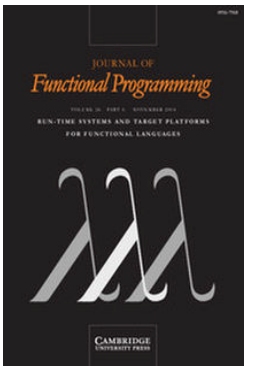


## A language for hierarchical data parallel design-space exploration on GPUs


BO JOEL SVENSSON, RYAN R. NEWTON and MARY SHEERAN



Journal of Functional Programming / Volume 26 / 2016 / e6
DOI: 10.1017/S0956796816000046, Published online: 17 March 2016

*JFP* **26**, e6, 53 pages, 2016.  
doi:10.1017/S0956796816000046

# *A language for hierarchical data parallel design-space exploration on GPUs*

BO JOEL SVENSSON and RYAN R. NEWTON

*Indiana University, School of Informatics and Computer Science, Bloomington, IN, USA*
(*e-mails:* bo.joel.svensson@gmail.com, rrnewton@indiana.edu)

MARY SHEERAN

*Chalmers University of Technology, Department of Computer Science and Engineering, Gothenburg, Sweden*
(*e-mail:* ms@chalmers.se)

**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 optimise 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 optimising compiler to make the aforementioned decisions, synthesising 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;

McDonell *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 composed 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, when using Accelerate, 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, Harris (2007) shows detailed, step by step, optimisation of reduction kernels in CUDA. 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, synchronisation 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.

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 *virtualisation* 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, 2015c). We hope that readers will feel inclined to experiment with the design and implementation of such kernels.[1]

### *1.1 Contributions*

This paper presents Obsidian in its current form. During the development of Obsidian, the following contributions have been made:

- *High-/low-level*: Obsidian started out based on earlier work in hardware design using functional languages. The idea was to implement a GPU programming language that was very similar to the hardware description language Lava (Bjesse *et al*., 1998) in order to test if a programming model similar to that of structural hardware design is suitable for GPU kernel implementation. The similarities to Lava are most apparent in early work on Obsidian (Svensson *et al*., 2008; Svensson *et al*., 2010). However, we have discovered that GPU programming differs from structural hardware design in that the target platform greatly constrains the shape of suitable programs. In particular, the user must control how an algorithm is divided up into non-communicating tasks, each of which can be tackled by a block of threads that can synchronise. He must also control the use of different forms of memory with different well-functioning access patterns. While we retain a Lava-like emphasis on higher order functions like reduce and scan, we have had to provide them in many more guises than we originally thought would be necessary. An example of this is the provision of *sequential* versions of such common combinators, which can be important to controlling task size and thus achieving high performance. Obsidian is unique among the functional DSLs for GPU programming in giving the user abstractions that permit very fine control both of the division of an algorithm into parallel and sequential sub-parts and of the deployment of the resulting tasks onto the GPU.
- *Push arrays*: Push arrays were first implemented as part of Obsidian (Claessen *et al*., 2012). They are a complement to the more standard delayed (or pull) arrays that are typically used in embedded languages. The combination of pull and push arrays allows for the generation of better code in embedded DSLs. Push arrays will be explained in more detail in Section 4.4.

[1] Download Obsidian at github. `www.github.com/svenssonjoel/Obsidian`

- *Monad reification*: Obsidian uses a simple monad reification technique (observing the underlying structure of a monadic computation) (Svenningsson & Svensson, 2013). At the same time, other Embedded Domain Specific Language (EDSL) implementors were also working on monad reification, leading to a number of independent solutions (Persson *et al*., 2012; Sculthorpe *et al*., 2013).
- *Hierarchy-level polymorphism*: The hierarchy polymorphism and type-level tagging of operations and arrays are a fairly new addition to Obsidian. The type-level modelling of the GPU hierarchy is touched upon in Svensson *et al*. (2014), but explained in more detail here. Note that the type level indices used by Obsidian are different from those in the Repa library (Keller *et al*., 2010). In Repa, an array is labelled differently, at the type level, depending on whether it is internally represented as a delayed or manifest array.

## 2 Background: The GPU and CUDA

Obsidian targets NVIDIA GPUs supporting CUDA (NVIDIA, 2015a), 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 1,536 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 shared local memory 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 elements 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 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 kernel implemented in this example reduces sub-arrays of a given input array. This reduction of sub-arrays can be used in the implementation of reduction of millions of elements as outlined in Section 7.1. In Obsidian, arrays are either of type `Pull s a` or `Push t s a`. The parameter `s` is the type of the array's length, and `a` its element type. The `t` parameter indicates at which level of the GPU hierarchy the array is computed. These array types are explained in more detail in Sections 4.2 and 4.4.

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 defined 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)
       → Pull Word32 a
       → Program t (Push t Word32 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. 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. Choosing 512 elements is just an example, any power of two works as long as the data fits in shared memory.

```
reductions :: Data a
           ⇒ (a → a → a)
           → Pull EWord32 a → Push Grid EWord32 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 dynamically linked in function. At capture, the number of threads to use per CUDA Block is specified, in this case 64. Obsidian will generate different code depending on the numeric parameter passed to capture. That is, capture 64 and capture 128 generate code specialised for either 64 or 128 threads. Any number is acceptable as the parameter to capture as long as it corresponds to a number of threads per block admissible by the GPU, currently between 1 and 1,024. The CUDA code that the kern handle refers to has been compiled for the specific array size and number of threads specified. In this case, kern operates on subarrays of length 512. The functions useVector and allocaVector are used to allocate space for the input and output of the kernel. 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 virtualisation.

Running the perform function prints the result:

```
[130816,392960]
```

The withCUDA function is the run function of a monad called CUDA that, together with its associated capture and <== functions, gives the programmer a way of running programs on the GPU directly from within a Haskell program. CUDA code is generated into files called gen0, gen1 and so on, located in the working directory. These files are not deleted after being compiled and dynamically loaded into the running executable, but rather left for inspection by the programmer.

## 4 Obsidian programming model: Overview and discussion

Obsidian is an EDSL, implemented in Haskell. When an Obsidian program is run, 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. Elliott (2003) presents an excellent introduction to compiling embedded languages.

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; these expressions are 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. This is a convention shared by several Haskell EDSLs and comes from an unfortunate definition of the standard Haskell `Ord` and `Eq` classes. These classes require that the result type of `<` and `==` is a `Bool`. In Obsidian, the equality operator resulting in an `EBool` is `==*`, while the standard `==` is still available for use in the meta-language.

There are restrictions on the elements used in an Obsidian program. These restrictions are expressed using the 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.

### *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 EDSLs (with the same representation being 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 subcomputations 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
```

Below is an example of a function on pull arrays. The `halve` function takes a pull array as input and returns a pair of pull arrays. The `!` operator represents indexing into a pull array.

```
halve :: (Integral i, ASize l) ⇒ i → Pull l a → (Pull l a, Pull l a)
halve arr = (Pull n2 (λix → arr ! ix),
             Pull  (n - n2) (λix → arr ! (ix + n2)))
  where n = len arr
        n2 = n `div` 2
```

This `halve` function creates two new views on the array passed in. The views index into the given array in slightly different ways. This is typical of operations that take pull arrays apart.

### *4.3 Programs*

The `Program t a` 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 takes as input an array, here `a Word32 e`. The `a` type can be either `Pull` or `Push t`. The `Push` array type is implemented in terms of programs and is thus defined in the following section. The elements of this input array are constrained by the `Data e` constraint. The `compute` function's return value of type `Program t`, 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 (Claessen *et al*., 2012). 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 individually to a *writer* function. Thus, the push-function is itself higher order:

```
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.

```
-- | 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 an 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.

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 $ λwf →
    forAll (sizeConv n) $ λi →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 decision on the number of blocks and the number 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.

Since push arrays can be unintuitive, three examples are shown below. The `iota` function creates a push array with value $i$ at index $i$. The `aMap` function maps a function over a push array. And finally `ixMap` applies an index transformation to the array.

```
iota s = Push s $ λwf →
    do
      forAll (sizeConv s) $ λix → wf ix ix

aMap   f (Push s p) = Push s $ λwf → p (λe ix → wf (f e) ix)

ixMap  f (Push s p) = Push s $ λwf → p (λe ix → wf e (f ix))
```

The `iota` function iterates from 0 to $s - 1$ and at each index, `ix`, pushes the value, `ix`, through the write-function. Mapping a function `f` over the elements of a push array is done by creating a new push array. The new push array uses the existing array's push-function but alters the write function to first apply the function `f` to the elements. Mapping an indexing transformation works in a way very similar to `aMap`, but, instead, the transformation is applied to the index.

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 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.

### *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. A pull array supports 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. The 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 `(map f) . (map g)` which 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 `compute`d, 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, `t`, decides how it is computed. For example, if `t` is `Thread`, a sequential loop is instantiated (with trip count equal to array length); whereas, if `t` 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`. Tools for reasoning about the cost of indexing into pull arrays are not currently available to the programmer. One alternative is to track at the type level whether or not this array is manifest, as Repa does (Keller *et al.*, 2010). The provision of some means to estimate costs of computations is left as future work.

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)
               $ λix →ifThenElse (ix <* sizeConv n1)
                       (a1 ! ix)
                       (a2 ! (ix - sizeConv n1))
    where
      n1 = len a1
      n2 = len a2
```

Computing a pull array that is the result of an append 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 pattern 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) $ λwf →
      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 write-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. Notice that the `sumUp` function completely unrolls the recursion, creating an expression tree representing the summation of the array. This means that the length of the array must be known when evaluating the meta-program. It is not possible to implement a variant of `sumUp` that takes an input of type `Pull EWord32 EWord32`.

The code generated from the `sumUp` function defined 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 change affects 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 the 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 | Sequential execution |
| Warp | Yes | Yes | Lock-step execution |
| Block | Yes | Yes | Yes |
| Grid | Yes | No | No |

At levels `Thread`, `Warp` and `Block`, programs are also limited by the size of shared memory. That means that any intermediate array storage has to fit within shared memory. Shared memory size varies between GPU models, but 48 kB per multiprocessor is common. At the Grid level, the program's memory usage is limited by the global memory, usually a few gigabytes.

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 restriction is captured by a type class `*<=*`.

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

The `Compute` class has an instance for all levels in the hierarchy that support storing intermediate data in shared memory and synchronising. Storing in shared memory can be done at 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 the `Thread` level, the synchronisation points are simply implemented as sequential composition. On the `Warp` level, there is an extra caveat: the arrays operated upon should be marked as volatile, in the generated CUDA. If the array is not marked as volatile there is a chance that, on some CUDA architectures, the values will be kept in registers as an attempted optimisation. Obsidian marks arrays that are used within a warp as volatile to ensure that the data is consistent across threads.

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 Section 3, whose type is shown below. From now on, we make use of the `SPull` and `SPush` type aliases:

```
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, 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 increases the complexity of the type of `reduce2stage`. A less polymorphic version might specialise 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 array using the `splitUp` function. These arrays are then distributed over the parallel resources and computed on individually. Each distributed computation results in a pull array of either push arrays or programs.

If the function being mapped over the inner arrays uses shared memory, its result will be a value of type `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 forming a grid, for example. A pull of pull array would imply that any thread in the grid can access any element, but 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 to 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.

## 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`

```
-- 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.

to be *more polymorphic* than its counterpart in many DSLs that return 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'')
```

```
-- | 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 (t ~ t1) ⇒ ExecProgram t (Push t1) where

-- | Exec variants for specific levels of the hierarchy.
--   Less need for annotation in application code
execThread :: (ExecProgram Thread a, Data e)
        ⇒ Program Thread (a Word32 e)
        → Push Thread Word32 e
execThread = exec

execBlock :: (ExecProgram Block a, Data e)
          ⇒ Program Block (a Word32 e)
          → Push Block Word32 e
execBlock = exec


execWarp :: (ExecProgram Warp a, Data e)
         ⇒ Program Warp (a Word32 e)
         → Push Warp Word32 e
execWarp = exec

-- Auxiliary
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.

`BinOp` and `UnOp` provide 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 synchronisation). 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 `t`, 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, `t`, in parallel at the level directly above it, `Step t`.

The `Bind` and `Return` constructors enable a `Monad` instance for `Program t`; 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` datatype is converted to an intermediate representation based on lists. This intermediate representation makes hierarchy-level information concrete.

**2A Liveness Analysis:** The intermediate representation is analysed to discover the live ranges of arrays in shared memory. This stage annotates the intermediate representation with liveness information, which keeps track of where an array is created and at what point it can be freed.

**2B Memory Mapping:** The annotated intermediate representation 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 intermediate representaton are compiled into CUDA. This is where virtualisation 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
```

The example `prg1` takes just one input. Programs with more than one input array are, however, permitted as well. Reifying this program is as simple as applying it to a *named array* in global memory: `(Pull n (λ ix → Index ("input",[ix])))`.

The function then yields its push array result. That push array, in turn, is a `Program` parameterised on a write-function. Providing a write-function `(λ a ix → Assign "output" [ix] a)`, which writes to a named (global) array, completes reification, yielding a `Program` AST.

The resulting `Program` AST is converted into a representation where the `t` parameter is made concrete. The new representation is called `IM`.

```
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 t b
```

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.

```
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 48 kB). When using the `capture` function to compile an Obsidian program, the GPU device is queried for the amount of shared memory and number of memory banks; 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 comes into existence. A standard analysis computes the full live range of each array:

- Step through the 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 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. Each “malloc” request is serviced

with the first available memory segment of sufficient size. The maximum size ever used is tracked, and in the end this number is the total amount of shared memory needed for the kernel. Memory is 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. Exhausting shared memory can 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 the case-studies presented in this paper, 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 optimial. Improving the memory management system is left as future work. Even though the memory allocation process is not optimal, the memory layout it generates does not affect performance of the kernel, since arrays will not spill to global memory. However, it may waste memory and disallow a kernel that would compile had the memory layout been optimal.

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 virtualisation must be addressed. Most cases of this compilation are very simple, as many statements correspond directly to their CUDA counterparts. For example, an assignment statement, `SAssign nameE [ixE] e`, is compiled into a C statement of the form `arr_n[i] = j`, where i and j are the results of recursively translating the expressions `ixE` and `e`.

The interesting cases are those that deal with parallelism, such as the `SForAll` and `SDistrPar` statements. For example, compiling a parallel-for, `SForAll`, over threads in a block has the structure outlined in pseudo code below:

```
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*realThreads + threadIdx.x;
            body
          }
    goR = -- run the last r threads
          if (threadIdx.x < r) {
            ...
          }
```

Compilation of `DistrPar` performs a similar technique for the virtualisation 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 maximise 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 in the NVIDIA Thrust and Accelerate libraries. Accelerate is a much higher level DSL than Obsidian, but one with hand-tuned (though 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 Table 1.

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 $512\times512$ 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

Table 1. *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 vary the number of threads-per-block, while the rows vary image size (square images). Each benchmark was executed 1,000 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 |

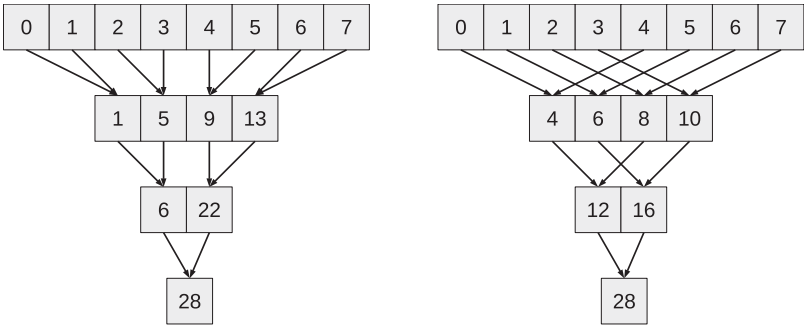


Fig. 7. *Left:* `evenOdds - zipWith` reduction, leads to uncoalesced memory accesses. *Right:* `halve - zipWith` reduction, leads to coalesced memory accesses. This coalescing is most important during the very first phase, when data is read from global memory.

means that images larger than 1024 pixels wide are possible (where 1,024 is the hardware limit on the 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.

### *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 be 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

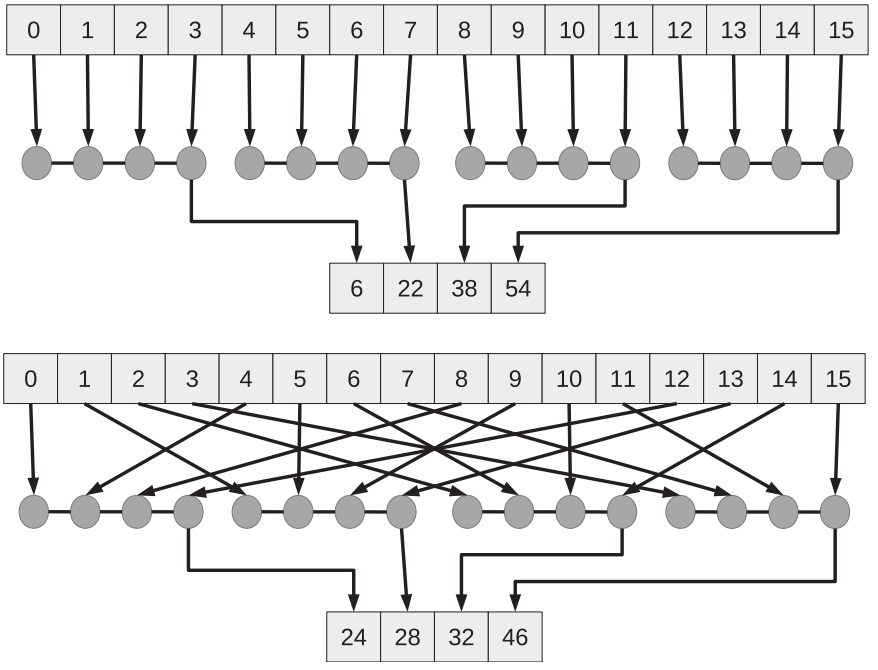


Fig. 8. *Top:* **BAD** Adding sequential reductions like this reintroduces memory coalescing issues. Consecutive threads no-longer access consecutive memory locations. *Bottom:* **GOOD** Using sequential reduction but maintaining coalescing.

implementation strategies and tuning parameters. While we only varied **threads-per-block** in the last subsection, in this subsection 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** - Seven 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. Figure 9 and Table 2 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 7. 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 (Table 2), due 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 elements 0 and 1, thread one accesses elements 2 and 3. It would have been much better if thread $i$ accessed element $i$ and $n + i$, for some stride $n$, which brings us to the next reduction kernel.

### 6.2.2 Reduction 2: Recursively halve and combine

The `red2` kernel, 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 7). 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 $ singleton $ f (arr ! 0) (arr ! 1)
  | otherwise =
    do
      let (a1,a2) = halve arr
      arr' ← compute (zipWith f a1 a2)
      red3 f arr'
```

This kernel cuts recursion off at length 2 and when the array reaches that length, the remaining two elements are combined directly using `f`. Performing this cutoff at two elements does not change the overall depth of the algorithm, but, since there is no call of `compute` in the last stage, the result will not be stored in shared memory. This optimisation has a very small 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 eight 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 8)—to the detriment of performance (Table 2).

#### 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
```

The coalesce function 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 specifies 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 n, we can push the tradeoff between the number of threads and the sequential work-per-thread further. The kernels 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 **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
```

### 6.3 Case study: Scan

In the scan case study, we compute all the prefix sums of a sequence of values using a binary associative operator (where scan is familiar to Haskell programmers as the

Table 2. *The numbers reported in Figure 9 represent the best parameter settings found. These settings are difficult to predict in advance. This table shows the best* **threads-per-block** *for each of the reduction kernels. Some of the kernels use virtualised threads and are highlighted. 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 |

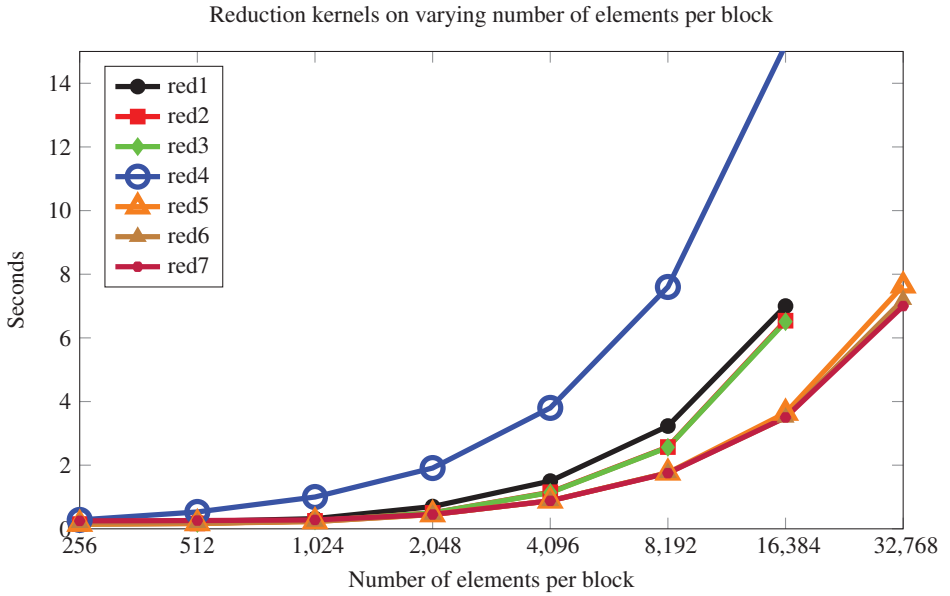


Fig. 9. The running time of 8,192 blocks executing a reduction kernel. The time reported is the sum of 1,000 executions of the 8,192 blocks grid, excluding transfer time of data to GPU memory. The $X$-axis varies **elements-per-block**, but each point represents the *best setting* for **threads-per-block**. These numbers are collected on an NVIDIA GTX680.

`scanl1` function). Given an array of values $a_0, a_1, \ldots, a_n$ and associative operator $\oplus$, the scan operation computes a new array:

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

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

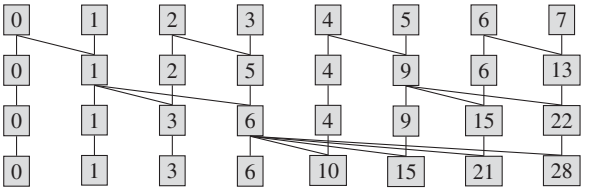


Fig. 10. Sklansky parallel prefix network.

- **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** - We use five different scan kernels. Three of the scan variants are based on the Sklansky network and two on the Kogge–Stone construction.

The total number of configurations is in this case 150. The results of these experiments are presented in Figure 12 and Table 3.

### *6.3.1 Sklansky scan*

We start by implementing a parallel prefix network attributed to Slansky (1960). This network follows a simple divide and conquer decomposition as shown in Figure 10. 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 thread must have a conditional to decide whether to be a copy or an 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 10, using as many threads as there are elements. Note that thread virtualisation 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
```

The `sklansky` function 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 may produce an unexpected result if the array used as input has delayed operations on it. The delayed operations will be replicated into each split of the array. In the context above, `unsafeBinSplit` is safe.

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
```

The function `fan` splits an array in the middle and combines the last element of the first half with each of the elements of the second half, using the operator `op`. This `fan` behaviour can be seen at each level in Figure 10. The `append` used in this function leads to conditionals in the generated code. In essence, each thread will execute a conditional asking "am I a copy thread or an operation thread?". These kinds of conditionals are particularly bad when threads within the same warp take different branches.

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. Notice 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.

```
phase :: Int
      → (a → a → a)
      → Pull Word32 a
      → Push Block Word32 a
phase i f arr =
  mkPush l (λwf → forAll sl2 (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
    sl2 = 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)]]
```

The `compose` function sequentialises 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 difference indicates that tweaking the way data is loaded into shared memory may be beneficial in that kernel.

The `sklansky3` code below improves the situation by breaking out a separate load stage that reads data in a coalesced way.

```
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
```

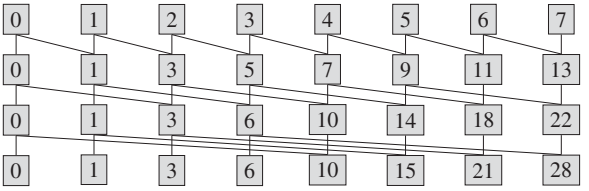


Fig. 11. Kogge–Stone parallel prefix network.

Here, we use `load 2` to realise loading of two 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.

```
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 12.

### 6.3.2 *Kogge–stone scan*

Figure 11 illustrates another approach to computing scan. The figure shows three stages. 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. In general, at stage $n$ the number of elements dropped is $2^{n-1}$. 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
```

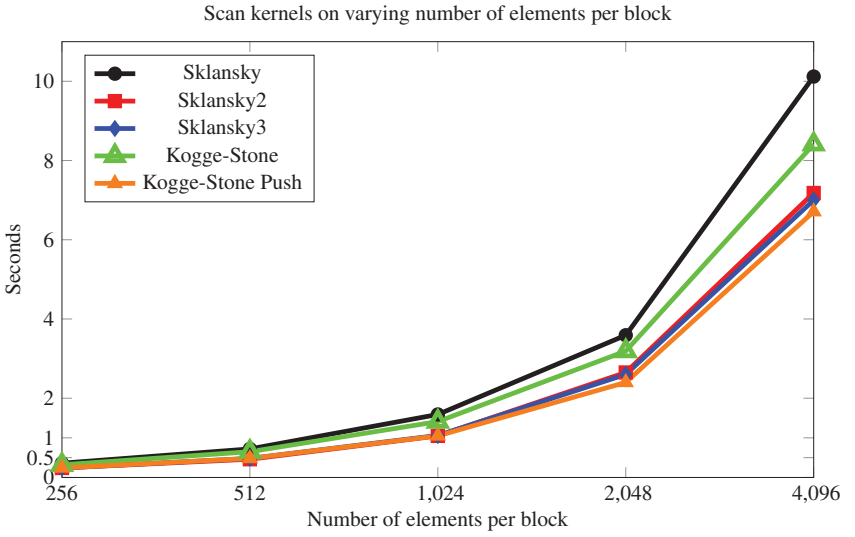


Fig. 12. The running time of 8,192 blocks executing scan kernel. The time reported is the sum of 1,000 executions of the grid of 8,192 blocks, excluding transfer time of data to GPU memory. The number of elements processed per block varies over the $X$-axis. Again, only the best threads-per-block setting at each elements-per-block is shown. These numbers are collected on an NVIDIA GTX680.

The Obsidian `zipWith` behaves the same as the standard `zipWith` on Haskell lists. That is, if the two inputs are of different length, the result has a length equal to the shortest of the inputs.

The implementation of `ksLocal` uses concatenation 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 `append` oped
```

Figure 12 shows a performance comparison of the scan kernels implemented here. The Kogge–Stone variant using push array concatenation was the fastest.

### *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

Table 3. *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 |

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 instances, and even more elements can be scanned per block.

This sequencing of parallel work, with a carry, can be implemented using a function called `sMapAccum` implemented in terms of lower level Obsidian functions. The `sMapAccum` function sequentially maps a computation over sub-arrays, while accumulating a small amount of state, `acc`, between iterations.

```
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
```

The `wrapKernCin` function takes a `ScanKernel`, a type alias that matches the type of the scan kernels, and transforms it into a kernel that also takes a carry-in value. The carry-in value is combined with the first element of the input before passing it to the wrapped scan kernel. The carry out is found in the result array at the last index.

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 scan algorithms implemented in this way perform on par with hand-tuned code as can been 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 behaviour 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, reductions are used to demonstrate the stitching together of 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 shows an example of how to invoke GPU computations from Haskell. Table 4 shows the running time for the above program executing a $2^{24}$ element reduction; in that figure and in Figure 13, we compare Obsidian against NVIDIA Thrust and Accelerate.

The evaluation of large reduction algorithms is done here in two different ways. First, in Table 4, 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 Table 4:

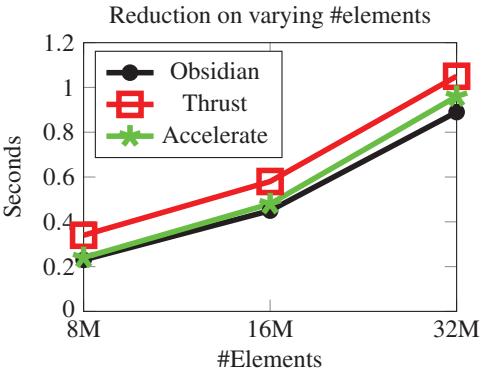


Fig. 13. The running time of reduction algorithms for larger data sizes. The time reported is the sum of 1,000 executions, excluding data transfer to and from the GPU memory. These numbers are collected on an NVIDIA GTX680. The presented Accelerate numbers are estimates based on a lower number of iterations as explained in Appendix A. The Obsidian numbers presented here come from the run with the parameter settings that performed the best.

- **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]`
- **reduction-kernel** - Seven different reduction kernels are used, (red1 to red7).
- **total-number-of-elements** - Always 16,777,216
- **number-of-blocks** - Always 4,096

In total, there are 42 different configurations.

In Figure 13, we try a different approach. The kernel used is kept constant, while block size, total number of elements and threads per block vary. The following list defines the configuration space:

- **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]`.

In total, there are 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.

### *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

Table 4. *Running times of $2^{24}$ (16M) element reduction using Obsidian. The results were obtained on an NVIDIA TESLA c2070 and on the GTX680. Each reduction procedure was executed 1,000 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* | Seconds† |
|---|---|---|---|---|
| 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‡ | 128 threads | 0.45 | | |

*Worst parameter setting for this kernel.
†Runtime at worst parameter setting.
‡Fastest Obsidian reduction variant.

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.
- 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 14, we ran a large number of experiments. The configuration space is described below:

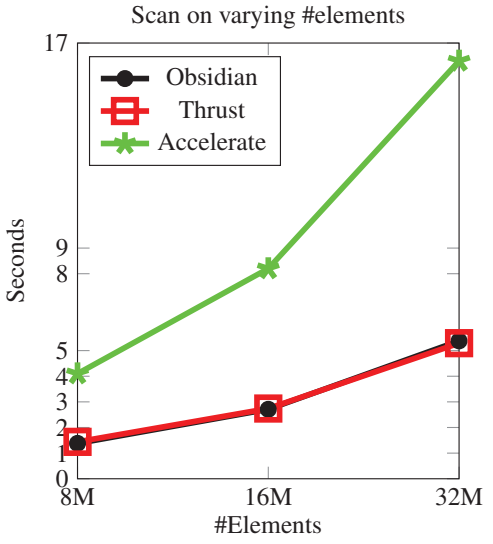


Fig. 14. The running time of scan algorithms for larger data sizes. The time reported is the sum of 1,000 executions, excluding data transfer to and from the GPU memory. These numbers are collected on an NVIDIA GTX680. The presented Accelerate numbers are estimates based on a lower number of iterations as explained in Appendix A.

- **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 1,980.

## 8 Evaluation

In this section, we evaluate the results obtained in the various case studies from Sections 6 and 7. The examples in Section 6 concern single kernels, either as a complete application (Mandelbrot) or as a building block for solving larger problems (Reduction and Scan). In Section 7, these building blocks are combined, via several kernel launches on the GPU, in order to perform reduction or scan of millions of elements.

The GPUs used to obtain performance measurements are the NVIDIA TESLA c2070 and the GTX680. The GTX680 is not used to drive any display, which could potentially introduce noise in the benchmark results. The TESLA c2070 is a compute capability 2.0 GPU with 14 multiprocessors each containing 32 CUDA cores for a

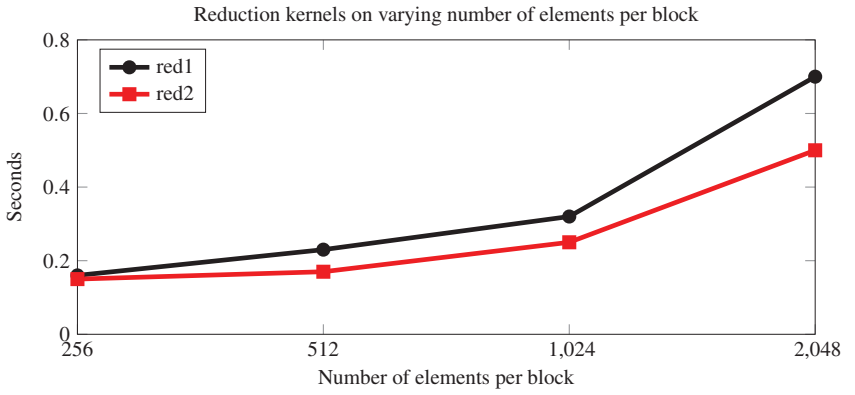


Fig. 15. This chart shows a comparison of `red1` and `red2` from Figure 9. The chart shows the impact of changing memory access pattern.

total of 448 CUDA cores. The GTX680 GPU supports compute capability 3.0 and has a total of 1,536 CUDA cores split over eight multiprocessors.

### *8.1 Reduction kernels*

The performance measurements in Section 6 compare different versions of kernels generated by Obsidian. The purpose of these measurements is to show the impact on performance of the changes to the Obsidian programs or of tuning the exposed parameters.

For example, Figure 15 shows a performance comparison between `red1` and `red2` . The chart shows that changing the memory access pattern improves performance, even on small sizes. From the information in Table 2, we see that the fastest kernels generated for these sizes used virtualised threads. This means that even though the GPU could run enough threads in parallel per block to compute these reductions, the fastest versions of the kernels sequentialise work per thread.

The kernel named `red4` is the first reduction kernel that explicitly sequentialises work. In this case, the sequential work is performed in a very different way compared to the implicit sequential work introduced by virtualisation. Here, each thread reads a number of elements and accumulates a sum before writing the result to shared memory. The sequential code introduced by virtualisation uses more memory, by storing intermediate results. The `red4` kernel is very slow, which can be explained by its memory access pattern. Figure 16 shows what happens when going from `red4` to `red5` in terms of running time. The `red5` code is in the group of the fastest reduction kernels generated in this case study.

### *8.2 Scan kernels*

In Section 6, two algorithms for computing prefix sums are implemented (Sklansky and Kogge–Stone). In this case, the performance increase comes from removing conditionals that take different paths on different threads within a warp. The difference between `sklansky` and `sklansky2` is mainly that `sklansky2` avoids

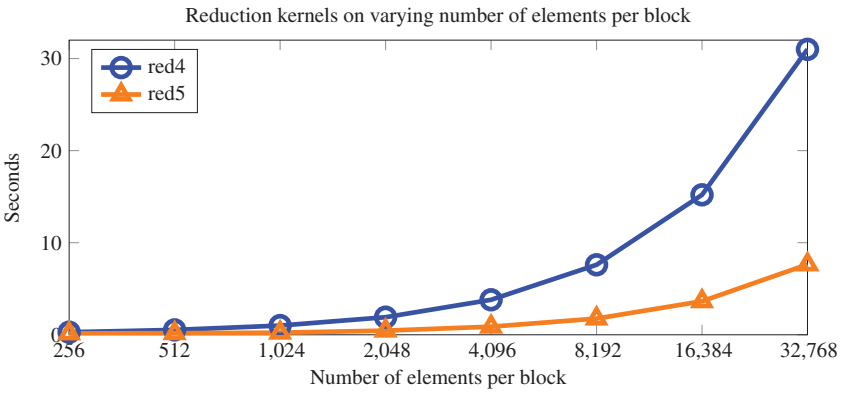


Fig. 16. This chart shows a comparison of `red4` and `red5` from Figure 9. The chart shows the impact of changing memory access pattern.

the conditionals by performing both a direct copy of an unchanged value and a combining operation on two values in each thread, in sequence. In `sklansky3`, an extra step is added where the data is read from global memory, making sure that this data is read in a more efficient, coalesced, way. This coalescing leads to a further slight improvement. The Kogge–Stone kernels, however, start out with what already looks like a fairly good memory access pattern. In Figure 12, the `kslocal` variant is faster than the original `sklansky`, most likely due to this memory access pattern. The `kslocal` kernel however has the same issue as `sklansky` when it comes to executing diverging conditionals, which in this case is fixed by switching from `append` on pull arrays to `append` on push arrays.

### *8.3 Benchmarks*

In Section 7, kernels generated using Obsidian are combined to solve larger instances of reduction and scan problems. The Obsidian implementations are compared to NVIDIA Thrust and to Accelerate.

Comparing to Thrust and Accelerate is, however, not entirely fair. The Thrust and Accelerate reduction code is applicable to arrays of any size, while code generated using Obsidian is specialised for arrays whose length is a multiple of a chunk size. This is a limitation of Obsidian and it should provide a performance advantage at those sizes it does support, since some degree of dynamic control flow can be avoided.

In Harris (2007), reduction is optimised incrementally leading to seven different kernels. In the end, a speedup of ≈30 times is obtained in comparison to the original naive reduction kernel. The steps taken to optimise the reduction kernel in CUDA are not directly comparable to our case study on reduction kernels. In Harris (2007), unrolling is applied as step 5 and 6, while unrolling is the default for all the reduction kernels implemented using Obsidian (except for loops introduced explicitly with `seqReduce` or resulting from virtualisation). In Harris (2007), two different kinds of kernel optimisations are considered, *algorithmic optimisation* and *code optimisations*. Algorithmic optimisations refer to changes of memory access pattern and amount of

work per thread and code optimisations include transformations like loop unrolling. In the NVIDIA reduction example, algorithmic optimisation gave greater payoff than code optimisation (roughly 12 times versus 2.5).

Finding the corresponding speedup values for the Obsidian case studies is difficult, because some optimisations are difficult to classify and because virtualisation can introduce loops implicitly. Which values would we compare? Table 4 shows the running time for reduction of 16 million elements; for `red1` the best found parameter settings are 2.5 times faster than the worst. Comparing the worst parameter setting for `red4` with the best parameter setting for `red7` shows a 6.2 times speedup. The greatest speedup obtained by tuning over the parameters alone is found for `red7`; it shows a difference of 3.2 times. If comparing the speedup as we apply optimisations from `red1` to `red7`, then only a 1.7 times speedup is obtained. But since the fastest variant of `red1` implicitly applies sequentialisation (via thread/block virtualisation) for its best performing parameter settings, it is not the same comparison, as in Harris (2007), between a naive reduction kernel and an optimised one.

After tuning the parameters, the Obsidian code performs on par with or slightly better than Thrust and Accelerate. The generated reduction kernels are all very fast, obtaining a throughput ranging from ≈110 GB/s (Thrust) up to ≈142 GB/s (Obsidian `red7`) on the GTX 680 with a memory bandwidth of 192 GB/s.

For scan on large input arrays, Obsidian and Thrust are equally matched, while Accelerate lags somewhat. We conjecture that the Accelerate performance could be improved by complicating the CUDA code used to implement the scan skeleton.

From the benchmark results, it is clear that being able to easily generate variants and tune over parameter settings is valuable. In Obsidian, the user has fine control over how a computation is divided up into parallel and sequential parts. The resulting ease of exploring the design space for a kernel by making syntactically small changes that radically change the behaviour of the generated code (for example, to adjust memory access patterns), while retaining a high-level description, is the main benefit of Obsidian over Thrust and Accelerate.

## 9 Related work

There are many languages and libraries for GPU programming. Starting at the low-level end of the spectrum we have CUDA (NVIDIA, 2015a). 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, 2015c). Thrust offers a programming model where details of the GPU architecture are completely abstracted away. Here, the programmer expresses algorithms using building blocks, such as: *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, 2015b), 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, but there does not yet seem to be a publication on this topic.

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 McDonell *et al*. (2013)). It seems to us that when using Accelerate, the programmer has no control over how to decompose a 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 option is 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 approach results in 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 patterns. 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, while Bergstrom and Reppy are attempting to implement nested data-parallelism in 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 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 thus different from both Accelerate and Obsidian.

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` (Oancea *et al*., 2012). They tackle performance problems related to GPU programming, such as bad memory access patterns and diverging branches. The Loo.py system (Kloeckner, 2015) takes a transformation-based approach to achieving performance, portability and productivity even further. It

provides a mechanism for user-controlled transformation of array programs, to make them more suited to data-parallel architectures such as GPUs. Both programs written specifically for Loo.py and in other languages (such as Fortran) can be transformed in this way, from Python. One possibility might be to explore the use of Loo.py as a backend for a Haskell-based higher level language.

## 10 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 the programmer 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 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 provide a novel abstraction, overcoming 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 straightforward 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 whenever possible. 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 virtualisation 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 computations 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 into 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.

## 11 Conclusion

Obsidian lends itself well to the kind of experimentation with low-level GPU details that results in the generation of efficient kernels. This prototyping aspect is illustrated in Section 6.2. The case studies also show how programmers 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 at exposing architectural details of the machine and giving the programmer fine control. This is partly because trying to provide simple programming idioms that permit control of low-level details relating to fine details of the GPU 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. We also thank the anonymous reviewers for the careful, detailed and thought provoking reviews.

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, E., Claessen, K., Sheeran, M., Svenningsson, J., Engdal, D. & Persson, A. (2011) The design and implementation of Feldspar: An embedded language for digital signal processing. In Proceedings of 22nd International Conference on Implementation and Application of Functional Languages, IFL'10. Berlin Heidelberg: Springer-Verlag, pp. 121–136.

Bergstrom, L. & Reppy, J. (2012) Nested data-parallelism on the GPU. In Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming. ICFP'12. New York, NY, USA: ACM, pp. 247–258.

Billeter, M., Olsson, O. & Assarsson, U. (2009) Efficient stream compaction on wide SIMD many-core architectures. In Proceedings of the Conference on High Performance Graphics. HPG '09. New York, NY, USA: ACM, pp. 159–166.

Bjesse, P., Claessen, K., Sheeran, M. & Singh, S. (1998) Lava: Hardware design in Haskell. In Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming, ICFP'98. New York, NY, USA: ACM, pp. 174–184.

Blelloch, G. (1996) Programming parallel algorithms. *Commun. ACM* **39**(3), 85–97.

Catanzaro, B., Garland, M. & Keutzer, K. (2011) Copperhead: Compiling an embedded data parallel language. In Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming. PPoPP '11. New York, NY, USA: ACM, pp. 47–56.

Chafi, H., Sujeeth, A. K., Brown, K. J., Lee, H., Atreya, A. R. & Olukotun, K. (2011) A Domain-specific Approach to Heterogeneous Parallelism. In Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming. PPoPP '11. New York, NY, USA: ACM, pp. 35–46.

Chakravarty, M. M. T., Keller, G., Lee, S., McDonell, T. L. & Grover, V. (2011) Accelerating Haskell array codes with multicore GPUs. In Proceedings of the 6th workshop on Declarative Aspects of Multicore Programming. DAMP'11. New York, NY, USA: ACM, pp. 3–14.

Claessen, K., Sheeran, M. & Svensson, B. J. (2012) Expressive array constructs in an embedded GPU Kernel programming language. In Proceedings of the 7th workshop on Declarative Aspects and Applications of Multicore Programming. DAMP '12. New York, NY, USA: ACM, pp. 21–30.

Elliott, C. (2003) Functional images. *The Fun of Programming*. "Cornerstones of Computing" series. Palgrave, pp. 131–150.

Elliott, C., Finne, S. & de Moor, O. (2003) Compiling embedded languages. *J. Funct. Program.* **13**(3), 455–481.

Guibas, L. J. & Wyatt, D. K. (1978) Compilation and delayed evaluation in APL. In Proceedings of the 5th Acm SIGACT-SIGPLAN Symposium on Principles of Programming Languages. POPL'78. New York, NY, USA: ACM, pp. 1–8.

Harris, M. (2007) *Optimizing parallel reduction in CUDA*. `"http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf"`.

Harris, M., Sengupta, S. & Owens, J. D. (2007) Parallel prefix sum (Scan) with CUDA. In *GPU Gems 3*, Nguyen, H. (ed), Boston Mass.: Addison Wesley, pp. 851–876.

Holk, E., Byrd, W. E., Mahajan, N., Willcock, J., Chauhan, A. & Lumsdaine, A. (2012) Declarative parallel programming for GPUs. In Proceedings of ParCo 2011 Applications, Tools and Techniques on the Road to Exascale Computing. Advances in Parallel Computing. Amsterdam: IOS Press, pp. 297–304.

Keller, G., Chakravarty, M. M. T., Leshchinskiy, R., Peyton Jones, S. & Lippmeier, B. (2010) Regular, shape-polymorphic, parallel arrays in Haskell. In Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming. ICFP'10. New York, NY, USA: ACM, pp. 261–272.

Kloeckner, A. (2015) Loo.py: From Fortran to performance via transformation and substitution rules. In Proceedings of the ACM SIGPLAN 2nd International Workshop on Libraries, Languages and Compilers for Array Programming. ARRAY'15. New York, NY, USA: ACM, pp. 1–6.

Mainland, G. & Morrisett, . (2010) Nikola: Embedding compiled GPU functions in Haskell. In Proceedings of the 3rd ACM Haskell Symposium. New York, NY, USA: ACM, pp. 67–78.

McDonell, T. L., Chakravarty, M. M. T., Keller, G. & Lippmeier, B. (2013) Optimising purely functional GPU programs. In Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming. ICFP'13. New York, NY, USA: ACM, pp. 49–60.

NVIDIA. (2015a) *CUDA C Programming Guide*. `"http://docs.nvidia.com/cuda/cuda-c-programming-guide/ index.html"`.

NVIDIA. (2015b) *NVIDIA CUB Library*. `"http://nvlabs.github.io/cub/"`.

NVIDIA. (2015c) *NVIDIA Thrust Library*. `"https://developer.nvidia.com/thrust"`.

Oancea, C. E., Andreetta, C., Berthold, J., Frisch, A. & Henglein, F. (2012) Financial software on GPUs: Between Haskell and Fortran. In Proceedings of the 1st ACM SIGPLAN workshop on Functional High-Performance Computing. FHPC'12. New York, NY, USA: ACM, pp. 61–72.

Persson, A., Axelsson, E. & Svenningsson, J. (2012) Generic monadic constructs for embedded languages. In *Implementation and Application of Functional Languages*, Andy Gill and Jurriaan Hage (eds), IFL '11. Berlin Heidelberg: Springer-Verlag, pp. 85–99.

Sculthorpe, N., Bracker, J., Giorgidze, G. & Gill, A. (2013) The constrained-monad problem. In Proceedings of 18th ACM SIGPLAN International Conference on Functional Programming, ICFP 2013. New York, NY, USA: ACM, pp. 287–298.

Sklansky, J. (1960) Conditional-sum addition logic. *IRE Trans. Electron. Comput.* **EC-9**(2), 226–231.

Stevens, R. T. (1989) *Fractal Programming in C*. M&T Books.

Svenningsson, J. & Axelsson, E. (2013) Combining deep and shallow embedding for EDSL. In *Trends in Functional Programming, TFP '12*, Loidl, H.-W. & Pea, R. (eds), Lecture Notes in Computer Science, vol. 7829. Berlin Heidelberg: Springer-Verlag, pp. 21–36.

Svenningsson, J. & Svensson, B. J. (2013) Simple and compositional reification of monadic embedded languages. In Proceedings of 18th ACM SIGPLAN International Conference on Functional Programming, ICFP'13. New York, NY, USA: ACM.

Svenningsson, J., Svensson, B. J. & Sheeran, M. (2013) Efficient counting and occurrence sort for GPUs using an embedded GPU programming language. In Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-Performance Computing. FHPC'13. New York, NY, USA: ACM pp. 50–63.

Svensson, B. J., Sheeran, M. & Newton, R. R. (2014) Design exploration through code-generating DSLs. *Commun. ACM* **57**(6), 56–63.

Svensson, J., Sheeran, M. & Claessen, K. (2008) Obsidian: A domain specific embedded language for parallel programming of graphics processors. In Proceedings of the International Conference on Implementation and Application of Functional Languages. IFL '08. Berlin Heidelberg: Springer-Verlag, pp. 156–173.

Svensson, J., Claessen, K. & Sheeran, M. (2010) GPGPU kernel implementation and refinement using Obsidian. *Procedia Comput. Sci.* **1**(1), 2065–2074.

Ulvinge, N. (2014) *Increasing Programmability of an Embedded Domain Specific Language for GPGPU Kernels using Static Analysis*. MSc Thesis, Dept. Computer Science and Engineering, Chalmers University of Technology.

## Appendix A. Accelerate performance numbers

Measuring performance of Accelerate programs turned out not to 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 behaviour on each invocation varies. For example, 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 difficult to reason about exactly when copying data to and from a device takes place.

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

For Obsidian and Thrust, each reduction and scan is run 1,000 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 led 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 1,000 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 1,000 iterations is extrapolated.