

A Language for Hierarchical Data Parallel Design-space Exploration on GPUs

BO JOEL SVENSSON

Indiana University

and

MARY SHEERAN

Chalmers University of Technology

and

RYAN NEWTON

Indiana University

Abstract

Graphics Processing Units (GPUs) offer potential for very high performance; they are also rapidly evolving. Obsidian is an embedded language (in Haskell) for implementing high performance kernels to be run on GPUs. We would like to have our cake and eat it too; we want to raise the level of abstraction beyond CUDA code and still give the programmer control over the details relevant to kernel performance.

To that end, Obsidian provides array representations that guarantee elimination of intermediate arrays while also using the type system to model the hierarchy of the GPU. Operations are compiled very differently depending on what level of the GPU they target, and as a result the user is gently constrained to write code that matches the capabilities of the GPU. Thus, we implement not Nested Data Parallelism but a more limited form that we call Hierarchical Data Parallelism.

We walk through case-studies that demonstrate how to use Obsidian for rapid design exploration or auto-tuning, resulting in performance that compares well to the hand-tuned kernels used in Accelerate and NVIDIA Thrust.

1 Introduction

Graphics Processing Units (GPUs) offer the potential for high-performance implementations of data parallel computations. Yet achieving top performance is recognised as a difficult task, requiring expert programmers with the ability and time to manually optimize use of on-chip storage, make granularity decisions, and match memory access patterns to the non-traditional constraints placed by GPU memory architectures. Accordingly, programs are written in low-level vendor-supplied programming environments, such as NVIDIA CUDA, where all these details are under programmer control.

One answer to the high cost of GPU programming is to attempt to *automate* the process, in particular by starting with a very high-level language and using an optimizing compiler to make the aforementioned decisions, synthesizing code in a language like CUDA. Indeed, many recent research projects have done just this, including: Copperhead (Catanzaro *et al.*, 2011), Accelerate (Chakravarty *et al.*, 2011; McDonnell *et al.*, 2013), Harlan (Holk

et al., 2012), and Delite (Chafi *et al.*, 2011). These languages are first and foremost *array languages*. Typical operations include mapping, filtering, scanning, and reducing array data. By restricting program structure, this language family gains at least one major benefit over more general purpose array languages: they can very effectively fuse series of array operations, eliminating temporary arrays. Decisions about what to fuse, and how to use local on-chip storage on the GPU, are completely automated by the DSL compiler and runtime.

Such automation certainly has considerable appeal. Many users would like to gain the benefits of GPU acceleration without studying the details of GPUs and GPU programming idioms. These are the users served by Accelerate, for example. Prefix sum becomes `scanl (+) 0 arr` without tuning parameters, and that is that. However, it is widely accepted that successful acceleration on GPUs often demands fine control of many details that are closely related to the GPU architecture, and even experts employ a *design exploration* process, experimenting with tradeoffs and making major changes to an initial version. For example, in reference (Harris, 2007) detailed optimisation efforts of reduction in CUDA is worked through step by step. The final reduction kernel is 30x faster than the naive CUDA implementation used as starting point. Our work on Obsidian tries to answer the question of whether or not the benefits of functional programming can be brought to the group of users of GPUs who wish to explore a variety of possible designs in the search for high performance.

The big question then is what forms of abstraction to provide. The intrepid GPU programmer needs to be able to control many details, including the arrangement of computations into threads, warps, blocks and grids, the number of kernels launched, the use of local memory on the GPU, synchronization points, memory access patterns and much more. The danger is that the user simply ends up writing CUDA in Haskell syntax; we would particularly like to avoid tedious index calculations. Our main approach to easing the job of the programmer while still providing fine control is the provision of *compositional* array operations that also offer *hierarchy polymorphism*. Obsidian uses a combination of *push* and *pull arrays*, in the meta-language (Section 5). It uses a *fusion by default* approach, even at the expense of work duplication, together with an explicit function for making arrays manifest in memory. In addition, Obsidian exposes the hierarchical nature of GPU hardware directly in the type system. Core operations work at any level (thread, warp, block, grid) but how they are compiled will vary greatly between levels. This use of the type system to model the GPU hierarchy allows us to implement not full Nested Data Parallelism (as for example in NESL and its successors (Blelloch, 1996)) but rather a limited form of hierarchy that is perfectly matched to the capabilities of the GPU. This is probably the most novel aspect of Obsidian.

Obsidian also eases the job of the programmer in other ways. GPUs have some hard-coded limits on aspects of programs such as the maximum number of threads allowed in a block, or the size of a warp. These constant limits do not apply to Obsidian programs, but rather there is *virtualization* of threads, warps and blocks, with the generated CUDA code obeying the limits. Also, the ease with which functional programs can be parameterised gives us a straightforward approach to the systematic generation and measurement of code variants, with the result that design exploration is very much easier than it is in CUDA. Obsidian also helps the programmer by automating the process of laying out intermediate

arrays in memory. This memory layout system takes care of liveness of arrays and reuse of space in shared memory and is done statically.

Obsidian has been under development for quite some time, and we have explored a variety of APIs. The version presented here has proved to work very well in developing key kernels for library operations such as reduction and prefix scan. The resulting kernels perform well—on a par with those in NVIDIA’s Thrust Library (NVIDIA, 2014c). We hope that readers will feel inclined to experiment with the design and implementation of such kernels¹.

2 Background: The GPU and CUDA

Obsidian targets NVIDIA GPUs supporting CUDA (NVIDIA, 2014a), a C-dialect for data-parallel programming. These GPUs are built on a scalable architecture: each GPU consists of a number of *multiprocessors*; each multiprocessor has a number of processing elements (cores) and an on-chip local memory that is shared between threads running on the cores. A GPU can come with as few as one such multiprocessor. The GPUs used in our measurements are an NVIDIA Tesla c2070 and a GTX680. The GTX680 GPU has eight multiprocessors, with a total of 1536 processing cores. On these cores, groups of 32 threads called *warps* are scheduled. There are a number of warp scheduling units per multiprocessor. Within a warp, threads execute in lockstep (SIMD); diverging branches, that is those that take different paths on different threads within a warp, are serialised, leading to performance penalties.

The scalable architecture design also influences the programming model. CUDA programs must be able to run on all GPUs from the smallest to the largest. Hence a CUDA program must work for any number of multiprocessors. The CUDA programming model exposes abstractions that fit the underlying architecture; there are *threads* (executing on the cores), *blocks* of threads (groups of threads run by a multiprocessor) and finally the collection of all blocks, which is called the *grid*.

The threads within a block can use the shared memory of the multiprocessor to communicate with each other. A synchronisation primitive, `__syncthreads()`, gives all the threads within a block a coherent view of the shared memory. There is no similar synchronisation primitive between threads of different blocks.

The prototypical CUDA kernel starts out by loading data from global memory. The indices into global memory for an individual thread are expressed in terms of the unique identifier for that block and thread. Some access patterns allow memory reads to be *coalesced*, while others do not, giving very poor performance. The patterns that lead to good performance vary somewhat between different GPU generations, but regular, consecutive accesses by consecutive threads within a warp are best. Accesses to local, shared, also have the property that certain patterns are more efficient. The shared memory is divided into banks and it is most efficient if the threads of a warp access data element residing in different banks. On current GPUs there are either 16 or 32 banks.

A CUDA program is expressed at two levels. Kernels are data-parallel programs that run on the GPU. They are launched by the controlling program, which runs on the CPU

¹ Download Obsidian at [github. www.github.com/svenssonjoel/Obsidian](http://github.com/svenssonjoel/Obsidian)

of the host machine. Obsidian is primarily a language for engineering efficient kernels, but, like other GPU DSLs, it also provides library functions for transparently generating, compiling, and invoking CUDA kernels from the high-level language in which Obsidian is implemented (Haskell). Unlike most GPU DSLs, Obsidian can also be used to generate standalone kernels, which can be called from regular CUDA or C++ programs—a common need when accelerating existing applications.

3 Introductory Obsidian Example: Reductions

Section 7 evaluates the performance of a series of reduction kernels. Here, we begin with a simple, concrete example of how to write and deploy a reduction kernel, saving Obsidian implementation details for later. The reduction code below works for arrays that have a length that is a power of two. We assume that the operator being reduced (or folded) is both associative and commutative. The function we define below, `reduce`, splits the array in the middle and then uses `zipWith` to apply the reduction operator to pairs of elements. It then proceeds to recursively reduce the resulting array.

```
reduce :: (Compute t, Data a)
  => (a -> a -> a)
  -> SPull a
  -> Program t (SPush t a)
reduce f arr
  | len arr == 1 = return $ push arr
  | otherwise   =
    do let (a1,a2) = halve arr
         arr' ← compute $ push $ zipWith f a1 a2
         reduce f arr'
```

In Obsidian one writes functions corresponding to kernels and a separate program that runs one or more such kernels in parallel over the GPU. This mirrors how one works in CUDA, and is different from (say) having parts of ones' Haskell (or Python/Java/ML...) program transparently implemented on the GPU. The program that distributes multiple instances of the reduction kernel is the `reductions` function below. It splits an input array into chunks, here of 512 elements, and performs the `reduce` kernel on each of those chunks.

```
reductions :: Data a
  => (a -> a -> a)
  -> DPull a -> DPush Grid a
reductions f arr = asGridMap body (splitUp 512 arr)
  where body a = execBlock (reduce f a)
```

Launching the `reductions` program on the GPU is done as follows.

```

perform :: IO ()
perform =
  withCUDA $ do
    kern ← capture 64 (reductions (+))

    useVector (V.fromList [0..1023 :: Int32]) $ λi →
      allocaVector 2 $ λo →
        do o <== (1,kern) <> i
           r ← peekCUDAVector o
           lift $ putStrLn $ show r

```

The first step is to capture the kernel; the Obsidian code is compiled into CUDA and the NVIDIA CUDA compiler `nvcc` is applied to it. The result `kern` is a handle to the compiled and linked function. At `capture`, the number of threads to use per CUDA Block is specified, in this case 64. The CUDA code that the `kern` handle is referring to has been compiled for the specific array size and number of threads specified. In this case `kern` operates on subarrays of length 512. Arrays in GPU memory are created to store the input and output of the GPU computation, `useVector` and `allocaVector` (taking as argument the number of elements to allocate space for). Then the kernel is launched using the `<==` operator; the number 1 specifies the number of blocks to use. Even though here only one (real) block is used, the kernel is run twice in sequence within that block. This is an example of block virtualization.

Running the `perform` function prints the result:

```
[130816,392960]
```

The `withCUDA` monad and its associated `capture` and `<==` functions, gives the programmer a way of running programs on the GPU directly from within a Haskell program. When doing so CUDA code will be generated into files called `gen0`, `gen1` and so on, located in the working directory. These files are not deleted after being compiled and linked into the running executable, but rather left for inspection by the programmer.

4 Obsidian Programming Model: Overview and Discussion

Obsidian is an Embedded Domain Specific Language (EDSL), implemented in Haskell. When running an Obsidian program a data structure is generated encoding an abstract syntax tree (AST). Embedded languages that generate ASTs are traditionally called *deeply embedded* languages. A shallow embedding, on the other hand, implements the DSL semantics directly at the point of each call into the EDSL API. A combination of the two approaches is often used, as one aims to find a sweet spot that combines the advantages of the two approaches, while avoiding the disadvantages of each (Svenningsson & Axelsson, 2013). This is what Obsidian does, as we shall see later. The AST is used for CUDA code generation, but our array representations have disappeared by the time we get to the AST. For an excellent introduction to compiling embedded languages, see reference (Elliott *et al.*, 2003).

Obsidian has two main parts, an array language with two main immutable array representations, *pull* and *push* arrays, and combinators for laying out computations onto a GPU.

4.1 Expressions

Obsidian’s target language includes expressions operating on scalar data; this language is captured by the `(Exp a)` GADT. For the types supported by Obsidian there are shorthands:

```

type EInt    = Exp Int
type EWord  = Exp Word
type EInt8   = Exp Int8
...
type EInt64 = Exp Int64
type EWord8 = Exp Word8
...
type EWord64 = Exp Word64
type EFloat = Exp Float
type EDouble = Exp Double
type EBool   = Exp Bool

```

Tuples are also supported but have no representation in the `Exp` data type; rather normal Haskell tuples are used.

Amongst the operations available on `Exp` expressions are arithmetic and bitwise operations exposed via instances of `Num` and `Bits`. There are also conditionals and boolean operations. Conditionals are expressed using an `ifThenElse` function:

```

class Choice a where
  ifThenElse :: Exp Bool → a → a → a

```

Boolean operators look like the traditional Haskell operators on `Bool`, but with a `*` appended to the operator name (a convention shared by several Haskell EDSLs). For example the equality operator resulting in an `EBool` is `(==*)`.

Restrictions apply on the elements used in an Obsidian Program. This takes the form of a constraint `Data a` on values of type `a`, which we saw in the code examples of Section 3. The `Data` class is the aggregation of the `Choice` and `Storable` classes.

```

class (Storable a, Choice a) ⇒ Data a

```

`Storable` should not be confused with the standard Haskell class for data that can be written into memory as raw bits; it is an Obsidian class that implements storing of data elements into shared memory. There are instances of `Storable` for base types and tuples up to a certain size. Note that there are *no* instances of `Data` for array types. Currently Obsidian only has support for storing flat arrays into memory.

4.2 Pull Arrays

A pull array is implemented as a length and a function from an expression representing an index to a value. This is a very standard approach to implementing arrays in embedded DSLs (the same representation is used in *Feldspar* (Axelsson *et al.*, 2011) and *Repa* (Keller *et al.*, 2010)), as it gives *fusion/deforestation by default*. We first came across it in Elliott’s work on *Pan* (Elliott, 2003), but similar ideas appeared much earlier, for example in the compilation of *APL* (Guibas & Wyatt, 1978).

The consumer of a pull array must apply the pull array function to each index of interest.

```
data Pull s a = Pull s (EWord32 → a)
```

The `s` parameter to `Pull` is the type of the length of the array. This type can be either `Word32` or `EWord32`. `Word32`, which we refer to as a static length, is used in most cases and is required, for example, when the array is stored into memory. The `EWord32` type, or dynamic length, is useful in some cases where the length does not need to be exactly known. Obsidian programs executed at the block level or below need a static size for any array computed *in parallel*. When composing fixed-size sub-computations into a grid, the number of such sub-computations can be dynamic—this is where dynamic sizes come in. For each type that is acceptable as an array length there is an instance of class `ASize` that supplies a single function, `sizeConv`, that enables conversion to a format used by Obsidian internally (Currently `Exp Word32`).

In code examples, we use shorthands for pull arrays of static and dynamic length:

```
type SPull a = Pull Word32 a
type DPull a = Pull EWord32 a
```

4.3 Programs

The `Program t` data type represents parallel and sequential computations on the GPU. A `Program` is parameterised on the level of the GPU on which it is to be executed; thus the `t` parameter can have one of the following types: `Thread`, `Warp`, `Block` or `Grid`. The `Program` type represents low level imperative programs and contains functionality for assigning to memory, allocating memory and iterating in sequence or parallel. Figure 3 lists some low-level functions related to the `Program` data type. Any Obsidian program that uses parallelism or shared memory thus involves this `Program` data type. For example, the function `compute` takes an array (a delayed, pull or push, array), computes all values and writes them to shared memory. The result of `compute` is always a pull array that reads values from the newly created array in shared memory:

```
class Compute t ⇒ ComputeAs t a where
  compute :: Data e ⇒ a Word32 e → Program t (Pull Word32 e)
```

The `compute` function has return type `Program t` that encodes an iteration schema over its input array and writes all elements to the *manifest* array it creates in memory. The `t` parameter is restricted by the `Compute t` constraint. There is an instance of `Compute` for `Thread`, `Warp` and `Block` since at these levels of the GPU hierarchy, shared memory can be used. There is no instance for `Grid`.

4.4 Push Arrays

Now enough Obsidian details have been explained to introduce push arrays. A push array has a length and a function, the *push-function*. The job of the push function is to generate a control structure that generates *all* elements of the array, pushing them one at a time to a *writer* function. Thus the push-function is itself higher order:

```

-- | reverse a pull or push array
reverse :: (Array array, ArrayLength array, ASize l)
        => array l a -> array l a
-- | Split a pull array at a given position
splitAt :: (Integral i, ASize l)
        => i -> Pull l a -> (Pull l a, Pull l a)
-- | Split a pull array in the middle
halve :: ASize l => Pull l a -> (Pull l a, Pull l a)
-- | Split an array up into chunks of a given size
splitUp :: (ASize l, ASize s, Integral s)
        => s -> Pull l a -> Pull l (Pull s a)
-- | Singleton pull or push array
singleton :: (Array a, ASize l) => e -> a l e
-- | Generate a pull or push array
generate :: (Functor (a s), Array a, ASize s)
        => s -> (EWord32 -> b) -> a s b
-- | Extract the last element from a pull array
last :: ASize l => Pull l a -> a
-- | Extract the first element from a pull array
first :: ASize l => Pull l a -> a
-- | like Prelude.take
take :: ASize l => l -> Pull l a -> Pull l a
-- | like Prelude.drop
drop :: ASize l => l -> Pull l a -> Pull l a
-- | Array of pairs to pair of arrays
unzip :: ASize l => Pull l (a,b) -> (Pull l a, Pull l b)
-- | Two arrays to an array of pairs
zip :: ASize l => Pull l a -> Pull l b -> Pull l (a, b)
-- | pair up the elements of an array
pair :: ASize l => Pull l a -> Pull l (a,a)
-- | flatten and array of pairs
unpair :: ASize l => Choice a => Pull l (a,a) -> Pull l a
-- | splits an array in the middle recursively n times
unsafeBinSplit :: Int
                -> (Pull Word32 a -> Pull Word32 b)
                -> Pull Word32 a
                -> Pull Word32 b
-- | Create a pull array
mkPull :: s -> (EWord32 -> a) -> Pull s a
-- | Create a push array
mkPush :: s
        -> ((a -> EWord32 -> Program Thread ()) -> Program t ())
        -> Push t s a

-- | Instances
instance Functor (Push t s) where
instance Functor (Pull s) where

```

Fig. 1. A selection of functions on pull arrays

```

class Array a where
  -- | Array of consecutive integers
  iota      :: ASize s => s -> a s EWord32
  -- | Create an array by replicating an element.
  replicate :: ASize s => s -> e -> a s e
  -- | Map a function over an array.
  aMap      :: (e -> e') -> a s e -> a s e'
  -- | Perform arbitrary permutations (dangerous).
  ixMap     :: (EWord32 -> EWord32) -> a s e -> a s e
  -- Requires Choice since the pull array implementation of it does.
  -- | Append two arrays.
  append    :: (ASize s, Choice e) => a s e -> a s e -> a s e
  -- | Statically sized array to dynamically sized array.
  toDyn     :: a Word32 e -> a EW32 e
  -- | Dynamically sized array to statically sized array.
  fromDyn   :: Word32 -> a EW32 e -> a Word32 e

```

Fig. 2. The Array class contains functionality that is shared between pull and push arrays

```

-- | A parallel forAll loop at level t
forall :: (t *<=* Block) => EWord32
        -> (EWord32 -> Program Thread ())
        -> Program t ()
-- | A sequential forAll loop at level t
seqFor :: EWord32 -> (EWord32 -> Program t ()) -> Program t ()
-- | Use a single thread of those available
-- | at level t to execute a program.
singleThread :: Program Thread () -> Program t ()
-- | Perform atomic op (Atomic a)
atomicOp :: Scalar a
          => Name          -- Array name
          -> Exp Word32   -- Index to operate on
          -> Atomic a     -- Atomic operation to perform
          -> Program Thread ()

-- Instances for Program t
instance Monad (Program t)
instance Functor (Program t)
instance Applicative (Program t)

```

Fig. 3. Low-level functions on programs. Most of the time, we expect Obsidian programmers not to need to use such low level functions. However, they are available for when very fine control is desired. In reference (Svenningsson *et al.*, 2013), very low-level programming in Obsidian is illustrated.

```

data Push t s a = Push s (PushFun t a)
type PushFun t a = Writer a -> Program t ()
type Writer a = a -> EWord32 -> Program Thread ()

```

A consumer of a push array needs to apply the push-function to a suitable writer. Commonly, the push-function is applied to a writer that stores its input value at the provided

input index into memory. This is what the `compute` function does when applied to a push array.

The function `push` converts a pull array to a push array:

```
push :: (t *<= Block, ASize s) => Pull s e -> Push t s e
push (Pull n ixf) =
  mkPush n $ \wf ->
    forAll (sizeConv n) $ \i -> wf (ixf i) i
```

This function sets up an iteration schema over the elements as a `forAll` loop. It is not until the `t` parameter is fixed in the hierarchy that it is decided exactly how that loop is to be executed. All iterations of the `forAll` loop are independent, so it is open for computation in series or in parallel.

The `t *<= Block` constraint is there to rule out conversion of a pull array to a push array at the Grid level. Allowing this conversion would mean that the decisions of numbers of block and numbers of elements to process per block would have to be taken automatically. In general the `*<=` type operator is used to restrict functions to operating up to a certain level.

A selection of functions on push arrays can be found in Figures 1 and 2.

Finally, some functions can be implemented on *both* pull arrays and on push arrays. This shared functionality is captured by an `Array` class (shown in Figure 2). There is also a class called `ArrayLength` with instances for all arrays that allow a `len` function, yielding the array's length. More array functions are shown in Figure 1, and some of these apply only to pull arrays.

4.5 Pull and Push Arrays: Important Differences

Neither pull nor push arrays represent data in memory; rather they represent two different ways of computing array elements. Pull arrays support efficient indexing, in that any element can be computed and accessed independently. A push array, on the other hand, encodes its own iteration schema. A consumer is forced to use the push array's built-in iteration pattern, and accessing any one element requires first computing the entire array.

The reason for having these two array representations is their complementary strengths and weaknesses. These properties of pull and push arrays are summarised in the table below.

Property	Pull	Push
Fusion	Yes	Yes
Parallel	Yes	Yes
Efficient indexing	Yes	No
Efficient concat	No	Yes
Efficient interleave	No	Yes

Operations on both pull and push arrays fuse by default. The classical example of this is that $(\text{map } f) \circ (\text{map } g)$ does not require an intermediate array stored in memory. Moreover,

conversion from pull to push array does not require intermediate storage either. Conversion in the other direction, push to pull, *does* require storage into memory. Push to pull array conversion is done using the `compute` function; this is the programmer's way to choose when *not* to fuse. Thus if an array resulting from an expensive computation is used in more than one place, using `compute` on that array ensures that the expensive computation takes place only once.

Both pull and push arrays allow for parallelism. In the case of push arrays, the iteration schema, parallel or sequential, is encoded in the array. When a push array is computed, that schema is executed yielding the array's elements. A pull array does not come with such a predetermined computation schema. When `compute` is used on a pull array, a schema determined by the hierarchy parameter, τ , decides how it is computed. For example, if τ is `Thread`, a sequential loop is instantiated (with trip count equal to array length); whereas, if τ is `Block`, a parallel loop, using the threads at the `Block` level, is created. Note that the array may be *longer* than the actual number of threads available at a particular level, meaning that computing an array must use *virtual threads*, multiplexed onto available physical threads. Currently, these virtual threads are implemented by wrapping an extra sequential loop around the parallel computation; thus large arrays are computed chunk by chunk. Section 5.1.3 covers code generation details.

Again, when accessing index i in a pull array no other element of that array need be touched. The cost of indexing into a pull array could, however, be entirely arbitrary, because the pull array represents a delayed computation at each element. Such a delayed computation could, for example, touch every element of some other array. Only when a pull array is the direct result of a `compute` is it guaranteed to be a traditional, $O(1)$, access to shared memory. Likewise accessing arrays that are *inputs* to the Obsidian program has the cost of a *global* memory read (a constant cost, but a significantly higher one). By contrast, a push array does not allow for efficient random access, rather it must be converted into a pull array with `compute`.

Concatenation and interleaving can be implemented on both pull and push arrays. On pull arrays, however, these functions are implemented using a conditional. The code for concatenating two pull arrays is shown below, this function is a member of the `Array` class (figure 2) and has the same name for both pull and push arrays:

```
append a1 a2 = mkPull (n1+n2)
               $ \ix →ifThenElse (ix < * sizeConv n1)
                           (a1 ! ix)
                           (a2 ! (ix - sizeConv n1))

where
  n1 = len a1
  n2 = len a2
```

When computing a pull array that is the result of an `append` this leads to a loop (parallel or sequential) that for each iteration executes a conditional.

```
for i in 0..(n1 + n2 - 1)
  data[i] := if (i < n1)
             then ...
             else ...
```

The case for interleaving pull arrays is worse still. There the conditional would take different paths for even and odd iterations. If executed in parallel on the GPU this creates *thread divergence* where half of the threads of each warp will be turned off at each point in time, wasting half of the GPU's arithmetic units.

A better approach is often to do away with the conditional and instead execute *two* loops. Indeed, we use this approach to concatenate and interleave push arrays:

```
append p1 p2 =
  mkPush (n1 + n2) $ \wf →
    do p1 <: wf
      p2 <: λa i → wf a (sizeConv n1 + i)
    where
      n1 = len p1
      n2 = len p2
```

Here the function `<:` applies a push array's push-function to a writer function. When using `compute` on a push array that is the result of `append` the generated code has the following form.

```
for i in 0..(n1-1)
  data[i] := ...
for i in 0..(n2-1)
  data[i+n1] := ...
```

These differences between pull and push arrays are the motivation for having both representations in Obsidian.

4.6 Compute and Parallelism

The function `compute` is used for storing intermediate arrays into memory. It is also important when it comes to expressing parallelism. Take the following function, for example, which sums up the elements of an array (like the reduction example earlier).

```
sumUp :: Pull Word32 EWord32 → EWord32
sumUp arr
  | len arr == 1 = arr ! 0
  | otherwise    =
    let (a1,a2) = halve arr
        arr'    = zipWith (+) a1 a2
    in sumUp arr'
```

This function halves the input array and performs element wise addition between the halves. Then it recurses and proceeds until there is only one element. This code implements sequential summation of an array as there is nothing in the function that realises the potential parallelism. One could imagine that `zipWith` would have a parallel implementation, but that is *not* the case here. `zipWith` just takes two pull arrays and produces a new one.

The code generated from the function above would have the following appearance.

```
output[0] = input[0] + input[4] +
           input[2] + input[6] +
           input[1] + input[5] +
           input[3] + input[7];
```

The code above is parallelised by inserting a `compute` operation. This changes the type of the function and also involves using `do` notation.

```
sumUp' :: Pull Word32 EWord32 → Program Block EWord32
sumUp' arr
  | len arr == 1 = return (arr ! 0)
  | otherwise    = do
    let (a1,a2) = halve arr
        arr' ← compute (zipWith (+) a1 a2)
    sumUp' arr'
```

In this function the result of the `zipWith` is computed and stored into shared memory, using `compute`. This computation of the pull array is performed using the threads of the block level in the hierarchy, as that is the hierarchy level of the resulting program. Thus, the above program becomes implicitly parallel in a *type-directed* manner. The code generated from this function will consist of a series of parallel stages:

```
parfor (i in 0 .. 3)
  imm0[i] = input[i] + input[i+4];
parfor (i in 0 .. 1)
  imm1[i] = imm0[i] + imm0[i+2];
parfor (i in 0 .. 0)
  output[i] = imm1[i] + imm1[i+1];
```

4.7 Programming the Hierarchy

As we've glimpsed so far, in Obsidian, the programmer is in control of how to lay out computations onto the GPU hierarchy. The hierarchy consists of four levels `Thread`, `Warp`, `Block` and `Grid`. There are limitations, imposed by the GPU hardware, on what can be done at the various levels. These limitations are summarised in the following table.

Level	Parallelism	Shared Memory	Thread synchronisation
Thread	No	Yes	No
Warp	Yes	Yes	Lock-step execution
Block	Yes	Yes	Yes
Grid	Yes	No	No

The types for both programs and push arrays (`Program t` and `Push t`) have a parameter `t` that designates at which level in the hierarchy they are computed. At the bottom of this hierarchy is the `Thread`. Then a type function called `Step` increments a hierarchy level to the next level above it.

```

data Thread
data Step t

type Warp = Step Thread
type Block = Step Warp
type Grid = Step Block

```

Some operations are only possible at a level less than or equal to a given `t`. This is captured by a type class `*<=*`.

```
class a *<=* b
```

The class `Compute`, that was used as a constraint on the `ComputeAs` class seen earlier, is an example of this. The `Compute` class has an instance for all levels in the hierarchy that support storing intermediate data in shared memory and synchronising. This can be done on any level less than or equal to `Block`.

```
class (t *<=* Block, Sync t, Write t) => Compute t
```

The `Sync` class has an instance for all hierarchy levels that allow synchronising. On levels `Thread` and `Warp`, this synchronisation costs nothing as threads within a warp are executing in lock-step². On the `Block` level, however, storing data into shared memory is followed by a thread barrier synchronisation. The `Write` class implements the actual writing into memory at those levels that support it. The `Sync` and `Write` classes are considered internal to `Obsidian`, while the `Compute` class is visible in the API exposed to the programmer.

We now return to the `reduce` function from an earlier example, whose type was:

```

reduce :: (Compute t, Data a) =>
  (a -> a -> a) -> SPull a -> Program t (SPush t a)

```

Thus `reduce` is hierarchy-generic, but restricted. The `t` parameter must be `Block` or lower in order to satisfy the `Compute t` constraint. To illustrate this hierarchy level polymorphism, below the same `reduce` kernel is instantiated on two levels separated by a `Step`.

```

reduce2stage :: (t *<=* Block
  , Step t *<=* Block
  , Compute t
  , Compute (Step t)
  , Data a)
=> (a -> a -> a)
-> SPull a -> Program (Step t) (SPush (Step t) a)
reduce2stage f arr = do
  arr' <- compute $ liftPar $ fmap body (splitUp 32 arr)
  reduce f arr'
  where body a = exec (reduce f a)

```

The code above instantiates the base reduction kernel `reduce` at two levels while keeping the result as hierarchy-level generic as possible. Of course, this leads to quite a bit of

² This is a property of warps that may not hold on future GPU architectures and already requires marking shared memory arrays as `volatile` on current architectures.

complexity in the type of `reduce2stage`. A less polymorphic version might specialize the function to do the reduction specifically on the `Warp` and `Block` levels, as follows.

```

reduce2stage' :: Data a
              => (a -> a -> a)
              -> SPull a -> Program Block (SPush Block a)
reduce2stage' f arr = do
  arr' <- compute $ asBlock $ fmap body (splitUp 32 arr)
  reduce f arr'
  where body a = execWarp (reduce f a)

```

Figure 4 lists the hierarchy-generic and hierarchy-specific functions that make up the hierarchy programming API. The examples above also make use of `exec` functions, such as `execBlock` and `execWarp`. These functions are used to run a `Program` yielding its payload. The `exec` functions are listed in Figure 5.

When programming the hierarchy, the approach is to split a pull array up into a nested pull arrays using the `splitUp` function. These arrays are then distributed over the parallel resources and computed on using a function resulting in either a `Push t` array or a `Program t` (yielding an array or value) for some `t` over the inner arrays. The result of this is a pull array of either push arrays or programs.

If the function being mapped over the inner arrays uses shared memory, its result type will be a `Program t`. If it does not use shared memory the result type could, however, potentially be a pull or a push array. The hierarchy programming functions in Figure 4 are designed to operate with pull of push arrays. Using pull of pull arrays in the hierarchy programming API would break down when for example forming a grid. A pull of pull array would imply that any thread in the grid can access any element. those elements reside in shared memory (local to a block). It is however, still possible for any thread of any block to put its element anywhere (that is push it to anywhere in global memory at the grid level). In this way, the combinations of types that match the programmer's API mirror the structure of the GPU and guide the programmer towards using idioms that match the GPU's capabilities.

```

-- Fixed hierarchy level functions
-- | Perform many Thread level computations in sequence
--   in a Thread
asThread :: ASize l => Pull l (SPush Thread b) -> Push Thread l b
asThreadMap :: (a -> SPush Thread b) -> SPull a -> SPush Thread b

-- | Perform a collection of level t computations in parallel or
--   sequence forming a Warp
asWarp :: (t *<=* Warp) => SPull (SPush t a) -> SPush Warp a
asWarpMap :: (t *<=* Warp) => (a -> SPush t b) -> SPull a -> SPush Warp b

-- | Perform a collection of level t computations in parallel
--   forming a Block
asBlock :: (t *<=* Block) => SPull (SPush t a) -> SPush Block a
asBlockMap :: (t *<=* Block) => (a -> SPush t b) -> SPull a -> SPush Block b

-- | Perform a collection of Block level computations forming a Grid
asGrid :: ASize l => Pull l (SPush Block a) -> Push Grid l a
asGridMap :: ASize l => (a -> SPush Block b) -> Pull l a -> Push Grid l b

-- Hierarchy level generic function
-- | execute a collection of level t computations at level (t+1)
liftPar :: ASize l => Pull l (SPush t a) -> Push (Step t) l a

-- | at level t perform a collection of level t computations
--   in sequence
liftSeq :: ASize l => Pull l (SPush t a) -> Push t l a

-- | use the threads of level t to run a collection
--   of Thread level computations
liftIn :: (t *<=* Block, ASize l) => Pull l (SPush Thread b) -> Push t l b

```

Fig. 4. Functions for programming the hierarchy. In each of these functions the resulting push array is the concatenation of the sequenced computations.

```

-- | Execute a program to yield its resulting array.
-- Programmer annotations of application code may be needed
class ExecProgram t a where
  exec :: Data e
        => Program t (a Word32 e)
        -> Push t Word32 e
-- Instances
instance (t *<=* Block) => ExecProgram t Pull
instance ExecProgram t (Push t)

-- | Exec variants for specific levels of the hierarchy.
-- Less need for annotation in application code
class ExecThread a where
  execThread :: Data e
              => Program Thread (a Word32 e)
              -> Push Thread Word32 e

class ExecWarp a where
  execWarp :: Data e
           => Program Warp (a Word32 e)
           -> Push Warp Word32 e

class ExecBlock a where
  execBlock :: Data e
            => Program Block (a Word32 e)
            -> Push Block Word32 e
-- Instances.
instance ExecThread (Push Thread)
instance ExecThread Pull
instance ExecWarp (Push Warp)
instance ExecWarp Pull
instance ExecBlock (Push Block)
instance ExecBlock Pull

-- Auxiliery
execThread' :: Data a => Program Thread a -> SPush Thread a
execWarp'   :: Data a => Program Warp a -> SPush Warp a
execBlock'  :: Data a => Program Block a -> SPush Block a

```

Fig. 5. The exec family of functions.

5 Obsidian Implementation

In this section, we show how to compile Obsidian into CUDA code, thus implementing the concepts in Section 4. First, the Obsidian compiler deals with two types of AST: scalar expressions (e.g. `EWord32`, see implementation of `Exp` below), and `Programs` (statements, see Figure 6). Scalar expressions include standard first-order language constructs (arithmetic, conditionals, etc).

```
data Exp a where
  Literal :: Scalar a => a -> Exp a

  Index   :: Scalar a => (Name, [Exp Word32]) -> Exp a

  If      :: Scalar a => Exp Bool -> Exp a -> Exp a -> Exp a

  BinOp   :: (Scalar a, Scalar b, Scalar c)
           => Op ((a,b) -> c) -> Exp a -> Exp b -> Exp c

  UnOp    :: (Scalar a, Scalar b) => Op (a -> b) -> Exp a -> Exp b
  ...
```

The `Exp` GADT defines the small language that is used at the element level in Obsidian. The `Scalar` class used here is not for the end user; we provide an instance for each scalar type that is representable in the `Exp` AST, namely numeric types. Haskell tuples are used to build product types of the form `(Exp a, Exp b)`, rather than `Exp (a,b)`. That Obsidian uses Haskell tuples rather than embedding its own representation of them into the `Exp` type is the reason for the `Choice` class (amongst other similar classes). Below, you can see how the `Choice` class enables `ifThenElse` to be *more polymorphic* than its counterpart in many DSLs that returns only results of the form `Exp a`.

```
class Choice a where
  ifThenElse :: Exp Bool -> a -> a -> a

instance Scalar a => Choice (Exp a) where
  ifThenElse (Literal False) e1 e2 = e2
  ifThenElse (Literal True)  e1 e2 = e1
  ifThenElse b e1 e2 = If b e1 e2

instance (Choice a, Choice b) => Choice (a,b) where
  ifThenElse b (e1,e1') (e2,e2') = (ifThenElse b e1 e2,
                                     ifThenElse b e1' e2')

instance (Choice a, Choice b, Choice c) => Choice (a,b,c) where
  ifThenElse b (e1,e1',e1'') (e2,e2',e2'') = (ifThenElse b e1 e2,
                                                ifThenElse b e1' e2',
                                                ifThenElse b e1'' e2'')
```

The `BinOp`/`UnOp` datatypes in `Exp` contain the basic operations supported at the element level. There are roughly 60 of those operations including arithmetic, trigonometric, comparison, bitwise and conversion operations.

Apart from the `Exp` datatype, the compiler also has to deal with the `Program` type, defined in Figure 6. The `Program` datatype represents mostly low-level operations such as `Assign` (assignment to named array in memory) and `Sync` (barrier synchronization). However, it also includes operations that, at least compared to CUDA, are slightly higher level; `ForAll` and `DistrPar` are examples of such operations.

The `ForAll` operation iterates a body (described by higher order abstract syntax) a given number of times over the resources at a given level τ , with the iterations being independent of each other. If the level is `Thread`, this is a sequential iteration; if it is `Block` or `Warp` it is parallel. `DistrPar`, on the other hand, is used to iterate a body at a given level, τ , in parallel at the level directly above it, `Step` τ .

There are also operations in `Program` that are concerned with implementation details. `Bind` and `Return` enable a `Monad` instance for `Program` τ ; how this works is explained in detail in reference (Svenningsson & Svensson, 2013). The `Identifier` operation is used internally to generate new names for intermediate arrays and variables.

Note that neither `Exp` nor `Program` mentions pull or push arrays. The pull and push array representations are a shallow embedding implemented on top of the `Program` and `Exp` datatypes. At the point where the compiler has an AST to work on, all traces of pull and push arrays have been replaced by lower level operations in the `Program` AST.

5.1 Compilation to CUDA

Compiling to CUDA requires the following steps, covered in this subsection:

- 1A Reification:** Haskell functions representing Obsidian programs are turned into ASTs, including generating names for arrays.
- 1B Stripping:** The `Program level` datatype is converted from a higher-order representation to a list of statements.
- 2A Liveness Analysis:** The `IM` is analyzed to discovering the live ranges of arrays in shared memory. This stage annotates the `IM` with liveness information, that keeps track of where an array is created and at what point it can be freed.
- 2B Memory Mapping:** The annotated AST goes through a simple abstract interpretation, simulating it against a mock-up memory in order to create a memory map.
- 3 CUDA Code Generation:** At this stage, explicit parallel loops in the `IM` are compiled into CUDA. This is where virtualization of threads, warps and blocks takes place.

5.1.1 Reification and Stripping

At this stage Obsidian functions (Haskell functions using the Obsidian library) are turned into ASTs. A complete Obsidian program has a type such as:

```
prg1 :: Pull EWord32 EWord32 → Push Grid EWord32 EWord32
```

(Although variable numbers of input arrays are permitted as well.) Reifying this program is as simple as applying it to a *named array* in global memory:

```
(Pull n (\ix → Index ("input", [ix]))).
```

The function then yields its push array result. That push array, in turn, is a `Program` parame-

```

data Program t a where

  Identifier :: Program t Identifier

  Assign :: Scalar a => Name -> [Exp Word32] -> (Exp a) -> Program Thread ()

  AtomicOp :: Scalar a
            => Name          -- Array name
            -> Exp Word32   -- Index to operate on
            -> Atomic a     -- Atomic operation to perform
            -> Program Thread ()

  Cond :: Exp Bool -> Program Thread () -> Program Thread ()

  SeqWhile :: Exp Bool -> Program Thread () -> Program Thread ()

  Break  :: Program Thread ()

  -- use threads along one level
  -- Thread, Warp, Block.
  ForAll :: (t *<=* Block) => EWord32 -> (EWord32 -> Program Thread ())
        -> Program t ()

  -- Distribute over Warps yielding a Block
  -- Distribute over Blocks yielding a Grid
  DistrPar :: EWord32 -> (EWord32 -> Program t ()) -> Program (Step t) ()

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

  -- Allocate shared memory in each MP
  Allocate :: Name -> Word32 -> Type -> Program t ()

  -- Automatic Variables
  Declare :: Name -> Type -> Program t ()

  Sync    :: Program Block ()

  -- Monad
  Return :: a -> Program t a
  Bind   :: Program t a -> (a -> Program t b) -> Program

```

Fig. 6. The Program GADT. Sequencing is provided via the monad bind operations. This allows sequences of statements in the AST to be generated using Haskell do notation, for example `do Allocate "arr1" 512 Int; ForAll 512 body; Sync`. In reference (Svenningsson *et al.*, 2013) we make use of the atomic operations represented here to implement sorting algorithms.

terized on a write-function. Providing a writer function ($\lambda a ix \rightarrow \text{Assign } \text{"output"}[ix] a$), which writes to a named (global) array, completes reification, yielding a Program AST.

Following this, the Program AST is converted into an representation where the `t` parameter is made concrete. The new representation is called IM.

```

type IMList a = [(Statement a,a)]

type IM = IMList ()

```

The parameter `a` is used to hold annotations on the nodes during subsequent compilation phases. The `Statement` type is very similar to `Program`, but sequencing of operations is replaced by the list type, `IMList`.

```

data Statement t = SAssign IExp [IExp] IExp
                 | SAtomicOp IExp IExp AtOp
                 | SCond IExp (IMList t)
                 | SSeqFor String IExp (IMList t)
                 | SBreak
                 | SSeqWhile IExp (IMList t)

                 | SForAll HLevel IExp (IMList t)
                 | SDistrPar HLevel IExp (IMList t)

                 | SAllocate Name Word32 Type
                 | SDeclare Name Type

                 | SSynchronize

data HLevel = Thread | Warp | Block | Grid

```

Here, `IExp` replaces `Exp` as the type for element-level expressions, and AST nodes have been explicitly annotated with types.

5.1.2 Liveness Analysis and Memory Mapping

The `compute` function, which introduces manifest arrays in shared memory, generates unique names for each intermediate array. CUDA does not provide any memory management facilities for shared memory, so in Obsidian we analyse kernel memory usage and create a memory map at compile time.

The amount of shared memory available in each GPU multiprocessor varies. (But it is always a small number, for example, 48KB.) When using the `capture` function to compile an Obsidian program, the GPU device is queried for the amount of shared memory and number of banks it is divided into, this information is used in the memory mapping procedure.

Shared memory is a limited resource. Making good use (and reuse) of it is important. The Obsidian `IM` AST already contains `Allocate` nodes (introduced by the `compute` function) that show where arrays come into existence. A standard analysis computes the full live range of each array:

- Step through list of statements in reverse. When encountering an array name for the first time it is added to a set of live arrays. The list of statements is annotated with this liveness information.
- When an `Allocate` statement is found, the array being allocated is removed from the set of live arrays.

Following this analysis phase, a memory map is constructed using a greedy strategy. This is done by simulating the AST execution together against an abstraction of the shared memory. The simulated shared memory is implemented as a list of free ranges and a list of allocated ranges. “malloc” requests are serviced with the first available memory segment of sufficient size. The maximum size ever used is tracked, and in the end this is the total amount of shared memory needed for this kernel. Kernels are allocated in such a way that the first element of the array falls in memory bank zero; padding is inserted to ensure this. If an array could start anywhere (for example in the middle of bank three), the programmer would have no chance of crafting good memory access patterns.

An Obsidian program may end up trying to use too much shared memory. This could happen if a kernel uses `compute` on many arrays that are all alive at the same time. If this happens, compilation of that program will fail at this stage. The alternative, to use global device memory (and warn the programmer) when shared memory is exhausted, would lead to drastically decreased performance. In practical use of Obsidian, we have not run into difficulties because of the limited availability of shared memory.

This greedy approach to memory management can potentially lead to memory fragmentation, and the greedy solution is certainly not optimal. However, (1) in practice we see local arrays either of the same size or shrinking sizes (divide and conquer), and (2) unlike traditional register allocation, this process primarily affects *whether* a kernel will compile, not its performance: since we do not spill arrays to global device memory. The upside of automatic shared memory management is that in Obsidian it is much easier than it is in CUDA to reuse and remap shared memory within a large kernel. The CUDA programmer would need to allocate a local array and then manually cast portions of it for reuse, which is tedious and error prone. This is another example of how the abstractions of functional programming can ease the life of the kernel developer, removing tedium so that time and effort can be spent on intelligently exploring the design space.

5.1.3 CUDA Code Generation

CUDA code is generated from the list of statements. This phase takes as a parameter the *number of real CUDA threads* for which the code should be generated. So it is here that resource virtualization must be addressed. The compilation is done using the `Language.C.Quote` library that allows us to mix in C syntax in our Haskell code. Most cases of this compilation are very simple, as many statements correspond directly to their CUDA counterparts. For example, an assignment statement is compiled as follows:

```
compileStm _ (Sassign name [ix] e) =
  [[cstm| $(compileExp name)[$(compileExp ix)] = $(compileExp e); ]]
```

The interesting cases are those that deal with parallelism, such as the `ForAll` and `DistrPar` statements. For example, compiling a parallel-for over threads in a block has the following structure:

```

compileStm realThreads (SForAll Block n body) = goQ ++goR
  where
    -- how to split the iteration space
    -- across the realThreads.
    -- q passes across all real threads
    -- followed by a stage of using r real threads
    q = n `quot` realThreads
    r = n `rem` realThreads

    goQ = for (int i = 0; i <q; ++i) {
      -- repurpose tid
      tid = i*nt + threadIdx.x;
      body
    }
    goR = -- run the last r threads
          if (threadIdx.x <r) {
            ...
          }

```

Compilation of `DistrPar` performs a similar technique for the virtualization of the available number of warps and blocks.

6 Case studies

Obsidian is designed to assist users in crafting high performance kernels. The aim is to remove some of the tedium of index manipulation and memory management, thus allowing more effort to be devoted to exploration of the design space and performance improvement. This exploration could be manual or could take the form of auto-tuning.

The following case studies start with a simple kernel, embarrassingly parallel with no inter-thread communication. Even with such a kernel, there is non-trivial tuning to maximize throughput. The remaining case studies consider key building blocks, reduce and scan, that have data-flow graphs involving much more communication. In a following section we compare these against the corresponding kernels within the NVIDIA Thrust and Accelerate implementation. Accelerate is a much higher level DSL but one with hand-tuned (but not auto-tuned) CUDA skeletons for patterns like scan and fold.

6.1 Case Study: Mandelbrot Fractals

We begin with a simple case; here we show how to implement the Mandelbrot fractal using Obsidian. It is an embarrassingly parallel program included as an example of a *complete* Obsidian application, with all code contained in this section. In fact, in spite of Mandelbrot's simplicity, even it exhibits performance complexities—the most efficient parameterisation (numbers of threads per block) differs between the two GPUs we test in Figure 7.

The Mandelbrot fractal is generated by iterating a function:

$$z_{n+1} = z_n^2 + c$$

where z and c are complex numbers. The method presented here is based on a sequential C program from reference (Stevens, 1989).

In order to generate the Mandelbrot fractal, one lets z_0 be zero and maps the x and y coordinates of the image being generated to the real and imaginary components of the c variable.

```
xmax, xmin :: EFloat
xmax = 1.2
xmin = -2.0

ymax, ymin :: EFloat
ymax = 1.2
ymin = -1.2
```

To obtain the well known and classical image of the set, we let the real part of c range over -2.0 to 1.2 as the x coordinate ranges from 0 to 512 , and similarly for the y coordinate and the imaginary component.

```
-- For generating a 512x512 image
deltaP, deltaQ :: EFloat
deltaP = (xmax - xmin) / 512.0
deltaQ = (ymax - ymin) / 512.0
```

The image is generated by iterating the function presented above. We map the height of the image onto blocks of executing threads. Each row of the image is computed by one block of threads. This means that for a 512×512 pixel image, 512 blocks of 512 threads are needed. The function to be iterated is defined below and called f . This function will be iterated until a condition holds (defined in the function $cond$). We count the number of iterations and break out of the iteration if it reaches 512 .

```
f :: EFloat → EFloat
  → (EFloat, EFloat, EWord32)
  → (EFloat, EFloat, EWord32)
f b t (x,y,iter) =
  (xsq - ysq + (xmin + t * deltaP),
   2*x*y + (ymax - b * deltaQ),
   iter+1)
  where
    xsq = x*x
    ysq = y*y

cond :: (EFloat, EFloat, EWord32) → EBool
cond (x,y,iter) = ((xsq + ysq) <= 4) && iter <= 512
  where
    xsq = x*x
    ysq = y*y
```

The number of iterations executed is used to decide which colour to assign to the corresponding pixel. In the function below, `seqUntil` iterates f until the condition $cond$ holds. Then the number of iterations is extracted and used to compute a colour value (out of 16 possible values).

```

iters :: EWord32 → EWord32 → SPush Thread EWord8
iters bid tid =
  fmap extract (seqUntil (f bid' tid') cond (0,0,1))
  where
    extract (_,_,c) = (w32ToW8 (c `mod` 16)) * 16
    tid' = w32ToF tid
    bid' = w32ToF bid

```

The final step is to run the iteration for each pixel location, by implementing a `genRect` function that spreads a sequential `Push Thread` computation across the grid.

```

genRect :: EWord32
         → Word32
         → (EWord32 → EWord32 → SPush Thread b)
         → DPush Grid b
genRect bs ts p = asGrid
                 $ mkPull bs
                 $ λbid → asBlock $ mkPull ts (p bid)

```

Generating the Mandelbrot image is done by generating a rectangle, applying the `iters` function at all points.

```

mandel :: DPush Grid EW8
mandel = genRect 512 512 body
  where
    body i j = execThread' (iters i j)

```

Running the `mandel` program and generating a raw output image is done as follows.

```

import qualified Data.Vector.Storable as V
import Data.ByteString as BS

performMandel :: IO ()
performMandel =
  withCUDA $
  do
    kern ← capture 256 mandel
    allocaVector (512*512) $ λo →
      do
        o <== (256,kern)
        r ← copyOut o

        lift $ BS.writeFile "fractal.out" (pack (V.toList r))

```

This is a case where virtualisation of threads and blocks helps the programmer. The code generates a 512×512 pixel image using 512 blocks each of 512 threads. The blocks and threads, however, can be virtual, meaning we can still generate the 512×512 image using for example 64 (real) blocks of each 64 (real) threads. It also means that images larger than 1024 pixels wide are possible. (1024 is the hardware limit on number of threads per block.) Also, in Obsidian, these limits can be broken without any burden on the programmer, which would not be the case if programming in CUDA. Using CUDA, the programmer would directly implement those sequential loops and the indexing arithmetic to go with them.

Fig. 7. Running times for the Mandelbrot program. The left table shows times measured on an NVIDIA GTX680 GPU. The right table shows times measured on an NVIDIA TESLA c2070. The columns varies the number of threads-per-block, while the rows varies image size (square images). Each benchmarks was executed 1000 times and the total time is reported in seconds. The transfer of data to or from the GPU is not included in the timing measurements.

size	32	64	128	256	512	1024	size	32	64	128	256	512	1024
256	0.25	0.17	0.12	0.21	0.33	0.60	256	0.44	0.38	0.41	0.36	0.41	0.98
512	0.71	0.43	0.34	0.41	0.69	1.16	512	1.44	1.16	1.17	1.16	1.14	2.00
1024	2.41	1.39	1.05	1.22	1.53	2.58	1024	5.12	3.96	3.95	3.98	4.17	4.75
2048	8.86	4.98	3.67	3.88	4.69	5.95	2048	18.80	14.53	14.38	14.48	14.84	17.50
4096	34.21	18.82	13.69	14.07	15.36	18.65	4096	72.12	55.36	54.94	55.16	55.67	61.89



Fig. 8. *Left*: evenOdds - zipWith reduction, leads to uncoalesced memory accesses. *Right*: halfve - zipWith reduction, leads to coalesced memory accesses. This coalescing is most important during the very first phase, when data is read from global memory.

6.2 Case Study: Reduction

In this section, we implement a series of reduction kernels. These Obsidian reductions take an associative operator as a parameter. In these benchmarks, the reduction will be addition only and the elements will be 32 bit unsigned integers. Some of the reduction kernels will also require that the operation is commutative.

To illustrate the kind of low level control that an Obsidian programmer has over expressing details of a kernel, each reduction kernel in the series has different optimisations applied. Many of the optimisations applied to the kernels can be found in a presentation from NVIDIA (Harris, 2007). This section focuses on local reduction kernels (on-chip storage only). In Section 7.1 these kernels are used as building blocks in the construction of reduction algorithms for millions of elements. Nevertheless, even these local kernels expose a large search space of both implementation strategies and tuning parameters. While



Fig. 9. *Left*: **BAD** Adding sequential reductions like this, reintroduces memory coalescing issues. Consecutive threads nolonger access consecutive memory locations. *Right*: **GOOD** Using sequential reduction but maintaining coalescing

we only varied **threads-per-block** in the last subsection, in this and the next we consider the following tuning parameters:

- **threads-per-block** - The number of threads per block used by each instance of the reduction kernel. This parameter takes on the values from [32, 64, 128, 256, 512, 1024]
- **elements-per-block** - The number of elements reduced by one instance of the kernel. This parameter takes on the values from [256, 512, 1024, 2048, 4096, 8192, 16384, 32768].
- **kernel-implementation** - 7 different reduction kernels (called red1 to red7). The largest elements-per-block size is only applicable using some of the reduction kernels (red4 to red7).

In our experiments, we vary all of these parameters resulting in 312 configurations in each benchmark run. Figures 10 and 11 show one way to aggregate these results.

6.2.1 Reduction 1: Recursively collapse adjacent

Our first attempt at reduction combines adjacent elements repeatedly. This approach is illustrated on the left of Figure 8. In Obsidian, this entails splitting the array into its even and its odd elements and using `zipWith` to combine these. This procedure is then repeated until there is only one element left. This kernel will work for arrays whose length is a power of two.

```
red1 :: Data a
  => (a -> a -> a)
  -> Pull Word32 a
  -> Program Block (SPush Block a)
red1 f arr
| len arr == 1 = return $ push arr
| otherwise   =
do
  let (a1,a2) = evenOdds arr
      imm ← compute (zipWith f a1 a2)
      red1 f imm
```

The above code describes what one block of threads does. To spread this computation out over many blocks and thus perform many simultaneous reductions, `asGridMap` is used:

```
mapRed1 :: Data a
  => (a -> a -> a)
  -> Pull EWord32 (SPull a)
  -> Push Grid EWord32 a
mapRed1 f arr = asGridMap body arr
  where
    body arr = execBlock (red1 f arr)
```

This kernel does not perform well (Figure 11), which may be attributed to its memory access pattern. Remember that one gets better performance on memory access when consecutive threads access consecutive elements. In this kernel consecutive threads access elements in a strided fashion. Thread zero accesses element 0 and n , thread one accesses element 2 and $n + 2$. It would have been much better if thread i accessed element i and $n + i$, which brings us to the next reduction kernel.

6.2.2 Reduction 2: Recursively halve and combine

`red2` tries to improve the memory access pattern by making consecutive threads access consecutive array elements. It does this by halving the input array and then using `zipWith` on the halves (see Figure 8). This choice can only be made if the operator is commutative.

```
red2 :: Data a
  => (a -> a -> a)
  -> Pull Word32 a
  -> Program Block (SPush Block a)
red2 f arr
| len arr == 1 = return push arr
| otherwise =
do
  let (a1,a2) = halve arr
      arr' ← compute (zipWith f a1 a2)
      red2 f arr'
```

6.2.3 Reduction 3: deforest the last shared memory copy

The two previous implementations of reduce write the final value into shared memory (as there is a `compute` in the very last stage). This means that the last element is stored into shared memory and then directly copied into global memory. This can be avoided by cutting the recursion off at length 2 instead of 1, and performing the last operation without issuing a `compute`.

```
red3 :: Data a
  => (a -> a -> a)
  -> Pull Word32 a
  -> Program Block (SPush Block a)
red3 f arr
| len arr == 2 =
return $ push $ fold1 f arr
| otherwise =
do
  let (a1,a2) = halve arr
      arr' ← compute (zipWith f a1 a2)
      red3 f arr'
```

This kernel takes cuts recursion off at length 2 and when the array reaches that length, a sequential fold over pull array sums up the remaining elements. Performing this cutoff at two elements does not change the overall depth of the algorithm, but since there is no force in the last stage the result will not be stored in shared memory. This optimisation has minimal effect on performance.

6.2.4 Reduction 4: Add sequential reduction, increase elements-in-sequence

Now we have a set of three basic ways to implement reduction and can start experimenting with adding sequential, per-thread, computation. `red4` uses `seqReduce`, which is provided by the Obsidian library and implements a sequential reduction that turns into a for loop

in the generated CUDA code. The input array is split into chunks of 8 that are reduced sequentially. The partial results are reduced using `red3`.

```
red4 :: Data a
      => (a -> a -> a)
      -> Pull Word32 a
      -> Program Block (SPush Block a)
red4 f arr =
  do arr' ← compute $ asBlockMap (execThread' ∘ seqReduce f)
                                     (splitUp 8 arr)
    red3 f arr'
```

Unfortunately, adding `seqReduce` reintroduces memory coalescing problems—because each thread reads consecutive elements in time, rather than striding at warp-sized steps (Figure 9)—to the detriment of performance (Figure 11).

6.2.5 Reduction 5, 6 and 7: restore coalescing, vary elements-in-sequence

Next, in `red5`, `red6` and `red7`, we address the coalescing problem by defining a function to split up the array into sub arrays, such that the elements in the inner arrays should be drawn from the original array in a strided fashion. Again, the idea is to maintain consecutive accesses by consecutive threads.

```
coalesce :: ASize l
          => Word32
          -> Pull l a
          -> Pull l (Pull Word32 a)
coalesce n arr =
  mkPull s (λi →
    mkPull n (λj → arr ! (i + (sizeConv s) * j)))
  where s = len arr `div` fromIntegral n
```

`coalesce` shows another benefit of high-level GPU meta-programming: index permutations need not pollute the consumer's code, they simply return new, first-class (delayed) arrays. With `coalesce` in place of `splitUp`, we implement a parameterised reduction kernel. The parameter `n` decides the degree of sequential work.

```
redParam :: Data a
          => Word32
          -> (a -> a -> a)
          -> Pull Word32 a
          -> Program Block (SPush Block a)
redParam n f arr =
  do arr' ← compute $ asBlockMap (execThread' ∘ seqReduce f)
                                     (coalesce n arr)
    red3 2 f arr'
```

Using the parameter we can push the tradeoff between number of threads and sequential work-per-thread further. `red5`, `red6` and `red7` vary this parameter to reduce 8, 16 and 32 elements in the sequential phase, respectively. That is, they differ only in varying the

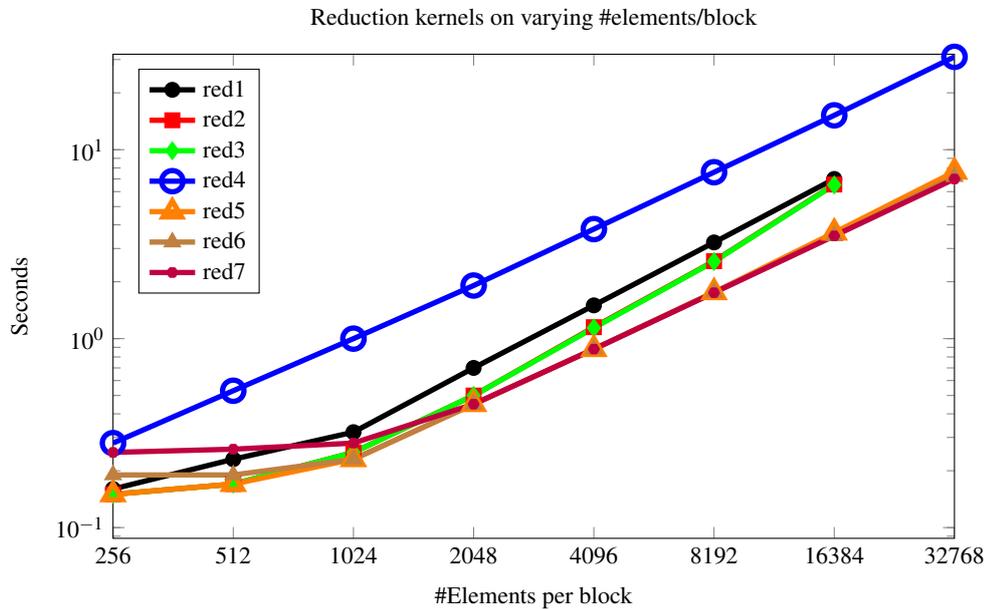


Fig. 10. The running time of 8192 blocks executing a reduction kernel. The time reported is the sum of 1000 executions of the 8192 blocks grid, excluding transfer time of data to GPU memory. The X-axis varies **elements-per-block**, but each point represents the *best setting* for **threads-per-block**. These number are collected on an NVIDIA GTX680.

Fig. 11. The **threads-per-block** setting that achieved the best time shown in Figure 10. These settings are difficult to predict in advance. Kernels that use virtualized threads are highlighted, note that there are many of these amongst the best selection. Again, **elements-per-block** varies over the X axis.

Kernel	256	512	1024	2048	4096	8192	16384	32768
red1	64	128	128	256	256	512	512	n/a
red2	64	128	64	128	256	512	512	n/a
red3	64	128	64	128	256	512	512	n/a
red4	64	64	128	64	64	64	128	512
red5	32	64	64	64	128	256	256	512
red6	32	32	64	64	128	128	256	256
red7	32	32	32	64	128	128	512	128

elements-in-sequence parameter. The performance of the fastest of these kernels is very satisfactory, at a level where the kernel is limited by memory bandwidth.

```

red5 = redParam 8
red6 = redParam 16
red7 = redParam 32
    
```

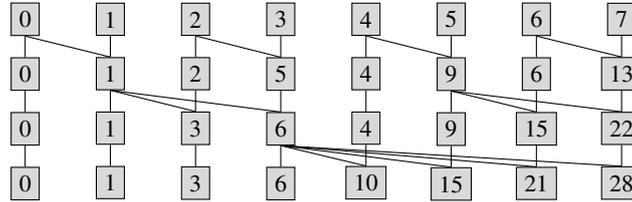


Fig. 12. Sklansky parallel prefix network

6.3 Case Study: Scan

Scan computes all the prefix sums of a sequence of values using a binary associative operator (and is familiar to Haskell programmers as the `scanl1` function). Given an array of values a_0, a_1, \dots, a_n and associative operator \oplus , the scan operation computes a new array:

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

During performance evaluation of the scan kernels developed here we vary the following tuning parameters:

- **threads-per-block** - The number of threads per block used by each instance of the scan kernel. This parameter takes on the values from $[32, 64, 128, 256, 512, 1024]$
- **elements-per-block** - The number of elements reduced by one instance of the kernel. This parameter takes on the values from $[256, 512, 1024, 2048, 4096]$.
- **kernel-implementation** - 5 different scan kernels. 3 of the scan variants are based on the Sklansky network and 2 on the Kogge-Stone construction.

The total number of configurations is in this case 150. The results of these experiments are presented in figures 14 and 15.

6.3.1 Sklansky Scan

Figure 12 shows a standard divide and conquer decomposition of scan. Data flows from top to bottom and boxes with two inputs are operators. At each level, exactly half of the boxes are operators and in an imperative language the algorithm would naturally be implemented in-place. Since we cannot easily express in-place algorithms currently in Obsidian, this means that we need to copy unchanged values into a new array during each phase. During a phase of compilation, Obsidian analyses memory usage and lays out intermediate arrays in memory. In the case of Sklansky scan kernels this leads to a ping-ponging behaviour between arrays occupying two areas of shared memory.

Also, the threads now do two different things (copy, or perform operation). One can have as many threads as elements, but then each must have a conditional to decide whether to be a copy or operation thread. Or we can launch half as many threads and have each of them

perform both a copy and an operation. We will show code for both of these options; the first is easier to implement.

The Obsidian code below implements the scan network from Figure 12, using as many threads as there are elements. Note that thread virtualization applies here, supporting arrays larger than the actual number of GPU threads. The limiting factor is the amount of shared memory.

```
sklansky :: Data a
  => Int
  → (a → a → a)
  → Pull Word32 a
  → Program Block (Push Block Word32 a)
sklansky 0 op arr = return $ push arr
sklansky n op arr =
  do let arr1 = unsafeBinSplit (n-1) (fan op) arr
      arr2 ← compute arr1
      sklansky (n-1) op arr2
```

This is a kernel generator; the (Haskell) `Int` parameter can be used to generate kernels of various sizes by setting it to the log base two of the desired array size.

The `unsafeBinSplit` combinator used in `sklansky` is part of the Obsidian library and used to implement divide and conquer algorithms. It divides an array recursively in half a number of times (first parameter) and applies a computation to each part (second parameter).

The `unsafeBinSplit` function is deemed “unsafe” because it will behave in an unsuspected way if the array used as input has delayed operations on it. The delayed operations will be replicated into each split of the array³. In practise this has seldom been a problem.

The operation applied in this case is `fan`:

```
fan :: Data a
  => (a → a → a)
  → SPull a
  → SPull a
fan op arr = a1 'append' fmap (op (last a1)) a2
  where
    (a1,a2) = halve arr
```

It is the array concatenation (`append`) used in this function that introduces conditionals into the generated code.

Both to avoid conditionals and to allow for larger scans per block, we move to two elements per thread. Each phase of the algorithm is a parallel for loop that is executed by half as many threads as there are elements to scan. The body of the loop performs one operation and one copy, using bit-twiddling to compute indices. Note the use of two *write functions* in sequence. Similar patterns were used in our implementations of sorting networks (Claessen *et al.*, 2012), for similar reasons.

³ There is a safe variant called `binSplit` that takes a mutable array as input

```

phase :: Int
  → (a → a → a)
  → Pull Word32 a
  → Push Block Word32 a
phase i f arr =
  mkPush 1 (λwf → forAll s12 (λtid →
do let ix1 = insertZero i tid
      ix2 = flipBit i ix1
      ix3 = zeroBits i ix2 - 1
      wf (arr ! ix1) ix1
      wf (f (arr ! ix3) (arr ! ix2) ) ix2))
  where
    l = len arr
    l2 = l `div` 2
    s12 = fromIntegral l2

```

For an input of length 2^n , n phases are composed as follows:

```

sklansky2 :: Data a
  ⇒ Int
  → (a → a → a)
  → Pull Word32 a
  → Program Block (Push Block Word32 a)
sklansky2 l f = compose [phase i f | i ← [0..(l-1)]]

```

`compose` sequences a list of programs, computing intermediate arrays between each step.

```

compose :: Data a
  ⇒ [Pull Word32 a → Push Block Word32 a]
  → Pull Word32 a
  → Program Block (Push Block Word32 a)
compose [f] arr = return $ f arr
compose (f:fs) arr = compose fs =<< compute (f arr)

```

Comparing the two kernels `sklansky` and `sklansky2` in the NVIDIA profiler indicates that `sklansky2`, while being faster than `sklansky` in many cases, has a worse memory loading behaviour. This indicates that tweaking the way data is loaded into shared memory may be beneficial in that kernel.

```

sklansky3 :: Data a
  ⇒ Int
  → (a → a → a)
  → Pull Word32 a
  → Program Block (Push Block Word32 a)
sklansky3 l f arr =
  do im ← compute $ load 2 arr
     compose [phase i f | i ← [0..(l-1)]] im

```

Here we use `load 2` to realise loading of 2 elements per thread but in a strided way that is more likely to lead to a good memory access pattern. This function is an example of one of the custom ways to create a push array from a pull array.

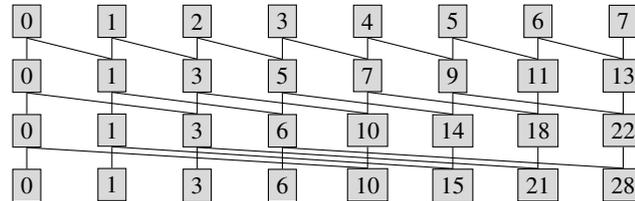


Fig. 13. Kogge-Stone parallel prefix network

```

load :: Word32 → Pull Word32 a → Push Block Word32 a
load n arr =
  mkPush m (λwf →
    forall (fromIntegral n') (λtid →
      do
        seqFor (fromIntegral n) (λix →
          wf (arr ! (tid + (ix*fromIntegral n'))
              (tid + (ix*fromIntegral n')))))
        where
          m = len arr
          n' = m `div` n

```

The results of these optimisations are shown in Figure 14.

6.3.2 Kogge-Stone Scan

Figure 13 illustrates another approach to computing scan. The figure shows three stages of this method of implementing scan. In stage one, the input array is zipped with itself with one element dropped. In stage two, the result of the previous stage is zipped with itself with two elements dropped. And in general, at stage n , 2^{n-1} . As with Sklansky, in each stage some values are copied unchanged. These unchanged values are the 2^{n-1} first elements. This algorithm performs more work than the Sklansky implementation, but it is very regular and therefore interesting to try out on the GPU.

The code below implements the Kogge-Stone prefix network kernel:

```

ksLocal :: Data a ⇒ Int → (a → a → a)
        → SPull a
        → Program Block (SPush Block a)
ksLocal 0 op arr = return $ push arr
ksLocal n op arr = do
  arr2 ← compute =<< ksLocal (n-1) op arr
  let m = 2^(n-1)
      a1 = drop m arr2
      oped = zipWith op arr2 a1
      copy = take m arr2
      all = copy `append` oped
  return $ push all

```

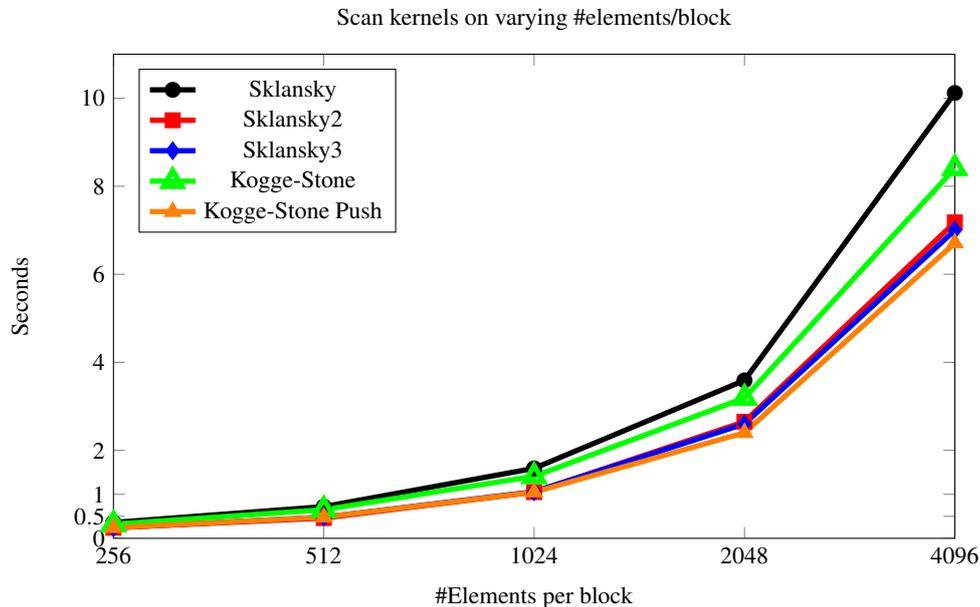


Fig. 14. The running time of 8192 blocks executing scan kernel. The time reported is the sum of 1000 executions of the 8192 blocks grid, excluding transfer time of data to GPU memory. The number of elements processed per block varies over the X-axis. Again, the only the best threads-per-block setting at each elements-per-block is shown. These number are collected on an NVIDIA GTX680.

This algorithm uses a concatenation and in the implementation above this is concatenation is performed on pull arrays. A small change to the program switches to a concatenation of push arrays.

```

ksLocalP :: Data a => Int -> (a -> a -> a)
           -> SPull a
           -> BProgram (SPush Block a)
ksLocalP 0 op arr = return $ push arr
ksLocalP n op arr = do
  arr2 <- compute =<< ksLocalP (n-1) op arr
  let m   = 2^(n-1)
      a1  = drop m arr2
      oped = push $ zipWith op arr2 a1
      copy = push $ take m arr2
  return $ copy 'concP' oped

```

Figure 14 show a performance comparison of the scan kernels implemented here. Out of tried kernels the Kogge-Stone variant using push array concatenation was the fastest.

The first experiment tests each of the kernels in the setting of a large reduction. The second experiment tries to drill down and find if tweaking the other parameters, for example number of blocks will give further improvement. In this case at 16 million elements the best value is the same between the two experiments.

Fig. 15. Shows what number of threads performed best for a given kernel and number of elements to process per block.

Kernel	256	512	1024	2048	4096
Sklansky	128	128	256	512	1024
Sklansky2	128	128	256	256	512
Sklansky3	128	128	256	512	512
Kogge-Stone	64	128	256	512	512
Kogge-Stone Push	64	128	256	512	512

6.4 Scan: More Work Per Block

In the reduction kernels of Section 6.2, we used sequential computation per thread to increase performance. The scan kernels implemented above will use virtual threads when the local scan is sufficiently large. The amount of shared memory available does however become a limiting factor when trying to increase the size of the local scan. One way to circumvent this problem is to have each block perform more than one scan in sequence and pass a carry value from the previous to the next instance. Thus, the shared memory used by the earlier instances can be reused by the later, and even more elements can be scanned per block.

This sequential sequencing of parallel work, with a carry, can be implemented using the function `sMapAccum` from the Obsidian library.

```
sMapAccum :: (Compute t, Data acc, ASize l)
  => (acc -> Pull l a -> Program t (acc, Push t l b))
  -> acc
  -> Pull l (Pull l a)
  -> Push t l b
```

In section 7.2 this approach is used to implement efficient scan algorithms for millions of elements. Still, the kernel building blocks used are the ones described above, only wrapped with code for handling input carry values and producing a carry out.

```
wrapKernCin :: Data a
  => ScanKernel a
  -> Int -> (a -> a -> a) -> a -> SPull a
  -> Program Block (a, SPush Block a)
wrapKernCin kern n op cin arr = do
  arr' <- compute $ applyToHead op cin arr
  arr'' <- compute $ execBlock $ kern n op arr'
  return (last arr'', push arr'')
where
  applyToHead op cin arr =
    let h = fmap (op cin) $ take 1 arr
        b = drop 1 arr
    in h 'append' b
```

Now a kernel that performs many scans in series connected via carry in - carry out can be implemented.

```

sklanskies :: Data a
            => Int -> (a -> a -> a) -> a -> SPull a -> SPush Block a
sklanskies n op acc arr =
  sMapAccum (wrapKernCin sklansky n op) acc (splitUp (2^n) arr)

```

The addition of `sMapAccum` to Obsidian is recent and we have previously struggled to implement scan algorithms that compete with hand tuned code but have now succeeded as can be seen in the data presented in section 7.2.

7 Combining kernels to solve large problems

We have seen how, with Obsidian, we can experiment with details of kernel code generation. In Section 6, we saw that the description of a local kernel involves its behavior when spread out over many blocks. However, solving large problems must sometimes make use of many different kernels or the same kernel used repeatedly. Here we use reductions of as an example of how to stitch together combinations of kernels.

7.1 Large Reductions

We implement reduction of large arrays by running local kernels on blocks of the input array. If the local kernel reduces n elements to 1 then this first step reduces $numBlocks * n$ elements into $numBlocks$ partial results. The procedure is then repeated on the $numBlocks$ elements until there is one value.

```

launchReduce = withCUDA (
  do let n = blocks * elts
        blocks = 4096
        elts = 4096
        kern ← capture 32 (mapRed5 (+) ∘ splitUp elts)

    (inputs :: V.Vector Word32) ←
      lift (mkRandomVec (fromIntegral n))
    useVector inputs (λi →
      allocaVector (fromIntegral blocks) (λ o →
        allocaVector 1 (λ o2 → do
          do o <== (blocks,kern) <> i
            o2 <== (1,kern) <> o
          copyOut o2))))

```

The code above is one example of our API for writing CPU-side host-programs, though it is also possible to call Obsidian-generated kernels from CUDA code as well. Figure 17 shows the running time for the above program executing a 2^{24} element reduction; in that figure and in Figure 16, we compare Obsidian against NVIDIA Thrust and Accelerate.

The evaluation of large reduction algorithms is done here in two different ways. First, in figure 17 we vary reduction kernel and the number of threads. The number of blocks launched and the total input array size is kept constant. The following list specifies the configuration space used in figure 17

- **threads-per-block** - The number of threads per block used by each instance of the scan kernel. This parameter takes on the values from [32, 64, 128, 256, 512, 1024]

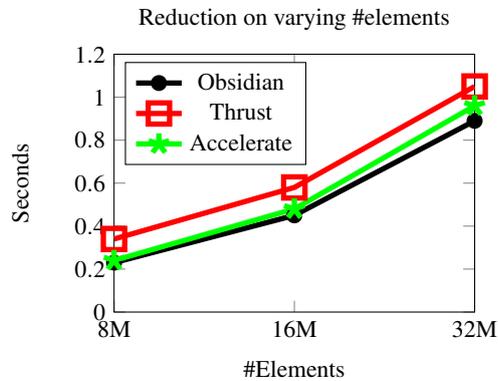


Fig. 16. The running time of reduction algorithms for larger data sizes. The time reported is the sum of 1000 executions, excluding data transfer to and from the GPU memory. These numbers are collected on an NVIDIA GTX680. The presented Accelerate numbers are estimates based on a lower number of iterations as explained in Section 7.3. The Obsidian numbers presented here come from the run with the parameter settings that performed the best.

- **reduction-kernel** - Seven different reduction kernels are used, (red1 to red7).
- **total-number-of-elements** - Is always 16777216
- **number-of-blocks** - Is always 4096

In total there are 42 different configurations.

In figure 16 we try a different approach. The kernel used is kept constant. While block size, total number of elements and threads per block vary. The configuration space is described below:

- **threads-per-block** - The number of threads per block used by each instance of the scan kernel. This parameter takes on the values from [32, 64, 128, 256, 512, 1024]
- **number-of-blocks** - The number of blocks on the GPU. This parameter takes on the values from [16, 32, 64, 128, 256, 512, 1024].
- **total-number-of-elements** - The size of input array that is reduced to a single value. This parameter takes on the values [8388608, 16777216, 33554432].

This results in 126 different configurations. The kernel used in the experiment is an adaptation of red5 (section 6.2) that selects an appropriate sequential depth given the number of blocks and total number of threads. The three reduction kernels red5, red6 and red7 are all very similar and all perform very well, they differ only in the amount of sequential work performed. This is what motivates using that kind of kernel in this experiment.

Fig. 17. Running times of 2^{24} (16M) element reduction using Obsidian. The results were obtained on a NVIDIA TESLA c2070 and the GTX680. Each reduction procedure was executed 1000 times, and the total execution time is reported in the table. Seven different reduction kernels (red1 to red7) are compared each with varying parameter settings of number of threads per block. The best threads per block setting for each kernel is listed in the table.

Variant	Parameter	Seconds	Parameter ^a	Seconds ^b
On Tesla C2070:				
red1	256 threads	0.75	32	2.11
red2	256 threads	0.80	32	2.41
red3	256 threads	0.80	32	2.41
red4	512 threads	1.07	1024	2.08
red5	256 threads	0.71	1024	1.88
red6	256 threads	0.69	1024	1.97
red7	128 threads	0.72	1024	1.97
On GTX680:				
red1	256 threads	0.77	32	1.95
red2	256 threads	0.59	32	1.90
red3	256 threads	0.59	32	1.89
red4	64 threads	1.92	1024	2.72
red5	128 threads	0.45	1024	1.08
red6	128 threads	0.45	1024	1.15
red7	128 threads	0.45	1024	1.45
Comparison on GTX680:				
Thrust		0.58		
Accelerate		0.48		
Obsidian ^c	128 threads	0.45		

^a Worst parameter setting for this kernel.

^b Runtime at worst parameter setting.

^c Fastest Obsidian reduction variant.

7.2 Large Scans

There are many different ways to implement scan algorithms on a GPU (Billeter *et al.*, 2009) (Harris *et al.*, 2007). The approach implemented in the benchmark used in this section uses both reduction kernels (from Section 6.2) and scan kernels (from Section 6.3).

Below is an outline of the algorithm.

- The input array is divided into equal size chunks.
- Each chunk is reduced using a reduction kernel. This step yields a 'carry' value for each chunk.

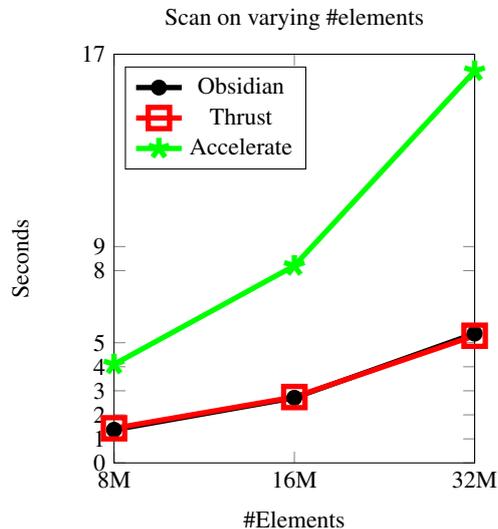


Fig. 18. The running time of scan algorithms for larger data sizes. The time reported is the sum of 1000 executions, excluding data transfer to and from the GPU memory. These numbers are collected on an NVIDIA GTX680. The presented Accelerate numbers are estimates based on a lower number of iterations as explained in Section 7.3.

- The array of carry values is scanned using a scan kernel. This step is cheap; there will be a small number of values to process here (as many as the number of chunks). However, it does require an inclusive scan kernel. Fortunately this can be implemented as a small wrapper around the already implemented scan kernels.
- The chunked input array and the array of carry values are processed by a grid of scan kernels taking a carry in. This step concludes the computation.

In the search for the scan implementation resulting in the performance numbers in figure /reffig:largescangraphs we ran a large number of experiments. The configuration space is described below:

- **threads-per-block** - The number of threads per block used by each instance of the scan kernel. This parameter takes on the values from [32, 64, 128, 256, 512, 1024]
- **Scan-kernel** - Three different scan kernels adapted for carry in.
- **Inclusive-scan-kernel** - Five different inclusive scan kernels. Varying these had little to none impact given how little data this stage operates upon.
- **total-number-of-elements** - Takes on values from [8388608, 16777216, 33554432]
- **number-of-blocks** - This parameter takes on values from [16, 32, 64, 128, 256, 512, 1024]

The total number of configurations run was 1980.

7.3 Accelerate Performance Numbers

Measuring performance of Accelerate programs turned out to not be easy. Running an Accelerate computation is done using a function called `run`, this function exposes a pure interface (using `unsafePerformIO` internally) and its operational behavior on each invocation varies. For example, at the first time an Accelerate program is `run` the CUDA compiler, `nvcc`, may be invoked to compile the skeletons used by that program (or it may hit a cache on disk), taking up to around two seconds. It is also hard to reason about exactly when copying data to and from device takes place.

In the sections above, Obsidian and Thrust reduction and scan implementations are compared together with Accelerate where the measurement of interest was running time on the GPU—compilation and data transfer excluded. To obtain these numbers for Accelerate we used the NVIDIA profiler (`nvprof`).

In the case of Obsidian and Thrust each reduction and scan was run 1000 times and the total time that took on the GPU is reported. This approach was chosen in order to get representative performance numbers in an average case. Trying to do the same in Accelerate lead to two issues. Running the program repeatedly on the same array, had the effect that the program was run only once and the result shared. Running the program on 1000 different arrays hit a problem with current Accelerate memory management, where old arrays were not freed on the GPU and ended up filling the device memory. Because of these issues the Accelerate numbers are obtained by running a smaller number of iterations and the running time for 1000 iterations is extrapolated.

8 Related work

There are many languages and libraries for GPU programming. Starting at the low-level end of the spectrum we have CUDA (NVIDIA, 2014a). CUDA is NVIDIA's name for the programming model and extended C language for their GPUs. It is the capabilities of CUDA that we seek to match with Obsidian, while giving the programmer the benefits of having Haskell as a meta programming language.

While remaining in the imperative world, but going all the way to the other end of the high-level - low-level spectrum, we have the NVIDIA Thrust Library (NVIDIA, 2014c). Thrust offers a programming model where details of GPU architecture are completely abstracted away. Here, the programmer expresses algorithms using building blocks like: *Sort*, *Scan* and *Reduce*. Thrust is designed to be agnostic of any particular parallel framework (CUDA, OpenMP, Sequential CPU etc.). It has a CUDA backend, but does not explicitly expose CUDA-specific details. Difficulties in maintaining and developing high performance kernels for use in Thrust led to the development of a lower level library called CUB (NVIDIA, 2014b), specifically for CUDA C++, and providing generic, reusable block-wide primitives. CUB is lower level than Thrust, and the two libraries can be used together. CUB and Obsidian work at similar levels of abstraction and have similar aims, we believe. It would be interesting to find out more about how CUB is being used by practitioners.

Accelerate is a language embedded in Haskell for GPU programming (Chakravarty *et al.*, 2011). The abstraction level is comparable to that of Thrust. In other words, Accelerate hides most GPU details from the programmer. Accelerate provides a set of operations (that are parallel and suitable for GPU execution, much like in Thrust) implemented as skeletons. Recent work has permitted the optimisation of Accelerate programs using fusion techniques to decrease the number of kernel invocations needed (see reference (McDonnell *et al.*, 2013)). It seems to us that when using Accelerate the programmer has no control over how to decompose his computation onto the GPU or how to make use of shared memory resources. For many users, remaining entirely within Haskell will be a big attraction of Accelerate.

The version of Obsidian described here does not try to use any compiler optimisation techniques. Instead, we are expecting that the CUDA compiler will apply a good set of techniques, from common subexpression elimination to more GPU specific transformations. The intention is to leave all important decisions in the hands of the programmer. Another options to try to build knowledge of GPU-related trade-offs into the compiler, making it more clever, and removing fine control from the programmer. This option was explored as part of a masters thesis project at Chalmers (Ulvinge, 2014). This gives a system in which many decisions are taken by the compiler. Ulvinge's work explored the use of standard compiler optimisations like loop tiling, and of program analyses to guide choices of memory access pattern. It would be interesting to further explore this approach, to find a sweet spot between full programmer control and (possibly mystifying) compiler optimisations.

Nikola (Mainland & Morrisett, 2010) is another language embedded in Haskell that occupies the same place as Accelerate and Thrust on the abstraction level spectrum. The systems above are all for flat data-parallelism, Bergstrom and Reppy are attempting nested data-parallelism by implementing a compiler for the NESL language for GPUs (Bergstrom & Reppy, 2012).

The Copperhead (Catanzaro *et al.*, 2011) system compiles a subset of Python to run on GPUs. Much like other languages mentioned here, Copperhead identifies usages of certain parallel primitives that can be executed in parallel on the GPU (such as reduce, scan and map). But Copperhead also allows the expression of nested data-parallelism and is in that way different from both Accelerate and Obsidian.

In reference (Oancea *et al.*, 2012), Oancea et al. use manual transformations to study a set of compiler optimisations for generating efficient GPU code from high-level and functional programs based on map, reduce and scan. They tackle performance problems related to GPU programming, such as bad memory access patterns and diverging branches. Obsidian enables easy exploration of decisions related to these issues.

9 Discussion

Our work on Obsidian investigates whether the benefits of functional programming can be brought to GPU kernel programmers who wish to explore a variety of possible designs in the search for high performance. Achieving high performance typically involves choosing a good algorithm that decomposes in a way that matches the structure of the GPU. The subparts will likely be individual kernels and the kernel implementor must decide on the function and memory access pattern (including input size) of each subpart. Often, it makes sense to try many different arrangements of the parts in a design exploration and parameter tuning phase. In CUDA, an important part of what one expresses is the behaviour of a single thread—how it decides what data to access based on its identity, what operations it does on that data, where it places intermediate and final results. Obsidian programs must encode the same information, but they do so by expressing the behaviour of the entire program (and how it operates on arrays) rather than by considering a single thread that will be launched many times. Because one typically expresses functions on arrays using familiar higher order functions like `map` or `zipWith`, rather than using indexing, this in itself removes a large burden of index manipulation from the programmer.

We find that Obsidian does indeed bring the benefits of functional programming to the process of writing CUDA kernels. Obsidian programming gives the kind of fine control that CUDA does, while at the same time providing abstractions that remove some of the tedium, particularly index calculations, and ease the search for high performance solutions. Our case studies demonstrate both the process of finding high performance solutions and the fact that the resulting kernels do indeed have performance comparable to NVIDIA's own Thrust library.

Some of the standard benefits of functional programming come into play when one uses Obsidian. Parameterisation eases the exploration of several possible implementations, as demonstrated in the case studies. Polymorphism makes it easy to change the type of a generated kernel with a tiny edit. These are run of the mill benefits, but they are actually important in our code generating DSL, and we feel that they should not be forgotten. And perhaps even higher order functions that capture common idioms, in this case of array programming, should be counted here. The “wrapping” of kernels to form carry chains in the large scan example is a classic example of something that is easy to do in a functional language, and much harder to do in a less expressive language.

Generating high performance CUDA code is a complex task, riddled with pitfalls. We have succeeded in doing so, while keeping the kernel specifications in Obsidian reasonably concise, through the combination of a variety of ideas. The hierarchy types (for thread, warp, block and grid) allow the same array function to be compiled in different ways, again easing the burden on the programmer. Push arrays are a novel abstraction that overcome some of the weaknesses of the much more standard pull (or delayed) arrays, while also guaranteeing fusion. Pull arrays are easy to understand and it is easy to implement the standard library functions (such as `map`, `zip`, `zipWith` and `permutations`) on them. Push arrays are harder to grasp, but they offer fine control to the programmer. Without them, we would not have been able to achieve satisfactory performance of generated kernels. One might wonder why we don't just give the programmer pull arrays and use program transformations to produce good loop structures in any case. Our choice has been to leave the programmer completely and firmly in control. There are no surprises.

The ease of writing functions like `coalesce` that control memory access patterns is also of central importance, especially when combined with the assistance that Obsidian provides in the layout management of CUDA shared memory arrays. The idea of virtualization as a way to hide hardware-related constraints (such as number of threads per block) from the user is simple. But it is actually quite hard to convey what a relief it is to the user! Each such easing of the programmer's burden frees up intellectual capacity for the quest for high performance. A final part of the puzzle is the escape hatch to lower level programming that Obsidian provides. The net effect is that Obsidian is now (finally) a good vehicle for those who wish to produce high performance CUDA code, enabling both fine control of the generated code and easing the necessary parameter tuning (Svensson *et al.*, 2014).

The capabilities of the GPU are changing and evolving. For example, it is now possible to do warp-local computation that exchange values between threads using a set of `shuffle` instructions. These kernels do not need to use shared memory to the same extent as the ones we generate. It would be interesting to try to incorporate these capabilities in Obsidian, especially since we already have the warp abstraction.

The programming idioms used in the large scan example, including the “wrapping” of kernels to make carry chains between them, are suggestive of some more general constructs. It would be interesting to explore a layer above Obsidian that documents and encodes as combinators many standard constructions in GPU kernel programming.

10 Conclusion

Obsidian lends itself well to the kind of experimentation with low-level GPU details that allow for the implementation of efficient kernels. This is illustrated in Section 6.2. The case study also shows how we can compose kernels and thus *reuse* prior effort.

The use of GPU-hierarchy generic functions makes the kernel code concise. The hierarchy generic and specific functions provide an easy way to control placement of computation onto levels of the hierarchy. The typing-design used to model the GPU hierarchy also rules out many programs that we cannot efficiently compile to the GPU.

While other approaches to GPU programming in higher level languages deliberately abstract away from the details of the GPU, we persist in our aim of exposing architectural details of the machine and giving the programmer fine control. This is partly because trying to provide simple but effective programming idioms is an interesting challenge. More importantly, we are fascinated by the problem of how to assist programmers in making the subtle algorithmic decisions needed to program parallel machines with programmer-controlled memory hierarchies, and exotic constraints on memory access patterns. This problem is by no means confined to GPUs, and it is both difficult and pressing.

Acknowledgments

Push arrays were invented by Koen Claessen. The implementation of push arrays in Obsidian is targeted at GPUs and restricted compared to Koen’s more general idea. Koen has also been a source of important insights and tips that have improved this work greatly.

We thank Henning Thielemann, Josef Svenningsson and Trevor McDonell for a lot of great feedback on the Obsidian implementation.

This research has been funded by the Swedish Foundation for Strategic Research (which funds the Resource Aware Functional Programming (RAW FP) Project) and by the National Science Foundation award 1337242.

References

- Axelsson, Emil, Claessen, Koen, Sheeran, Mary, Svenningsson, Josef, Engdal, David, & Persson, Anders. (2011). The Design and Implementation of Feldspar: an Embedded Language for Digital Signal Processing. *Proc. 22nd International Conference on Implementation and Application of Functional Languages, IFL’10*. Springer Verlag.
- Bergstrom, Lars, & Reppy, John. (2012). Nested data-parallelism on the GPU. *Pages 247–258 of: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming, ICFP’12*. ACM.
- Billeter, Markus, Olsson, Ola, & Assarsson, Ulf. (2009). Efficient stream compaction on wide SIMD many-core architectures. *Proceedings of the Conference on High Performance Graphics*. ACM.
- Blelloch, Guy. (1996). Programming Parallel Algorithms. *Communications of the ACM*, **39**(3).

- Catanzaro, Bryan, Garland, Michael, & Keutzer, Kurt. (2011). Copperhead: Compiling an Embedded Data Parallel Language. *Pages 47–56 of: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*. PPOPP '11. ACM.
- Chafi, Hassan, Sujeeth, Arvind K., Brown, Kevin J., Lee, HyoukJoong, Atreya, Anand R., & Olukotun, Kunle. (2011). A Domain-specific Approach to Heterogeneous Parallelism. *Pages 35–46 of: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*. PPOPP '11. ACM.
- Chakravarty, Manuel M.T., Keller, Gabriele, Lee, Sean, McDonell, Trevor L., & Grover, Vinod. (2011). Accelerating Haskell Array Codes with Multicore GPUs. *Proceedings of the sixth workshop on Declarative Aspects of Multicore Programming*. DAMP'11. ACM.
- Claessen, Koen, Sheeran, Mary, & Svensson, Bo Joel. (2012). Expressive Array Constructs in an Embedded GPU Kernel Programming Language. *Proceedings of the 7th workshop on Declarative Aspects and Applications of Multicore Programming*. DAMP '12. ACM.
- Elliott, Conal. (2003). Functional Images. *The Fun of Programming*. “Cornerstones of Computing” series. Palgrave.
- Elliott, Conal, Finne, Sigbjørn, & de Moor, Oege. (2003). Compiling Embedded Languages. *Journal of Functional Programming*, **13**(2).
- Guibas, Leo J., & Wyatt, Douglas K. (1978). Compilation and Delayed Evaluation in APL. *Pages 1–8 of: Proceedings of the 5th Acm SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '78. New York, NY, USA: ACM.
- Harris, Mark. (2007). *Optimizing parallel reduction in CUDA*. "<http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf>".
- Harris, Mark, Sengupta, Shubhabrata, & Owens, John D. (2007). Parallel Prefix Sum (Scan) with CUDA. Nguyen, Hubert (ed), *GPU Gems 3*. Addison Wesley.
- Holk, Eric, Byrd, William E, Mahajan, Nilesh, Willcock, Jeremiah, Chauhan, Arun, & Lumsdaine, Andrew. (2012). Declarative Parallel Programming for GPUs. *Pages 297–304 of: Applications, Tools and Techniques on the Road to Exascale Computing, Proc. ParCo 2011*. Advances in Parallel Computing. IOS Press.
- Keller, Gabriele, Chakravarty, Manuel M.T., Leshchinskiy, Roman, Peyton Jones, Simon, & Lippmeier, Ben. (2010). Regular, shape-polymorphic, parallel arrays in Haskell. *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ICFP'10. New York, NY, USA: ACM.
- Mainland, Geoffrey, & Morrisett, Greg. (2010). Nikola: Embedding Compiled GPU Functions in Haskell. *Pages 67–78 of: Proceedings of the third ACM Haskell Symposium*. ACM.
- McDonell, Trevor L., Chakravarty, Manuel M.T., Keller, Gabrielle, & Lippmeier, Ben. (2013). Optimising Purely Functional GPU Programs. *18th ACM SIGPLAN International Conference on Functional Programming*. ICFP'13. ACM.
- NVIDIA. (2014a). *CUDA C Programming Guide*.
- NVIDIA. (2014b). *NVIDIA CUB Library*.
- NVIDIA. (2014c). *NVIDIA Thrust Library*.
- Oancea, Cosmin E., Andreetta, Christian, Berthold, Jost, Frisch, Alain, & Henglein, Fritz. (2012). Financial software on GPUs: between Haskell and Fortran. *Pages 61–72 of: Proceedings of the 1st ACM SIGPLAN workshop on Functional High-Performance Computing*. FHPC'12. New York, NY, USA: ACM.
- Stevens, Roger T. (1989). *Fractal Programming in C*. M&T Books.
- Svenningsson, Josef, & Axelsson, Emil. (2013). Combining Deep and Shallow Embedding for EDLSL. *Pages 21–36 of: Loidl, Hans-Wolfgang, & Pea, Ricardo (eds), Trends in Functional Programming*. Lecture Notes in Computer Science, vol. 7829. Springer Berlin Heidelberg.

- Svenningsson, Josef, & Svensson, Bo Joel. (2013). *Simple and Compositional Reification of Monadic Embedded Languages*. 18th ACM SIGPLAN International Conference on Functional Programming, ICFP 2013.
- Svenningsson, Josef, Svensson, Bo Joel, & Sheeran, Mary. (2013). Efficient Counting and Occurrence Sort for GPUs using an Embedded GPU Programming Language. *2nd ACM SIGPLAN Workshop on Functional High-Performance Computing*. FHPC'13. ACM.
- Svensson, Bo Joel, Sheeran, Mary, & Newton, Ryan R. (2014). Design Exploration Through Code-generating DSLs. *Commun. acm*, **57**(6), 56–63.
- Ulvinge, Niklas. (2014). *Increasing programmability of an embedded domain specific language for GPGPU kernels using static analysis*. MSc thesis, Chalmers University of Technology.