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

Joel Svensson, Mary Sheeran, and Koen Claessen

Chalmers University of Technology, Göteborg, Sweden

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

1 Introduction

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

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

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

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

2 Connection patterns for hardware design and parallel programming

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

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

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

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

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

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

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

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

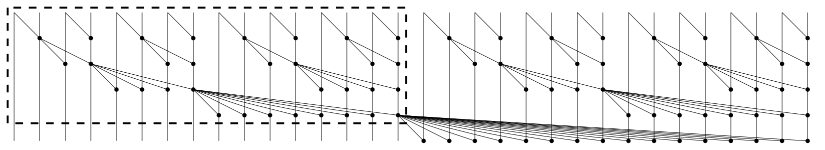

Fig. 1. The Sklansky construction for 32 inputs. It recursively computes the parallel prefix for each half of the inputs (corresponding to the use of `two` in the definition) and then combines the last output of the lower (left) half with each of the outputs of the upper (right) half. The dotted box outlines the recursive call on the lower half of the inputs.

In this work, we plan to investigate the use of connection patterns, and more generally an emphasis on *structure*, in parallel programming. We have chosen to target GPUs partly because of available expertise among our colleagues at Chalmers, and partly because reading papers about General Purpose GPU (GPGPU) programming gave us a sense of déjà vu. Programs are illustrated graphically, and bear a remarkable resemblance to circuit modules that we have generated in the past using Lava. We see an opportunity here, as there is an extensive literature, going back to the 1960s, about implementing algorithms on silicon that may provide clues about implementing algorithms on GPUs. This literature does not seem to have yet been scrutinised by the Data Parallel Programming or GPGPU communities. This is possibly because GPUs are moving closer to simply being data parallel machines, and so work on library functions has taken inspiration from earlier work on Data Parallel Programming, such as Blelloch's NESL [**?**]. But some of the restrictions from the early data parallel machines no longer hold today; for instance broadcasting a value to many processors was expensive in the past, but is much easier to do on modern GPUs. So a construction like Sklansky, which requires such broadcasting, should now be reconsidered, and indeed we have found it to give good results in our initial experiments (writing directly in CUDA). In general, it makes sense to spread the net beyond the standard data parallel programming literature when looking for inspiration in parallel algorithm design. We plan to explore the use of "old" circuit design ideas in programming library functions for GPUs.

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

3 Graphics Processing Units, accessible high performance parallel computing

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

See reference [**?**] for information about the very similar AMD GPU architecture. We have used the NVIDIA architecture, but developments are similar at AMD. Intel's Larrabee processor points to a future in which each individual core is considerably more powerful than in today's GPUs [**?**].

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

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

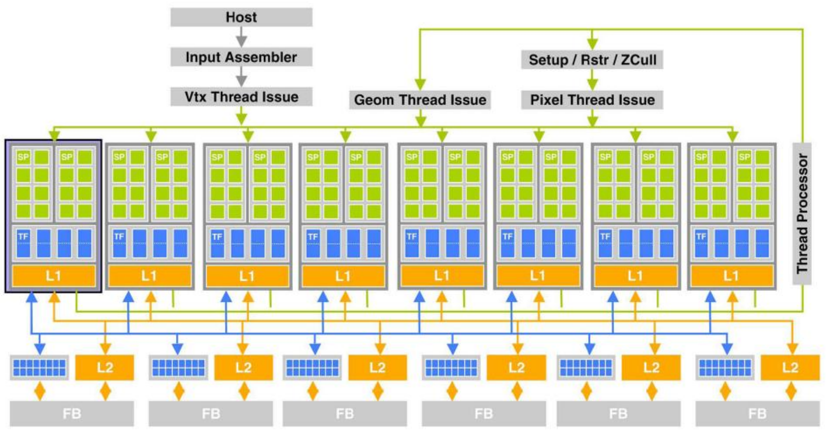

Fig. 2. The NVIDIA 8800GTX GPU architecture, with 8 pairs of multiprocessors. Diagram courtesy of NVIDIA.

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

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

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

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

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

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

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

4 Obsidian: a domain specific embedded language for GPU programming

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

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

The most notable differences between the two implementations are that Obsidian functions are monadic and that a datatype `Arr` is used where Haskell lists are used in Lava. The pattern matching on the list used in Lava is here replaced by guards. The function `len :: Arr s a -> Int` gives the length of the array. These differences lead to a slightly different programming style.

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

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

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

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

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

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

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

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

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

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

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

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

4.1 Implementation

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

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

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

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

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

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

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

```
type Exp a = E DExp

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

As an example consider the program:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

type IC = [Statement]
```

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

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

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

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

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

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

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

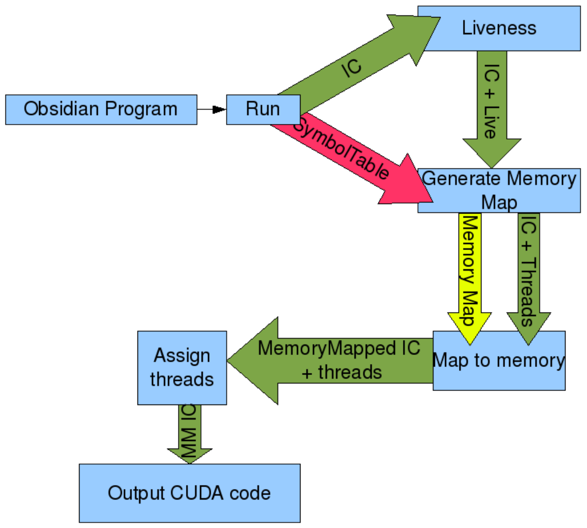

Fig. 3. Steps involved in generating CUDA C code from Obsidian description. The boxes represent functions and the arrows represent data structures.

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

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

Now, the symbol table together with the `ICLive` object is used to lay out the arrays in memory. Arrays that had type `SArr` are assigned storage in the shared memory and arrays of type `GArr` in Global memory. The result of this stage is a *Memory Map*. This is a mapping from names to positions in memory. The picture also shows that another output from this stage is intermediate code annotated with thread information, call it `ICT`. This is now done in a separate pass over the IC but it could be fused with the memory mapping stage, saving a pass over the IC. The ICT is just a list of statements and the number of threads assigned to compute them:

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

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

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

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

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

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

Fig. 4. Generated CUDA code.

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

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

Fig. 5. A small part of the CUDA code generated from the recursive implementation of the Sklansky parallel prefix algorithm.

5 Results

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

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

6 Discussion

6.1 Our influences

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

6.2 Related work on GPU and GPGPU programming languages

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

The more general purpose languages aim to abstract away from the graphics heritage of GPUs, and target a larger group of programmers. The thesis by Lucas presents CGiS, an imperative data-parallel programming language that targets both GPUs and SIMD capable CPUs – with the aim being a combination of a high degree of abstraction and a close resemblance to traditional programming languages [?]. BrookGPU (which is usually just called Brook) is a classic example of a language [?] designed to raise the level of abstraction at which GPGPU programming is done. It is an extension of C with embedded kernels, aimed at arithmetic-intense data parallel computations. C is used to declare streams[3], CG/HLSL (the lower level GPU languages) to declare kernels, while function calls to a runtime library direct the execution of the program. Brook had significant impact in that it raised the level of abstraction at which GPGPU programming can be done. The language Sh also aimed to raise the level of abstraction at which GPUs were programmed [?]. Sh was an embedded language in C++, so our work is close in spirit to it. Sh has since evolved into the

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

RapidMind development platform [?], which now supports multicores and Cell processors as well as GPUs. The RapidMind programming model has arrays as first class types. It has been influenced by functional languages like NESL and SETL, and its program objects are pure functions. Thus it supports both functional and imperative styles of programming. A recent PhD thesis by Jansen asserts that there are some problems with RapidMind's use of macros to embed the GPU programming language in C++, including the inability to pass kernels (or shader programs) as classes [?]; the thesis proposes GPU++ and claims improvement over previous approaches, particularly through the exploitation of automatic partitioning of the programs onto the available GPU hardware, and through compiler optimisations that improve runtime performance.

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

6.3 Lessons learned so far in the project

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

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

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

7 Future work

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

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

8 Conclusion

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