Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 1 (2012) 2065–2074

ELSEVIER

Procedia Computer Science

www.elsevier.com/locate/procedia

International Conference on Computational Science, ICCS 2010

GPGPU kernel implementation and refinement using Obsidian

Joel Svensson[1], Koen Claessen, Mary Sheeran

CSE Dept., Chalmers University of Technology, Gothenburg, Sweden

Abstract

Obsidian is a domain specific language for data-parallel programming on graphics processors (GPUs). It is embedded in the functional programming language Haskell. The user writes code using constructs familiar from Haskell (like `map` and `reduce`), recursion and some specially designed combinators for combining GPU programs. NVIDIA CUDA code is generated from these high level descriptions, and passed to the `nvcc` compiler [1]. Currently, we consider only the generation of single kernels, and not their coordination.

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

Keywords: Data-parallel, Embedded language, GPUs, Haskell

1. Introduction

Multicore and manycore processors are becoming increasingly common. Modern graphics processing units (GPUs) are examples of manycore processors. Today GPUs come with hundreds of processing elements capable of managing thousands of threads [2]. The Obsidian project is about exploring ways to program these new machines.

Obsidian is a domain specific language for general purpose programming on GPUs (GPGPU), capable of generating code for modern NVIDIA GPUs. In [3] we describe a version of Obsidian that uses a monadic interface. In the current version of Obsidian the monad has been replaced by another data structure, closely related to Arrows [4], representing GPU programs. Although the details are left out because of space limitations, the use of this GPU program representation can be seen in section 2.2. For a description of the implementation see [5] and for more general information on embedded language implementation see [6].

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 [7].

URL: `joels@chalmers.se` (Joel Svensson)

doi:10.1016/j.procs.2010.04.231

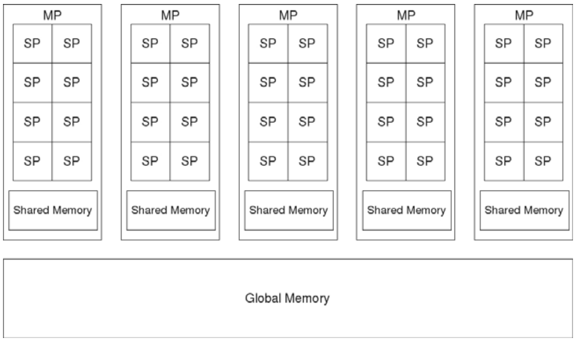

Figure 1: Conceptual image of a CUDA enabled GPU.

1.1. NVIDIA GPUs and CUDA

Starting with the 8000 series of GPUs (the G80 architecture), NVIDIA's GPUs came with a unified architecture, meaning that all the processing elements on the GPU are of the same kind. This was different from the previous generation's GPUs, which typically had two kinds of processing elements, fragment and vertex processors. Now these two kinds of processors are replaced by a single kind with capabilities surpassing both of the old ones. This new unified architecture together with development tools for GPGPU programming go under the name CUDA (Compute Unified Device Architecture) [1]. 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* consisting of 32 threads that are executed in an SIMD fashion on the MPs. Conditionals that take different paths within a warp have a negative effect on performance; the two diverging paths will in fact be executed sequentially [8].

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

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

1.2. Aims of Obsidian

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

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

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

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 [9]. 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 [10].

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

2.1. Array Language

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

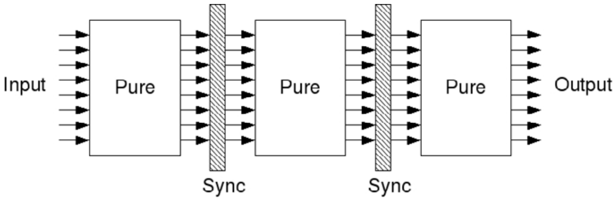

Figure 2: A GPU program of type `a :-> b` can be thought of as pure computations interspersed by syncs.

2.2. GPU Programs

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

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

Informally we can think of `a :-> b` as representing programs that operate as illustrated in figure 2. This figure shows a program that performs some computation using a number of threads followed by a barrier synchronisation, and so on. The contents of the boxes marked with *Pure* can be thought of as containing an array program.The type constructor `:->` is a variant of an arrow, which in turn is a generalisation of a monad [5]. One way to create a GPU program is by using the function `pure` of type `(a -> b) -> a :-> b`. For example the array language program `fmap (+1)`, that increments every element of an array, can be lifted to a GPU program:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3. Outline of Code Generation

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

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

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

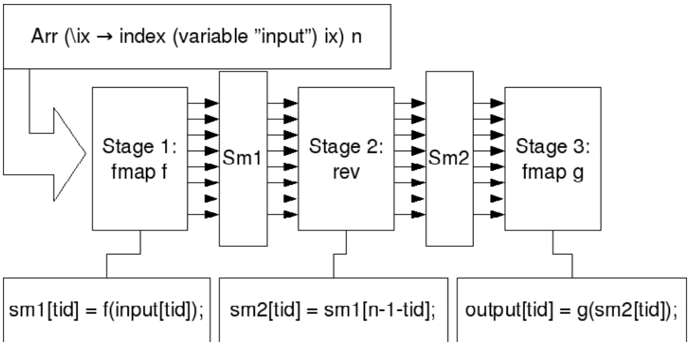
Arr (\ix → index (variable "input") ix) n
Stage 1: fmap f
Sm1
Stage 2: rev
Sm2
Stage 3: fmap g
sm1[tid] = f(input[tid]);
sm2[tid] = sm1[n-1-tid];
output[tid] = g(sm2[tid]);

Figure 3: Outline of the code generation procedure.

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

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

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

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

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

From this representation, which is essentially the list `[fmap f, rev, fmap g]`, the CUDA code is generated as outlined in figure 3. An input array is created and applied to the first stage of the computation, `(fmap f)`, resulting in:

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

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

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

4. Case Studies

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

4.1. Sorting

Sorting is a popular function to place on the GPU, see [11, 12]. *Odd-Even merge sort*, due to Batcher, uses a merger component called the Odd-Even merger. This merger merges two sorted arrays. In Obsidian this merger and sorter can be implemented using the combinators `two`, `ilv`, `evens` and `odds`. For a detailed description of these combinators see [13].

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

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

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

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

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

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

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

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

4.2. Parallel Prefix

This subsection shows the implementation of a parallel prefix (also called scan) kernel, known as `sklansky` after J. Sklansky [14]. This kernel will then be optimised step-by-step using Obsidian. This optimisation effort uses an experimental feature of Obsidian called `syncHow`. This is different from the normal `sync` in the way it assigns work to threads. For a more thorough explanation of `syncHow` see [5]. For an explanation of the parallel-prefix operation see [15].

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

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

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

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

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

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

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

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

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

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

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

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

4.2.1. 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 in the same way as in [16]. The table below shows the results of using the different parallel prefix kernels from above in an algorithm for computing the prefix sums of 2^{20} elements.

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

The table above shows the running times of six different Sklansky kernels. All runtime measurements where performed on an NVIDIA 9800GX2 using one GPU. The code labeled *Hand Optimised* was written directly in CUDA. This kernel was the result of two afternoons of optimisation effort by two people. The three following kernels *sklansky* to *sklansky2* are generated from the given Obsidian programs. The version called *sklansky3* is identical to `sklansky2` except is in-place. Even the very first version, *sklansky* performs reasonably well, but by using the experimental features the performance can be pushed quite close to the hand optimised version. The fastest kernel, called *NVIDIA SDK*, is the one supplied with the CUDA SDK. This kernel is based on the Brent-Kung network and its implementation is shown in [16]. The `NVIDIA SDK` kernel is highly optimized with regards to its memory access pattern and so the code is considerably more complicated than that of any of the Obsidian parallel-prefix kernels.

5. Future work

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

In order to get to the quite efficient version of the `sklansky` kernel in section 4, the experimental `How` argument to `sync` was needed. This needs to be explored further. `How` as it is today, any function of type `IndexE -> [IndexE]`, offers too much freedom and with that the risk of introducing errors. We are searching for an elegant model for expressing how and what to compute, and for ways to integrate this with the language. This is our biggest research challenge.

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. More work needs to be done in order to find a good way to produce efficient code for a larger set of combinators. This may very well involve designing new combinators that do not need to be nested as the current ones do.

The CUDA code generated by Obsidian uses very few registers. In fact, the generated code is under-utilizing the resources of the GPU. The generated code also contains many common subexpressions. This hints that applying a step of common subexpression elimintaion would be beneficial, increasing the register use and reducing the repeated computation of values. Performing common subexpression elimination would also lower the instruction count, which is quite high to begin with, because loops are unrolled. The register usage and instruction count values on which these conclusions are based where obtained using the CUDA profiler supplied as part of the CUDA toolkit [17].

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

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

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

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 [1]. CUDA supplies a slightly extended version of C in which the programmer can specify GPU kernels and the controlling CPU program in the same language. In CUDA the programmer writes a single program parameterised over a thread identity. Compare this to Obsidian where the program describes a computation over an array.

There are a number of other C/C++ based languages that target GPGPU programmers, for example Brook[18] and RapidMind[19]. Brook, CUDA and RapidMind are major improvements from what was previously available for the programmer interested in general purpose computations on the GPU.

Higher level approaches are also being investigated. PyGPU embedds a GPGPU programming language in Python[20]. PyGPU makes use of Python's introspective abilities to generate efficient code.

Like Obsidian, Accelerate is embedded in Haskell [21]. Accelerate however, is higher level language than Obsidian. Where the purpose of Obsidian is to implement basic algorithmic building blocks such as reductions and prefix sums, Accelerate provide these building blocks as primitives.

Vertigo is another GPU programming language embedded in Haskell[22]. Vertigo can be used to describe procedureal surfaces and textures and generates efficient DirectX9 shader code.

There are also many examples of domain specific languages (DSLs) that target not only GPUs but also other platforms, such as multicore machines. Of particular interest to us is Microsoft's Accelerator project [23]. Interestingly, Accelerator includes the same restriction to "hardware-like" algorithms as Obsidian does.

7. Discussion and 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 and to broaden the available set of combinators for combining GPU programs.

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

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

Acknowledgements

This research is funded by the Swedish Research Council.

References

[1] NVIDIA, NVIDIA CUDA Programming Guide 2.0 (2008).
URL `http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf`

[2] NVIDIA, Technical brief: Nvidia geforce 8800 gpu architecture overview.
URL `http://www.nvidia.com/page/8800_tech_briefs.html`

[3] J. Svensson, M. Sheeran, K. 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, 2009.

[4] J. Hughes, Generalising monads to arrows, Sci. Comput. Program. 37 (1-3) (2000) 67–111.

[5] J. Svensson, K. Claessen, M. Sheeran, GPGPU Kernel Implementation using an Embedded Language: a Status Report, dept. of Computer Science and Engineering, Chalmers, Tech. Report Number 2010:1 (2010).
URL `http://www.cse.chalmers.se/~joels/writing/GPGPU_tech.pdf`

[6] C. Elliott, S. Finne, O. de Moor, Compiling Embedded Languages, Journal of Functional Programming 13 (2).

[7] M. Pharr, R. Fernando, GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation, Addison-Wesley Professional, 2005.

[8] NVIDIA CUDA Best Practices Guide.
URL `http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide_2.3.pdf`

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

[10] H. Nguyen, GPU Gems 3: 3D and General Programming Techniques for GPUs, Addison-Wesley Professional, 2007.

[11] E. Sintorn, U. Assarsson, Fast parallel GPU-sorting using a hybrid algorithm, J. Parallel Distrib. Comput. 68 (10) (2008) 1381–1388.

[12] N. Satish, M. Harris, M. Garland, Designing efficient sorting algorithms for manycore GPUs, in: Proc. IEEE Int. Symp. on Parallel & Distributed Processing, 2009, pp. 1–10.

[13] K. Claessen, M. Sheeran, S. Singh, The design and verification of a sorter core, in: Proc. of Conference on Correct Hardware Design an Verification Methods (CHARME), Lecture Notes in Computer Science, Springer Verlag, 2001.

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

[15] G. E. Blelloch, Prefix sums and their applications, Tech. Rep. CMU-CS-90-190, School of Computer Science, Carnegie Mellon University (Nov. 1990).

[16] M. Harris, S. Sengupta, J. D. Owens, Parallel prefix sum (scan) with cuda, in: H. Nguyen (Ed.), GPU Gems 3, Addison Wesley, 2007.

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

[18] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, P. Hanrahan, Brook for GPUs: stream computing on graphics hardware, ACM Trans. Graph. 23 (3) (2004) 777–786.

[19] M. D. McCool, Data-parallel programming on the Cell BE and the GPU using the RapidMind Development Platform, presented at the GSPx Multicore Applications Conference (2006).

[20] C. Lejdfors, L. Ohlsson, Implementing an embedded GPU language by combining translation and generation, in: SAC '06: Proceedings of the 2006 ACM symposium on Applied computing, ACM, New York, NY, USA, 2006, pp. 1610–1614.

[21] S. Lee, M. M. Chakravarty, V. Grover, G. Keller, GPU Kernels as Data-Parallel Array Computations in Haskell, in: Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods (EPHAM), 2009.

[22] C. Elliott, Programming graphics processors functionally, in: Proceedings of the 2004 Haskell Workshop, ACM Press, 2004.

[23] D. Tarditi, S. Puri, J. Oglesby, Accelerator: using data parallelism to program GPUs for general-purpose uses, in: Proc. 12th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), ACM, 2006, pp. 325–335.