Expressive Array Constructs in an Embedded GPU Kernel Programming Language

Koen Claessen Mary Sheeran Bo Joel Svensson

Chalmers University of Technology, Department of Computer Science and Engineering, Gothenburg, Sweden
koen@chalmers.se/ms@chalmers.se/joels@chalmers.se

Abstract

Graphics Processing Units (GPUs) are powerful computing devices that with the advent of CUDA/OpenCL are becomming useful for general purpose computations. Obsidian is an embedded domain specific language that generates CUDA kernels from functional descriptions. A symbolic array construction allows us to guarantee that intermediate arrays are fused away. However, the current array construction has some drawbacks; in particular, arrays cannot be combined efficiently. We add a new type of *push arrays* to the existing Obsidian system in order to solve this problem. The two array types complement each other, and enable the definition of combinators that both take apart and combine arrays, and that result in efficient generated code. This extension to Obsidian is demonstrated on a sequence of sorting kernels, with good results. The case study also illustrates the use of combinators for expressing the structure of parallel algorithms. The work presented is preliminary, and the combinators presented must be generalised. However, the raw speed of the generated kernels bodes well.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; Concurrent, distributed, and parallel languages; D.3.4 [*Programming Languages*]: Processors—Code generation

General Terms Languages, Algorithms, Performance

Keywords Arrays, Data parallelism, Embedded Domain Specific Language, General Purpose GPU programming, Haskell

1. Introduction

Graphics Processing Units (GPUs) are parallel computers with hundreds to thousands of processing elements. The CUDA and OpenCL languages make available the power of the GPU to programmers interested in general purpose computations. In CUDA and OpenCL, the programmer writes *kernels*, Single Program Multiple Data (SPMD) programs that are executed by groups of threads on the available processing elements of the GPU.

CUDA and OpenCL are general purpose programming languages, mirroring the increased capabilities of a modern GPU to target that domain. However, these languages lack compositionality. Also, being based in C/C++ means that the core idea in a program may not be easily visible.

DAMP'12, January 28, 2012, Philadelphia, PA, USA.

1.1 Embedded DSLs for GPGPU programming

We are aiming for a GPU programming language that is more concise than mainstream languages such as CUDA and OpenCL. Obsidian is a domain specific embedded language (DSEL) implemented in Haskell. When an Obsidian program is run, a representation of the program is created as a syntax tree. For more information on EDSL implementation see [9]. The program representation generated when running an Obsidian program is compiled into CUDA code. We are also working on an OpenCL backend.

Our approach is different from that of other Haskell DSELs targetting GPUs [4, 12, 13]. We do not try to abstract away from all the peculiarities of GPU programming, but rather provide a higher level language in which to experiment with them. For instance, Accelerate provides a standard set of basic operations such as `map`, `reduce` and `zipWith` as built in skeletons, implemented with the help of small, predefined, hand-tuned CUDA kernels [4]. Obsidian, on the other hand, allows the user to experiment with the *generation* of small kernels for fixed size array inputs from higher level descriptions. It is intended to allow the user to play with the kinds of tradeoffs that are important when writing such high performance building blocks; in this paper, the main consideration is the number of array elements of the input and output that are manipulated by a single thread in the generated CUDA code. An important aspect of Obsidian is the symbolic array representation used, along with its associated `sync` operation. As we shall see, the `sync` operation allows the programmer to guide code generation and control parallelism and thread use [10].

In Obsidian, a kernel that sums an array can be expressed as:

```
sum :: Array IntE -> Kernel (Array IntE)
sum arr | len arr == 1 = return arr
        | otherwise    = (pure (fmap (uncurry (+)) . pair)
                           ->- sync
                           ->- sum) arr
```

The result of running this kernel on an eight element input array, `runKernel sum (namedArray ``input'' 8)`, is an intermediate representation of the computation (shown in slightly pretty-printed form):

```
arr0 = malloc(16)
par i 4 {
arr0[i] = ( +  input[( *  i 2 )] input[( +  ( *  i 2 ) 1 )] );
}Sync
arr1 = malloc(8)
par i 2 {
arr1[i] = ( +  arr0[( *  i 2 )] arr0[( +  ( *  i 2 ) 1 )] );
}Sync
arr2 = malloc(4)
par i 1 {
arr2[i] = ( +  arr1[( *  i 2 )] arr1[( +  ( *  i 2 ) 1 )] );
}Sync
```

The named intermediate arrays in this representation are then laid out in GPU shared memory and CUDA code can be generated (here for arrays of length eight)[1]:

```
__global__ void sum(int *input0,int *result0){
  unsigned int tid = threadIdx.x;
  unsigned int bid = blockIdx.x;
  extern __shared__ unsigned char sbase[];
  (( int *)sbase)[tid] =
    (input0[((bid*8)+(tid*2))]+
     input0[((bid*8)+((tid*2)+1))]);
  __syncthreads();
  if (tid<2){
    (( int *)(sbase + 16))[tid] =
      ((( int *)sbase)[(tid*2)]+
       (( int *)sbase)[((tid*2)+1)]);
  }
  __syncthreads();
  if (tid<1){
    (( int *)sbase)[tid] =
      ((( int *)(sbase+16))[(tid*2)]+
       (( int *)(sbase+16))[((tid*2)+1)]);
  }
  __syncthreads();
  if (tid<1){
    result0[(bid+tid)] = (( int *)sbase)[tid];
  }
}
```

1.2 Arrays in Obsidian

An array is represented by an indexing function and a length:

```
data Array a = Array (UWordE -> a) Word32
```

This array representation has served us well. It has these properties:

- Fusion of operations is automatic.
- It naturally describes a data-parallel computation suitable for CUDA/OpenCL generation.
- Many basic operations can be implemented: `map`, `zipWith` etc.

Using this array representation in a DSEL is not new; the first occurence that we know of is in Pan [8]. Similar array representations have also later been used in Feldspar [1], and more recently also in the Repa library [11]. Functions for indexing and getting the length of arrays are as follows:

```
(!) :: Array a -> UWordE -> a
(Array ixf _) ! ix = ixf ix

len :: Array a -> Word32
len (Array _ n) = n
```

A *Functor* instance for the `Array` datatype is

```
instance Functor Array where
  fmap f arr = Array (\ix -> f (arr ! ix)) (len arr)
```

Now, composed applications of `fmap` will be automatically fused. This is illustrated in the example program below and the CUDA generated from it.

```
mapFusion :: Array IntE -> Kernel (Array IntE)
mapFusion = pure (fmap (+1) . fmap (*2))

__global__ void mapFusion(int *input0,int *result0){
  unsigned int tid = threadIdx.x;
  unsigned int bid = blockIdx.x;

  result0[((bid*32)+tid)] = ((input0[((bid*32)+tid)]*2)+1);
}
```

Both of these code listings need explanation. In the Haskell code, `mapFusion` has type `Array IntE -> Kernel (Array IntE)`; `Kernel` is a state monad that accumulates CUDA code as well as provides new names for intermediate arrays. Neither of these features of the monad is activated by this example though. The `pure` function is defined using the monad's `return` as `pure f a = return (f a)`. In this case, it lifts a function of type `Array IntE -> Array IntE` into a kernel.

The generated CUDA code computes the result array using a number of threads equal to the length of that array. In this case, the kernel was generated to deal with arrays of length 32. The important detail to notice in the CUDA code is that there is no intermediate array created between the `(*2)` and the `(+1)` operations.

The `mapFusion` example could just as well have been implemented using the kernel sequential composition combinator, `->-`.

```
mapFusion :: Array IntE -> Kernel (Array IntE)
mapFusion = pure (fmap (*2)) ->- pure (fmap (+1))
```

Exactly the same CUDA code is then generated.

In some cases, it is necessary to force computation of intermediate arrays. This can be used to share partial computations between threads and to expose parallelism. In Obsidian, the tool for this is called `sync`, a built-in kernel. Using `sync` as follows prevents fusion of the two operations:

```
mapUnFused :: Array IntE -> Kernel (Array IntE)
mapUnFused = pure (fmap (*2)) ->- sync ->- pure (fmap (+1))
```

The generated CUDA code now stores an intermediate result in local shared memory before moving on.

```
__global__ void mapUnFused(int *input0,int *result0){
  unsigned int tid = threadIdx.x;
  unsigned int bid = blockIdx.x;
  extern __shared__ unsigned char sbase[];
  (( int *)sbase)[tid] = (input0[((bid*32)+tid)]*2);
  __syncthreads();
  result0[((bid*32)+tid)] = ((( int *)sbase)[tid]+1);
}
```

Intermediate arrays are laid out in the `sbase` array in shared memory. Since we may store arrays of many different types in the same locations of the shared memory at different times during the execution, the type casts used in the code above are necessary.

1.3 Sync and parallelism

The `sync` operation also enables the writing of parallel reduction kernels. A reduction operation is an operation that takes an array as input and produces a singleton array as output.

First, we define `zipWith` and `halve` on Obsidian arrays.

```
zipWith :: (a -> b -> c) -> Array a -> Array b -> Array c
zipWith op a1 a2 = Array (\ix -> (a1 ! ix) `op` (a2 ! ix))
                   (min (len a1) (len a2))

splitAt :: Word32 -> Array a -> (Array a, Array a)
splitAt n arr =
  (Array (\ix -> arr ! ix) n ,
   Array (\ix -> arr ! (ix + fromIntegral n)) (len arr - n))

halve arr = splitAt ((len arr) `div` 2) arr
```

A reduction kernel that takes an array whose length is a power of two and gives an array of length one can be defined recursively. Defining kernels recursively results in completely unrolled CUDA kernels, and kernel input size must be known at compile time. The approach to reduction taken here is to split the input array into two halves and then apply `zipWith` of the combining function to the two halves, repeating the process until the length is one.

```
reduceS :: (a -> a -> a) -> Array a -> Kernel (Array a)
reduceS op arr | len arr == 1 = return arr
               | otherwise    =
                 (pure ((uncurry (zipWith op)) . halve)
                  ->- reduceS op) arr
```

Since the output of this kernel is of length one, and the number of elements in the output array specifies the number of threads used to compute it, this function, `reduceS`, defines a sequential reduction. The generated code for arrays of length eight is

[1] An alignment qualifier for shared memory has been omitted to save space in the listings showing generated code

```
__global__ void reduceSAdd(int *input0,int *result0){
  unsigned int tid = threadIdx.x;
  unsigned int bid = blockIdx.x;

  result0[(bid+tid)] =
    (((input0[((bid*8)+tid)]+
       input0[((bid*8)+(tid+4))])+
      (input0[((bid*8)+(tid+2))]+
       input0[((bid*8)+((tid+2)+4))]))+
     ((input0[((bid*8)+(tid+1))]+
       input0[((bid*8)+((tid+1)+4))])+
      (input0[((bid*8)+((tid+1)+2))]+
       input0[((bid*8)+(((tid+1)+2)+4))])));
}
```

Sequential reduction is not very interesting for GPU execution, but the fix is simple. A well placed use of `sync` indicates that we want to compute, after each `zipWith` phase, the intermediate arrays using as many threads as that intermediate array is long. The effect is shown in the code below.

```
reduce :: Syncable Array a
          => (a -> a -> a) -> Array a -> Kernel (Array a)
reduce op arr | len arr == 1 = return arr
              | otherwise    =
                (pure ((uncurry (zipWith op)) . halve)
                 ->- sync
                 ->- reduce op) arr
```

The CUDA code for reduction with addition on eight elements is

```
__global__ void reduceAdd(int *input0,int *result0){
  unsigned int tid = threadIdx.x;
  unsigned int bid = blockIdx.x;
  extern __shared__ unsigned char sbase[];
  (( int *)sbase)[tid] =
    (input0[((bid*8)+tid)]+
     input0[((bid*8)+(tid+4))]);
  __syncthreads();
  if (tid<2){
    (( int *)(sbase + 16))[tid] =
      ((( int *)sbase)[tid]+
       (( int *)sbase)[(tid+2)]);
  }
  __syncthreads();
  if (tid<1){
    (( int *)sbase)[tid] =
      ((( int *)(sbase+16))[tid]+
       (( int *)(sbase+16))[(tid+1)]);
  }
  __syncthreads();
  if (tid<1){
    result0[(bid+tid)] = (( int *)sbase)[tid];
  }
}
```

In this generated CUDA, three phases can be identified. The first uses four threads to compute a four element intermediate array; the second uses two threads, and so on. At the very end, a single thread copies the result from local shared memory to global memory.

1.4 Drawbacks of Obsidian Arrays

The previous subsection described positive aspects of the array representation that we have used so far. There are, however, circumstances in which this Array representation is too restricted.

Take the problem of concatenating two arrays. Using the array representation described above, the only way to concatenate two arrays is to introduce a conditional into the indexing function. If `f` and `g` are the indexing functions of two arrays that are to be concatenated, and `n1` is the length of the first array, the indexing function of the result must be

```
new ix = if (ix < n1)
         then f ix
         else g (ix - n1)
```

The following program concatenates two arrays:

```
catArrays :: (Array IntE, Array IntE) -> Kernel (Array IntE)
catArrays = pure conc
```

When it is used to generate a CUDA kernel that concatenates two arrays of length 16, the following code is the result:

```
__global__ void catArrays(int *input0,int *input1,int *result0){
  unsigned int tid = threadIdx.x;
  unsigned int bid = blockIdx.x;

  result0[((bid*32)+tid)] =
    (tid<16) ? input0[((bid*16)+tid)] :
               input1[((bid*16)+(tid-16))];
}
```

Now, conditionals like these are *bad* in code to execute on a GPU, with its wide-SIMD data-parallel model. Separating the operation into two assignments and using half as many threads gives much higher performance.

```
__global__ void catArraysByHand(int *input0,
                                int *input1,
                                int *result0){
  unsigned int tid = threadIdx.x;
  unsigned int bid = blockIdx.x;

  result0[((bid*32)+tid)] = input0[((bid*16)+tid)];
  result0[((bid*32)+tid+16)] = input1[((bid*16)+tid)];
}
```

There are cases where code with conditionals is not that bad. An expert on NVIDIA GPUs in particular may say that code with the condition `(tid < 32)` is fine, since 32 is the SIMD width of those GPUs. However, any number that is not a multiple of 32 would lead to poor performance, so in general this is a problem. Worse still, *zipping* two arrays together and then *unpairing* (to get an array of elements) leads to code that takes two different paths depending on odd or even *thread id*. When a GPU executes such code, it shuts down half of the threads and computes the two paths in sequence.

```
zippUnpair :: (Array IntE, Array IntE) -> Kernel (Array IntE)
zippUnpair = pure (unpair . zipp)
```

The `zipp` and `unpair` operations are defined as follows:

```
zipp :: (Array a, Array b) -> Array (a, b)
zipp (arr1,arr2) =
     Array (\ix -> (arr1 ! ix, arr2 ! ix))
           (min (len arr1) (len arr2))

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

Code generated from the `zippUnpair` program exhibits really poor performance; at any time half of the threads are shut down.

```
__global__ void zippUnpair(int *input0,
                           int *input1,
                           int *result0){
  unsigned int tid = threadIdx.x;
  unsigned int bid = blockIdx.x;

  result0[((bid*64)+tid)] =
    ((tid%2)==0) ? input0[((bid*32)+(tid>>1))] :
                   input1[((bid*32)+(tid>>1))];
}
```

If we wrote this CUDA program by hand, we would, again, split it up into two phases so that all threads can progress in parallel.

The arrays described so far, with an indexing function and a length, have been nicknamed *Pull arrays* for how they describe how to compute an element by *pulling* data from a number of places. Using just Pull arrays, we have been unable to solve the problems described so far in this section. The solution is to add a complementary array type to Obsidian.

2. Push Arrays

In order to improve low level control for the programmer, *Push arrays* are added to Obsidian. The old Pull arrays are still available, along with the new array type.

Some operations, typically involving taking arrays apart, are easily described using Pull arrays, giving efficient code. In those cases, using a Push array would add complexity in the implementation for no performance benefit. Other operations cannot be implemented efficiently with Pull arrays, but Push arrays then provide the solution. This duality is apparent when looking at operations on Pull arrays such as `halve` and `conc` (for concatenate). The `halve` function is efficient since it introduces no diverging conditionals. The `conc` function, on the other hand, introduces conditionals. Concatening two arrays using the `concP` combinator, implemented on Push arrays, allows us to generate the desired code:

```
catArrayPs :: (Array IntE, Array IntE) -> Kernel (ArrayP Int)
catArrayPs = pure concP

__global__ void catArrayPs(int *input0,int *input1,int *result0){
  unsigned int tid = threadIdx.x;
  unsigned int bid = blockIdx.x;

  result0[((bid*32)+tid)] = input0[((bid*16)+tid)];
  result0[((bid*32)+(16+tid))] = input1[((bid*16)+tid)];

}
```

Compared to the CUDA code for `catArrays`, this kernel uses only 16 threads instead of 32. At each step of the computation, all the threads are fully busy doing exactly the same thing, which is the preferred mode of execution on the target platform.

2.1 What are Push Arrays?

The idea behind Push arrays is to have a way to describe where elements are supposed to end up. In some sense, a Push array produces a collection of Index/Value pairs. This makes Push arrays complementary to Pull arrays. For example, it is possible for a Push array to output several elements at the same index (which we probably need to control carefully). Push arrays should permit us to provide more expressive operations on arrays to the user, including an operation similar to Haskell's `filter` on lists. Here, we consider a different advantage of adding push arrays: finer control over patterns of thread use in generated code.

A Push array consists of three parts: a function in continuation passing style, a `Program` datatype and an array datatype.

```
type P a = (a -> Program) -> Program
```

For another example of using continuations and a more complete description of their meaning and application, see [5].

The `Program` datatype has now been adopted as Obsidian's internal representation of CUDA programs.

```
data Program
  = Skip
  | forall a. Scalar a => Assign Name UWordE (Exp a)
  | Par (UWordE -> Program) Word32
  | Allocate Name Word32 Type
  | Synchronize
  | ProgramSeq Program
               Program
```

Even Obsidian programs that never explicitly uses a Push array will also be represented by this datatype.

Note that the `Par` constructor, the *parallel for loop*, could potentially introduce nesting, which would lead to *nested data-parallelism*. We do not compile nested data parallelism into CUDA, and right now this is guaranteed by taking care not to introduce any nesting in the library functions provided. Some of the simpler cases of nestedness should be possible to take care of quite easily. For example, one extra level of nesting could be done by sequential execution in each thread of the GPU; using sequential computations per thread has been shown to be beneficial [3]. But for the general case of arbitrary nesting, some method of flattening is needed. We also assume that both `Allocate` and `Synchronize` occur only at the top level in objects of type `Program`.

Now, a Push array is a function in continuation passing style coupled with a length.

```
data ArrayP a = ArrayP (P (UWordE, a)) Word32
```

There is a function that takes an array and turns it into a Push array, called `push`. This function is defined for both Pull and Push arrays:

```
class Pushable a where
  push :: a e -> ArrayP e

instance Pushable ArrayP where
  push = id

instance Pushable Array where
  push (Array ixf n) =
     ArrayP (\func -> Par (\i -> func (i,(ixf i))) n) n
```

Going in the other direction, from a Push array to a Pull array, is a costly operation; it involves writing all the elements to GPU memory followed by creating a Pull array that represents reading them. The task of writing intermediate values to memory has traditionally been up to the `sync` operation in Obsidian. Therefore, in this version, `sync` is overloaded to operate on both Pull and Push arrays. This means that the `sync` operation can be used both on arrays of type `Array` and of type `ArrayP`. The result type, however, is always `Array`.

When a Push array is synced, it is applied to a continuation that writes the elements into a named array in memory. The name to use is obtained through the `Kernel` monad.

```
targetArray :: Scalar a => Name -> (UWordE,Exp a) -> Program
targetArray n (i,a) = Assign n i a
```

After applying the Push array to `targetArray <name>`, the `sync` operation proceeds by storing away a representation of the program that computes the array called `<name>`; it returns a Pull array that reads elements from that same array.

Now we have seen enough of the implementation of Push arrays to be able to look at some operations. Earlier, we saw that the array concatenation function `conc` on Pull arrays leads to inefficient code. The Push version of this operation, called `concP` can be implemented as follows:

```
concP :: (Pushable arr1,
          Pushable arr2) => (arr1 a, arr2 a) -> ArrayP a
concP (arr1,arr2) =
  ArrayP (\func -> f func
                   *>*
                   g (\(i,a) -> func (fromIntegral n1 + i,a)))
         (n1+n2)
  where
     ArrayP f n1 = push arr1
     ArrayP g n2 = push arr2
```

The function `concP` takes two arrays, that can be Push or Pull arrays, and concatenates them into a single Push array. It does so by creating a sequential program, using the `*>*` operator for `Program` sequential composition. An example use of this combinator has already been displayed in the `catArrayPs` example.

The `zippUnpair` example shows a drawback similar to that of `catArrays` using Pull arrays. In this case, the problem is that the `unpair` function introduces a conditional that takes different paths depending on odd or even *thread id*. A Push array implementation of the `unpair` operation called `unpairP` can be given as follows:

```
unpairP :: Pushable arr => arr (a,a) -> ArrayP a
unpairP arr =  ArrayP (\k -> f (everyOther k)) (2 * n)
  where
    ArrayP f n = push arr
```

```
everyOther :: ((UWordE, a) -> Program ())
              -> (UWordE, (a,a)) -> Program ()
everyOther f  = \(ix,(a,b)) -> f (ix * 2,a) *>* f (ix * 2 + 1,b)
```

Just like `concP`, this function takes either a Push or Pull array as input, and produces a Push array as result.

Rewriting the example from earlier using `unpairP` gives:

```
zippUnpairP :: (Array IntE, Array IntE) -> Kernel (ArrayP IntE)
zippUnpairP = pure (unpairP . zipp)
```

In this case, the generated code looks as follows:

```
__global__ void zippUnpairP(int *input0,int *input1,int *result0){
  unsigned int tid = threadIdx.x;
  unsigned int bid = blockIdx.x;

  result0[((bid*64)+(tid*2))] = input0[((bid*32)+tid)];
  result0[((bid*64)+((tid*2)+1))] = input1[((bid*32)+tid)];
}
```

Again, we get CUDA code that uses half as many threads as the inefficient version, but all threads are occupied at all times. This uses the resources more efficiently.

Being able to generate the kind of code that we have just seen is something we have desired for a long time. We believe that Push arrays are an important tool for obtaining high performance kernels. The results in section 3 bear this out.

3. Application

3.1 Sorting on a GPU

In this section, we introduce combinators that express patterns of computation on whole arrays, and show their application to the development of fast sorting kernels. By kernel, we mean a computation that is performed by multiple threads, each performing the same computation, in a single block. The computation is performed entirely on the GPU, operating on a short array, which has been placed into shared memory. On the GPU on which we perform measurements, the maximum number of threads in a single block is 512. Thus, we will build and benchmark a sequence of kernels that sort 512 inputs. Our first kernels are implemented using one thread per array element. Next, we show how Push arrays allow us to move to having each thread operate on two array elements, giving a substantial performance improvement.

Sorting kernels are typically used as building blocks in larger programs to sort much larger sequences of inputs. In section 3.6, we show how to build a sorter for large arrays from small building blocks, including small kernels for sorting and merging that are generated from Obsidian. Our small kernels are constructed in the form of sorting and merging *networks*, building on Batcher's bitonic merger [2] and on the periodic balanced merger [7]. We chose also to implement the large sorter used in benchmarking the small kernels as a sorting network. However, large sorters that are not themselves sorting networks (with typical examples being radix sort and quicksort) often call small sorting networks when they need to sort small arrays during their execution. Thus, small, fast sorting kernels have a variety of uses.

3.2 Describing Batcher's bitonic merger

The bitonic merger is typically presented as a recursive construction and we have earlier explored ways to describe and analyse it in both (our) Ruby and in Lava [6, 14]. Here, we consider iterative descriptions using similar combinators.

Figure 1 illustrates the merger for 16 inputs. Data flows from left to right. The vertical lines indicate components that operate on two array elements, placing the minimum onto the lower (abstract) wire, and the maximum onto the other output of the component.

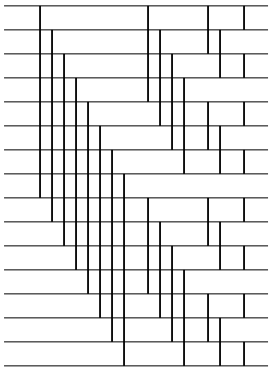

Figure 1. A diagram of a 16 input bitonic merging network, using a style that is standard in the literature. Note that in each stage containing 8 min/max or comparator components, all 8 operate on independent parts of the input and so can proceed in parallel.

The leftmost *stage* operates (for $n = 16$) on elements that are 8 apart. the next stage deals with elements that are 4 apart, and so on.

We introduce a combinator `ilv1`, for *interleave*, that captures this pattern. `ilv1 i f g` applies `f` to elements 2^i apart, producing the output at the lower of the two input indices; it applies `g` to the same pairs of elements, producing the output on the upper index. Defining `stage i` to be `ilv1 i min max`, the four stages in the diagram are simply `stage` applied to3, 2 1 and 0. The definition of `ilv1` makes use of the fact that flipping the bit i of an index (using the function `flipBit`) gives the index of the element that will be combined with it using the functions`f` and `g`. The decision about whether to apply `f` or `g` is made by looking at the value of bit i. As we shall see, the use of the Obsidian `ifThenElse` produces conditionals in the resulting CUDA.

```
lowBit :: Int -> UWordE -> Exp Bool
lowBit i ix = (ix .&. bit i) ==* 0

flipBit :: Bits a => Int -> a -> a
flipBit = flip complementBit

ilv1 :: Choice a =>
        Int -> (b -> b-> a) -> (b -> b -> a) ->
        Array b -> Array a
ilv1 i f g arr = Array ixf (len arr)
  where
    ixf ix = let l = arr ! ix
                 r = arr ! newix
                 newix = flipBit i ix
             in (ifThenElse (lowBit i ix) (f l r) (g l r))
```

Expressing `ilv1` using bit-flipping may seem strange, but it has the advantage that it actually applies the desired pattern of computation repeatedly over larger input arrays. Now, for 2^n inputs, a Haskell list containing the n calls of this interleave combinator are built:

```
bmerge :: Int -> [Array IntE -> Array IntE]
bmerge n = [istage (n-i) | i <- [1..n]]
  where istage i = ilv1 i min max
```

Finally, the `compose` function makes each element of the list into a kernel (using `map pure`) and places a `sync` between each kernel (using `composeS`).

```
compose :: (Scalar a) =>
           [Array (Exp a) -> Array (Exp a)]
           -> Array (Exp a) -> Kernel (Array (Exp a))
compose = composeS . map pure

runm k = putStrLn$ CUDA.genKernel "bitonicMerge"
         (compose (bmerge k)) (namedArray "inp" (2^k))
```

Note that `bmerge k` works on inputs of length 2^{k+j}, for $j > 0$, applying the merger to sub-sequences of length 2^k. The CUDA code for `bmerge 4` on 16 inputs (with some newlines inserted) is

```
*Main> runm 4
__global__ void bitonicMerge(int *input0,int *result0){
  unsigned int tid = threadIdx.x;
  unsigned int bid = blockIdx.x;
  extern __shared__ unsigned char sbase[];
  (( int *)sbase)[tid]
     = ((tid&8)==0)
      ? min(input0[((bid*16)+tid)],input0[((bid*16)+(tid^8))])
      : max(input0[((bid*16)+tid)],input0[((bid*16)+(tid^8))]);
  __syncthreads();
  (( int *)(sbase + 64))[tid]
     = ((tid&4)==0)
      ? min((( int *)sbase)[tid],(( int *)sbase)[(tid^4)])
      : max((( int *)sbase)[tid],(( int *)sbase)[(tid^4)]);
  __syncthreads();
  (( int *)sbase)[tid]
     = ((tid&2)==0)
      ? min((( int *)(sbase+64))[tid],(( int *)(sbase+64))[(tid^2)])
      : max((( int *)(sbase+64))[tid],(( int *)(sbase+64))[(tid^2)]);
  __syncthreads();
  (( int *)(sbase + 64))[tid]
     = ((tid&1)==0)
      ? min((( int *)sbase)[tid],(( int *)sbase)[(tid^1)])
      : max((( int *)sbase)[tid],(( int *)sbase)[(tid^1)]);
  __syncthreads();
  result0[((bid*16)+tid)] = (( int *)(sbase+64))[tid];
```

3.3 Modifying the bitonic merger

The bitonic merger for which we have just generated a kernel is known to sort so-called bitonic sequences, which include sequences whose first half is sorted in one direction and whose second half is sorted in the other direction. This fact can be used to build the well-known bitonic sorting network. However, a GPU implementation typically needs to check, for each comparator, whether or not it should sort upwards or downwards, see for instance the simple CUDA implementation shown in Appendix A. We choose here to modify the merger so that it sorts two concatenated sequences that are sorted in the *same* direction. We do this by using a well-known trick, reversing half of the input to the merger. It turns out that we can also reverse the same half of the output of the first stage of the network, without affecting overall behaviour. The resulting network, `tmerge`, shown in Figure 2, encourages us to develop a new combinator to describe the characteristic V-shaped pattern that results in the first stage. The combinator is modelled on `ilv1`. The only difference is that the "partner" of an index is found not by flipping bit i, but by flipping bits 0 to i, using function `flipLSBsTo`. The implementation of `vee1` is got from that for `ilv1` by replacing the call of `flipBit` by one of `flipLSBsTo` (and we could also have chosen to make a more generic function that is parameterised on this *partner* function).

```
flipLSBsTo :: Int -> UWordE -> UWordE
flipLSBsTo i = (`xor` (oneBits (i+1)))

vee1 :: Choice a =>
        Int -> (b -> b-> a) -> (b -> b -> a) ->
        Array b -> Array a
vee1 i f g arr = Array ixf (len arr)
  where
    ixf ix = let l = arr ! ix
                 r = arr ! newix
                 newix = flipLSBsTo i ix
             in (ifThenElse (lowBit i ix) (f l r) (g l r))


tmerge :: Int -> [Array IntE -> Array IntE]
tmerge n = vstage (n-1): [istage (n-i) | i <- [2..n]]
  where
    vstage i = vee1 i min max
    istage i = ilv1 i min max
```

Now that we have a merger that sorts sub-sequences containing two concatenated sorted sequences, it is easy to make a tree of them. The list of kernels to be composed now becomes

```
tsort1 :: Int -> [Array IntE -> Array IntE]
tsort1 n = concat [tmerge i | i <- [1..n]]
```

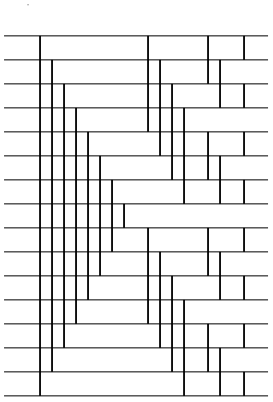

Figure 2. 16 input merging network, *tmerge*. The first stage is made with a new combinator that we call vee, while the remaining stages are as in the bitonic merger. This network sorts an input that consists of two half-sized sorted sequences, giving the opportunity to build a tree-shaped sorting network.

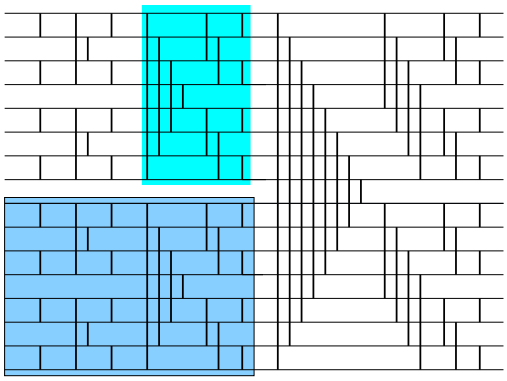

Figure 3. A 16-input sorter made from a tree of *tmerge* mergers. 8 2-input mergers feed 4 4-input mergers, followed by 2 8-input and one 16-input merger. A sorter on 8 inputs is shaded, as is a merger on 8 inputs, above it.

The resulting sorting network is shown in Figure 3.
The following call writes the resulting CUDA to file `tsort1.cu` and this is the first *generated* CUDA kernel whose performance is measured in section 3.6.

```
runs1 k
  = writeFile "tsort1.cu" $ CUDA.genKernel "tsort1"
      (compose (tsort1 k)) (namedArray "inp" (2^k))
```

The generated code uses one thread per array element (just as the `bitonicMerge` kernel shown above did). The next step is to move to using Push as well as Pull arays, so as to be able to generate more efficient code from essentially the same sorter construction.

3.4 New combinator implementations using Push arrays

It is in combining the results of the `f` and `g` functions that we run into difficulty using just Pull arrays. Earlier, we saw how to use Push arrays to implement `concP`, which concatenates two arrays. Here, we use exactly the same approach to make a new version of the *interleave* combinator. The results of applying the `f`s and `g`s are combined into a Push array, in the right order.

```
ixMap :: (UWordE -> UWordE) -> ArrayP a -> ArrayP a
ixMap f (ArrayP p n) = ArrayP (ixMap' f p) n

ixMap' :: (UWordE -> UWordE)
          -> P (UWordE, a)
          -> P (UWordE, a)
ixMap' f p = \g -> p (\(i,a) -> g (f i,a))

insertZero :: Int -> UWordE -> UWordE
insertZero 0 a = a `shiftL` 1
insertZero i a
  = a + (a .&. fromIntegral (complement (oneBits i :: Word32)))

ilv2 :: Choice b =>
        Int -> (a -> a -> b) -> (a -> a -> b) ->
        Array a -> ArrayP b
ilv2 i f g (Array ixf n)
   = ArrayP (\k -> app a5 k *>* app a6 k) n
  where
    n2 = n `div` 2
    a1 = Array (ixf . left) (n-n2)
    a2 = Array (ixf . right) n2
    a3 = zipWith f a1 a2
    a4 = zipWith g a1 a2
    a5 = ixMap left (push a3)
    a6 = ixMap right (push a4)
    left = insertZero i
    right = flipBit i  . left
    app (ArrayP f _) a = f a
```

This new combinator can now replace `ilv1` in the bitonic merger, giving a kernel that runs considerably faster. We will use that kernel to build a large sorter later.

The implemenation of `vee2` is almost identical to that of `ilv2`, with `flipBit i` replaced by `flipLSBsTo` as before (so that, again, one would in fact make a more generic function for building such combinators). Now, we just need to replace the `ilv1` and `vee1` combinators in the tree sorter with `ilv2` and `vee2` , to get a verrsion that uses half as many threads:

```
tmerge2 :: Int -> [Array IntE -> ArrayP IntE]
tmerge2 n = vstage (n-1) : [ istage (n-i) | i <- [2..n]]
  where
    vstage i = vee2 i min max
    istage i = ilv2 i min max


tsort2 :: Int -> [Array IntE -> ArrayP IntE]
tsort2 n = concat [tmerge2 i | i <- [1..n]]
```

As we shall see in section 3.6, the resulting code is significantly faster. This is because it uses one thread per two array elements, and the code no longer contains any conditionals.

In order to go faster still, we resort to building a different sorting network, of exactly the same size as the bitonic sorter, but based instead on the balanced period merger of Dowd et al [7]. This involves the introduction of one new combinator that can be seen as a mixture of the `ilv` and `vee` combinators already introduced.

3.5 A sorter built from the balanced periodic merger

First, we note that the balanced periodic merger contains multiple uses of the now familiar vee-shaped pattern, see Figure 4.

```
bpmerge2 :: Int -> [Array IntE -> ArrayP IntE]
bpmerge2 n = [vstage (n-i) | i <- [1..n]]
  where vstage i = vee2 i min max
```

Now Dowd et al proved that the balanced periodic merger sorts two *interleaved* sorted sequences. So, taking an iterative view of the resulting sorter, we want to build a tree of mergers as before, but the smaller mergers should be interleaved, rather than operating on adjacent sub-sequences. There should be one merger on the right hand end of the network; left of that, there should be two mergers that operate on the odd and even elements, and each of them should in turn be fed by two interleaved mergers, and so. The sorter is illustrated, for 16 inputs in Figure 5. Just to the left of the final

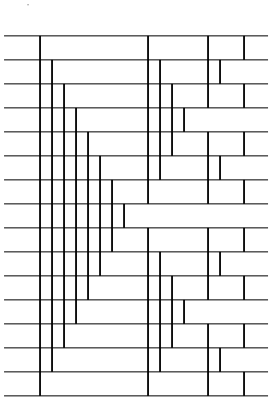

Figure 4. 16 input periodic balanced merging network

balanced merger, one of the two interleaved mergers is shown using dotted lines. It operates on a completely different set of inputs from the other 8-input merger in the same part of the tree.

The most straightforward way to give an iterative description of this sorter is to introduce a new combinator that is a combination of `ilv` and `vee`. The only thing that we need to change is the *partner* function. This time we will flip not the least significant bits from position 0 to position i but from position i to $i + j$.

```
-- flip bits from position i to position i+j inclusive
flipBitsFrom :: Bits a => Int -> Int -> a -> a
flipBitsFrom i j a = a `xor` (fromIntegral mask)
  where
    mask = (oneBits (j + 1):: Word32) `shiftL` i

ilvVee1 :: Choice a =>
           Int -> Int ->
           (b -> b-> a) -> (b -> b -> a) ->
           Array b -> Array a
ilvVee1 i j f g arr = Array ixf (len arr)
  where
    ixf ix = let l = arr ! ix
                 r = arr ! newix
                 newix = flipBitsFrom i j ix
             in (ifThenElse (lowBit (i+j) ix) (f l r) (g l r))

ilvVee2 :: Choice b => Int -> Int ->
           (a -> a -> b) -> (a -> a -> b) ->
           Array a -> ArrayP b
ilvVee2 i j f g (Array ixf n)
  = ArrayP (\k -> app a5 k *>* app a6 k) n
  where
    n2 = n `div` 2
    a1 = Array (ixf . left) (n-n2)
    a2 = Array (ixf . right) n2
    a3 = zipWith f a1 a2
    a4 = zipWith g a1 a2
    a5 = ixMap left (push a3)
    a6 = ixMap right (push a4)
    left = insertZero (i+j)
    right = flipBitsFrom i j . left
    app (ArrayP f _) a = f a
```

For both variants of the combinator, we simply add to the `ilv` definitions a new `Int` parameter, `j`, and replace `flipBit i` by `flipBitsFrom i j`. We also insert the zero bit (when calculating the left index) at position `i + j` rather than just at position `i`. `ilvVee` is a generalisation of both `ilv` and `vee`. `ilvVee i 0` has the same behaviour as `ilv i`, and `ilvVee 0 (j-1)` is the same as `vee j`. The `i` parameter controls the degree of interleaving, and the `j` parameter controls the size of the vee-shaped blocks.

For 16 inputs, the parameters to `ilvVee` that describe the periodic merger on the right of the construction are $i = 0$ (for no interleaving) paired with 3, 2, 1 and 0 for the decreasing size of

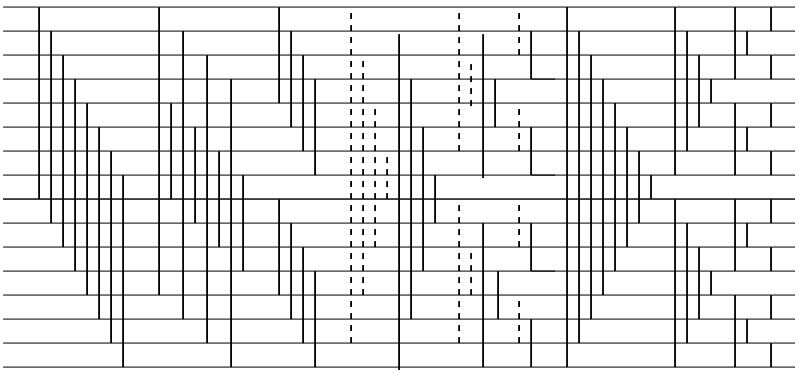

Figure 5. A sorter based on the idea that the periodic balanced merger network sorts two interleaved sorted sequences. It consists of two half-sized sorters, one working on the odd elements of the input and one on the even, followed by the balanced merger. The diagram indicates using dotted lines the balanced merger that is the final (rightmost) part of one of the half-size sorters.

```
unsigned int arrayLength = 1 << LOG_L_SIZE;
unsigned int diff = LOG_L_SIZE - LOG_S_SIZE;
unsigned int blocks = arrayLength / S_SIZE;
unsigned int threads = S_SIZE / 2;

sortSmall<<<blocks, threads,4096>>>(din,din);

for(int i = 0 ; i < diff ; i += 1){
  vSwap<<<blocks/2,threads*2,0>>>(din,din,(1<<i)*S_SIZE);

  for(int j = i-1; j >= 0; j -= 1)
    iSwap<<<blocks/2,threads*2,0>>>(din,din,(1<<j)*S_SIZE);

  bmergeSmall<<<blocks,threads,4096>>>(din,din);}
```

Figure 6. CUDA code for our large sorter. `sortSmall` and `bmergeSmall` are replaced by Obsidian-generated kernels in the experiments. `vSwap` and `iSwap` are handwritten CUDA kernels that perform one column of compare and swap operations in the vee and ilv shapes respectively, and are parameterised on the stride. (Our generated kernels have fixed input and output size.)

the vee-shaped blocks. Next, to the left, the mergers are interleaved ($i = 1$) and there are three stages with vee-shaped blocks of decreasing size ($j = 2, 1, 0$), see Figure 5. The following code gives an iterative description of the construction for 2^n inputs:

```
vsort :: Int -> Array IntE -> Kernel (Array IntE)
vsort n = composeS . map pure $ [istage (n-i) (i-j)
                                 | i <- [1..n], j <- [1..i]]
  where istage i j = ilvVee2 i j min max
```

The resulting generated code uses one thread per 2 indices.

3.6 Measuring performance of the generated kernels

We have measured the performance of generated 512-input sorting and merging kernels by plugging them into a larger sorter written in CUDA. The sorter has exactly the structure shown in Figure 3. Figure 7 shows the location of smaller sorters and mergers, and of vSwap and iSwap kernels for 16 inputs. Larger sorters simply have more columns of mergers, each preceded by a vSwap and a number of iSwaps. The overall structure of the resulting CUDA code is shown in Figure 6. Because `iSwap` is used repeatedly, we also wrote kernels corresponding to compositions of two and three of them (avoiding memory accesses between the columns). That is, we replaced the loop containing `iSwap` above by

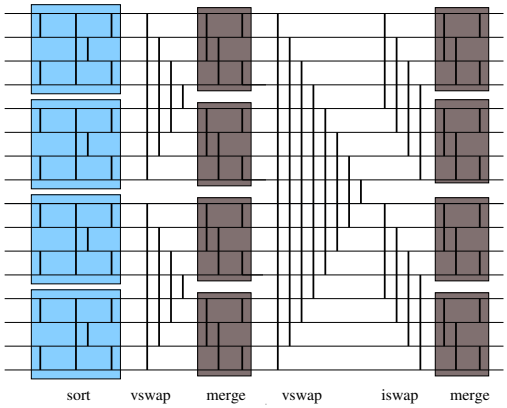

Figure 7. This diagram shows the tree-shaped sorting network that we saw earlier, but with 4 input sorting and merging kernels indicated. This is the structure of the network that we have used to implement large sorters, in which the small kernels having $2^9 = 512$ inputs in all cases. For 2^{24} inputs overall, the resulting sorter has one column of small sorters and $24 - 9 = 15$ of small mergers.

```
for(int j = i-1; j >= 0; j -= 3){
  if (j==0)
    iSwap<<<blocks/2,threads*2,0>>>(din,din,(1<<j)*S_SIZE);
    else
    {if (j==1)
       iSwap2<<<blocks/4,threads*2,0>>>(din,din,(1<<j)*S_SIZE);
       else
       iSwap3<<<blocks/8,threads*2,0>>>(din,din,(1<<j)*S_SIZE);}}
```

The Obsidian implementation and all code for examples in the paper are available at `http://www.cse.chalmers.se/~joels/expressive.html`.

Table 3.6 shows the performance figures for the large sorter for 5 different $512-$input small sorter kernels. `tsort1` is defined above, and is a variant of bitonic sort that does not require `if` statements to determine the direction of sorting of pairs, as all comparison operations place the minimum at the same index as the lower input. `tsort2` is the same construction, but built with the combinators `ilv2` and `vee2` that result in the use of one thread per two indices. `vsort`, the fastest kernel, uses a generalisation of those two combinators, and again uses one thread per two indices.

k	20	21	22	23	24	24(CPU)
bitonic(CUDA)(512)	5	12	25	54	117	2741
tsort1(512)	4	10	22	47	102	2742
tsort2(256)	4	9	23	45	98	2741
vsort1(512)	4	10	21	47	102	2747
vsort(256)	4	9	20	44	96	2783

Table 1. Sorting time (ms) for 2^k inputs, $20 \leq k \leq 24$. The GPU used is an NVIDIA GTX480. Each line shows the result using a particular small 512 input sort kernel with the indicated number of threads as `sortSmall` in the CUDA code above. The small merge kernel used is our generated `bmerge2`, with two indices per thread and 512 inputs. Memory transfer time (GPU-CPU) is not shown. In the 2^{24} input case using `vsort`, the total sorting time, including memory transfers was 141 ms. The rightmost column shows the time taken for the C quicksort function `qsort`, compiled with `gcc -O3`, to sort the same inputs as those sorted on the GPU on an i7-920 2.8GHz CPU.

	time	percent
bitonic(CUDA)	30203	32.07
tsort1	15823	19.72
vsort1	14955	18.76
tsort2	11562	15.37
vsort	9228	12.67
bitonic (NVIDIA SDK)	23815	6.55

Table 2. time: GPU time (in μs) spent in calls to small sorter kernels in the initial phase of the sorter for 2^{24} inputs. percent: percentage of total GPU time spent in the small sort kernel. The final line shows the numbers for the bitonic code (for large sorts) that is distributed with the the NVIDIA SDK. Its structure also starts with a phase of small sorting kernels. It takes 274 ms to sort 2^{24} key-value pairs on an NVIDIA GTX480 GPU. It could be sped up by using our `vsort` kernel, and also by (hand) fusing single "column" kernels as we did with `iSwap`, although the need to calculate directions of comparators in the bitonic network would complicate this. Our CUDA code is simpler because all comparators point in the same direction in the sorter construction.

`vsort1` is the same construction as `vsort`, but with `ilvVee2` replaced by `ilvVee1`, and so using one thread per index. As a reference small kernel implementation, we include a simple hand-coded bitonic sort that uses one thread per index (see code in Appendix). None of the kernels has been subject to optimisations related to warp size.

We also recorded the GPU time spent in the calls to the small sorters alone (using the NVIDIA CUDA Visual profiler), see Table 3.6. The `tsort1` and `vsort1` kernels, which are built using only pull arrays, (and which have one index per thread in the generated code) are noticeably slower than those that have two indices per thread. The generation of the latter kernels is made possible by the introduction of push arrays.

Although none of our generated kernels is optimised (for example with respect to warp size), their performance is, nevertheless, very good. We are working on automated warp size-related optimisations. It would also be interesting to explore the fusion of adjacent columns of comparators in the small kernels; omitting a `sync` would cause this fusion to happen but we also need to modify the threading behaviour (doubling the number of indices per thread).

4. Discussion

Push arrays form a new approach to array representation in DSELs. We do not know of similar approaches in the literature, despite the fact that the notions of demand and data flow may feel familiar to the reader who considers Pull and Push arrays. The addition of Push arrays to Obsidian seems highly beneficial. With this new feature, the user gains finer grained control over the code generated and the resulting CUDA kernels perform considerably better than before. This was illustrated in the series of sorters explored in section 3. The performance of `vsort` is sufficiently good that it can be used as a first phase in a larger sorter (written in CUDA) that can sort 16M elements in 96 ms, while an i7-920 CPU takes around 2740 ms. Further speed improvements look possible, both in the coordination code and in the kernels. An obvious next step would be to investigate the generation of the `iSwap` and `vSwap` kernels from Obsidian. (This is not currently possible because of assumptions that we made about the interfaces to kernels and about how *thread ids* are used. We will look into ways to relax our assumptions.)

The series of kernels also illustrates how the use of combinators brings a form of reuse, and makes design exploration easier. Our experience of using similar combinators in the Lava hardware description language [6] was that a relatively small set of combinators went a long way. So, although we introduced three combinators here, `ilv`, `vee` and `ilvVee`, which includes the other two, we do not believe that every new kernel development exercise would demand a completely new set of combinators. We expect to provide the user with a well-documented set of combinators, so that users can get access to this style of programming without having to develop their own combinators, and without having to think too much about bit-hacking. The bit-manipulation approach chosen to define our combinators automatically created functions that apply to sub-sequences of the input that are of an appropriate length.

In this paper, we made combinators for the special case of two input, two output operations (built from two two-input funtions that we typically called `f` and `g`). This approach should be generalised to deal with blocks that have 2^k inputs and outputs. Also, we made a compound combinator from `ilv` and `vee`, but generalising to more than two input components would allow for composing combinators, and indeed for recursive descriptions that could be unrolled. Then, ignoring `syncs`, a recursive description of `vsort` could be something like

```
vsortR 0 = id
vsortR n = bpmergeR n . ilv2 1 (vsortR (n-1))
```

It would then be necessary to optimise the code generated from multiple applications of `ilv2 1`, for example, whereas here we have forced the user to figure out both the unrolling and the combinations. Moving to more general combinators would also give the opportunity to provide predefined combinators that capture more of the commonly used threading patterns (for instance k indices per thread rather than the 1 and 2 shown here).

The integration of Push arrays into Obsidian raises some new questions. Previously, there was a direct correspondence between the length of an array and the number of threads used to compute it, which allowed the user to write an initial program without worrying about threads at all, and then to tweak the Obsidian program if he was not satisfied with the threading behaviour of the resulting kernel. Now, as can be seen in the `catArrayPs` example and in the sorters, this correspondence can be broken using Push arrays. The `catArrayPs` example and two of the sorters use half as many threads as the number of elements. For users who are very concerned about the speed of the generated kernels, getting this control through using Push arrays in a particular pattern is clearly a good thing. But adding a second, different way to control thread

use in the generated code certainly complicates matters, and further case studies are needed to confirm that the complication pays off.

The addition of Push arrays also adds the possibility to include potentially unsafe operations in Obsidian, for example by writing multiple array elements to the same index, or by discarding elements. This new expressiveness will have to be carefully controlled. On the positive side, it offers the possibility to encode functions like `filter` from Haskell that are simply not expressible using only Pull arrays. Being able to implement `filter` would make programming kernels in obsidian feel much more like programming in Haskell – a welcome loosening of the strait-jacket. Once that is done, it will be time to develop a very simple coordination language to allow programming of entire GPU applications that make use of the kind of small kernel building blocks developed here.

A. Appendix

```
__device__ inline void swap(int & a, int & b)
{
   int tmp = a;
   a = b;
   b = tmp;
}

__global__ static void bitonicSort(int * values, int *results)
{
   extern __shared__ int shared[];

   const unsigned int tid = threadIdx.x;
   const unsigned int bid = blockIdx.x;

   // Copy input to shared mem.
   shared[tid] = values[(bid*NUM) + tid];

   __syncthreads();

   // Parallel bitonic sort.
   for (unsigned int k = 2; k <= NUM; k *= 2)
   {
     // bitonic merge
     for (unsigned int j = k / 2; j>0; j /= 2)
       {
           unsigned int ixj = tid ^ j;

           if (ixj > tid)
           {
               if ((tid & k) == 0)
               {
                   if (shared[tid] > shared[ixj])
                   {
                       swap(shared[tid], shared[ixj]);
                   }
               }
               else
               {
                   if (shared[tid] < shared[ixj])
                   {
                       swap(shared[tid], shared[ixj]);
                   }
               }
           }
           __syncthreads();
       }
   }

   // Write result.
   results[(bid*NUM) + tid] = shared[tid];
}
```


Acknowledgments

This research has been funded by the Swedish Foundation for Strategic Research (which funds the RAW FP Project) and by the Swedish Research Council.

References

[1] E. Axelsson, K. Claessen, M. Sheeran, J. Svenningsson, D. Engdal, and A. Persson. The Design and Implementation of Feldspar an Embedded Language for Digital Signal Processing. In *Proceedings of the 22nd international conference on Implementation and application of functional languages*, IFL'10, pages 121–136, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] K. E. Batcher. Sorting Networks and Their Applications. In *Proc. AFIPS Spring Joint Computer Conference 32*, pages 307–314, 1968.

[3] M. Billeter, O. Olsson, and U. Assarsson. Efficient stream compaction on wide SIMD many-core architectures. In *Proc. Conf. on High Performance Graphics*, HPG '09, pages 159–166. ACM, 2009.

[4] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell array codes with multicore GPUs. In *Proc. sixth workshop on Declarative Aspects of Multicore Programming*, DAMP '11, pages 3–14. ACM, 2011.

[5] K. Claessen. A poor man's concurrency monad. *J. Funct. Program.*, 9(3):313–323, 1999.

[6] K. Claessen, M. Sheeran, and S. Singh. The Design and Verification of a Sorter Core. In *Proc. Int. Conf. on Correct Hardware Design and Verification Methods (CHARME)*, volume 2144 of *Springer LNCS*, pages 355–369, 2001.

[7] M. Dowd, Y. Perl, and M. S. Larry Rudolph. The Periodic Balanced Sorting Network. *Journal of the ACM*, 36:738–757, 1989.

[8] C. Elliott. Functional Images. In *The Fun of Programming*, "Cornerstones of Computing" series. Palgrave, Mar. 2003.

[9] C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(2), 2003.

[10] Joel Svensson. Obsidian: GPU Kernel Programming in Haskell. Technical Report 77L, Computer Science and Enginering, Chalmers University of Technology, Gothenburg, 2011. Thesis for the degree of Licentiate of Philosophy.

[11] G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 261–272, New York, NY, USA, 2010. ACM.

[12] B. Larsen. Simple optimizations for an applicative array language for graphics processors. In *Proc. sixth workshop on Declarative Aspects of Multicore Programming*, DAMP '11, pages 25–34. ACM, 2011.

[13] G. Mainland and G. Morrisett. Nikola: embedding compiled GPU functions in Haskell. In *Proc. third ACM Symposium on Haskell*. ACM, 2010.

[14] M. Sheeran. Describing Butterfly Networks in Ruby. In *Functional Programming*, pages 182–205. Springer Workshops in Computing, 1989.