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

Bo Joel Svensson
Indiana University
joelsven at indiana dot edu

Mary Sheeran
Chalmers University of Technology
ms at chalmers dot se

Ryan R. Newton
Indiana University
rrnewton at indiana dot edu

April 4, 2014

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 kernel performance.

To that end Obsidian includes guaranteed elimination of intermediate arrays and predictable space/time costs, while also providing array functions that are polymorphic across different levels of the GPUs' hierarchical structure, providing a limited form of nested data parallelism.

We walk through case-studies that demonstrate how to use Obsidian for rapid design exploration or auto-tuning, resulting in better performance than hand-tuned kernels in an existing GPU language.

1 Introduction

Graphics Processing Units (GPUs) offer the potential for high-performance implementations of data parallel computations. Yet achieving top performance is recognized as a difficult task, requiring expert programmers with the ability and time to manually optimize use of on-chip storage, make granularity decisions, and match memory access patterns to the [non-traditional] constraints placed by GPU memory architectures (*i.e.* not just temporal memory patterns, but the coordination of accesses across groups of threads). Accordingly, programs are written in a low-level vendor-supplied programming environments, such as NVIDIA CUDA, where all these details are under programmer control.

One answer to the high cost of GPU programming is to attempt to *automate* the process, in particular by starting with a very high-level language and using an optimizing compiler to make the aforementioned decisions, synthesizing code in a language like CUDA. Indeed, many recent research projects have done just this, including: Copperhead [4], Accelerate [6, 16], Harlan [12], and Delite [5]. These languages are first and foremost *array languages*, intentionally restricted versions of older languages such as APL [13], and Matlab. Typical operations include mapping, filtering, scanning, and reducing array data. By restricting program structure, this language family gains one major benefit over more general purpose array languages: they can very effectively fuse series of array operations, eliminating temporary arrays.

Pitfalls of abstraction The problem with aggressive abstraction approaches to GPU programming, is that they remove the control necessary for the *design exploration* process that remains a critical of the current process for porting algorithms to the GPU. Much like a computer architect, a programmer working

to GPU-accelerate an application kernel must go far beyond their initial version (typically ported from CPU code), and must iterate through several different designs, experimenting with tradeoffs. Often the final result is more than an order of magnitude faster than the starting point.By contrast, a language like *Accelerate* abstracts GPU programming to the point that there is a single way to express each communication pattern, for example prefix sum becomes "`scanl (+) 0 arr`", with no tuning parameters. In fact, *all* of the following optimization tools are lost:

- Controlling how many kernels are launched
- Controlling which arrays are mapped to on-chip (local) memory.
- Controlling synchronizations points (*e.g.* `__syncthreads`)

Further, because systems like Accelerate depend on compiler optimization for performance, there is not a fixed cost model for the time and space cost of operations, which may or may not be fused, deforesting intermediate arrays. One day, hopefully this automation will work well enough to obsolete the human from the performance tuning process, but it hasn't yet.

A language for rapid design exploration In this paper we argue that it is possible to make a more surgical strike in choosing what to abstract in GPU programming. We propose a small language, Obsidian, that leaves the above controls in the programmers hands while providing three key benefits over CUDA programming:

1. Abstracting over constant limits (*virtualization* of warps and blocks)
2. Systematic generation of code variants, traditionally addressed in domain-specific languages (DSLs) by *metaprogramming*, which enables both design exploration and easier auto-tuning.
3. *Compositional* array operations that also offer *hierarchy polymorphism*: the same programming primitives at thread, warp, block, and grid level.

The most unique benefit of Obsidian is in the last point. First, Obsidian uses a combination of *push-* and *pull-arrays*, in the meta-language (Section 4). 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. This makes the cost model fully transparent. Second, Obsidian *exposes* the hierarchical nature of GPU hardware (directly in the type system), while core data operations work at any level. As such it allows a limited form of nested data parallelism (NDP [3]), with nestings only as deep as the machine hierarchy itself[1]. Obsidian is the first language to our knowledge to use either of these techniques.

In this paper, we present the design and implementation of Obsidian and demonstrate that where high-level DSLs have highly tuned fixed operations (such as a prefix sum), we can generate those same results and also explore the nearby design landscape. Moreover, where high-level DSLs *fail* to produce good performance, Obsidian provides the tools to drill down and fix the problem. Yet in spite of that low-level control, embedding, metaprogramming, and novel array representations enable better code reuse than CUDA, comparable to higher level DSLs.

2 Background: The GPU and CUDA

Obsidian targets NVIDIA GPUs supporting CUDA [18], 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 of these multiprocessors. The GPUs

[1] Again, without employing any automatic optimizations, of which NDP flattening transformations would be one example.

used in our measurements are an NVIDIA Tesla c2070 and a GTX 680. The GTX680 GPU has eight multiprocessors, with a total of 1536 processing cores. On these cores, groups of 32 threads called *warps* are scheduled. There are a number of warp scheduling units per multiprocessor. Within a warp, threads execute in lockstep (SIMD); diverging branches, that is those that take different paths on different threads within a warp, are serialised, leading to performance penalties.

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

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

The prototypical CUDA kernel starts out by loading data from global memory. The indices into global memory for an individual thread are expressed in terms of the unique identifier for that block and thread. Some access patterns allow memory reads to be *coalesced*, while others do not, giving very poor performance. The patterns that lead to good performance vary somewhat between different GPU generations, but regular, consecutive accesses by consecutive threads within a warp are best.

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 GPU-accelerating existing applications.

3 Obsidian Programming Model

Obsidian is an Embedded Domain Specific Language (EDSL), implemented in Haskell. When running an Obsidian program—which is really just a Haskell program using the Obsidian libraries—a data structure is generated encoding an abstract syntax tree (AST) in a small embedded language. Embedded languages that generate ASTs are traditionally called *deeply embedded* languages[2]. The creation of an AST offers flexibility in interpretation of the DSL. In Obsidian's case the AST is used for CUDA code generation. For an excellent introduction to compiling embedded languages, see reference [9]. As a result of the embedding, the following function, when invoked, does not immediately increment any array elements. Rather, computation is both deferred and extracted into an AST:

```
incLocal arr = fmap (+1) arr
```

EDSLs in Haskell, like those in Scala [5] and C++ [17], tend to use an *overloading* approach, resolved at compile time, to extend operations like `(+)` to work over AST types in addition to actual numbers. Dynamic languages instead tend to use *introspection* [4, 11] to disgorge the code contents of a function object and acquire an AST for domain-specific compilation. While these AST-extraction methods are largely interchangeable, there are other issues of representation that have a big effect on what is possible in the DSL compiler, namely: *array representation*.

In Obsidian, there are two different [immutable] array representations, *Pull* and *Push* arrays, neither of which commits to an in-memory, *manifest* data representation. Pull arrays are implemented as a function from index to element, with an associated length. A consumer of a pull array needs to iterate over those indices of the array it is interested in and apply the pull array function at each of them. A push array, on the other hand, encodes its own iteration schema. Any consumer is forced to use the push array's built-in

[2] An alternative to a deep embedding is a shallow embedding, one that does not generate ASTs but rather implement the DSL directly in terms of their semantics.

iteration pattern[3]. Indexing is a cheap operation on pull arrays, but on push arrays it requires traversal of the entire array in worst case. Both representations can safely avoid bounds checks for typical combinations of array producers and consumers.

The `incLocal` function above operates on pull arrays, so both its input and output type are `(Pull` *size num*`)`, *e.g.* `(Pull Word32 EWord32)`. The difference between a `Word32` and a `EWord32` is related to the embedded nature of Obsidian. An `EWord32` (short for `Exp Word32`) is a data structure (an AST) while an `Word32` is a value. The `Word32` type (rather than `EWord32`) is used for lengths of arrays in local memory; thus ensuring that these array sizes are known when Obsidian CUDA code generation occurs. For simplicity of presentation we will err on the side of monomorphism, avoiding generic types where they are not directly required to illustrate the point. For example:

```
incLocal :: Pull Word32 EWord32 → Pull Word32 EWord32
```

Adding parallelism "Local", in the name of the function above, is a hint that we're not yet entirely done. While `incLocal` completely describes the *computational* aspects of this example, it does not describe how that computation is laid out on the GPU. Obsidian, like CUDA, differentiates between *Thread*, *Block* and *Grid* computations. While CUDA provides no abstraction for warps, Obsidian does. The programmer specifies how the computation is laid out over the available parallel resources. For example, a sequential computation to be carried out by each thread. Many instances of a sequential computation can then be run in parallel across the threads of a Warp, Block or Grid.

For example, to turn the parallelism-agnostic `incLocal` function into a function that executes GPU-wide, we use *push* to apply an iteration schema:

```
incPar :: Pull Word32 EWord32 → Push Grid Word32 EWord32
incPar arr = push (incLocal arr)
```

This function is still *cheap* in the sense that it does not make the array manifest in memory. The behavior of the push array is also type-directed; if we had changed `Grid` to `Thread`, we would get a sequential rather than parallel loop. Likewise, if we see a `(Push Block` *size num*`)` array, we know it is an array computed in parallel across the threads within *one* block on the GPU.

In CUDA, blocks are limited to a maximum of 1024 thread. This limitation does *not* hold in Obsidian, because threads within a block are virtualized. Virtualization of threads is explained further in Section 4. Hiding these hardware limits makes it easier to quickly switch between different mappings of loop nests onto the hardware hierarchy—one of the main benefits of Obsidian for enabling design exploration. Second, because parallel loops are *implicit* in CUDA kernels (unlike, *e.g.*, OpenMP or Cilk), switching between parallel and sequential loops in CUDA requires changing much more code than a one-word tweak to the array type. Third, Obsidian arrays offer a modularity advantage: the logic of the program can be defined at a point far removed from where loop structure decisions are made.

Limited nested parallelism To explore interesting loop structures, we need nested array operations. In Obsidian, we can split arrays into chunks of size n with `splitUp`, and then concatenate them again with `pConcat`, obeying this law:

```
pConcat (splitUp n arr) == push arr
```

The `splitUp` function takes a chunk size (a `Word32`), a known-at-compile-time value[4]. However, the *length* of an array can be either static or dynamic (`Word32` or `EWord32`). Many Obsidian functions are limited to static sizes; code generation depends on this. The dynamic lengths are an added convenience—after specifying a local [fixed-size] computation, it can be launched over a varying number of GPU blocks. For a description of all core Obsidian array functions, see Figure 14, and for the full type signatures of `splitUp` and other operations, see Figure 11 in the Appendix.

[3]Non-parallel push arrays are similar to the *generators* found in many languages, but Obsidian's staged execution removes the necesity for any coroutine-like control flow with "yield".

[4]Obsidian compile time is Haskell runtime; so, as is typical for metaprogramming systems, it is still possible to build arbitrary computations that construct these "static" Obsidian values.

```
-- Enter into hierarchy
tConcat :: Pull l (Push Thread Word32 a) → Push t l a

-- Step upwards in hierarchy
pConcat :: Pull l (Push Word32 t a) → Push (Step t) l a

-- Remain on a level of the hierarchy
sConcat :: Pull l (Push t Word32 a) → Push t l a
```

Figure 1: GPU hierarchy programming API, contains functions to spread computation across parallel resources in a level of the GPU hierarchy. These could be combined into a single polymorphic `concat` operation, but doing so would lose the benefits of type inference (requiring tedious explicit type annotations on every `concat`).

The next program describes how to spread local work out over several of the GPU blocks. The input to this function is an array of arrays, with each inner array as the input to an instance of `incLocal`.

```
increment :: Pull _ (Pull _ _) → Push Grid _ _
increment arr = pConcat (fmap body arr)
  where body a = push (incLocal a)
```

The `increment` program uses `pConcat` to execute several instances of `incLocal` in parallel across the block level of the GPU hierarchy, thus forming a grid. The type of `pConcat` forces the computation to step up *one level* in the hardware hierarchy. It's signature is

```
Pull l (Push s t a) → Push (Step t) l a
```

where `(Step t)` is a type-level function that transforms, *e.g.* Warp into Block. Because `(Step t) = Grid` in the `increment` function above, the type checker inferred that `t = Block`.

But why does `pConcat` return a push array? Because it is more efficient for `pConcat` to build its own iteration schema (for example, pushing chunk 1, chunk 2, etc in sequence), rather than form a a chain of conditionals (based on index i are we in chunk n?).

Loop structure experimentation The application of `pConcat` in `increment` creates a nested parallel loop structure equivalent to: `parfor (...) { parfor (...) body(..); }` But this is only one of the possible decompositions of this computation over the parallel resources of the GPU. Another way would be to create a loop nesting with a sequential innermost loop, wrapped in two parallel for loops. This decomposition is shown below.

```
increment2 :: Pull _ (Pull _ (Pull _ _)) → Push Grid _ _
increment2 arr = pConcat (fmap body arr)
  where body a = tConcat (fmap push (fmap incLocal a))
```

which corresponds to a loop-nest `parfor/parfor/for`. Another reason loop-structure changes are difficult in CUDA is that there is only *one level* of implicit parallel loops (the kernel). Simulating nested parallel loops requires tedious index computations, which, here, Obsidian handles automatically.

In Figures 12 and 13, the generated code for `increment` and `increment2` are shown. The block level and grid level parallel for loops are implicit in the CUDA programming model. For loops in the generated code are sequential loops that originate either from sequential loops directly from our Obsidian program (`increment2`), or from programs that "spill" over the threads-per-block and blocks-per-grid limits (Section 4).

Programs and Parallelism Finally, in addition to push/pull array values and expressions, Obsidian contains one more AST data type called `Program`, capturing program *effects* such as a push array feeding its outputs into a [manifest] array in local storage. In Section 4, we will see how push arrays internally encode their iteration schemas by generating snippets of `Program` AST. As is standard for monadic values in Haskell, a `(Program t a)` value represents a *computation*, which, when run, results in a value of type `a`.

Programs are parameterized on a level of the GPU hierarchy, `(Program` *level*`)`. The type system ensures that only programs that meet the GPU constraints can be generated: For example, threads participating in a barrier synchronization must always be in the same block. Or: no `(Program Thread)` can be created that uses a parallel for loop. For an Obsidian program to exhibit parallelism it needs to either result in a push array, or `(Program` *level* `(Pull` *size num*`))`.

Transparent Cost Model One of the goals of Obsidian is to provide a transparent cost-model. Thus the user should clearly know how much memory and computation each operation requires, and also how large of an expression their Obsidian kernel generates. As one example of a program that can get us into trouble, consider the following function for summing an array of numbers:

```
sumUp :: Pull Word32 EWord32 → EWord32
sumUp arr
  | len arr == 1 = arr ! 0
  | otherwise    =
     let (a1,a2) = halve arr
         arr2    = zipWith (+) a1 a2
     in sumUp arr2
```

Here, `zipWith` (a two-argument *map*) operates on pull arrays and returns another. Following Obsidian's *de-facto* fusion policy, it does not use any memory for arrays. However, because the divide-and-conquer recursion above happens at compile time, `sumUp` generates a large $O(N)$-sized expression to sum all the elements of the array[5]! For example:

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

For small arrays, this might be code might be ideal. But `sumUp` would need to be used with care; it precludes parallelism, and it shouldn't be used on larger arrays.

3.1 Using Force: Parallelism and Shared Memory

Of course, arrays can't always stay *non-manifest*. The Obsidian library comes with a family of "force"-functions (`force`, `forcePull`), which serve three roles:

1. **Make array manifest in memory:** For sharing of computed results between threads.
2. **Expose parallelism:** Forcing a pull array `(forcePull arr)` sets up an iteration schema over its range and computes the pull array function at each index. The result of forcing a pull array is a `(Program` *level* `(Pull` *size num*`))` array. forcing a push array instantiates the iteration schema encoded in the push arrays and writes all elements to memory using that strategy. Forcing a `(Push` *level size num*`)` array results in a `(Program` *level* `(Pull` *size num*`))` array.
3. **Conversion:** from push array to pull array, enabling cheap indexing.

A single call to `forcePull` transforms the `sumUp` program into a binary tree shaped parallel reduction:

```
sumUp' :: Pull Word32 EWord32 → Program Block EWord32
sumUp' arr
  | len arr == 1 = return (arr ! 0)
  | otherwise    =
   do let (a1,a2) = halve arr
      arr2 ← forcePull (zipWith (+) a1 a2)
      sumUp' arr2
```

[5]This problem, *over elaboration*, is a potential user error in all embedded DSLs. For example, in Intel ArBB (embedded in C++), if one forgets to use `for_` instead of `for` they evaluate a loop at compile time that was meant for runtime (fulling unrolling it).

This version has also been changed to Haskell's `do` notation for monadic computations. The statement `arr2 ←forcePull ...` creates a manifest intermediate array that all threads within that block can access. The code generated from `sumUp'` has the following form:

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

3.2 Programming Blocks and Warps

The `increment` example in section 3 already showed how to apply a hierarchy-agnostic function on pull arrays at different levels of the GPU's hierarchy. To have a complete cost-model, it is also important for the user to understand the meaning of memory operations at the `Warp` and `Block` levels, and the rules for automatic synchronization insertion. Here we will illustrate those rules with a simple example:

```
agnostic arr =
  do imm1 ← forcePull (fmap (+1) arr)
     imm2 ← forcePull (fmap (*2) imm1)
     imm3 ← forcePull (fmap (+3) imm2)
     return (push imm3)
```

Because the `agnostic` function uses force, some constraints apply. For example, this push array cannot be instantiated at the grid level, as we did with the previous `incLocal` example. Rather, we must instantiate `agnostic` at the `Block` level or below, where synchronized communication via shared memory is possible.

As with `increment`, if we want to distribute the `agnostic` function over individual blocks, we can take a larger array, chunk it with `splitUp 256 arr`, and then `fmap` the `agnostic` function over each chunk, and finally flatten the result back out with `pConcat`, which generates code following this pattern:

```
parfor (i in 0..255) {
   imm1[i] = input[blockID * 256 + i] + 1;
   __syncthreads();
   imm2[i] = imm1[i] * 2;
   __syncthreads();
   imm3[i] = imm2[i] + 3;
   __syncthreads();
}
```

Note that each stage is followed by a barrier synchronization operation[6]. It is also possible to place the `agnostic` computation on the warp level. This can be done by splitting the input pull array into a three level nested pull array: *e.g.* `fmap (splitUp 32) (splitUp 256 arr)`. Each warp of a blocks operate on the innermost chunks, and the resulting code follows this pattern:

```
parfor (i in 0..255) {
  warpID = i / 32;
  warpIx = i % 32;
  imm1[warpID * 32 + warpIx] =
    input[blockID * 256 + warpID * 32 + warpIx] + 1;
  imm2[warpID * 32 + warpIx] =
    imm1[warpID * 32 + warpIx] * 2;
  imm3[warpID * 32 + warpIx] =
    imm2[warpID * 32 + warpIx] + 3;
}
```

All the synchronization operations disappeared, because a warp-level program is naturally lockstep (SIMD/SIMT).

[6]Indeed, in this simple example the synchronizations are unnecessary, and the user should not have used `forcePull`!

Constructor	Arguments	Notes	Description
`Assign`	*name*, *val_exp*, *ix_exp*	$name[ix] = val$	-
`ForAll`	*range*, *body*	body is represented by a function	The *body* is a Thread-level program that is executed *range* number of times on a level (Thread, Warp, Block, Grid)
`DistrPar`	*range*, *body*	body is represented by a function	The *body* is a level `t` program that is spread out in parallel over level `(Step t)` in the hierarchy
`SeqFor`	*range*, *body*	body is represented by a function	A sequential loop, the program remains on the same level of the hierarchy as the *body*
`Allocate`	*name*, *size*, *type*	-	Allocate space for array *name* in shared memory
`Declare`	*name*, *type*	-	declare a variable *name*
`Sync`		-	Barrier synchronization across all threads of a block
`Seq`*	*program*, *program*	-	sequences of statements

Figure 2: A list of some constructors from the program AST data type, `(data Program t a)`.
**In the implementation sequences of statements are not really provided by a `Seq` constructor, but rather via making the `Program` data type a monad. Sequencing is then 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}`*

4 Obsidian Implementation

The Obsidian compiler deals with two types of ASTs: scalar expressions (*e.g.* `EWord32`), and `Programs` (statements). Scalar expressions include standard first-order language constructs (arithmetic, conditionals, etc). Obsidian source expressions such as `(5+1)`, elaborate into standard Haskell algebraic datatypes[7], *e.g.* `(BinOp Add (Literal 5) (Literal 1))`. There are also two GPU specific variables in the `Exp` grammar, `ThreadIdx` and `BlockIdx` referring to the [eventual] CUDA thread identity—for internal use by the Obsidian compiler only. The second AST, `Program`, is Obsidian's imperative core language, with data constructors listed in Figure 2.

Pull arrays Pull arrays are indeed implemented as functions from index to [expression] value. This is a common representation for *immutable* arrays and allows easy implementation of many interesting operations, such as `map`, `zipWith` and permutations.

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

The embedded language Pan [8] used a similar representation for images and was the main inspiration for Obsidian's pull arrays. Contemporary languages Feldspar [1] and Repa [14] also use the same array representation.

Push arrays Push arrays are implemented on top of the `Program` data type. Where pull arrays are a function that returns an element for each index, a Push array is a *code generator*: a function that returns a `Program` action.

```
data Push t s a =
   MkPush s ((a → EWord32 → Program Thread ())
             → Program t ())
```

Each push array is waiting to be passed a *receiver* function, which takes a value (`a`) and index (`EWord32`), and generates single-threaded code to store or use that value. Given a receiver, a push array is then responsible

[7] GADTs actually, in the current implementation: `https://github.com/svenssonjoel/Obsidian`

for generating a program that traverses the push array's iteration space, invoking the receiver as many times as necessary.

Warp/Block Virtualization The length of an array, the `s` parameter, can be either static (a Haskell known-at-compile-time value) or dynamic (a runtime value). Static lengths are used for local (or block) computations, with those lengths determining shared memory consumption and parallel and sequential loop sizes. When, an array size is larger than the hardware limit on a warp or block size, compiler-enabled *virtualization* of blocks and warps occurs. Implementing this only requires inserting an additional sequential loop at the relevant level, to make multiple passes.

4.1 Push and pull array interplay

Forcing arrays to memory (Section 3.1) is a function overloaded on hierarchy level. It's type is:

```
force :: Push t Word32 a → Program t (Pull Word32 a)
```

with very different implementations at each level (*i.e.* different `t`'s). For example, below is pseudo code of `force` at the block level:

```
force (MkPush size p) = do
   name ← gensymname
   Allocate name size type
   p (Assign name)
   Sync
   return (MkPull size (λix →Index name ix))
```

Converting in the other direction, pull array to a push array, is cheap and is done using a function called `push` that also behaves differently (sequentially or in parallel) at different levels of the GPU hierarchy. This is implemented as a type class:

```
class Pushable t where
  push :: ASize s ⇒ Pull s e → Push t s e
```

`ASize`, an additional type class, has instances for both the static and dynamic lengths, both of which are internally converted (via `sizeConv`) into `Exp`, after noting the known sizes.

There are four instances of the Pushable class, for each level of the hierarchy:

```
instance Pushable Thread where
  push (MkPull n ixf) =
    MkPush n
     (λwf → SeqFor (sizeConv n) (λi → wf (ixf i) i))

instance Pushable Warp where
  push (MkPull n ixf) =
    MkPush n
     (λwf → ForAll (sizeConv n) (λi → wf (ixf i) i))
-- Block: same structure as Warp
-- Grid: same structure as Warp
```

Now, the `push` function captures just one possible way to convert a pull array into push array—with one write per thread. Conversion of pull arrays into push arrays can be done in many ways. For example, more than one element per thread one could be written by each thread, and then choices of stride length come into play. For example, one specialized "push"-function available in the Obsidian library is `load`:

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

```
m = len arr
n' = m `div` n
```

The `load` function combines sequential and parallel loops in pushing the pull array. The reason it is called `load` is its intended use as a initial load coalescer (to coalesce the first load a kernel performs from global memory).

Finally, just like push arrays, pull arrays can be forced (made manifest in memory). A Pull array is forced by converting them to push `forcePull arr = force (push arr)`

4.2 Compilation to CUDA

During Haskell evaluation, operations like `fmap` and `zipWith` disappear, leaving the an explicit AST with `Exp`, `Program`, and the `MkPush`/`MkPull` constructors. After this point, the Obsidian compiler begins, and proceeds through the following phases:

1A Reification: Haskell functions representing Obsidian programs are turned into ASTs, including generating names for arrays.

1B Stripping: The `Program` *level* datatype is converted from a higher-order representation to a list of statements (`IM` datatype) that make the hierarchy level of parallel loops concrete.

2A Liveness Analysis: The `IM` is analyzed to discovering the live ranges of arrays in shared memory. This stage annotates the `IM` with liveness information, that keeps track of where an array is created and at what point it can be freed.

2B Memory Mapping: The annotated AST goes through a simple abstract interpretation, simulating it in order to create a memory map. Then, all arrays are renamed with direct accesses to its allotted memory offset.

3 CUDA Code Generation: At this stage, explicit for loops in the `IM` are compiled into CUDA. This is where virtualization of threads, warps and blocks take place.

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
```

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

```
(MkPull n (λix →Index "input" ix)).
```

The function then yields its push array result. That push array, in turn, is a `Program` parameterized on a write-function. Providing the push array with a receiver-function. Providing a receiver, such as `(λa ix →Assign "output" a ix)`, which writes to a named (global) array, completes reification.

Liveness Analysis and Memory Mapping The `force` functions, that introduce 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.

There are 48Kb of shared memory available on each GPU multiprocessor, so it is a limited resource. Making good use (and reuse) of it is important. Obsidian `Program` AST already contains `Allocate` nodes, that shows where an arrays comes into existence, and we compute the full live range each array with a standard analysis:

- Step through list of statements in reverse. When encountering an array name for the first time it is added to a set of live arrays. The list of statements is annotated with this liveness information.

- When an `Allocate` statement is found, the array being allocated is removed from the set of live arrays.

Following this analysis phase a memory map is constructed using a greedy strategy. This is done by simulating the AST execution together against an abstraction of the shared memory. The simulated shared memory is implemented as a list of free ranges and a list of allocated ranges. "malloc" requests are serviced with the first available memory segment of sufficient size. The maximum size ever used is tracked, and in the end this is the total amount of shared memory needed for this kernel. After creating the memory map, the list of statements is traversed again and all array names are replaced with their location in shared memory.

Finally, this can potentially lead to memory fragmentation, and an the greedy solution is certainly not optimial. However, (1) in practice we see local arrays either of the same size or shrinking sizes (divide and conquer), and (2) unlike traditional register allocation, this process primarily affects *whether* a kernel will compile, not its performance: we do not spill to main memory. The upside of automatic shared memory managemnent is that it makes it much easier to reuse and remap shared memory within a large kernel, than it would be in CUDA. In CUDA you would need to allocate a local array and then manually cast portions of it for reuse—tedious and error prone.

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

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

The interesting cases are those that deal with parallelism: *e.g.* the `ForAll` and `DistrPar` statements. For example, compiling a parallel-for over threads in a block would have the following structure:

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

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

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

5 Case studies

The question we want to ask about Obsidian, is not directly "how fast is it"? Because the program synthesis abstractions we have described do not add overhead, achievable performance remains the same as CUDA

generally. Rather, we explore how Obsidian helps navigate the design space around a solution (manually, or through an auto-tuner).

The following case studies start with a simple kernel, embarassingly parallel with no inter-thread communication. Even with such a kernel, there is non-trivial tuning to maximize throughput. The remaining case studies consider key building blocks, reduce and scan, that have data-flow graphs involving much more communication. We compare these against the corresponding kernels within the *Accelerate* implementation, a much higher level DSL but one with hand-tuned (but not auto-tuned) CUDA skeletons for patterns like scan and fold[8].

5.1 Mandelbrot Fractals

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 to generate the fractal presented here is based on a sequential C program from reference [21].

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

```
xmax =  1.2 :: EFloat; xmin = -2.0 :: EFloat
ymax =  1.2 :: EFloat; ymin = -1.2 :: EFloat
```

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 range from 0 to 512 and similarly for the y coordinate and the imaginary component.

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

The image is generated by iterating the function presented above. We map the height of the image onto blocks of executing threads. Each row of the image is computed by one block of threads. This means that for a 512×512 pixel image, 512 blocks 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 if they reach 512 we break out of the iteration.

```
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 (x,y,iter) = ((xsq + ysq) <* 4) &&* iter <* 512
  where
    xsq = x*x
    ysq = y*y
```

The number of iterations that are 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).

[8] In fact, Accelerate and similar languages can *only* generate applications that are chains of these skeletons. So in some sense measuring these skeletons captures everything interesting about that language from a performance perspective.

size	32	64	128	256	512	1024
256	0.25	0.17	**0.12**	0.21	0.33	0.60
512	0.71	0.43	**0.34**	0.41	0.69	1.16
1024	2.41	1.39	**1.05**	1.22	1.53	2.58
2048	8.86	4.98	**3.67**	3.88	4.69	5.95
4096	34.21	18.82	**13.69**	14.07	15.36	18.65

size	32	64	128	256	512	1024
256	0.44	0.38	0.41	**0.36**	0.41	0.98
512	1.44	1.16	1.17	1.16	**1.14**	2.00
1024	5.12	3.96	**3.95**	3.98	4.17	4.75
2048	18.80	14.53	**14.38**	14.48	14.84	17.50
4096	72.12	55.36	**54.94**	55.16	55.67	61.89

Figure 3: Running times for the Mandelbrot program. The left table shows times measured on an NVIDIA GTX680 GPU. The rigth table shows times measured on an NVIDIA TESLA c2070. The columns varies the number of threads per block, while the rows varies image size. Each benchmarks was executed 1000 times and the total time is reported in seconds. The transfer of data to or from the GPU is not included in the timing measurements.

Figure 4: 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.*

```
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 iterations for each pixel location, by implementing a `genRect` functions 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 =
  pConcat (mkPull bs (λbid →
      (tConcat (mkPull ts (p bid)))))
```

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

```
mandel = genRect 512 512 iters
```

5.2 Reduction

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

To illustrate the kind of low level control that an Obsidian programmer has over expressing details of a kernel, we show a series of reduction kernels, each with different optimisations applied. Many of the optimisations applied to the kernels can be found in a presentation from NVIDIA [10].

Figure 5: Left: ***BAD*** *Adding sequential reductions like this, reintroduces memory coalescing issues. Consecutive threads nolonger access consecutive memory locations.* Right: ***GOOD*** *Using sequential reduction but maintaining coalescing*

This section focuses on local reduction kernels (on-chip storage only). The construction of large reduction algorithms from these kernels will be illustrated in section 6.

5.2.1 Reduction 1

Our first attempt at reduction combines adjacent elements repeatedly. This approach is illustrated on the left of Figure 4. 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 :: MemoryOps a
      ⇒ (a → a → a)
      → Pull Word32 a
      → Program Block a
red1 f arr
  | len arr == 1 = return (arr ! 0)
  | otherwise    =
    do let (a1,a2) = evenOdds arr
       imm ← forcePull (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, `pConcat` is used, as before:

```
mapRed1 :: MemoryOps a
        ⇒ (a → a → a)
        → Pull EWord32 (SPull a)
        → Push Grid EWord32 a
mapRed1 f arr = pConcat (fmap body arr)
  where
    body arr = singletonPush (red1 f arr)
```

This kernel does not perform well (Figure 6), which may be attributed to its memory access pattern. Remember that one gets better performance on memory access when consecutive threads access consecutive elements, which happens if each thread accesses elements that are some stride apart.

5.2.2 Reduction 2

`red2` lets each thread access elements that are further apart. It does this by halving the input array and then using `zipWith` on the halves (see Figure 4). This choice can only be made if the operator is commutative.

```
red2 :: MemoryOps a
      ⇒ (a → a → a)
      → Pull Word32 a
      → Program Block a
red2 f arr
  | len arr == 1 = return (arr ! 0)
  | otherwise    =
    do let (a1,a2) = halve arr
       arr2 ← forcePull (zipWith f a1 a2)
       red2 f arr2
```

5.2.3 Reduction 3

The two previous implementations of reduce write the final value into shared memory (as there is a `force` 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 `force`.

```
red3 :: MemoryOps a
           ⇒ Word32
           → (a → a → a)
           → Pull Word32 a
           → Program Block a
red3 cutoff f  arr
  | len arr == cutoff =
    return (foldPull1 f arr)
  | otherwise =
    do let (a1,a2) = halve arr
       arr2 ← forcePull (zipWith f a1 a2)
       red3 cutoff f arr2
```

This kernel takes a `cutoff` as a parameter and when the array reaches that length, sequential fold over pull array is used to sum up the remaining elements. setting the `cutoff` to two does not change the overall depth of the algorithm, but since there is no force in the last stage the result will not be stored in shared memory.

5.2.4 Reduction 4

Now we have a set of three basic ways to implement reduction and can start experimenting with adding sequential, per thread, computation. `red4` uses `seqReduce`, which is provided by the Obsidian library and implements a sequential reduction that turns into a for loop in the generated CUDA code. The input array is split into chunks of 8 that are reduced sequentially. The partial results are reduced using the previously implemented (`red3`).

```
red4 :: MemoryOps a
           ⇒ (a → a → a)
           → Pull Word32 a
           → Program Block a
red4 f arr =
  do arr2 ← force (tConcat (fmap (seqReduce f)
                                 (splitUp 8 arr)))
     red3 2 f arr2
```

As can be seen by the running times in Figure 6, this optimisation did not come out well. The problem is that it reintroduces memory coalescing issues (see Figure 5).

5.2.5 Reduction 5

With `red5`, the coalescing problem is dealt with by defining a new function to split up the array into sub arrays. The idea is that the elements in the inner arrays should be drawn from the original array in a strided fashion.

```
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
```

With `coalesce` in place of `splitUp`, `red5` can be defined as:

```
red5 :: MemoryOps a
      ⇒ (a → a → a)
      → Pull Word32 a
      → Program Block a
red5 f arr =
  do arr2 ← force (tConcat (fmap (seqReduce f)
                                 (coalesce 8 arr)))
     red3 2 f arr2
```

5.2.6 Reductions 6 and 7

Lastly, we try to push the tradeoff between number of threads and sequential work per thread further. `red6` and `red7` represent changing `red5` to reduce 16 and 32 elements in the sequential phase. The performance of the fastest of these kernels is very satisfactory, at a level where the kernel is *memory bound*, that is, constrained by memory bandwidth.

We augment `red5` with a parameter saying how much sequential work should be performed.

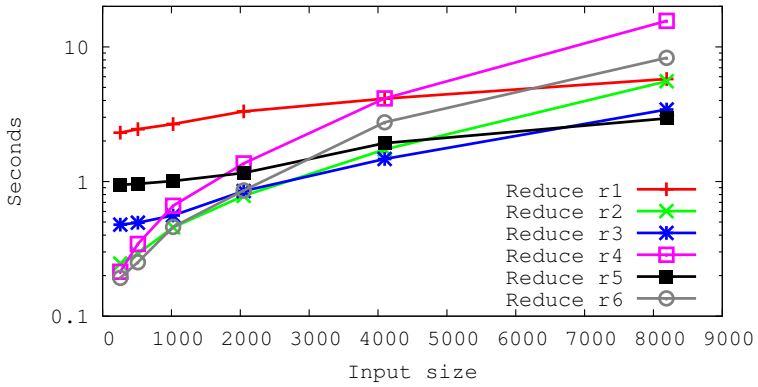
```
  red5' :: MemoryOps a
            ⇒ Word32
            → (a → a → a)
            → Pull Word32 a
            → Program Block a
  red5' n f arr =
    do arr2 ← force (tConcat (fmap (seqReduce f)
                                   (coalesce n arr)))
       red3 2 f arr2
```

```
  red6 f arr = red5' 16 f arr

  red7 f arr = red5' 32 f arr
```

Lines of Code Figure 7 lists the number of lines of code for each of the reduction kernels. The reduction benchmarks was based, in spirit, on the reduction optimization tutorial from by NVIDIA [10], and as a comparison the CUDA kernels shown in that tutorial we estimate to range between 10 and 19 lines of code; not counting lines containing just a "}" character or type signatures. Likewise for our Obsidian code the type signature has been left out of the count and we have not counted the lines in the very restrictive vertical space offered in the papers format, but rather how the code would look using more standard line length. Notable in the lines of code count is that as we apply more optimisations reuse of prior effort leads to less and less added new work, this is one strength of meta programming.

There are important differences between the sequence of reduction optimizations performed in this section and those described in reference [10]. **First**, they do not employ unrolling of the kernel until the very last



Kernel	256	512	1024	2048	4096	8192	16384
Reduce r1	**64**	**64**	**128**	**128**	**256**	**512**	**512**
Reduce r2	**64**	**64**	**128**	**256**	**256**	**512**	**512**
Reduce r3	**64**	**64**	**128**	**128**	**256**	**512**	**512**
Reduce r4	32	64	128	256	512	**64**	**128**
Reduce r5	32	64	64	256	256	**256**	**256**
Reduce r6	32	32	64	128	256	512	512

Figure 6: Top: The best *time for each kernel variant at each input size. Bottom: the thread setting that achieved that best time. These settings are difficult to predict in advance. Kernels that use virtualized threads are highlighted, note that there are many of these amongst the best selection.*

step. The Obsidian approach, using Haskell recursion to implement the reduction kernels lead to unrolled code by default. **Second**, in the NVIDIA tutorial they apply an optimization that computes on elements before ever storing anything in shared memory. This is something that we also get for free in Obsidian and would actually need to add code to get the kind of reduction that stores the elements in shared memory before operating on them in the first stage. The code that needs to be added is a use of `forcePull` on the input array as step one in the reduction kernel.

5.3 Scan

Scan computes all the prefix sums of a sequence of values using a binary associative operator (and is familiar to Haskell programmers as the `scanl1` function).

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

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

Figure 8 shows a standard divide and conquer decomposition of scan. Data flows from top to bottom and boxes with two inputs are operators. At each level, exactly half of the boxes are operators and in an imperative language the algorithm would naturally be implemented in-place. Since we cannot easily express in-place algorithms currently in Obsidian, this means that we need to copy unchanged values into a new array during each phase. During a phase of compilation, Obsidian analyses memory usage and lays out

Kernel	Lines	Cumulative	Total
red1	5	5	7
red2	5	5	7
red3	5	5	7
red4	3	8	10
red5	3	8	10
red6	1	9	11
red7	1	9	11

Figure 7: The figure shows number of lines of code for the different reduction kernels. The Lines *column contains number of lines in the body of that particular reduction function, reuse of prior effort not included. The* Cumulative *column includes reuse of previously implemented kernels in the count. The* Total *column also includes extra lines for distributing the reductions over blocks (using* `pConcat`, `fmap` *and* `push`*). This distribution code is identical for all of the reduction kernels.*

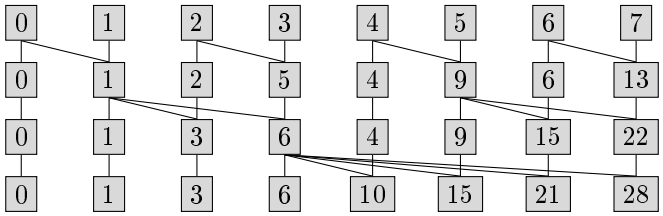


Figure 8: Sklansky parallel prefix network

intermediate arrays in memory. In the case of Sklansky scan kernels this leads to a ping-ponging behaviour between two arrays in shared memory.

Also, the threads now do two different things (copy, or perform operation). One can have as many threads as elements, but then each must have a conditional to decide whether to be a copy or operation thread. Or we can launch half as many threads and have each of them perform both a copy and an operation. We will show code for both of these options; the first is easier to implement.

The Obsidian code below implements the scan network from Figure 8, using as many threads as there are elements. Note that thread virtualization applies here, so arrays larger than the actual number of GPU threads can be processed. The limiting factor is the amount of shared memory.

```
sklansky :: (Choice a, MemoryOps 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 = binSplit (n-1) (fan op) arr
    arr2 ← forcePull arr1
    sklansky (n-1) op arr2
```

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

The `binSplit` 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 operation applied in this case is `fan`:

```
fan :: Choice a
     ⇒ (a → a → a)
     → SPull a
     → SPull a
fan op arr =  conc a1 (fmap (op c) a2)
    where
      (a1,a2) = halve arr
      c = a1 ! fromIntegral (len a1 - 1)
```

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

5.3.1 Two elements per thread

Both to avoid conditionals and to allow for larger scans per block, we move to two elements per thread. Each phase of the algorithm is a parallel for loop that is executed by half as many threads as there are elements to scan. The body of the loop performs one operation and one copy, using bit-twiddling to compute indices. Note the use of two *write functions* in sequence. Similar patterns were used in our implementations of sorting networks [7], 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
```

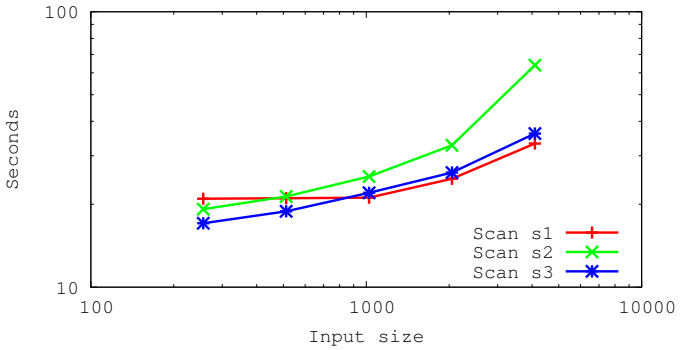
For an input of length 2^n, n phases are composed as follows:

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

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

```
compose :: MemoryOps 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 =
  do
    let arr1 = f arr
    arr2 ← force arr1
    compose fs arr2
```

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



Kernel	256	512	1024	2048	4096
Scan s1	256	256	512	1024	1024
Scan s2	256	256	512	512	512
Scan s3	128	128	512	512	512

Figure 9: As with Figure 6, the left shows only the best *times for any threadsetting at each size.*

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

Here we use `load 2` to realise loading of 2 elements per thread but in a strided way that is more likely to lead to a good memory access pattern. This function is an example of one of the custom ways to create a push array from a pull array mentioned in section 4.1. The results of these optimisations are shown in Figure 9. To make an even faster local and global scan, we would need to add more sequential work, as we did in the reductions.

6 Combining kernels to solve large problems

With Obsidian, we can experiment with details during the implementation of a single kernel. In section 5, we saw that the description of a local kernel involves its behavior when spread out over many blocks. However, solving large problems must sometimes make use of many different kernels or the same kernel used repeatedly. Here the procedure of making use of combinations of kernels is explained using large reduction as an example.

6.1 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.

Variant	Parameter	Seconds	Parameter*	Seconds*
ACC	Loop	2.767		
ACC	AWhile	2.48		
Red1	256 threads	0.751	32	2.113
Red2	256 threads	0.802	32	2.413
Red3	256 threads	0.799	32	2.410
Red4	512 threads	1.073	1024	2.083
Red5	256 threads	0.706	1024	1.881
Red7	128 threads	0.722	1024	1.968

Figure 10: Running times of 2^{24} element reduction using Obsidian or Accelerate. The results were obtained on a NVIDIA TESLA c2070. Each reduction procedure was executed 1000 times, and the total execution time is reported in the table. Two different methods for executing the Accelerate (ACC) reduction repeatedly was tested. There variants are refered to as "Loop" and "AWhile". A large number of experiments was performed on the our benchmark reductions (Red1 to Red7) and the best threads per block setting is listed in the table. The two colums on the right show the number of threads - kernel combination that perform the worst.*

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

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

The code above is one example of our API for writing CPU-side host-programs, though it is also possible to call Obsidian-generated kernels from CUDA code as well. Figure 10 shows the running time for the above program executing a 2^{24} element reduction compared against Accelerate.

7 Related work

There are many languages and libraries for GPU programming. Starting at the low-level end of the spectrum we have CUDA [18]. 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 [19]. Thrust offers a programming model where details of GPU architecture are completely abstracted away. Here, the programmer expresses algorithms using building blocks like: *Sort*, *Scan* and *Reduce*.

Data.Array.Accelerate is a language embedded in Haskell for GPU programming [6]. The abstraction level is comparable to that of Thrust. In other words, Accelerate hides most GPU details from the programmer. Accelerate provides a set of operations (that are parallel and suitable for GPU execution, much like in Thrust) implemented as skeletons. Recent work has permitted the optimisation of Accelerate programs using fusion techniques to decrease the number of kernel invocations needed (see reference [16]). It seems to us that when using Accelerate the programmer has no control over how to decompose his computation onto the GPU or how to make use of shared memory resources. For many users, remaining entirely within Haskell will be a big attraction of Accelerate.

Nikola [15] is another language embedded in Haskell that occupies the same place as Accelerate and Thrust on the abstraction level spectrum. The systems above are all for flat data-parallelism, Bergstrom and Reppy are attempting nested data-parallelism by implementing a compiler for the NESL language for GPUs [2].

The Copperhead [4] system compiles a subset of Python to run on GPUs. Much like other languages mentioned here, Copperhead identifies usages of certain parallel primitives that can be executed in parallel on the GPU (such as reduce, scan and map). But Copperhead also allows the expression of nested data-parallelism and is in that way different from both Accelerate and Obsidian.

In reference [20], Oancea et al. use manual transformations to study a set of compiler optimisations for generating efficient GPU code from high-level and functional programs based on `map`, `reduce` and `scan`. They tackle performance problems related to GPU programming, such as bad memory access patterns and diverging branches. Obsidian enables easy exploration of decisions related to these issues.

8 Conclusion

Obsidian lends itself well to the kind of experimentation with low level GPU details that allow for the implementation of efficient kernels. This is illustrated in section 5.2. The case study also show hows we can compose kernels and thus *reuse* prior effort.

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

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

References

[1] Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. The Design and Implementation of Feldspar an Embedded Language for Digital Signal Processing. IFL'10, Berlin, Heidelberg, 2011. Springer Verlag.

[2] Lars Bergstrom and John Reppy. Nested data-parallelism on the GPU. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP 2012)*, pages 247–258, September 2012.

[3] Guy Blelloch. Programming Parallel Algorithms. *Communications of the ACM*, 39(3), 1996.

[4] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: Compiling an Embedded Data Parallel Language. Technical Report UCB/EECS-2010-124, EECS Department, University of California, Berkeley, Sep 2010.

[5] Hassan Chafi, Arvind K Sujeeth, Kevin J Brown, HyoukJoong Lee, Anand R Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. In *ACM SIGPLAN Notices*, volume 46, pages 35–46. ACM, 2011.

[6] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell Array Codes with Multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, DAMP '11, New York, NY, USA, 2011. ACM.

[7] Koen Claessen, Mary Sheeran, and Bo Joel Svensson. 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. ACM, 2012.

[8] Conal Elliott. Functional images, 2000.

[9] Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(2), 2003.

[10] Mark Harris. Optimizing parallel reduction in CUDA. `"http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf"`.

[11] Stephan Herhut, Richard L Hudson, Tatiana Shpeisman, and Jaswanth Sreeram. Parallel programming for the web. In *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism, HotPar*, volume 12, pages 1–6, 2012.

[12] Eric Holk, William E Byrd, Nilesh Mahajan, Jeremiah Willcock, Arun Chauhan, and Andrew Lumsdaine. Declarative parallel programming for gpus. In *PARCO*, pages 297–304, 2011.

[13] Kenneth E Iverson. A programming language. In *Proceedings of the May 1-3, 1962, spring joint computer conference*, pages 345–351. ACM, 1962.

[14] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. 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, 2010. ACM.

[15] Geoffrey Mainland and Greg Morrisett. Nikola: Embedding Compiled GPU Functions in Haskell. In *Proceedings of the third ACM Haskell symposium*, pages 67–78. ACM, 2010.

[16] Trevor L. McDonell, Manuel M.T. Chakravarty, Gabrielle Keller, and Ben Lippmeier. Optimising Purely Functional GPU Programs, 2013. 18th ACM SIGPLAN International Conference on Functional Programming, ICFP 2013.

[17] Chris J. Newburn, Byoungro So, Zhenying Liu, Michael McCool, Anwar Ghuloum, Stefanus Du Toit, Zhi Gang Wang, Zhao Hui Du, Yongjian Chen, Gansha Wu, Peng Guo, Zhanglin Liu, and Dan Zhang. Intel's array building blocks: A retargetable, dynamic compiler and embedded language. CGO '11, Washington, DC, USA, 2011. IEEE Computer Society.

[18] NVIDIA. CUDA C Programming Guide.

[19] NVIDIA. NVIDIA Thrust Library.

[20] Cosmin E. Oancea, Christian Andreetta, Jost Berthold, Alain Frisch, and Fritz Henglein. Financial software on gpus: between haskell and fortran. In *Proceedings of the 1st ACM SIGPLAN workshop on Functional high-performance computing*, FHPC '12, pages 61–72, New York, NY, USA, 2012. ACM.

[21] Roger T. Stevens. Fractal Programming in C, 1989. M&T Books.

Appendix

```
-- Array creation
mkPull :: l → (EWord32 → a) → Pull l a
mkPush :: l
        → ((a → EWord32 → Program Thread ())
        → Push t l a

-- Map on pull and push arrays
fmap :: (a → b) → Pull l a → Pull l b
fmap :: (a → b) → Push t l a → Push t l b

-- Elementwise operations
zipWith :: (a → b → c)
         → Pull l a
         → Pull l b
         → Pull l c

-- Splitting
splitUp :: ASize l
         ⇒ Word32
         → Pull l a
         → Pull l (Pull Word32 a)
coalesce :: ASize l
          ⇒ Word32
          → Pull l a
          → Pull l (Pull Word32 a)

-- Array indexing
(!) :: Pull l a → EWord32 → a

-- Array convesion
push :: Pull l a → Push t l a

-- Make array manifest in memory
force :: Push t Word32 a
       → Program t (Pull Word32 a)
forcePull :: Pull Word32 a
           → Program t (Pull Word32 a)
```

Figure 11: Array programming API: Full type signatures for Obsidian functions.

```
__global__ void increment(uint32_t* input0,
                          uint32_t n0,
                          uint32_t* output1)
{
    uint32_t bid = blockIdx.x;
    uint32_t tid = threadIdx.x;

    for (int b = 0; b < n0 / 256U / gridDim.x; ++b) {
        bid = blockIdx.x * (n0 / 256U / gridDim.x) + b;
        output1[bid * 256U + tid] =
          input0[bid * 256U + tid] + 1U;
        bid = blockIdx.x;
        __syncthreads();
    }
    bid = gridDim.x * (n0 / 256U / gridDim.x) + blockIdx.x;
    if (blockIdx.x < n0 / 256U % gridDim.x) {
        output1[bid * 256U + tid] =
          input0[bid * 256U + tid] + 1U;
    }
    bid = blockIdx.x;
    __syncthreads();
}
```

Figure 12: CUDA code generated from the `increment` *program.*

```
__global__ void increment2(uint32_t* input0,
                           uint32_t n0,
                           uint32_t* output1)
{
    uint32_t bid = blockIdx.x;
    uint32_t tid = threadIdx.x;

    for (int b = 0; b < n0 / 256U / gridDim.x; ++b) {
        bid = blockIdx.x * (n0 / 256U / gridDim.x) + b;
        for (int i0 = 0; i0 < 32U; ++i0) {
            output1[bid * 256U +
                    (tid * 32U + i0)] =
              input0[bid * 256U + (tid * 32U + i0)] +
                1U;
        }
        bid = blockIdx.x;
        __syncthreads();
    }
    bid = gridDim.x * (n0 / 256U / gridDim.x) + blockIdx.x;
    if (blockIdx.x < n0 / 256U % gridDim.x) {
        for (int i0 = 0; i0 < 32U; ++i0) {
            output1[bid * 256U +
                    (tid * 32U + i0)] =
              input0[bid * 256U + (tid * 32U + i0)] +
                1U;
        }
    }
    bid = blockIdx.x;
    __syncthreads();
}
```

Figure 13: CUDA code generated from the `increment2` *program.*

function	input	output	description
pConcat	nested (Pull of Push)	flat Push array	Compute the inner Push arrays in parallel over a level of the hierarchy.
tConcat	nested (Pull of Push)	flat Push array	Spread computation of the inner sequential Push arrays over the threads at a given level in the hierarchy.
sConcat	nested (Pull of Push)	flat Push array	Remain at a level in the hierarchy, inner push arrays are computed in sequence (Potentially with internal parallelism at level Warp and above.
push	Pull	Push	Push to Pull array conversion.
force	Push	Pull	Makes array manifest in memory.
forcePull	Pull	Pull	Makes array manifest in memory.
fmap	$function(a \rightarrow b)$, Pull/Push	same array type as input	elementwise computation.
splitAt	n, Pull	pair (Pull, Pull)	Split a Pull array at a given index.
splitUp	n, Pull	nested (Pull of Pull)	Split an array into chunks of size n.
coalesce	n, pull	nested (Pull of Pull)	Split an array into chunks of size n. elements in the inner array are pulled from the input array using a stride.
evenOdds	Pull	pair (Pull,Pull)	Split input into its even and odd indices.
replicate	n, a	Pull	Create a Pull array with n elements all identical to a.
singleton	a	Pull	Create a one element Pull array.
singletonPush	Program t a	Push	Create a one element Push array.
last	Pull	a	Pull the last element from input array.
first	Pull	a	Pull the first element from input array.
conc	Pull, Pull	Pull	Concatenate two pull arrays. function inserts a conditional into the pull array.
concP	Push, Push	Push	Concatenate two Push arrays. Often more efficient than concatenating Pull arrays.
zip	Pull, Pull	Pull of pairs	Pair up elements from two Pull arrays.
unzip	Pull of pairs	Pair (Pull, Pull)	Split an array of pairs into two arrays.
upair	Pull of pairs	flat Pull array	Flatten an array of pairs.
binSplit	n, $function(Pull \rightarrow Pull)$, Pull	Pull	Splits input array in the middle recursively n times, then applies $function$ to each part and concatenates the result. Purpose is divide and conquer kernels.
seqReduce	op, Pull	Push	Reduction implemented as a sequential loop in a Push array.
seqIterate	n, $function$, $init$	Push	Sequentially iterate a function in a Push array.
seqUntil	$function$, $cond$, $init$	Push	Sequentially iterate until $cond$ is true.
seqScan	op, Pull	Push	Sequential scan.

Figure 14: Array programming API. Core functions (top), and a sampling of derived library functions in Obsidian (middle), and built-in sequential primitives (bottom).