Design Exploration through Code-generating DSLs

High-level DSLs for low-level programming

Bo Joel Svensson, Indiana University
Mary Sheeran, Chalmers University of Technology
Ryan Newton, Indiana University

DSLs (domain-specific languages) make programs shorter and easier to write. They can be stand-alone—for example, LaTeX, Makefiles, and SQL—or they can be embedded in a host language. You might think that DSLs embedded in high-level languages would be abstract or mathematically oriented, far from the nitty-gritty of low-level programming. This is not the case. This article demonstrates how high-level EDSLs (embedded DSLs) really can ease low-level programming. There is no contradiction.

A gentle introduction to EDSLs can be found in the previous article in this series: “Domain-specific Languages and Code Synthesis Using Haskell,” in which Andy Gill considers the pros and cons of implementing a deeply embedded DSL, compared with a stand-alone compiler. Here, the story continues, posing a slightly different question: If you need to produce high-performance, low-level code in a language such as C or CUDA, is it worthwhile to use (or create) a code-generating EDSL, or should you just buckle down and write the low-level code by hand? Our answer is that the EDSL may well be a reasonable option.

WHY GENERATE HIGH-PERFORMANCE PROGRAMS?

In 2010, one of this article’s authors was tasked with porting high-performance benchmarks to run on a new computer architecture. One of these benchmarks was a parallel sort designed to use vector instructions[6]—SSE instructions (streaming SIMD extensions)
to be precise. To accomplish vectorization, this sort bottoms out to a bitonic sorting network once the arrays reach a small size. Sorting networks such as these are directly vectorizable. Of course, benchmarks of this kind have no reputation for maintaining clean abstractions in the code. Indeed, this particular benchmark had hard-coded sorting networks with 16, 32, and 64 inputs, using many pages of repetitive and inscrutable code:

```
...
xmm10 = _mm_min_ps(xmm1, xmm6);
xmm11 = _mm_max_ps(xmm1, xmm6);
xmm12 = _mm_min_ps(xmm2, xmm5);
xmm13 = _mm_max_ps(xmm2, xmm5);
xmm14 = _mm_min_ps(xmm3, xmm4);
xmm15 = _mm_max_ps(xmm3, xmm4);
...
```

This representation of the program has obvious drawbacks. For one, how can it be ported to a new architecture with different vector instructions? It's impossible—you have to write a new version. Yet, a glance at the bitonic sorter Wikipedia entry (http://en.wikipedia.org/wiki/Bitonic_sorter) shows the algorithm is a simple and elegant recursive function. Why can't that essential structure be captured, removing the repetition from the example? Repetition can at least be reduced using C preprocessor macros:

```
#define SORT16(a,b,c,d,e,f,g,h,i,j,k,l,m,n,p,q,r,s,t,u,v,w,x,y,z,za) \
  i = _mm_min_ps(a,h); \
  j = _mm_max_ps(a,h); \
  ...
```

`SORT16` can then be reused to define `SORT32`. Still, C preprocessor macros do *not* offer the necessary compile-time programmability. Rather, the best solution is to write a simple *program generator* script that prints the repetitive code above to a file as output.

The lesson here is that software engineers should not hesitate to write programs that generate programs. If you can recognize regularities in your programs, then program generation is probably an option. There are plenty of sophisticated technologies to help with this *metaprogramming* (MetaOCaml, Scheme macros, Template Haskell, C++ template metaprogramming, etc.). Indeed, because the DSLs discussed here are embedded, their host programs are in fact program generators (i.e., metaprograms). These DSLs, however, also impose additional structure that yields several benefits.

DSLS OFFER SAFETY

The script alluded to before is very primitive; it emits each generated program directly as a string. An EDSL instead emits an abstract syntax tree for the DSL program, and with that you can do whatever you like (see Gill's article for more details). In fact, if structured properly, an EDSL used to generate code can offer safety guarantees in the generated code; a well-typed EDSL program should guarantee well-typed generated code. This is a great way to find errors early. For example, if the EDSL is embedded in Haskell, the Haskell type checker finds errors that would have led to broken low-level programs. Finding those errors by debugging the low-level generated code could be much slower—for example, in embedded systems where build and debug times are long.

You can go even further and restrict the DSL to particular idioms so as to gain extra safety properties. For example, you can guarantee only memory-safe generated programs, as in Galois Inc.'s C code generation in its Ivory DSL (http://smaccmpilot.org/languages/).

DSLS ENABLE SMARTER COMPILERS

These days, DSLs for high performance focus on parallelism, but the degree to which parallel implementation details are exposed to the user varies among DSLs. Some, such as SQL and very high-level array DSLs,[2,3,4,11,16,21,22] hide everything but abstract data transformations, which are naturally parallel. These compilers can achieve good performance by limiting the kinds of communication allowed between parallel code regions, enforcing structured data-processing patterns such as `map` and `reduce`, and removing features that make auto-parallelization difficult (e.g., aliasing, pointers, and arbitrary control flow). By leveraging these restrictions, you can often apply radical code transformations. An example of this would be fusion, the removal of intermediate data structures (e.g., replacing `map f (map g arr)` with `map (f ∘ g) arr`).

When this approach works well—that is, when the compiler can map the latent parallelism in the declarative specification onto the target architecture efficiently—it is an attractive option. Sometimes, though, experimentation is necessary to find a good parallel decomposition that matches the target, which might be a GPU (graphics processing unit) or FPGA (field-programmable gate array), for example. The user then wants not only fine control over the generated code, but also easy ways to change it. For example, Kansas Lava, the HDL described in the aforementioned article by Andy Gill, is expressly intended for design-space exploration, and case studies in forward error correctors show how this leads to high-performance designs. At a higher level of abstraction, the Bluespec (http://www.bluespec.com/high-level-synthesis-tools.html) behavioral language allows algorithm experts to generate high-performance FPGA accelerators. A major selling point is that it is quick and easy to do architectural exploration, particularly using highly parameterized DSL code.

These same benefits apply to low-level software domains, as well as hardware. In general, not all performance tweaking and tuning can be automated, which creates a need for DSLs that enable systematic user-guided performance exploration.

DSLS ABSTRACT CODE-GENERATION TACTICS

How should we give the programmer control over the generated code, along with easy ways to vary it? Because EDSL programs perform arbitrary computation at program-generation time, they can also encapsulate reusable code-generation tactics, in the guise of regular functions and data types. Let's take a look at an example: deferred array representations,[1,5,7,10] which represent the ability to generate elements , rather than elements already residing in memory (in this, they are similar to the concepts of generators or iterators found in many languages, but the specifics are quite different).

To explain deferred arrays, let's start with a simple program using the `map` and `reduce` functions from before:

```
let arr2 = map f arr1
    sum = reduce (+) arr2
    prd = reduce (*) arr2
...
```

Does `arr2` require memory for its storage (and, therefore, memory traffic to read and write it)? In most traditional languages the answer is an unqualified yes. In high-level array DSLs—which may or may not use deferred arrays—fusion optimizations *may* eliminate the intermediate array. This is possible because in many of these DSLs,[2,4,11,16] arrays are immutable and the DSLs themselves may even be side effect-free, making fusion opportunities easy to recognize in the compiler. Thus, very often a `map` operation *can* fuse into a `reduce` operation, which means that the (parallel or sequential) loop that is eventually created for `reduce` contains the `map`'d function inside that loop.

In the example, however, `arr2` is used twice. Most array DSLs will choose not to inline `arr2` if it means duplicating work, or they will use a heuristic based on a static cost estimate for `f` to decide whether work duplication is worth it to enable fusion. Explicitly deferred arrays, on the other hand, leave this decision up to the programmer. They represent an array in terms of the computation that generates it, and support a fusion-by-default approach. The `map` in the previous example would be duplicated and fused into each `reduce`, unless the user explicitly makes an array "real" in memory by applying a function called `force`. Moreover, deferred arrays gain an additional benefit in the embedded DSL context: they never suffer function-call overhead for the functions used to represent arrays, because they are inlined as the EDSL host program performs code generation.

Deferred-array data types are one example of encapsulating code-generation tactics with a nice API. They provide programmers with high-level array operations, while retaining control over memory usage. They can also help avoid bounds checks without risk, when composing known producer and consumer functions from the provided library (e.g., `map` and `reduce`). In a later section, we describe a language that provides even more control, with different control-flow patterns encapsulated in different variants of deferred arrays (push and pull). Finally, the article shows how deferred arrays can be used to make operations such as `map` highly generic (e.g., a `map` operation that, depending on the context in which it is applied, generates a sequential loop *or* a parallel loop, and, if parallel, can operate at one of multiple scales inside a hierarchical parallel architecture, generating very different code in each case).

DSLS CAN MIX IT UP (DESIGN EXPLORATION)

That last point—that high-level data operators can generate different code based on the context in which they are used—moves closer to the goal of design exploration. In what other ways can a DSL allow programmers to explore different implementation strategies, while changing many fewer lines of code than if they had to convert the generated code manually? One way is to take advantage of program generation to build highly parameterized designs. In a hardware-synthesis language, such as Bluespec or Lava, one wants designs that are flexible and do not commit to a specific circuit area. Likewise, later in this article, you will see GPU programs that are parameterized over the number of GPU hardware threads to use, sometimes changing the generated program structure (injecting sequential loops) if not enough threads are provided. This just-in-time determination of program structure would be difficult to accomplish in handwritten code. With it, we might accept a *list* of GPU programs in the host program and provision the GPU to run all those computations continuously and simultaneously, with each program adapting to the number of threads it is allotted.

Finally, with increased design-exploration power, a common desire is to auto-tune by generating many variants of a program, testing them all, and selecting the winner. Several DSLs internally perform such an auto-tuning process,[8,14,15] though that is beyond the scope of this article. Instead, this article aims to help you weigh the pros and cons of embedded DSLs that generate low-level code. It does so by presenting one concrete example: Obsidian, an embedded DSL that generates CUDA for GPU programming. First, let's review the CUDA programming model, before describing Obsidian in the remainder of this article.

A CUDA PRIMER

CUDA is a C dialect for GPU programming that NVIDIA provides. (Readers with previous experience with CUDA can skip this section.)

GPUs began as hardware accelerators for generating computer graphics, but have become increasingly general purpose, driven both by graphics programmers who want more flexibility and by a new breed of programmer, motivated not by graphics but by a desire to accelerate the data-parallel parts of non-graphics applications.

A typical computer system containing a CUDA-capable GPU is split into two parts: the *host*, which refers to the CPU and main memory of the computer; and the *device,* which consists of the GPU and its associated memory. The device usually comes in the form of a PCI Express card holding the GPU and memory. The GPU consists of a number of MPs (multiprocessors), each containing some number of functional units (*CUDA cores* in NVIDIA terminology). Each MP also contains local shared memory, which can be regarded as a programmer-managed cache. The CUDA cores within an MP can communicate via this shared memory.

MASSIVE PARALLELISM

A GPU is capable of managing a large number of threads in flight. It thrives when thousands of threads (doing mostly identical work) are launched together. It has a hierarchical structure managing all these threads. Onto each MP are launched several *blocks* of threads, each consisting of up to 1,024 threads. Threads within a block (and only within the block) can communicate using the shared memory of the MP. The collection of blocks launched onto the GPU is called the *grid*. A group of 32 consecutive threads within a block is called a *warp*. Warps execute in lock-step; all threads perform the same instruction at any given moment. Threads within a warp that do not partake in the computation are turned off.

SCALABLE ARCHITECTURE

GPUs may contain as little as one or as many as 15 MPs. A CUDA program should be able to run on any of the GPUs along the scale. The CUDA programming model mirrors the GPU hierarchy. The programmer writes an SPMD (single-program multiple-data) program that is executed by the threads within a block. Many instances of this program also execute across all blocks launched. All instances are independent and can be launched in any order, or in parallel, across available MPs.

The upshot of all this is that programming a GPU really doesn't much resemble programming a multicore machine with two, four, or eight cores. Programmers need to launch thousands of threads and figure out how they should communicate within the constraints of the GPU. This means that they need to think about warps, blocks, and grids, like it or not. (As will be seen, however, a DSL can still make it *easier* to map abstract data transforms onto the hierarchy.)

The references at the end of this article include links to further reading on CUDA programming and GPU architecture.[12,13]

THE OBSIDIAN LANGUAGE

In CUDA, parallel `for` loops are implicit; the computation is described at the element level. Which elements to access (where to load and store data) is expressed as a function of a thread's identity—that is, threads must ask "where am I?" and answer the question by computing with `blockIdx`, `blockDim`, and `threadIdx`. Obsidian programs, in contrast, describe array-to-array computations at an aggregate level, replacing indexing arithmetic with collective operations. For example:

```
vecAdd v1 v2 = zipWith (+) v1 v2
```

The vecAdd function takes two arrays, v1 and v2, as input and performs elementwise addition using the zipWith library function (zipWith is just a map over two arrays; see figure 1 for a selection of library functions). vecAdd is not a complete Obsidian program, however; information about how to map this program onto the GPU hierarchy is needed and will be provided in the program's types.

To give precise types to operations such as vecAdd requires an understanding of deferred arrays described in the first section. Specifically, there are *push* and *pull* array variants. The vecAdd program operates on pull arrays. Its type is:

```
vecAdd :: Pull EFloat -> Pull EFloat -> Pull EFloat
```

That is, vecAdd takes two pull arrays of EFloat values as input and returns a pull array. (Values with types prefixed by E are actually expression trees, as explained in the next section.) Pull arrays are implemented as a length plus a function from index to value:

```
type Pull a = (Size, (EWord32 -> a))
```

This representation of arrays provides fusion of operations for free. For example, with the above definition of a pull array, the following equations show how a function mapped over a pull array is actually absorbed (composed) into the function-based representation it already uses:

```
map f arr == map f (sz,g) == (sz, f ∘ g)
```

No intermediate data is written to memory for the output of g, but in some cases it is desirable to write the elements to memory—for example, revisiting the example from the first section, but now

FIGURE 1 **A Selection of Operations on Pull and Push Arrays**

Operation	Input	Output	Description
map function	pull array	pull array	apply function to each element of input array
map function	push array	push array	apply function to each element of input array
zipWith function	two pull arrays	pull array	pairwise version of map
reverse	pull array	pull array	reverses a pull array
splitUp n	pull array	nested pull arrays	splits a pull array into chunks of size n
halve	pull array	pair of pull arrays	splits a pull array in the middle
push	pull array	push array	converts from pull to push representation
force	push array	pull array	makes array manifest in memory
forcePull	pull array	pull array	makes array manifest in memory

assuming that `f2` is expensive in the following code:

```
let arr2 = map f2 arr1
    sum = reduce (+) arr2
    prd = reduce (*) arr2
...
```

Here `arr2` is used twice, and the programmer should have the power to ensure that `arr2` is computed and fully stored in memory. This is where `force` functions come in:

```
do arr2 <- forcePull (map f2 arr1)
   let sum = reduce (+) arr2
       prd = reduce (*) arr2
   ...
```

The reason for switching to Haskell's `do` notation is that `forcePull` is a *monadic* function. Monads are Haskell's way of encoding side effects, and in this program the monad in question is Obsidian's `Program` monad, which roughly corresponds to CUDA code. The return type of `forcePull` is `Program t (Pull a)`, where `t` is a parameter designating a level in the GPU hierarchy (`Thread, Warp, Block,` or `Grid`).

With the Program monad, we can define Push arrays as well.

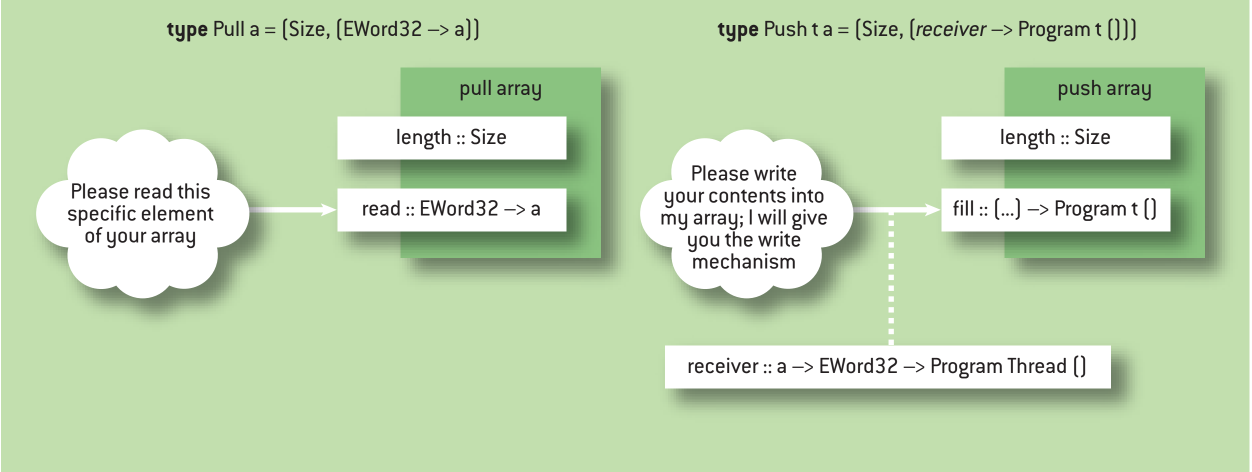
```
type Push t a = (Size, ((a -> EWord32 -> Program Thread ()) -> Program t ()))
```

Like pull arrays, push arrays consist of a length and a function. The difference is that the function in the push array (the filler function) results in a program that encodes an iteration schema at level `t` in the hierarchy. The argument to the filler function is itself a function (the receiver function). A pull array fulfills requests for specific elements, whereas a push array only allows a bulk request to push *all* elements via a receiver function. This relationship is pictured in figure 2. When invoked, the filler function creates the loop structure, but it inlines the code for the receiver inside that loop. A push array with its elements computed by `f` and receiver `rcv` might generate: `for(i∈[1,N]) { rcv(i,f(i)); }`

More specifically, when forcing a push array to memory, each invocation of `rcv` would write one memory location, `A[i] = f(i)`.

The reason that Obsidian has both pull and push arrays is related to performance. Some operations are very easy to express on pull arrays and result in good code, such as `map, zipWith,` and permutations. An operation such as `append`, however, which takes two pull arrays and concatenates them, leads to a conditional within the lookup function. When forcing such a concatenated array, this conditional will be executed in every iteration of the generated `for` loop (or by each thread). Push arrays, on the other hand, encode their own iteration schema, so rather than executing a conditional, when concatenating push arrays their inherent loop schemas are executed in sequence, neither loop executing a conditional in each iteration.

FIGURE 2 **The Interface Between Push/Pull Arrays and Their Clients**


type Pull a = (Size, (EWord32 –> a))
type Push t a = (Size, (receiver –> Program t ()))
pull array
length :: Size
read :: EWord32 –> a
Please read this specific element of your array
push array
length :: Size
fill :: (...) –> Program t ()
Please write your contents into my array; I will give you the write mechanism
receiver :: a –> EWord32 –> Program Thread ()

PUSH AND PULL CONVERSION

The result of `force` (for push arrays) and `forcePull` (for pull arrays) is always a pull array. That means a push array can be converted to a pull array by application of `force`. Applying `force` always has a cost in memory, however, and it realizes the full cost of computing all elements.

Going in the other direction—converting from pull array to push array—requires a function called `push`. This function does not have any cost directly associated with it. It merely changes the representation (though switching to the push representation means losing the ability to compute only *part* of the array). Note, however, that pull arrays do not have the `t` parameter like push arrays—they are hierarchy-level-agnostic. Using `push` on a pull array and fixing the `t` parameter of the result locks the array into a given level of the hierarchy, as in the following:

```
vecAdd :: Pull EFloat -> Pull EFloat -> Push Grid EFloat
vecAdd v1 v2 = push (zipWith (+) v1 v2)
```

Now the `vecAdd` program is complete: `push` converts the result array to a push array, and the result type is fixed as `Push Grid`, so the iteration becomes `Grid`-level parallel.

DESIGN-SPACE EXPLORATION

Even a simple operation such as reduction requires design-space exploration for good performance on the GPU. Sequential reductions per thread can be combined with parallel binary-tree-shaped reduction over multiple threads in a block. Both the degree of sequential work and the number of threads used for the parallel work can be varied. This section describes a local (on-chip, shared-memory) `reduce` algorithm. It performs one reduction per GPU block. This can serve as a building block in a full-scale reduction algorithm over all the GPU threads.

The following code implements a recursive parallel reduction. In each step, the input array is split in the middle, and the elements from the two resulting arrays are added together pairwise, resulting in an intermediate array half as long. The intermediate array is forced to memory using `forcePull`, writing to shared memory using one thread per element (in this case, at the `Block` level).

```
reduceLocal :: Scalars a => (a -> a -> a) -> Pull a -> Push Block a
reduceLocal f arr = singletonPush (loop arr)
  where loop arr | len arr == 1 = return (arr ! 0)
                 | otherwise =
                     do let (a1,a2) = halve arr
                        arr' <- forcePull (zipWith f a1 a2)
                        loop arr'
```

Next, many instances of the block-level reduction algorithm are combined to form a grid computation. The following code distributes the computation over multiple blocks using `map` and `pConcat`. The `pConcat` function is for *nested parallelism*; it concatenates the results computed in each block and is in charge of the parallel distribution of work across blocks. (A related function is `sConcat`, which performs sequential work within one block or one thread.)

```
reduceGrid :: Scalars a => (a -> a -> a) -> Pull a -> Push Grid a
reduceGrid f arr = pConcat (map (reduceLocal f) chunks)
  where chunks = splitUp 4096 arr
```

The `forcePull` function prevents fusion of operations, so leaving it out in some stages of the reduction is a way of trading off sequential and parallel execution. An `optForce` *n* function could be written that forces only arrays shorter than *n* elements:

```
optForce n arr = if len arr <= n
                 then forcePull arr
                 else return arr
```

The `optForce` function could replace `forcePull` in `reduceLocal`. The `n` parameter in `optForce` is one possible tuning parameter to add to `reduceLocal`; another is the `splitUp` factor (`4096` above). By parameterizing on the force cutoff and the `splitUp` factor, a family of different reduction codes can be generated from the same EDSL description.

Obsidian provides yet another tuning parameter by default: the number of real GPU threads to allow per block. This parameter is given by the Obsidian user to the code generator and can range over the number of threads per block supported by CUDA (1–1,024). Given the execution model of GPUs, however, only multiples of the warp size (32) make sense.

Allowing the parameters to vary over the following ranges:

- `splitUp` factor: $s \in \{32,64,128,\ldots,8192\}$
- `optForce` cut-off: $c \in \{32,64,\ldots,s\}$
- Real Threads: $t \in \{32,64,96,\ldots,min(1024,s/2)\}$

would create 929 variants of the reduction code for auto-tuning. For benchmark results for reduction kernels, refer to the authors' recent technical report.[20]

GENERATING CUDA CODE FROM THE EDSL

Up to this point we have seen the programmer's view of Obsidian, with dips into implementation details here and there. But how does it all work? The first article in this series (Gill) shows much of the needed infrastructure: overloading, reification, and expression data types. Obsidian is no different; the `EWord32` and `EFloat` types seen throughout this article correspond to expression trees. The `Program` monad is reified into abstract syntax representing a program in an imperative, CUDA-like language.

One new concept presented here, compared with Gill's article, is the use of pull and push arrays. These deferred arrays rely on extracting program fragments as abstract syntax (i.e., deep embedding), but the functions that represent push and pull arrays are not themselves captured in abstract syntax. Rather, they are like macros that desugar during Haskell execution, *before* the Obsidian CUDA-emitting compiler is invoked (shallow embedding).[17]

OUTLINE OF CUDA CODE GENERATION

• **Function reification.** EDSL functions (such as `vecAdd`) are applied to a symbolic array, causing it to be evaluated as a Haskell program and yielding its result. The functions Obsidian can reify have push arrays as results. After the application to a symbolic array, therefore, you have a push array. The push array is a function that generates a `Program`, and applying that push array's filler function to yet another symbolic input extracts a final, complete `Program`.

• **Monadic reification.** Because the `Program` type is a monad, a monad reification technique is applied (see Gill), and the result is an AST (abstract syntax tree) describing a program in a CUDA-like imperative language. This AST, however, has explicit parallel `for` loops rather than CUDA's implicit ones.

• **Shared memory analysis.** All intermediate arrays (resulting from `force`) have unique names in the CUDA-like AST, but they still must be laid out in shared memory. Liveness analysis finds the first and last use of each array, and a memory map is generated. Named arrays in the AST are replaced by exact locations in shared memory.

• **CUDA generation and thread virtualization.** This is the final step in code generation. Most aspects of the CUDA-like AST translate directly into CUDA source. The explicit parallel loops, which may have larger iteration spaces than the number of actual CUDA threads, need some further conversion. If the AST contains a loop, "`parFor(i∈[1,512]){body})`", but the code generator was instructed to use only 256 threads, then this `parFor` will happen in two stages, implemented by injecting an additional sequential loop.

CONCLUSION

This article has highlighted a number of potential benefits of DSL technology. If you are going to give embedded DSLs a try, however, there are some downsides to watch out for as well. First, error messages can be problematic, as these are expressed in the terminology of the host language. If Obsidian programmers try to use some feature at the incorrect level in the GPU hierarchy (e.g., a too-deep nesting of parallel for loops by applying too many `pConcats`), then the error message will likely state that there are missing instances of some Haskell class, rather than explaining why what they tried cannot be done on a GPU. Furthermore, new DSLs naturally lack the library and tooling ecosystems of large established languages.

Using an EDSL such as Obsidian, which exposes underlying architectural structure, allows control over the details that determine performance. Choosing exactly what to abstract and what to expose, however, remains important. For example, in Obsidian programmers can trade off sequential and parallel work and use shared memory (via `force`), but the details of shared-memory layout of arrays, or managing their lifetimes in that memory, are automated. Further, just as important is what the DSL *prevents* the user from doing. In Obsidian, programmers are gently, but firmly, prevented from writing DSL code that would produce CUDA that doesn't match what the GPU can cope with, such as the too-deep nestings just mentioned.

Writing CUDA code and hand-tuning for performance is often a tedious task. Design decisions made early may be hard to revert and require much of the code already written to be replaced. It is thus appealing to generate variants of CUDA code from a highly parameterized description in a DSL, and then select the best performing, empirically. Our experiments in previous work[20] show that the best variant of particular code may differ from one GPU to another.

How much work does it take to produce a code-generating DSL such as Obsidian? In Obsidian's case there have been years of experiments and redesigns—leading to a Ph.D.![18] But don't let that put you off. The current version is relatively small (2,857 lines in the language implementation and 1,629 lines in the compiler part), and it is freely available to study and copy.[19] Further, modern libraries for parsing, code generation from templates (for example, http://hackage.haskell.org/package/language-c-quote), and compiler construction[9] make constructing DSLs easier than expected. So, next time you find yourself with repetitive low-level code, why not consider an embedded code-generating DSL instead?

ACKNOWLEDGMENTS

Push arrays were invented by Koen Claessen. We thank Andy Gill for his valuable feedback on this article. Work on Feldspar and Obsidian was supported by the Swedish Foundation for Strategic Research in the RAW FP (Resource Aware Functional Programming) project and by the National Science Foundation award 1337242.

REFERENCES

1. Axelsson, E., Claessen, K., Sheeran, M., Svenningsson, J., Engdal, D., Persson, A. 2011. The design and implementation of Feldspar, an embedded language for digital signal processing. In *Implementation and Application of Functional Languages*. Springer Verlag.
2. Catanzaro, B., Garland, M., Keutzer, K. 2011. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming:* 47-56.
3. Chafi, H., Sujeeth, A. K., Brown, K. J., Lee, H. J., Atreya, A. R., Olukotun, K. 2011. A domain-specific approach to heterogeneous parallelism. In *ACM SIGPLAN Notices* 46: 35-46.
4. Chakravarty, M. M. T., Keller, G., Lee, S., McDonell, T. L., Grover, V. 2011. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the 6th Workshop on Declarative Aspects of Multicore Programming.*
5. Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R. R., Bradshaw, R., Weizenbaum, N. 2010. FlumeJava: easy, efficient data-parallel pipelines. In *ACM SIGPLAN Notices* 45: 363-375.
6. Chhugani, J., Nguyen, A. D., Lee, V. W., Macy, W., Hagog, M., Chen, Y.-K., Baransi, A., Kumar,

S., Dubey, P. 2008. Efficient implementation of sorting on multicore SIMD CPU architecture. In *Proceedings of the VLDB Endowment* 1(2):1313-1324.

7. Claessen, K., Sheeran, M., Svensson, B. J. 2012. Expressive array constructs in an embedded GPU kernel programming language. In *Proceedings of the 7th Workshop on Declarative Aspects and Applications of Multicore Programming*.
8. Filipovič, J., Madzin, M., Fousek, J., Matyska, L. 2013. Optimizing CUDA code by kernel fusion—application on BLAS. arXiv preprint arXiv:1305.1183.
9. Keep, A. W., Dybvig, R. K. 2013. A nanopass framework for commercial compiler development. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*: 343-350.
10. Keller, G., Chakravarty, M. M. T., Leshchinskiy, R., Peyton Jones, S., Lippmeier, B. 2010. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*.
11. Newburn, C. J., So, B., Liu, Z., McCool, M., Ghuloum, A., Du Toit, S., Wang, Z. G., Du, Z. H., Chen, Y., Wu, G., Guo, P., Liu, Z., Zhang, D. 2011. Intel's array building blocks: a retargetable, dynamic compiler and embedded language. International Symposium on Code Generation and Optimization.
12. NVIDIA. 2013. CUDA C Programming Guide; http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.
13. NVIDIA. 2012. Nvidia's Next Generation CUDA Compute Architecture: Kepler GK110; http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.
14. Püschel, M., Moura, J. M. F., Johnson, J. R., Padua, D., Veloso, M. M., Singer, B. W., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., et al. 2005. Spiral: code generation for DSP transforms. *Proceedings of the IEEE* 93(2): 232-275.
15. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image-processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation:* 519-530.
16. Scholz, S.-B. 2003. Single assignment C: efficient support for high-level array operations in a functional setting. *Journal of Functional Programming* 13(6): 1005-1059.
17. Svenningsson, J., Axelsson, E. 2013. Combining deep and shallow embedding for EDSL. In *Trends in Functional Programming*, eds. H.-W. Loidl and R. Pea. *Lecture Notes in Computer Science* 7829: 21-36. Berlin/Heidelberg: Springer.
18. Svensson, B. J. 2013. Embedded languages for data-parallel programming. Ph.D. thesis. Department of Computer Science and Engineering, Chalmers University of Technology.
19. Svensson, B. J. 2014. Obsidian GitHub Repository; https://github.com/svenssonjoel/Obsidian.
20. Svensson, B. J., Sheeran, M., Newton, R. R. 2014. A language for nested data parallel design-space exploration on GPUs. Technical Report 712. Indiana University; http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR712.
21. Tang, Y., Chowdhury, R. A., Kuszmaul, B. C., Luk, C.-K., Leiserson, C. E. 2011. The pochoir stencil compiler. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures:* 117-128.
22. Thies, W., Karczmarek, M., Amarasinghe, S. 2002. StreamIt: a language for streaming applications. In *Compiler Construction:* 179-196. Springer.

LOVE IT, HATE IT? LET US KNOW
feedback@queue.acm.org

BO JOEL SVENSSON earned his Ph.D. in computer science from Chalmers University of Technology in Gothenburg, Sweden, in 2013. His advisors were Mary Sheeran, Koen Claessen, and Josef Svenningsson. He is currently a postdoc at Indiana University working with Ryan R. Newton on a project related to EDSLs and parallelism.

MARY SHEERAN is a professor in the software technology division and member of the functional programming group at Chalmers University of Technology, Gothenburg, Sweden. She has been working with domain-specific languages since the early 1980s, when she worked on a functional hardware description language. She remains interested in how functional programming can assist in low-level programming or hardware design. She has a background in electrical engineering and a D.Phil. in computation from Oxford University.

RYAN R. NEWTON received his Ph.D. in computer science from MIT in 2009. He then conducted research on parallel programming tools as part of Intel's Developer Products Division. In 2011, he joined Indiana University, where his research focuses on language-based approaches to the programming challenges posed by future architectures. To this end, he and his students invented a new concurrent data abstraction (LVars) for deterministic parallelism and are working on domain-specific compilers for array languages and distributed stream processing.