logical expressions to generate good code.

The following listing shows the implementation of two, it in turn uses the function `twoFF` that performs the actual transformation of the array programs between the syncs:

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

The implementation of the `twoFF` function is obtained from `f' = conc . (\(x,y) -> (f x, f y)) . halve` by inlining the definitions of `conc` and `halve`, the conditionals are then moved inwards as much as possible and replaced by bitwise logic.

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

3.4 Experimental features

In order to support the experimental versions of the `sync` function. The type (`:->`) is extended with some more information in the `Sync` case:

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

The SyncInfo type holds information about whether or not to store data in-place and if the code generator should insert a `__syncthreads` or not:

```
data SyncInfo = SyncInfo {inPlace :: Bool,
                          inWarp  :: Bool}
```

The `How` type is currently a Haskell function from an index to a list of indices:

```
type How = IndexE -> [IndexE]
```

The meaning of the `How` function is that, if given a thread ID as input the output will contain a list of thread IDs that the given input thread shall pretend to be. That is input is a "real" thread and outputs are "virtual" threads.

Of course, representing the `How` argument in this way has some obvious risks. The programmer is free to provide any function (that fits the type). For example the function (`\x -> [0]`), that tells every thread to pretend to be thread zero. There are some ideas on how to improve this situation, description of these are defered to the future work section (section 5).

However, the `syncHow` approach is clearly worth exploring further. It decouples the notions of "what to compute" and "how to compute it" even further, which is desired.

3.5 Code generation

In order to generate C code from an Obsidian description it is first necessary to gather some information about the program. Two important pieces of information are the number of threads needed to compute the result and the amount of shared memory needed to hold intermediate values.

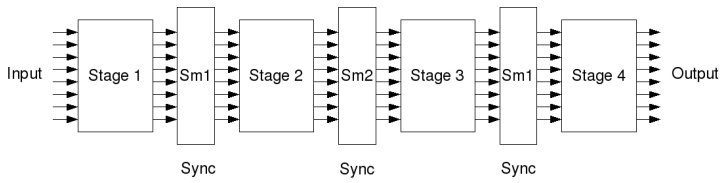

Figure 14: A program alternately using the arrays sm1 and sm2 for storage.

A GPU program can be seen as a sequence of computations separated by barrier synchronisations. The amount of shared memory that is needed for a program in total is the maximum of the memory requirements of the computations separated by synchronisations. The number of threads needed is obtained in the exact same way. In order to get the shared memory requirements of the program an input of the desired size is needed, as indicated by the pseudo code below:

```
sharedMemReq :: a :-> b -> a -> Int
sharedMemReq (Pure _) _ = 0
sharedMemReq (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 the `input` is needed.

When the number of threads and amount of shared memory needed has been computed it is time generate the actual code. When generating the Code for a particular stage of the program the number of threads needed to compute that stage is needed. If the number of threads needed for the program in total is higher than the number of threads needed in a given stage, the code for that stage will be enclosed in a conditional that disables a number of threads. Examples of where this happens are the `mySum` programs from section 2.2.1.

The generated code uses two shared memory arrays to perform the computations, these arrays are called `sm1` and `sm2` and are used alternately. Figure 14 shows an example of how a program uses the two shared memory arrays.

Figure 15 indicates how code is generated for the following program given functions `f :: Arr IntE -> Arr IntE` and `g :: Arr IntE -> Arr IntE`:

```
example :: Arr IntE :-> Arr IntE
```

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

If we overlook the extra `how` arguments to the `Sync` constructor, the datastructure that represents the above program looks like this:

```
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 datastructure, C code is generated by applying a symbolic array as input to the function `f1`. Given the symbolic array `Arr (\ix -> index (variable ‘‘input’’) ix) n` the result of this is an expression corresponding to `f(input[tid])`. The result of this is supposed to be stored in the `sm1` shared memory array followed by a barrier synchronisation giving:

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

Following this an input must be created to be used as symbolic input to the next function, `f2`. This symbolic input must represent an array located in `sm1`: `Arr (\ix -> index (variable ‘‘sm1’’ ix) n)`. This symbolic array can be turned into an array of `FData` and supplied to `f2` as input giving an expression `sm1[n-1-tid]` as result. Now the result of this stage should be stored into the `sm2` array:

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

The same procedure is repeated for the last stage but here since we have arrived at an instance of the Pure combinator we know we are at the end of the program. the result of this computation should be written to an array called `output`, giving a symbolic array `Arr (\ix -> index (variable ‘‘output’’ ix) n)` should be used here. It is also not necessary to add a barrier synchronisation at this point.

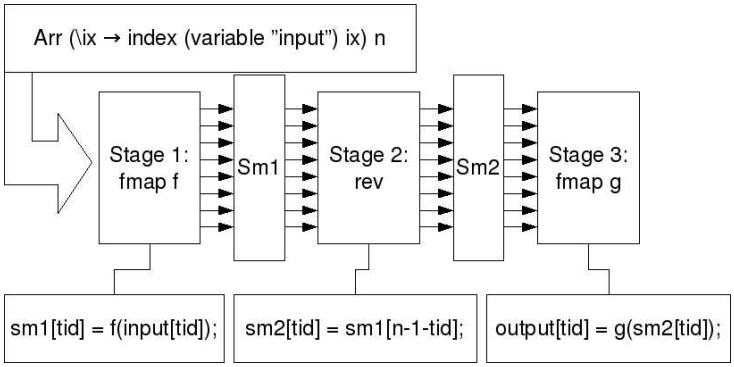

Figure 15: A sketch of how code is generated for a program.

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

The body of the program given by this procedure in total is shown below:

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

The code above looks slightly different from the examples of real generated code shown on other places in this document. The difference is that the other code shown uses a couple of macros for indexing. Instead of having `sm1[tid]` this is written, using the macros, as `ix_int` if the data is integers, `ix_float` if floating point numbers. The data itself is stored in arrays of 32-bit words. The reason for this is that a choice was made to store the elements of a pair in an array of pairs next to each other in memory. This means that the shared memory array, `sm1` for example, may contain interleaved floating point and integer data.

This section showed an outline of how C code is generated given an GPU program. Many details have been left out but the key ideas are present in the description.

4 Case Studies

This section describes a few slightly larger Obsidian programs. One of the case studies, on prefix networks, also contains some performance measure-

ments.

4.1 Reduction

Reduction is simply a generalisation of the `mySum` concept shown in section 2.2. However, this highlights one of the strengths of a higher level language compared to CUDA, where a similar abstraction would be harder to express. It is possible that resorting to C++ and Templates would offer the same capabilities to the CUDA programmer.

```
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
```

This can be easily 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 even be used to sum up an array of 3D vectors. The `Vec3` type is defined as follows:

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

An addition function on these vectors can be implemented like this:

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

And `reduce` is applicable without change:

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

4.2 Mergers

Mergers are components that are capable of taking two sorted sequences and turning them into one sorted sequence. These mergers can then be used to implement sorting algorithms. This section describes the implementation of two different mergers, one based on the butterfly network and one called the odd-even merger [8].

The butterfly merger , `bfly`, is capable of merging two sequences correctly if the first one is sorted and the second is sorted in the reversed order. The actual `bfly` program will take these two sequences as a single array. Figure 16 illustrates the access pattern of an 8 input butterfly.

The Obsidian program below that implements the butterfly takes a function from pairs to pairs as argument. As argument the desired *compare and swap* function should be supplied:

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

The `Int` parameter to the merger should be the *log* of the array length desired.

In order to run the merger a compare and swap function is needed:

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

This compare and swap function will be used in the following examples of mergers and also in the next section about sorters.

The butterfly merger can be executed on the GPU:

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

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

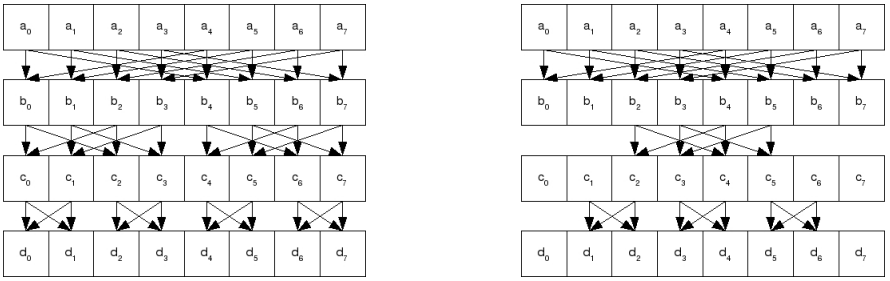

Figure 16: The Butterfly network and the Odd-Even merging network.

The next merger is called Bathcher's *Odd-Even merger*, this merger merges two sorted sequences [2]. This is different from the butterfly merger which required one of the sequences to be sorted in the reversed order. Figure 16 shows the data access pattern of this merger.

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

Using the same compare and swap component as in the previous example, executing this merger on the GPU works as follows:

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

4.3 Sorting Networks

Sorting is a popular function to place on the GPU, see [22, 7, 21]. This section describes two different sorting algorithms known as *Odd-Even sort* and *Bitonic sort*.

Odd-Even sort uses the Odd-Even merger shown previously. The algorithm follows a divide and conquer approach, using `two`. The array is split in two halves and recursively sorted. Following this, the two sorted sub-arrays are merged using the Odd-Even merger.

```
sortOE :: Int -> (Arr IntE :-> Arr IntE)
```

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

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

The next example of a sorting network is Bitonic sort. Bitonic sort uses the butterfly merger instead and needs to reverse one of the sub-arrays before merging them. Besides that, the algorithms follow similar recursive decompositions.

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

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

4.4 Parallel Prefix

This subsection shows the implementation of a parallel prefix (also called scan) kernel, known as `sklansky` after J. Sklansky [23]. This kernel will then be optimised step-by-step using Obsidian.

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

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

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 [4].

Figure 17 shows the recursive decomposition of the sklansky parallel prefix network. The sklansky parallel prefix algorithm is implemented by splitting

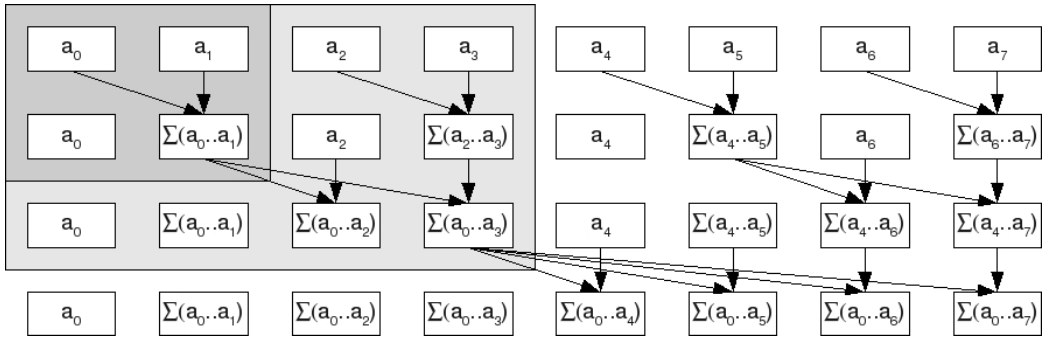

Figure 17: The sklansky parallel prefix network.

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

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

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

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

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

Since the code generated by Obsidian is not by default in-place with regard to shared memory, each thread in the 256 threaded program needs to both perform the operation between two elements and copy one value unchanged. This is desired because it would mean that each thread performs the exact

same operations which means that there is no risk for divergence within a warp.

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

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

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

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

A last tweak to apply is to use the in-place version of `sync`. In code this looks as follows:

```
sklansky3 :: (Flatten a, Choice a) =>
             Int -> (a -> a -> a) -> (Arr a :-> Arr a)
sklansky3 0 op = pure id
sklansky3 n op = two (sklansky3 (n-1) op) ->- pure (fan op)
          ->- if (n < 5)
               then syncIPWarpHow (pairNth 256)
               else syncIPHow (pairNth 256)
```

4.4.1 Parallel Prefix Sums on large arrays

The Sklansky kernels given above can be used in an algorithm that computes the parallel prefix of a large array. This is done through an approach similar to that used in the summation of a large array as shown in section 1.1.1. The large array is split up into chunks of 512 elements, each of these are scanned using the kernel. The kernels needs to be slightly modified so that they also output their maximum to a separate array of block maximums. The array of block maximums are then scanned recursively and the results of that are distributed to the chunks of the large array, see figure 18.

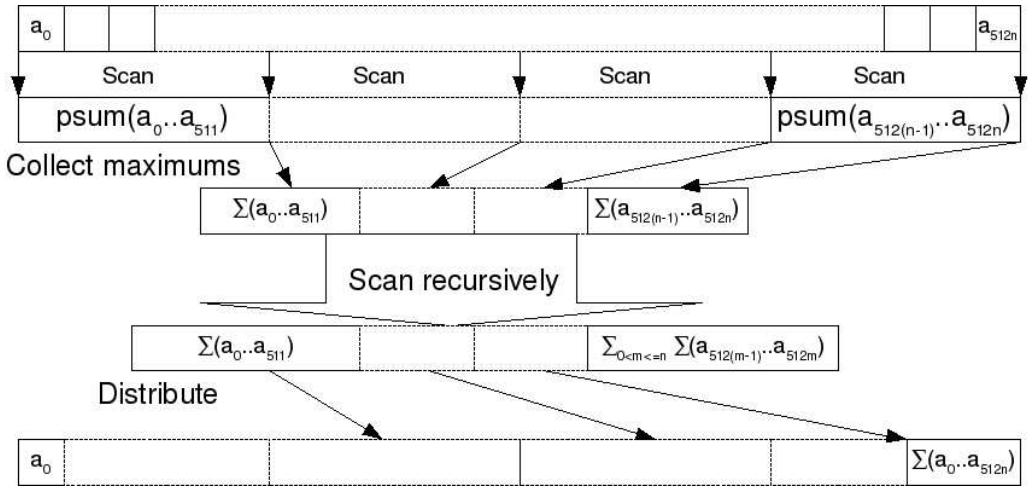

Figure 18: The large array is divided into chunks. Each chunk is scanned using the small parallel prefix kernel. The maximum values are stored in a separate array that is in turn scanned and then distributed over the chunks.

The table below shows the results of using the different parallel prefix kernels from above in an algorithm for computing the prefix sums of 2^{20} elements.

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

The table above shows the running times of five different Sklansky kernels. The one labeled as *Hand Optimised* was written directly in CUDA. This

kernel was the result of two afternoons of optimisation effort by two people. The four other kernels *sklansky* to *sklansky3* are generated from the given Obsidian programs. Even the very first version, *sklansky1* performs very well but by using the experimental features the performance can be pushed quite close to the hand coded version.

5 Future work

Obsidian is work in progress and as such it changes a lot. This document also described some experimental features of Obsidian. Future work will consist of evaluating these experimental features and maybe integrating them more thoroughly into the language.

The experiments with a `How` parameter, even though the current approach has many problems, shows a path to decoupling the notions of what to compute and how to compute it even further. This is something that must be considered for future versions. The `How` parameter must also be made more safe and compositional. One possible approach to that is to make the `How` argument an array and move it into the GPU program, it currently exists outside of the program in question. In the datatype below, the array that is suposed to be written to memory is zipped with an array of identities of threads. Each element is written to memory by the thread it whose ID it is paired up with:

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

Since the length of arrays are known, the same information that the `How` as a function provided can be derived from the (`Arr (IndexE,FData)`).

Another possibility is to replace the `How` parameter by more constructors in the `a :-> b` type. These new constructors could have different meaning in regard to division of work. In this setting `Sync` could mean; compute the array and write it to memory using one thread per element. Another constructor `Parl` could take two programs and compute the first program on one array using threads 0 to n and compute the second program using threads $n+1$ to m. Which of these two approaches, `How` as an array or more constructors, that will work out best needs to be explored.

Section 4 on case studies showed that it is possible to generate quite efficient code from Obsidian programs. The current version generates efficient code

for very specific uses of the `two` combinator, for example the kind used in the `sklansky` parallel prefix network. This is depending on a set of compiler optimisations on bitwise logical expressions. The set of optimisations in use now produces efficient code for certain uses of `two` but is of limited value for the `ilv` combinator. More work need to be done in order to find a good way to produce efficient code more reliably. It is also clear in some of the examples that common sub-expression elimination could be worth exploring. However, some care must be taken to strike a balance between recomputation and register use.

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

6 Related work

GPUs are becoming more and more interesting to use in non-graphical applications. A modern GPU is a manycore machine with, today, hundreds of processing elements. The question of how to program these machines arises, NVIDIA's answer is CUDA [18]. CUDA supplies a slightly extended version of C in which the programmer can specify GPU kernels and the controlling CPU program in the same language. There are a number of other C/C++ based languages that target GPGPU programmers, for example Brook[6] and RapidMind[15]. Brook, CUDA and RapidMind are major improvements from what was previously available for the programmer interested in general purpose computations on the GPU. Before these, the GPGPU programmer only had the graphics API to work with and needed to translate his or her programs into graphics vocabulary in order to use the GPU.

Higher level approaches are also being investigated. PyGPU embedds a GPGPU programming language in Python[14]. PyGPU makes use of Python's introspective abilities to generate efficient code.

Like Obsidian, GPUGen is embedded in Haskell [13]. GPUGen however, is higher level language than Obsidian. Where the purpose of Obsidian is to implement basic algorithmic building blocks such as reductions and prefix sums, GPUGen provide these building blocks as primitives.

Vertigo is another GPU programming language embedded in Haskell[9]. However, Vertigo is targeting graphical applications.

There are also many examples of languages that do not specifically target GPUs but experiment with new methods and ways of parallel programming.

In this category we have Sequoia, where the memory hierarchy is in focus. Sequoia programs can for example be compiled to the Cell BE architecture. Data-Parallel Haskell extends Haskell with parallel arrays and operations on parallel arrays[12]. Data-Parallel Haskell implements the nested data-parallel paradigm and is in that sense following in the path of NESL[5].

7 Conclusion

Obsidian is work in progress and there are many loose ends to tie up and paths left to explore. In section 4, the strengths of Obsidian show; it is possible to express quite complex algorithms using short and elegant programs. The case studies also show that it is possible to generate quite efficient code from these high level descriptions. However, this needs more work in order to more reliably produce efficient code, not just generate efficient code for very specific uses of certain combinators. This indicates that more work needs to be done in optimising generated code, or maybe that another set of combinators with properties different from those currently available is needed.

Another benefit of a higher level language such as Obsidian compared to CUDA is the ability to reuse code. There is an example of this too in the case studies section where a single reduction program can be used to generate code to compute the maximum, minimum and sum very easily.

In Obsidian it is easy to describe an initial prototype solution to a problem, such as for example the `mySum` kernels in section 2.2.1. It requires no deep knowledge of the GPU architecture but works out of the box. The prototype implementation can be tweaked into a more efficient implementation by performing small changes to the Obsidian code that often lead to quite large differences in the generated C code. An example of this is `mySum1` against `mySum2` where adding a single `sync` leads to a radically different C program.

In order to get to the quite efficient version of the `sklansky` kernel in the case studies section, the experimental `How` function was needed. As stated in section 5, this needs to be explored further. `How` as it is today, any function `IndexE -> [IndexE]`, offers too much freedom and with that risk to introduce errors. Hopefully some elegant model for expressing how and what to compute will evolve over time.

One drawback of the hardware like approach used with Obsidian is that we cannot describe algorithms where the size of the output is data dependent. An example of such a data dependent algorithm is `filter`. The `filter` function takes a sequence of elements and a predicate and produces a sequence

of those elements for whom the predicate holds.

Obsidian offers the GPGPU programmer a higher level language while trying not to sacrifice too much performance. When Programming in CUDA C the indexing arithmetic often gets quite complex, see section 1.1.1. This is a common trait of data-parallel programming in C like languages. One Goal of Obsidian is to be able to express these algorithms without the complex index manipulations; instead the data access pattern is captured in the use of functions such as `pair` and `two` or in the recursive structure of the Obsidian program.

Obsidian is also an improvement over CUDA in the area of code reuse. The `reduce` function from section 4 is an example of this. Obsidian is also compositional in another way than CUDA. Reusing kernels as building blocks in other kernels is in CUDA not realistic, while in Obsidian it is the prefered way to write programs. Take as an example of this the mergers and sorters in section 4. In CUDA you would design one kernel from scratch implementing the merger and sorter simultaneously.

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

References

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

[2] 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.

[3] 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.

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

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

[6] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, August 2004.

[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] 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.

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

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

[11] 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.

[12] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M T Chakravarty. Harnessing the multicores: Nested data parallelism in haskell. In Ramesh Hariharan, Madhavan Mukund, and V Vinay, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2008)*, volume 2 of *Leibniz International Proceedings in Informatics*, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[13] Sean Lee, Manuel M. Chakravarty, Vinod Grover, and Gabriele Keller. GPU Kernels as Data-Parallel Array Computations in Haskell. http://www.cse.unsw.edu.au/ chak/papers/gpugen.pdf, 2009.

[14] 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.

[15] Michael D. McCool. Data-parallel programming on the Cell BE and the GPU using the RapidMind Development Platform. 2006.

[16] Hubert Nguyen. *GPU Gems 3*. Addison-Wesley Professional, August 2007.

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

[18] NVIDIA. *NVIDIA CUDA Programming Guide 2.0*. 2008.

[19] NVIDIA. *NVIDIA CUDA Best Practices Guide*, July 2009.

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

[21] 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.

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

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