Simple Optimizations for an Applicative Array Language for Graphics Processors

Bradford Larsen

Department of Computer Science
Tufts University
blarsen@cs.tufts.edu

Abstract

Graphics processors (GPUs) are highly parallel devices that promise high performance, and they are now flexible enough to be used for general-purpose computing. A programming language based on implicitly data-parallel collective array operations can permit high-level, effective programming of GPUs. I describe three optimizations for such a language: automatic use of GPU shared memory cache, array fusion, and hoisting of nested parallel constructs. These optimizations are simple to implement because of the design of the language to which they are applied but can result in large run-time speedups.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—*Code generation, Compilers, Optimization*; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming

General Terms Languages, Design

1. Introduction

Graphics processing units (GPUs) are highly parallel accelerator devices, offering high-performance computing capabilities superior to those of traditional processors. Due to increased hardware flexibility and improved programming tools, GPUs have frequently been used for non-graphics computation, e.g., for SAT solving [Manolios and Zhang 2006], state space search [Edelkamp et al. 2010], and physics simulations [Elsen et al. 2006]. Early work in general-purpose GPU programming repurposed graphics programming models such as OpenGL or Direct3D (e.g., Manolios and Zhang [2006]); more recent efforts have involved NVIDIA's CUDA dialect of C++ (e.g., Edelkamp et al. [2010]). Although CUDA is a significant improvement over previous methods, it is too low-level for easy use: for example, to find the largest element of a vector *efficiently* requires over 150 lines of CUDA; in a typical sequential programming language, this problem can be solved with at most a few lines of code.

What an ideal general-purpose GPU programming model might be is still an open question. One promising approach is that of *array programming*, where one describes how to transform an array as a whole using collective operations. Array programming is not a new idea, going back at least to APL [Iverson 1962]. It has particular advantages in the context of high-performance parallel processing, because many collective array operations are implicitly data-parallel and have well-known parallel implementations [Blelloch 1989, 1996; Sengupta et al. 2007].

Recent efforts have explored the possibility of targeting GPUs through array programming [Catanzaro et al. 2010; Lee et al. 2009; Mainland and Morrisett 2010]. This paper shows that an applicative array programming model supported by simple optimizations can enable users to write GPU kernels whose performance is competitive with hand-written CUDA code but whose source code is a fraction of the size of CUDA. In particular:

- I describe how to automatically make use of GPU shared memory cache in stencil operations to reduce memory traffic (§5), and show experimentally that it improves performance by up to a factor of eight (§8).
- I describe a simple array fusion scheme to further reduce memory traffic and eliminate the need for temporary arrays (§6), and show experimentally that it improves performance by up to a factor of three (§8).
- I describe how some nested data-parallel programs can be easily transformed into equivalent non-nested programs through hoisting, similar to loop-invariant code motion performed by compilers in traditional imperative languages (§7).

These optimizations are implemented in the compiler for the research language Barracuda, a compositional, applicative array programming language for GPUs, embedded within Haskell (§3). Although more general optimization techniques are known (e.g., the flattening transform for nested data-parallel programs [Blelloch 1996]), the optimizations presented in this paper are simple to implement—requiring almost no analysis—due to the language design of Barracuda.

2. Background: Efficient GPU Code

The GPU is structured at a very high level as shown in Figure 1. Each GPU has several distinct memory areas, and data can only be transferred among these areas in fixed, well-defined ways. The latest GPUs possess hundreds of cores organized into dozens of *multiprocessors*, each with its own private memory, known as *shared memory*, which can be accessed as quickly as a register. Access to *device memory* located on the GPU itself is shared by all the GPU multiprocessors, and can be done within hundreds of GPU cycles. The CPU of the system controls data movement between main memory and device memory, which takes thousands of GPU cycles.

The core of a parallel algorithm for the GPU must be expressed as a set of single-instruction, multiple-data computations called

DAMP'11, January 23, 2011, Austin, Texas, USA.

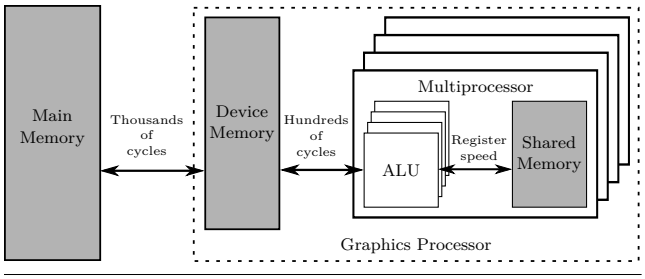

Figure 1. Block diagram of the GPU.

kernels. A single kernel is run in parallel on a GPU, on multiple data, by thousands of lightweight *kernel threads*. In the simplest case, each thread executing the kernel computes a function for a single element of the input arrays. These kernel threads are organized into gangs of *thread blocks*, which are in turn organized into a *grid*, which are scheduled onto the GPU's multiprocessors; in other words, work must be assigned to GPU threads through a two-level decomposition of the problem. For example, an element-wise transformation on a vector of 2^{20} elements might assign one element to each of 2^{20} threads, organized and scheduled as 4096 groups of 256-thread blocks. When a kernel is called it must be supplied with parameters that specify the data-parallel execution configuration in addition to its data arguments.

2.1 CUDA

The high-level programming model described previously is compatible with a wide variety of GPU architectures. To provide access to this parallel hardware, and to hide differences between architectures (which change rapidly), NVIDIA provides CUDA, a dialect of C++ that is aware of the GPU.

CUDA offers a means of expressing a kernel: it is a C++ procedure that uses no recursion and no indirect calls and is executed purely for its side effect on device memory. Kernel code may not refer to main memory; it can only access the GPU's device memory and shared memory.

CUDA offers a small set of primitives for managing memory on the GPU and for transferring data between main memory and device memory: most importantly, `cudaMalloc` and `cudaFree` allocate and deallocate memory on the GPU, and `cudaMemcpy` transfers data between main memory and device memory. Data to be stored in shared memory is marked with the `__device__` qualifier, and data transfer between shared memory and device memory is done through regular assignment statements within kernel code. Explicit use of synchronization primitives by threads within a block is necessary within kernel code when shared memory is used.

2.2 Keys to good parallel performance on the GPU

Several factors determine whether GPU code will be efficient. In roughly descending order of impact, these include:

1. minimization of data movement between the GPU and CPU;
2. choice of appropriate block and grid dimensions;
3. proper memory alignment and array indexing patterns; and
4. explicit use of shared memory when appropriate.

Each can significantly impact performance—i.e., by factors of two or more—and it can be difficult to get them all correct.

Data movement between GPU and CPU GPU code cannot access the system's main memory, and so the data required for any GPU operation must be copied into device memory. After GPU code has finished and further processing with the result is desired (e.g., printing it or sending it over the network), it is necessary to copy the result back into main memory, which takes thousands of cycles. An application that makes efficient use of the GPU will involve minimal copying between these memory spaces.

Block and grid dimensions A problem must be decomposed for a grid of thread blocks to run on a GPU. Only certain thread block dimensions (roughly, powers of two) are efficiently executed by the GPU; furthermore, it is important that there be enough thread blocks so that the GPU hardware can be fully utilized. A poor choice of block or grid dimensions can greatly hinder performance.

Memory alignment and array indexing patterns When arrays are properly aligned within device memory and all threads within a thread block use one of a few prescribed memory access patterns, access to device memory can be batched or *coalesced*, reducing the number of memory transactions required, which can have a dramatic impact on overall performance. However, the necessary conditions for memory coalescing to occur are subtle and the details are hard to get right.

Shared memory Each GPU multiprocessor contains fast on-chip shared memory that is accessible by all threads within a thread block executing on that multiprocessor. Since shared memory can be accessed so much faster than device memory, shared memory should be used in kernel code where multiple threads within a block would access the same memory locations. Shared memory is analogous performance-wise to L1 cache in a CPU, but differs sharply in the fact that shared memory caching must be done explicitly by the programmer.[1]

3. Barracuda: An Applicative Array Language

Barracuda is a simple array language for GPU programming, and is the vehicle used to test the impact of the optimizations presented in this paper. Barracuda supports array programming with rank-1 arrays (i.e., vectors) and rank-2 arrays (i.e., matrices). Furthermore, Barracuda is statically typed: arrays in Barracuda are parametrized by element type, which is either that of floating-point numbers, integers, or Boolean values; these element types can be easily mapped to types supported by the GPU.

Barracuda provides three primitive collective operations on its arrays:

Map Applies a scalar function element-wise to some number of arrays of the same rank.

Reduce Accumulates a scalar value from an array using a binary reducing function, e.g, addition.

Slice Creates a sub-array of the same rank from another array, given start, stop, and stride information.

The set of supported operations and types in Barracuda is small, reducing the amount of implementation effort required while still being sufficient for meaningful proof-of-concept work.

Barracuda is an applicative, total language: it has no notion of side effects or assignment, and it is not possible to write non-terminating or undefined programs. This is not as restrictive as it may first seem, as many operations on arrays can be expressed functionally. Because Barracuda programs lack side effects and must terminate, it is easier to reason about them, and code generation is simplified, as no sophisticated analysis is required. Furthermore, Barracuda's array primitives are compositional: the array primitives can be freely nested rather than having to be explicitly sequenced, which allows a more declarative programming style.

[1] NVIDIA's Fermi GPUs support implicit caching; however, NVIDIA still recommends explicit use of shared memory for best performance.

Barracuda is designed for offline generation of GPU code. Its primary use case is this: a programmer writing an application in C++ wants to run certain array operations on a GPU, for sake of performance. Rather than writing the GPU code by hand, the programmer writes the operations at a high level in Barracuda, which is then compiled into efficient CUDA code and C++ wrapper code. In other words, Barracuda functions are the unit of compilation. Barracuda functions are compiled into C++ procedures that wrap CUDA code, and the user of the generated code need not be aware of any of the low-level details of GPU programming. In particular, Barracuda hides completely all the performance issues of GPU programming discussed in section 2.2 except for the minimization of data movement between the CPU and GPU, which it puts entirely into the hands of the programmer.

Barracuda is the vehicle by which the optimizations described in this paper are explored; all the optimizations ought to be applicable in other languages sharing Barracuda's essential attributes. These attributes are the following:

Support for array programming The language emphasizes collective array operations, such as *map*, *reduce*, and *slice*.

Compositionality The collective operations can be freely nested.

Referential transparency The array operations are applicative, lacking side effects.

Compiled to a GPU The language targets a GPU, which possesses fast on-chip shared memory cache that can be exploited in operations with highly local memory access patterns.

The remainder of this section gives an overview of Barracuda. Details of the embedding are included in Appendix A.

3.1 Examples

Two example Barracuda programs appear in Figure 2. The first is an implementation of root mean square error. It is written similarly to the way one could write a version operating on regular Haskell lists, modulo different types and different names for certain functions, e.g., `VExp Float` instead of `[Float]`, and `vmap` instead of `map`. (These alternative function names were chosen to avoid shadowing the functions in the Haskell Prelude.) Other functions, such as `sqrt` and the arithmetic operators, are overloaded and use the normal Haskell names. This example demonstrates composition of vector primitives: `vzipWith`—a map operator that applies a binary function element-wise to two vector—appears inside the vector argument of `vmap`, which itself appears inside the vector argument of `vsum`, a vector reduction.

The second example, an implementation of weighted moving average over a vector of floats, is more complicated. It shows the use of Haskell as a macro system for Barracuda: the weight parameters, given as a Haskell list rather than a Barracuda vector, are incorporated at compile-time. This example also crucially uses the vector slice primitive of Barracuda: to compute an n-point weighted moving average, the `wmAvg` function slices the input vector into n overlapping slices, multiplies each slice by its corresponding weight, and adds up all the results element-wise. This function is an example of a *stencil operation*, where the function applied at each element depends on the values of neighboring elements.

4. Compiling Barracuda to a GPU

Barracuda functions are compiled into CUDA code and C++ wrapper code, as seen from a very high level in Figure 2. Barracuda defines C++ types corresponding to each of the Barracuda vector and matrix types: an expression of type `SExp Int` in Barracuda maps to `int` in C++; `SExp Float` maps to `float`; and `SExp Bool` maps to `bool` in C++. The C++ component of Barracuda defines two tem-

RMSE in Barracuda

```
rmse :: VExp Float -> VExp Float -> SExp Float
rmse x y = sqrt (sumDiff / len)
  where
    len = fromIntegral (vlength x)
    sumDiff = vsum (vmap (^2) (vzipWith (-) x y))
```

Generated code for RMSE

```
// CUDA kernel; not exported
__global__ void
rmse_kernel (const float *x,
             const float *y,
             float *partial_results);

// Wrapper procedure; exported in interface
void
rmse (const gpu_vector<float> &x,
      const gpu_vector<float> &y,
      float &result)
{
    ...  // invoke kernel; finish reduction
}
```

Weighted moving average in Barracuda

```
wmAvg
  :: [Float]          -- weights
  -> VExp Float       -- input vector
  -> VExp Float
wmAvg ws xs = foldr1 (vzipWith (+)) slices
  where
    slices :: [VExp Float]
    slices = zipWith weightVec [0..] ws

    weightVec :: SExp Int -> Float -> VExp Float
    weightVec i w = vmap (* float w) slice
      where
        slice = vslice xs (i, sliceLen + i, 1)
        sliceLen = vlength xs - int (length ws)
```

Generated code for weighted moving average

```
// CUDA kernel; not exported
__global__ void
wmAvg_kernel (const float *xs, float *result);

// Wrapper procedure; exported in interface
void
wmAVg (const gpu_vector<float> &xs,
       gpu_vector<float> &result)
{
    ...  // invoke kernel
}
```

Figure 2. Two Barracuda programs and their corresponding generated code. Root mean square error is shown above; weighted moving average is shown below. The Barracuda implementations operate on Barracuda vectors and mirror what might be written in regular Haskell operating on Haskell lists: `VExp a`, `vmap`, `vsum`, `vlength`, and `vzipWith` correspond to `[a]`, `map`, `sum`, `length`, and `zipWith` in Haskell. The `int` and `float` functions in the second example are explicit conversions from Haskell values into Barracuda values. The function `vslice` denotes a sub-vector.

plate classes for vectors and matrices residing on the GPU, which are mapped to Barracuda's `VExp` and `MExp` type constructors:

```
template <class T>
class gpu_vector;

template <class T>
class gpu_matrix;
```

In other words, the location of arrays in the generated code is explicit in the types, and the caller of the code is responsible for deciding when to copy between device memory and main memory, which is one of the most costly operations of CUDA programming. These template classes are implemented so that memory is properly aligned, which is a necessary condition for memory coalescing.

The unit of compilation in Barracuda is the function. Barracuda functions are compiled into C++ procedures using the following parameter passing conventions:

- Input arguments are passed by constant reference; and
- the output argument is passed by reference.

By returning the result by reference, the onus of memory management for function outputs is placed on the caller. This is an inconvenience, but it gives the caller precise control over when allocations are made and allows for in-place array updates, both of which are important for high-performance computing.

4.1 Compiling the Array Primitives

In the simplest case, a Barracuda function does not use any of the array primitives, and therefore consists only of scalar code. Such code is easily compiled into CUDA. The interesting cases involve use of the array primitives, which, when generating parallel code (see section 7 for discussion of the alternative), are compiled into CUDA kernels and kernel invocation code.

Map A map operation applies an n-ary scalar mapping function to n arrays element-wise. A CUDA kernel is generated for the mapping function, and kernel invocation code is inserted into the C++ wrapper code being generated. In the current implementation, the generated kernel has each thread apply the mapping function to only a single set of n elements. More work-efficient code would have each thread apply the function to multiple elements, although it is not always a clear performance win; experimenting with this is left as future work.

Reduce A reduce operation accumulates a scalar value by repeatedly applying a binary reducing function to elements of an array. Similarly to the way the map operation is compiled, a CUDA kernel is generated for the reducing function, and kernel invocation code is inserted into the C++ wrapper code being generated. The Barracuda compiler generates code for reduction that uses a logarithmic fan-in: each CUDA thread block reduces a block of elements from the array using shared memory, and after the kernel execution finishes, the CPU performs a final reduction of the partial results. The reduction code Barracuda generates is based on the final optimized implementation described by Harris [2008].

Slice A slice operation names a sub-array of a larger array. For instance, a slice of a one-dimensional array specifies start index, stop index, stride, and the array to slice. In the case of a slice operation that appears at the top level of a Barracuda function, special copying routines provided by the Barracuda runtime are used. In the more interesting case where a slice expression occurs as an array argument of another array primitive, the slice is compiled by translating indexes into array slice into the appropriate indexes of the original array. Barracuda's array slices have no run-time reification, and are handled at compile time.

5. Shared Memory Optimization

As mentioned in section 2, graphics processors possess a limited on-chip shared memory space that can be accessed much more quickly than device memory. When a data-parallel array operation computes a function at each element using that element's value and the values of neighboring elements, large performance gains can be realized by generating CUDA code that uses the GPU's shared memory cache rather than accessing device memory repeatedly. The weighted moving average code listed in Figure 2 is an example: the input vector is sliced n times (once for each of n weight parameters), with each slice being shifted over one element from the previous slice. So, with $n = 15$, each vector element would be read 15 times by different threads.

The Barracuda compiler identifies array operations where use of shared memory is applicable, and in such situations it generates code that uses shared memory cache, storing one element at a time for each thread within a thread block. It does this as follows:

1. Collect all uses of array variables that are used in at least two unique *overlapping contexts*.
2. When generating kernel code for the function, for each such array variable, generate code to load the cache from device memory, and replace each corresponding array indexing expression with code that accesses shared memory when in bounds and that accesses device memory otherwise.

The *context* of an array variable refers to array slicing: an array variable can be referred to in *unsliced* or in *sliced* fashion. Two contexts are said to *overlap* if it can be statically determined that there exist array elements included in both contexts. Whether two contexts overlap is determined by simple rules:

1. Two identical array variables overlap.
2. An array variable and a slice of that variable overlap.
3. Two slices s_1 and s_2 of variable v overlap if in each dimension, the start and stop parameters of s_1 and s_2 form overlapping intervals and the overlap is a significant.[2]

If the expressions that specify slice parameters involve only compile-time constants and accessors for array extents (e.g., vector length), the Barracuda compiler will be able to determine conclusively whether two contexts overlap, using the above rules along with special handling of accessors of array extents.

Consider the forward difference operator, an archetypal stencil computation of the sort used in iterative methods. The forward difference operator is defined by the equation

$$\Delta \mathrm{v}(x) = \mathrm{v}(x+1) - \mathrm{v}(x)$$

where x and $x + 1$ are valid indexes for v. A Barracuda function for this transformation might be written as so:

```
forwardDiff :: VExp Float -> VExp Float
forwardDiff xs = vzipWith (-) xs'' xs'
  where
    xs'' = vslice xs (1, vlength xs, 1)
    xs'  = vslice xs (0, vlength xs - 1, 1)
```

Compiling this function without shared memory optimization would result in a kernel that would have each thread read from device memory twice, for two different elements of `xs`.

When compiling with shared memory optimization enabled, the compiler determines that variable named `xs` is used in two different slice contexts (named by `xs'` and `xs''`), and that `xs''`

[2] *Significant* being defined in the current implementation as 50% of the corresponding CUDA thread block dimension, or in other words, 128 for vector slices and 8 for matrix slices.

is the same as `xs'`, only shifted over one index. The Barracuda compiler has a special case for such "shifted" slices: only if the shift amount is enough to cause the contexts to be so different that they could not possibly significantly overlap will non-optimized code be generated. With the forward difference example, when shared memory optimization is enabled, the compiler generates a kernel in which the majority of threads read from device memory only once: the threads first load from device memory into shared memory, and then work from that. Conditional code is necessary to handle the cases where a thread is at the boundary of a thread block, or at the boundary of the vector.

6. Array Fusion

The array primitives in Barracuda can be freely nested. For instance, in the root mean square error example in Figure 2, `vzipWith` (vector map on two vectors) occurs within `vmap`, which occurs within `vsum` (a vector reduction). Naive compilation of this expression would generate code that evaluates from the most nested outward, using two temporary vectors and three CUDA kernel invocations. This code would be correct but inefficient. However, since Barracuda is applicative and total, it is always safe to fuse array arguments of map, reduce, and slice that are themselves array expressions.

The map and reduce primitives both involve scalar functions. When compiling the body of a CUDA kernel for an array expression, it is necessary to generate an indexing expression for the array expression. When indexing a composite array expression, the compiler composes the scalar functions in the appropriate way so that they are fused.

Consider a Barracuda function involving a multiply-and-add on a vector with some constants:

```
mulAdd :: VExp Float -> VExp Float
mulAdd xs = vmap (+ 1) (vmap (* 2) xs)
```

The vector indexing code for this composite vector expression would look roughly like $\mathrm{xs}[\mathrm{idx}] * 2 + 1$.

In Barracuda, fusion is guaranteed for array primitives that appear as array arguments: no temporaries will be allocated, and only a single pass over the input arrays will be performed. This fusion technique is most similar to that used in the Blitz++ array library for C++ [Veldhuizen 1998].

7. Hoisting Nested Array Operations

The array primitives in Barracuda are compositional, and so they can be used in any type-correct context, including within mapping and reducing functions. However, if the map and reduce primitives were always compiled into parallel code, a nested data-parallel target language would be required. Unfortunately, the CUDA programming model does not allow arbitrary nested data-parallel code—kernels can contain only sequential code. Instead of implementing the well-known and more general *flattening transformation* to convert nested data-parallel programs into flat data-parallel programs [Blelloch et al. 1993], Barracuda takes a simple, much less general approach based on hoisting, which is similar to loop-invariant code motion performed by compilers for imperative languages.

To add the sum of one vector to each element of a second, one might write

```
addSum :: VExp Float -> VExp Float -> VExp Float
addSum arr1 arr2 = vmap f arr2
  where f y = vsum arr1 + y
```

Here, `vsum`, an array reduction, is used within the mapping function given to `vmap`. In this case, the subexpression containing the nested array operation, `vsum arr1`, is independent of the argument of the mapping function, and can safely be hoisted out.

However, it is possible to write nested data-parallel functions that cannot be transformed by hoisting. Consider the following:

```
ndp :: VExp Float -> VExp Float -> VExp Float
ndp arr1 arr2 = vmap f arr2
  where f v = vsum (vmap (\x -> x + v) arr1)
```

For each element `v` of `arr2`, this function would compute the sum of adding `v` to each element of `arr1`. Here, the nested array operations cannot be hoisted out because they depend upon the value of the array element given to the mapping function. When nested array operations cannot be hoisted, the operation is implemented by emitting a sequential loop within the generated CUDA kernel; this is similar to the approach taken in Nikola [Mainland and Morrisett 2010] and Copperhead [Catanzaro et al. 2010]. In these cases, the Barracuda compiler emits a warning that the code will be inefficient: the principal motivation for using graphics processors is performance.

Earlier versions of Barracuda experimented with techniques to make expression of troublesome constructs impossible, but all proved unsatisfactory. A design goal of Barracuda was that it allow the array operations to be composed, and that it as be free of restrictions as possible. Disallowing nested data-parallel programs through type system techniques hindered expression, either by eliminating compositionality or preventing kernel functions from closing over their environments.

The hoisting technique presented here is hardly novel, and is not as general as the flattening transform, but it is simple to implement in an applicative array language like Barracuda: if a nested array expression does not contain variables bound by reducing or mapping functions, hoist the nested expression out and have it store its result in a temporary, and replace the original nested expression with a reference to the temporary. In the case of an array reduction (which results in a scalar value), the cost of a temporary is virtually free.

8. Experimental Results

I performed three experiments. First, I ran several benchmarks comparing Barracuda-generated code to handwritten implementations to demonstrate the basic viability of Barracuda. Second, I compared the run time of Barracuda-generated code with and without array fusion to measure its performance impact. Finally, I compared the run time of Barracuda-generated code with and without shared memory optimization to measure its performance impact.

In all the benchmarks, only the computation time was measured, not the time to copy data between main memory and device memory, as the copying costs would be the same for handwritten and Barracuda implementations. All of the benchmarks were run on a system with a quad-core Intel Q6600 CPU, 8 GB RAM, and a GeForce 8800GT graphics card with 512 MB RAM, running 64-bit Ubuntu Linux 10.04 and CUDA 3.2.

These benchmarks ran within microseconds even for large arrays, which complicated measurement of execution time, as such short runs approached the resolution of the timers available on the system. To minimize this source of error and to get consistent measurements, each benchmark was run repeatedly for thirty seconds, with the number of runs recorded. Only the operation being timed was executed within this loop; the requisite allocation and initialization of arguments was done outside of the timing loop. The values for the input arguments were chosen pseudo-randomly. The total execution time of a benchmark was then divided by the number of loop iterations performed to arrive at the average execution time.

The benchmarks that work with vectors were run with power-of-two sizes. The benchmarks that work with matrices were run

with square matrices with power-of-two dimensions. This was done for simplicity and also because power-of-two vectors and matrices satisfy the memory alignment properties necessary for good CUDA performance without extra padding. Note, however, that the Barracuda-generated code is not restricted to run with power-of-two vectors or square matrices, but is general.

8.1 Basic Performance Measures

Operations from the BLAS library [Lawson et al. 1979] and Black-Scholes option pricing are benchmarks that frequently appear in work on systems for high-performance array computing (e.g., see Lee et al. [2009] and Mainland and Morrisett [2010]), as they test the basic viability of such systems. I compared the performance of handwritten CUDA code provided in the CUDA SDK or the cuBLAS library to that of Barracuda-generated code. The two BLAS operations chosen were SAXPY and SDOT; SAXPY is expressed as a vector map operation in Barracuda, and SDOT is expressed as a vector reduction.

Results of these benchmarks are shown in Figure 3. In the SAXPY benchmark, we see that the Barracuda version always runs faster than the cuBLAS version, which is a surprising result. A simple modification of Barracuda's generated code to use the same block and grid dimensions as the cuBLAS implementation indicates that the differences in these factors are not the cause of Barracuda's superior performance. Without access to the cuBLAS source code it is difficult to say conclusively, but it seems that the cuBLAS implementation allows for non-unit stride in its vector arguments and does not have a code path specialized unit stride.[3]

A Barracuda version of Black-Scholes option pricing was compared to the implementation found in the CUDA 3.2 SDK examples. Black-Scholes option pricing is another example of an element-wise vector transformation, but with a much more complex and expensive function than SAXPY. This benchmark demonstrates the viability of a high-level language for more realistic computations that one would like to run on a graphics processor. We see that once the vectors are large enough to warrant running on a GPU, the performance of the Barracuda version is within five percent of the handwritten version from the CUDA SDK.

The Black-Scholes benchmark points out a deficiency in Barracuda: Barracuda programs are single expressions, and there are no tuple types, meaning only a single result can be returned. The handwritten version can evaluate both call and put option prices within a single pass, and work can be shared between the two evaluations, i.e., it has two output vectors. Because only a single result can be returned by Barracuda programs, one would have to call two different Barracuda-generated procedures to evaluate both call and put option prices, which would require two passes over the input vectors and would eliminate the possibility of sharing intermediate results. Barracuda could be made more generally usable by allowing functions to return multiple results.

8.2 Impact of Array Fusion

To test the impact of array fusion, a benchmark from the RMSE program of Figure 2 was created, both with and without array fusion.[4] The results are shown in Figure 4. We see a speedup of three when the input vectors are large.

[3] A Barracuda user could generate an implementation that used non-unit stride by using Barracuda's vector slicing capability, so long as the stride was known at compile time.

[4] Note that Barracuda always uses array fusion when generating code; the non-fused version was synthesized by calling several Barracuda-generated routines for the various nested subexpressions of RMSE.

8.3 Impact of Shared Memory Optimization

To test the impact of shared memory optimization, I ran three stencil computation benchmarks: forward difference, a two-dimensional Jacobi iterative solver, and a 15-point weighted moving average. These three benchmarks all demonstrate the combined use of map and slice array primitives, and all three result in output arrays that are smaller than the inputs.[5]

Forward difference The forward difference function given in section 5 was tested with and without shared memory. We see in the results that once the vectors become large enough, the performance of the shared memory version becomes several times faster than the unoptimized version, and appears to approach a four times speedup in the limit. One might expect the speedup of the shared memory version to approach two at the limit, because each element of the vector (with the exception of the first and last) is read by two threads. In this case, the use of shared memory facilitates memory coalescing, and so fewer overall memory transactions are required, explaining the greater-than-two speedup.

Weighted moving average Weighted moving average is a complicated one-dimensional stencil operation. For this benchmark, the example code from Figure 2 was given the weights for Spencer's 15-point moving average.[6] When the vectors become large, shared memory optimization results in an eight times speedup.

Jacobi iteration stencil Solving a two-dimensional discretization of Poisson's equation is a more complicated demonstration of a stencil computation than forward difference. We see from the results that when the matrices become large enough, use of shared memory improves performance by roughly a factor of two.

9. Related Work

There have been many higher-level languages designed for programming GPUs, most based on array programming or stream programming. Brook is a C-like language that treats the GPU as a stream processor [Buck et al. 2004]. Sh is a shader and stream programming language embedded within C++, making heavy use of the template mechanism and preprocessor [McCool et al. 2004]. Copperhead [Catanzaro et al. 2010] is a system for Python that compiles a subset of Python code to GPU code, in particular various vector primitives such as map and reduce.

Most similar to Barracuda are array languages embedded within Haskell. Accelerate [Lee et al. 2009] provides an imperative array language and an online compiler targeting CUDA. Accelerate offers a richer set of primitives and supported types than Barracuda; in particular, product types and the prefix scan primitive are supported. Accelerate disallows expression of nested data-parallel programs by giving its parallel array primitives monadic result type, and hence making them non-compositional. Nikola [Mainland and Morrisett 2010] is an applicative array language and online compiler targeting CUDA. Nikola demonstrates how more Haskell language constructs, such as function application and let-expressions, can be used to denote object language constructs. Array fusion is not discussed in either the work on Accelerate or the work on Nikola, and neither of these systems hoist nested array expressions.

There is little work on automatic use of GPU shared memory cache; none of the other systems described here implement the shared memory optimization described in this paper. Silberstein et al. [2008] described dynamic spatial and temporal caching using shared memory, implemented in software, whereas the shared memory optimization described in this paper is entirely static.

[5] For discussion of how in-place array updates are made possible in Barracuda, see Appendix B.2.

[6] -3, -6, -5, 3, 21, 46, 67, 74, 67, 46, 21, 3, -5, -6, and -3.

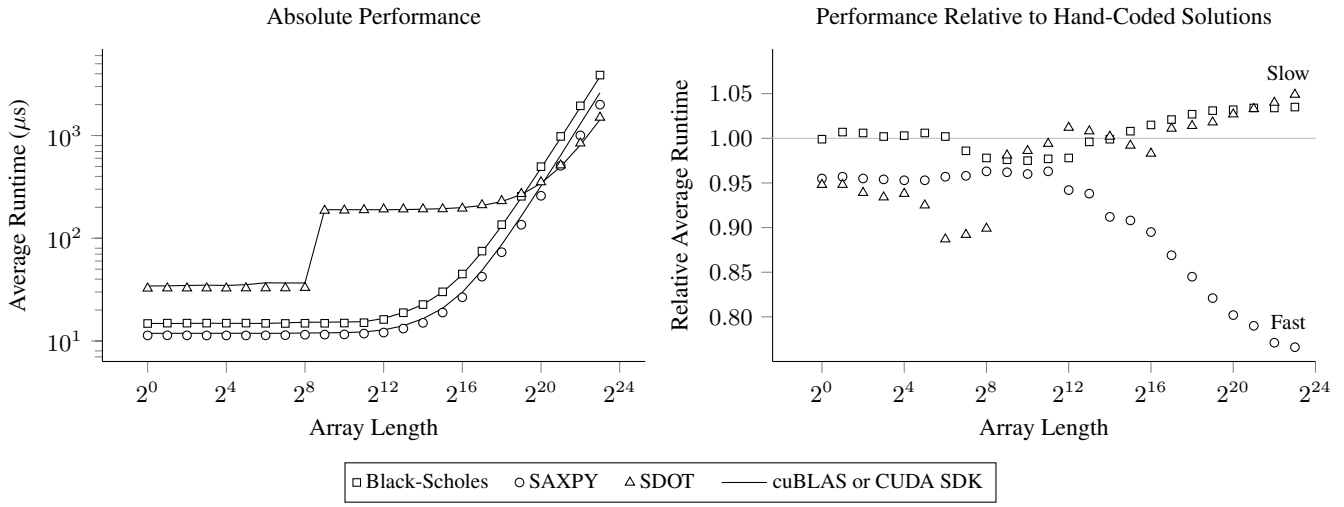

Figure 3. Performance measures of Barracuda programs. Benchmarks are from Lee et al. [2009], Mainland and Morrisett [2010], and Lawson et al. [1979]. Only computation time on the GPU was measured, not the time needed to copy data to and from the GPU. The left graph shows the absolute run times for the Barracuda versions, and the corresponding cuBLAS or CUDA SDK versions; the performance of the Barracuda code is close to that of the reference implementations. Relative run times for the same Barracuda programs appear in the graph on the right, showing that on all benchmarks, Barracuda performs within 5% of hand-written code. On SAXPY, Barracuda is 5–20% faster.

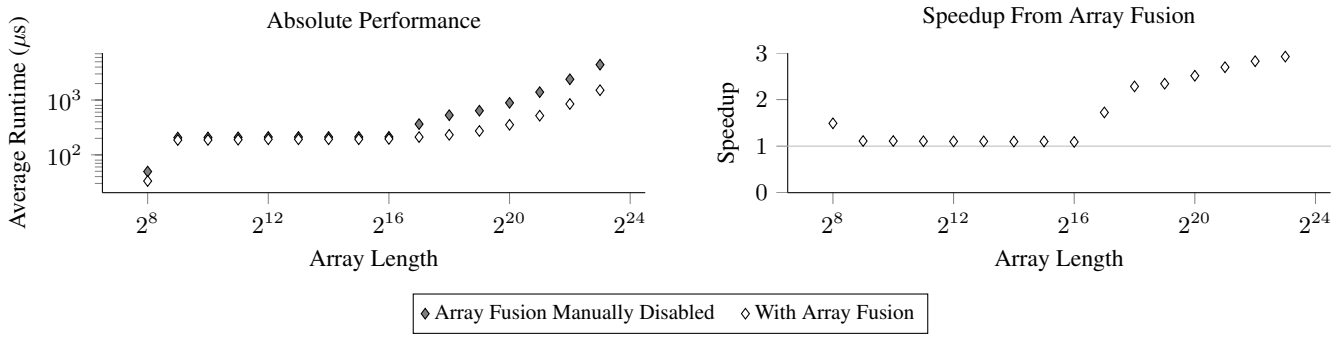

Figure 4. Performance measurements for the Root Mean Square Error Barracuda program with and without array fusion. Only computation time on the GPU was measured, not the time needed to copy data to and from the GPU.

10. Conclusion

This paper has shown that using a high-level, applicative, compositional array programming language can hide most of the complexity of GPU programming, yet can have performance competitive with handwritten code. This paper described three simple optimizations—shared memory optimization, array fusion, and hoisting of nested array expressions—and showed that the former two can result in large speedups. Finally, in virtue of the language being applicative and compositional, these optimizations are simple to implement, requiring almost no analysis.

10.1 Challenges for Embedded Languages

The experience of embedding a language within Haskell caused me to run into two difficulties. First, when deeply embedding a language within another, one would like to have natural, lightweight concrete syntax for the embedded language. Haskell makes a pleasant metalanguage, as it supports definition of new infix operators and overloading through type classes, but it would be beneficial if more Haskell language constructs could be overloaded, such as conditional expressions, let-expressions, and lambda expressions.

It is difficult to enforce complex static semantics of an object language statically in the metalanguage. For example, it might be desirable to statically guarantee against expression of non-hoistable nested data parallel functions in Barracuda. Although a metalanguage with a more expressive type system (e.g., full dependent types) could allow embedding of more complicated semantics of an object language, it would be a great boon if such semantics could be expressed in a more explicit way than through the type system of the metalanguage.

10.2 Future Work

Barracuda is a limited language. It is sufficiently expressive to describe certain element-wise transformations on arrays and matrices, but is unable to express operations like matrix multiplication and sorting in a single function. A more generally useful embedded array language would feature more array primitives, such as *scan* operations, which alone can be used to express many realistic data-parallel algorithms [Blelloch 1989]. The expressiveness of Barracuda could further be improved by allowing multiple results to be returned from functions. Furthermore, the language could be

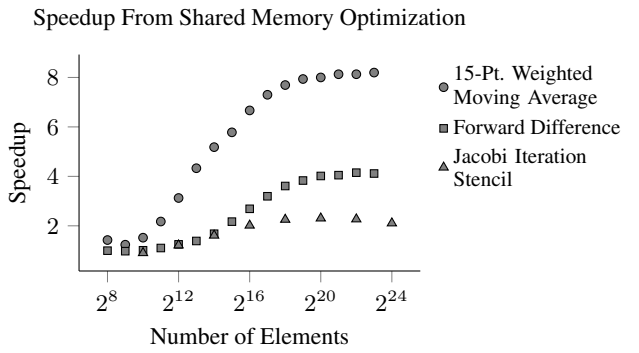

Figure 5. Run-time speedup for Barracuda programs compiled with shared memory optimization. As with the other results, only computation time on the GPU was measured, not the time needed to copy data to and from the GPU. The Jacobi Iteration Stencil benchmark was run with square matrices.

made more reusable by adding *rank polymorphism* [Keller et al. 2010], e.g., so that array map could be used to express element-wise operations on arrays of arbitrary dimensionality.

Barracuda is designed to be used as an offline compiler for generating code to be used in a performance-oriented C++ application. It would be advantageous if Barracuda could also be used to access a graphics processor entirely from within Haskell as an online compiler, as both Accelerate [Lee et al. 2009] and Nikola [Mainland and Morrisett 2010] allow.

The fusion technique used in Barracuda is simple. More powerful, sophisticated techniques for data structure fusion have been developed [Coutts et al. 2007; Gill et al. 1993], which would be important in a richer array language where the simple fusion scheme presented here would be insufficient.

There are several ways in which the generated code could be improved, for instance, through the elimination of unnecessary array bounds checking and through specialization of kernels for arrays of certain lengths. Such techniques could improve performance of generated code even further.

Finally, a more useful array-centric language might have multiple targets, e.g., multicore CPUs, GPUs, the Cell processor, and other high-performance architectures.

Acknowledgments

I gratefully acknowledge the help of Norman Ramsey and Philip Hatcher in proofreading drafts of this paper and in suggesting ways to better present the experimental results.

This work was supported in part by the NASA Space Grant Graduate Fellowship and NSF grant IIS-0082577.

References

G. E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38(11):1526–1538, 1989. ISSN 0018-9340.

G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996. ISSN 0001-0782.

G. E. Blelloch, J. C. Hardwick, S. Chatterjee, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. In *PPOPP '93: Proceedings of the fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 102–111, New York, NY, USA, 1993. ACM. ISBN 0-89791-589-5.

I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM.

B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an embedded data parallel language. Technical Report UCB/EECS-2010-124, EECS Department, University of California, Berkeley, September 2010.

J. Cheney and R. Hinze. First-class phantom types. Technical Report TR2003-1901, Cornell University, July 2003.

D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: from lists to streams to nothing at all. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, pages 315–326, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-815-2.

S. Edelkamp, D. Sulewski, and C. Yücel. Perfect hashing for state space exploration on the GPU. In R. I. Brafman, H. Geffner, J. Hoffmann, and H. A. Kautz, editors, *Proceedings of the 29th International Conference on Automated Planning and Scheduling, ICAPS 2010, Toronto, Ontario, Canada, May 12–16, 2010*, pages 57–64. AAAI Press, May 2010.

C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(3):455–481, May 2003.

E. Elsen, M. Houston, V. Vishal, E. Darve, P. Hanrahan, and V. Pande. N-body simulation on GPUs. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, page 188, New York, NY, USA, 2006. ACM. ISBN 0-7695-2700-0.

A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *FPCA '93: Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, New York, NY, USA, 1993. ACM. ISBN 0-89791-595-X.

M. Harris. Optimizing parallel reduction in CUDA. PDF, 2008. Provided in the documentation of the CUDA 3.2 SDK.

K. E. Iverson. A programming language. In *AIEE-IRE '62 (Spring): Proceedings of the May 1–3, 1962, spring joint computer conference*, pages 345–351, New York, NY, USA, 1962. ACM.

T. Johnsson. Lambda lifting: transforming programs to recursive equations. In *Proceedings of a Conference on Functional Programming Languages and Computer Architecture*, pages 190–203, New York, NY, USA, 1985. Springer-Verlag New York, Inc. ISBN 3-387-15975-4.

G. Keller, M. M. Chakravarty, R. Leschinskiy, S. P. Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2010, pages 261–272, New York, NY, USA, September 2010. ACM. ISBN 978-1-60558-794-3.

C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979. ISSN 0098-3500.

S. Lee, M. M. T. Chakravarty, V. Grover, and G. Keller. GPU kernels as data-parallel array computations in Haskell. *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods*, 2009.

G. Mainland and G. Morrisett. Nikola: Embedding compiled GPU functions in Haskell. In *Proceedings of the third ACM Haskell symposium on Haskell*, pages 67–78, New York, NY, USA, September 2010. ACM. ISBN 978-1-4503-0252-4.

P. Manolios and Y. Zhang. Implementing survey propagation on graphics processing units. In A. Biere and C. P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121 of *Lecture Notes in Computer Science*, pages 311–324. Springer, 2006. ISBN 3-540-37206-7.

M. D. McCool, S. Du Toit, T. Popa, B. Chan, and K. Moule. Shader algebra. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 787–795, New York, NY, USA, 2004. ACM.

NVIDIA. *NVIDIA CUDA Programming Guide Version 3.2*. NVIDIA, 2010.

F. Pfenning and C. Elliott. Higher-order abstract syntax. *ACM SIGPLAN Notices*, 23(7):199–208, July 1988.

S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, pages

97–106, Aire-la-Ville, Switzerland, 2007. Eurographics Association. ISBN 978-1-59593-625-7.

M. Silberstein, A. Schuster, D. Geiger, A. Patney, and J. D. Owens. Efficient computation of sum-products on GPUs through software-managed cache. In *ICS '08: Proceedings of the 22nd Annual International Conference on Supercomputing*, pages 309–318, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-158-3.

T. L. Veldhuizen. Arrays in Blitz++. In D. Caromel, R. Oldehoeft, and M. Tholburn, editors, *ISCOPE '98: Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*, volume 1505 of *Lecture Notes in Computer Science*, pages 223–230, London, UK, 1998. Springer-Verlag. ISBN 3-540-65387-2.

H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 224–235, New York, NY, USA, 2003. ACM. ISBN 1-58113-628-5.

A. Embedding Barracuda

A.1 Language Representation

Barracuda is an embedded domain-specific language (EDSL) in Haskell. It is *deeply embedded*, meaning that a programmer writes Haskell code that constructs abstract syntax trees. These abstract syntax trees are manipulated and eventually compiled into GPU code. Although the metalanguage limits the choice of concrete syntax, Haskell's overloading and the ability to define new infix operators helps keep the syntax overhead low.

Barracuda functions are represented as Haskell functions operating on the abstract syntax tree (AST) types shown in Figures 6 and 7. The type constructors `SExp`, `VExp`, and `MExp` represent scalar, vector, and matrix expressions. The allowed element types of such expressions (i.e., `Float`, `Int`, and `Bool`) are instances of the class `Scalar`. These types are generalized algebraic data types, carefully defined so that ill-typed Barracuda programs are ill-typed Haskell programs, making ill-typed Barracuda programs impossible to construct [Cheney and Hinze 2003; Xi et al. 2003], which simplifies parts of the compiler implementation.

A.2 Smart Constructors

A Barracuda programmer does not construct ASTs directly, but instead uses *smart constructors*, i.e., Haskell functions that may perform some computation to construct values of a data type. In many cases, the smart constructors have both the same type and name as their corresponding AST constructors, shown in Figures 6 and 7, but lowercase instead of uppercase. In other cases, such as with arithmetic operations, the smart constructors are suitably overloaded so that syntactic overhead is low. For instance, numeric scalar expressions types in Barracuda are instances of the appropriate numeric Haskell type classes, allowing the normal arithmetic functions to be used to construct Barracuda program fragments. This overloading approach is frequently taken when deeply embedding languages within Haskell. Following Elliott et al. [2003], the smart constructors perform optimizations from the bottom up, including constant folding and algebraic simplifications.

Barracuda functions are represented using higher-order abstract syntax [Pfenning and Elliott 1988]. This permits the use of functions in the metalanguage (Haskell) to represent functions in the object language (Barracuda), allowing reuse of the metalanguage's name binding mechanism. In particular, in Barracuda, the constructors representing let-expressions, reduction, and mapping use Haskell functions in their representation.

```
-- allowed scalar types
class (Ord a, Show a, Typeable a) => Scalar a where
  ...  -- methods omitted

type Id = String
type Reducer a = SExp a -> SExp a -> SExp a

-- scalar expressions
data SExp :: * -> * where
    -- scalar constants
    SCFloat :: Float -> SExp Float
    ...
    -- primitive operations
    SFAdd :: SExp Float -> SExp Float -> SExp Float
    SFMul :: SExp Float -> SExp Float -> SExp Float
    VIdx :: (Scalar a) => Id -> SExp Int -> SExp a
    ...
    -- let-expressions
    SLet :: (Scalar a, Scalar b)
         => SExp a                -- expr. to bind
         -> (SExp a -> SExp b)    -- body
         -> SExp b
    ...
    -- reductions
    VReduce :: (Scalar a)
            => Reducer a    -- reducing function
            -> SExp a       -- initial value
            -> VExp a       -- vector to reduce
            -> SExp a
    ...
```

Figure 6. The representation of scalar expressions.

```
-- slice descriptors
data VSliceD = ...  -- definition omitted
data MSliceD = ...  -- definition omitted

-- vector expressions
data VExp :: * -> * where
    -- vector slices
    VSlice :: (Scalar a) => VSliceD -> Id -> VExp a
    -- array map expressions
    VMap :: (Scalar a, Scalar b)
         => (SExp a -> SExp b)
         -> VExp a
         -> VExp b
    ...

-- matrix expressions
data MExp :: * -> * where
    -- matrix slices
    MSlice :: (Scalar a) => MSliceD -> Id -> MExp a
    -- matrix map expressions
    MMap   :: (Scalar a, Scalar b)
           => (SExp a -> SExp b)
           -> MExp a
           -> MExp b
    ...
```

Figure 7. The representation of vector and matrix expressions.

A.3 Explicit Sharing

In order to avoid redundant computation of expressions, it is necessary for Barracuda to have a notion of sharing. Consider a function that squares its argument:

```
square :: (Num a) => a -> a
square x = x * x
```

When applied to a Barracuda expression, this Haskell function will result in code duplication. Consider:

```
addOneSquare :: SExp Float -> SExp Float
addOneSquare x = square (x + 1)
```

Although in the metalanguage the result of `x + 1` will be shared when used with `square`, the representation in the object language lacks this sharing. The following AST results from `addOneSquare`:

```
\x -> SFMul (SFAdd x (SCFloat 1.0))
            (SFAdd x (SCFloat 1.0))
```

Essentially, function application in Haskell acts as macro expansion in Barracuda. This can be problematic in complicated functions, as it results in too much inlining. What is needed is that sharing be explicit in Barracuda. To allow this, rather than recovering sharing using a common subexpression elimination pass, the `slet` smart constructor is used, which is a Barracuda let-expression for scalar expressions. The type of `slet` matches its AST counterpart:

```
slet
  :: (Scalar a, Scalar b)
  => SExp a                -- expression to bind
  -> (SExp a -> SExp b)    -- body
  -> SExp b
```

This construct indicates that its first argument is to be computed once and referenced potentially multiple times within the body (a higher-order function). Using this construct, one could write a new version of `addOneSquare` that will not result in code duplication:

```
square' :: (Num a, Scalar a) => SExp a -> SExp a
square' x = slet x (\x' -> x' * x')

addOneSquare' :: SExp Float -> SExp Float
addOneSquare' x = square' (x + 1)
```

The Barracuda AST resulting from `addOneSquare'` is

```
\x -> SLet (SFAdd x (SCFloat 1.0))
           (\x' -> SFMul x' x')
```

This lacks the duplication found in the previous implementation. Writing functions using `slet` to indicate sharing in the object language adds syntactic noise, but is crucial for generating reasonable code for complicated functions.

A.4 Specifying Names in the Generated Code

The Barracuda compiler can generate CUDA procedures for Barracuda expressions (i.e., values of type `SExp a`, `VExp a`, or `MExp a`, where `a` is one of the allowed element types), and for Haskell functions taking a Barracuda expression and returning something that can be compiled. In other words, Haskell values of type

$$T_1 E_1 \to \ldots \to T_n E_n$$

where each T_i is one of the type constructors `SExp`, `VExp`, or `MExp`, and each E_i is one of the allowed element types, can be compiled into CUDA procedures. The Haskell types that can be compiled are instances of the `Compilable` type class.

Because Barracuda generates code meant to be called directly by a C++ programmer, it is important that the generated procedures and their arguments have meaningful names. This is done by annotating Barracuda functions with name information via the `Function` data type:

```
data Function :: * where
  Function
    :: (Compilable a)
    => a                -- a Barracuda function
    -> String           -- function name
    -> [String]         -- input names
    -> String           -- output name
    -> Function
```

`Function` is a GADT that takes something the Barracuda compiler can handle and annotates it with name information to be used in the generated code.

B. Compilation Details

B.1 Lambda Lifting

A function in Barracuda closes over its lexical environment. For instance, consider a Barracuda function that adds a given scalar value to each element of an array:

```
addVal :: SExp Float -> VExp Float -> VExp Float
addVal x xs = vmap (\y -> x + y) xs
```

The mapping function `\y -> x + y` contains the closed-over variable `x`; in other words, `x` occurs free in the mapping function. This presents a challenge for compilation, as CUDA has no notion of closures. Lambda lifting [Johnsson 1985] is used to transform the mapping and reducing functions in Barracuda—which may contain closed-over variables—into corresponding functions with no closed-over variables. Applying this to the `addVal` function above, the kernel generated for the mapping function takes `x` as an additional argument that must be passed in with the `x` from the environment where the kernel is used.

B.2 In-place Array Updates

In high-performance array computing it is important that subsections of arrays can be updated in-place. However, Barracuda is an applicative language without assignment, which seems at odds with this requirement. The solution is simple: for array and matrix outputs, pass a *view*, or a runtime representation of an array or matrix slice, instead of an array or matrix itself. *view* is responsible for translating indexes into the appropriate indexes of the viewed array or matrix. Consider the Barracuda function that doubles each element of an array:

```
doubleArr :: VExp Float -> VExp Float
doubleArr xs = vmap (\x -> x * x) xs
```

This will be compiled into a C++ procedure with signature

```
void doubleArr (const gpu_vector<float> &xs,
                gpu_vector_view<float> &output);
```

The output argument is a reference of type `gpu_vector_view<float>` rather a reference of type `gpu_vector<float>`. The caller of this procedure could then specify that the output array is a sub-array of another using the `view` function from Barracuda's runtime, e.g.,

```
gpu_vector<float> arr1;
gpu_vector<float> arr2;
...
doubleArr(arr1, view(arr2, 10, 100));
```

This use of `view` would write the results of doubling `arr1` into `arr2` from locations 10 to 100. If no slicing of the output array is desired, the caller need not use `view`, but could instead simply pass `arr2`, as a `gpu_vector<T>` will be implicitly cast to a `gpu_vector_view<T>`.