# Compiling an Array Language to a Graphics Processor

BY

Bradford Larsen

B.A. Philosophy, University of New Hampshire (2009)

B.S. Computer Science, University of New Hampshire (2008)

THESIS

Submitted to the University of New Hampshire
in Partial Fulfillment of
the Requirements for the Degree of

Master of Science

in

Computer Science

December 2010

ALL RIGHTS RESERVED

© 2010

Bradford Larsen

This thesis has been examined and approved.

Thesis director, Philip J. Hatcher
Professor and Chair of Computer Science

R. Daniel Bergeron
Professor of Computer Science

Kai Germaschewski
Assistant Professor of Physics

Date

# CONTENTS

# LIST OF CODE SAMPLES

# LIST OF FIGURES

# ACKNOWLEDGMENTS

I would like to thank Professor Joachim Raeder for encouraging and funding me to attend the Path to Petascale workshop in 2009. Without that experience this work would not have happened.

This work was funded through the NASA Space Grant Graduate Fellowship and NSF grant IIS-0082577.

# ABSTRACT

**COMPILING AN ARRAY LANGUAGE TO A GRAPHICS PROCESSOR**

by

Bradford Larsen

University of New Hampshire, December, 2010

Graphics processors are significantly faster than traditional processors, particularly for numerical code, and in recent years have become flexible enough to permit general-purpose use, rather than just graphics use. NVIDIA's CUDA makes general-purpose graphics processor computing feasible, but it still requires significant programmer effort.

My thesis is that array programming can be an effective way to program graphics processors, and that a restricted, functionally pure array language coupled with simple optimizations can have performance competitive with handwritten GPU programs. I support this thesis through the research language Barracuda, an array language embedded within Haskell that generates optimized CUDA code.

# 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 an array *efficiently* requires over 150 lines of CUDA; in a typical 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 array 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 [Lee et al., 2009a, Mainland and Morrisett, 2010, Catanzaro et al., 2010]. This thesis shows that a simple array programming model supported by a few simple optimiza-

tions 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, this work makes the following contributions:

- I describe how to automatically make use of GPU shared memory cache in stencil operations to reduce memory traffic, and show experimentally that it can increase performance by up to a factor of eight.
- I describe a simple, yet effective array fusion scheme to further reduce memory traffic and eliminate the need for temporary arrays, improving performance by up to an order of magnitude as observed in experiments.
- I describe how certain nested data-parallel programs can be easily transformed into equivalent non-nested programs through hoisting.

These optimizations should be applicable to any array language targeting GPUs or similar accelerator hardware that allows (or requires) explicit cache control, such as the Cell processor. These optimizations are implemented in a compiler for the research language Barracuda, an applicative array programming language for GPUs that is compositional, allowing array primitives to be freely nested. Barracuda is implemented as an embedded language within Haskell.

# CHAPTER 1

# BACKGROUND

## 1.1 GPUs: Not Just for Graphics

In recent years, graphics processing units (GPUs) have been used for computationally intensive tasks other than graphics, such as SAT solving [Manolios and Zhang, 2006], state space search [Edelkamp et al., 2010], and physics simulations [Elsen et al., 2006]. This development has been motivated by the promise of superior performance of GPUs over more traditional processors, particularly for floating-point operations.

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 an array *efficiently* requires over 150 lines of CUDA; in a typical programming language, this problem can be solved with at most a few lines of code.

## 1.2 General-Purpose GPU Programming with CUDA

The CUDA programming language from NVIDIA is a C++ dialect that explicitly supports use of graphics cards for more general purposes [NVIDIA, 2010]. In synopsis, CUDA allows use of a GPU through data parallel execution of a restricted class of C++ procedures.

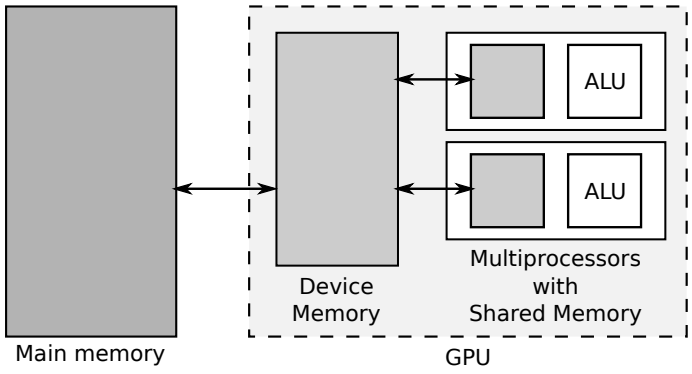


Figure 1.1: A block diagram of the GPU

### 1.2.1 Devices and Address Spaces

CUDA programming extends the C++ programming model with the notion of a *device* (i.e., a graphics processor) possessing many processing cores. In addition, each device has several additional address spaces associated with it, including device memory, shared memory, constant memory, and texture memory, which are distinct from each other and from the system's main memory, and data can only be transferred between the address spaces in fixed, well-defined ways. The rough organization of a system with a GPU is shown in Figure 1.1. The memory management procedures `cudaMalloc` and `cudaFree` are provided to allocate and free device memory, and the procedure `cudaMemcpy` can copy between main memory and device memory.

### 1.2.2 Kernels

A *kernel* is a C++ procedure that can be executed on the GPU. For example, a kernel used to compute the sine of each element of an array is shown in Listing 1.1. Kernel code is marked with the `__global__` qualifier, and is subject to several restrictions, including the following:[1]

- The return type must be `void`

[1]These conditions are somewhat relaxed with the latest NVIDIA GPUs and the most recent versions of CUDA; for instance, recursion *is* allowed starting with CUDA 3.0 on Fermi-based GPUs, and print statements can be executed from kernel code in debug mode.

```
__global__ void
array_sine_kernel (const float *in, float *out, const unsigned len)
{
    // Get the index into the array for the executing thread.
    const unsigned i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < len)
        out[i] = sin(in[i]);
}
```

Listing 1.1: A CUDA kernel to compute the sine of each element of an array

- Pass-by-reference arguments are not permitted
- Main memory is inaccessible
- The only allowed side effect is assignment
- Objects with virtual methods may not be used
- Recursion is not allowed

The CUDA programming model is mainly data parallel: a kernel specifies sequential code to be run by thousands of lightweight *kernel threads*, in parallel, on different data. In the kernel shown in Listing 1.1, each kernel thread is responsible for computing the sine of a single element from `in` and writing the result to the appropriate element of `out`. The element that a thread is to work on is determined from three special variables that are in scope in the body of a kernel procedure: `threadIdx`, `blockIdx`, and `blockDims`. Each of these variables is of type `dim3`, a struct with unsigned integer fields `x`, `y`, and `z`.

### 1.2.3 Problem Decomposition

To execute a kernel, the programmer must use another CUDA-specific construct, the *kernel invocation*. Invoking a kernel is similar to calling a regular procedure—one specifies the arguments to the kernel, but in addition, one must specify parameters that control the data parallel execution using special kernel invocation syntax.

To make effective use of a GPU, a kernel must be executed by thousands of kernel threads in data parallel fashion. Work is assigned to threads through a two-level decomposition of the problem. At the lower level, threads are organized into one-, two-, or three-

dimensional *thread blocks*. At the higher level, thread blocks are organized into a one- or two-dimensional *grid*. The CUDA runtime handles the scheduling of thread blocks to GPU multiprocessors.

Continuing our example, we see CUDA code to invoke the sine kernel in Listing 1.2. This is code suitable for the not-entirely-unreasonable use case where much of the processing is done on a CPU, but one wishes to accelerate certain operations by performing them on a GPU instead. The `in` and `out` parameters are pointers to main memory. However, CUDA kernels cannot access main memory, so we need to allocate temporary arrays on the GPU. (Use of temporary arrays is expensive and something to avoid as much as possible.) The first thing the wrapper code does is allocate temporary arrays on the GPU using `cudaMalloc`. Next, the input array (residing in main memory) is copied to the temporary input array (residing in device memory) using `cudaMemcpy`. The sine kernel is then invoked. Finally, the results in the temporary output array (residing in device memory) are copied into the true output array (residing in main memory), and the temporary arrays are freed using `cudaFree`.

In this example, the two-level data parallel decomposition of the problem uses a one-dimensional grid of one-dimensional thread blocks, with each thread block comprising 128 threads. The size of the grid is dependent upon the length of the arrays, as each kernel thread computes the transform for a single element of the array. For example, an array 256 elements long would be decomposed into two blocks of 128 threads each, as seen in Figure 1.2.

### 1.2.4 Shared Memory

As mentioned previously, CUDA exposes an additional *shared memory* address space to kernel code. Each thread block has shared memory that is accessible by all threads within the block. It is much faster to access shared memory than device memory: accessing shared memory can be as fast as accessing a register, whereas accessing device memory can take hundreds of cycles. Hence, in kernel code where multiple threads within a block would access the same memory locations, it makes sense to use shared memory instead of redundantly accessing the slower device memory. Shared memory is analogous performance-

```
void
array_sine_wrapper (const float *in, float *out, const unsigned len)
{
    // Allocate memory on the GPU for the arrays.
    float *in_d;
    float *out_d;

    const size_t arr_size = len * sizeof(float);

    cudaMalloc ((void **)&in_d, arr_size);
    cudaMalloc ((void **)&out_d, arr_size);

    // Copy the input array into GPU memory.
    cudaMemcpy (in_d, in, arr_size, cudaMemcpyHostToDevice);

    // Set up and invoke the kernel.
    const dim3 block_dim (128, 1, 1);
    const dim3 grid_dim ((unsigned)ceilf((float)len / 128f), 1, 1);

    array_sine_kernel<<<grid_dim, block_dim>>>(in_d, out_d, len);

    // Copy results back into CPU memory.
    cudaMemcpy (out, out_d, arr_size, cudaMemcpyDeviceToHost);

    // Release the GPU memory.
    cudaFree (out_d);
    cudaFree (in_d);
}
```

Listing 1.2: A CUDA wrapper procedure to invoke the kernel of Listing 1.1

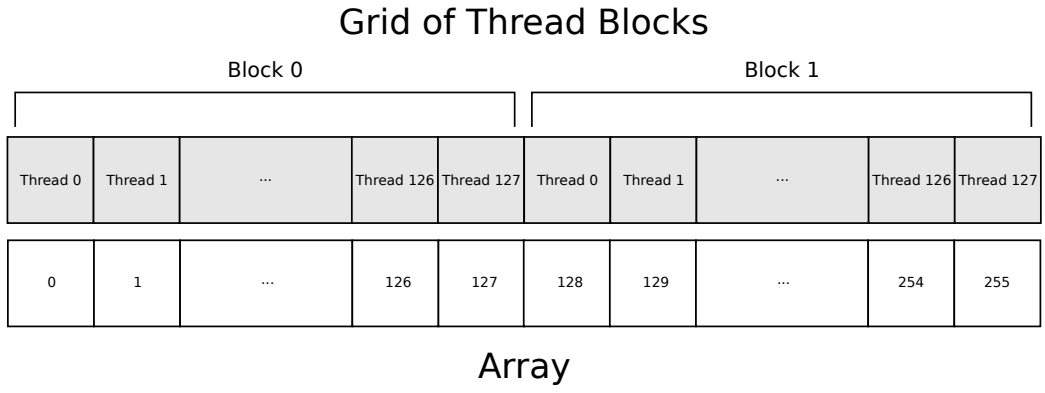


Figure 1.2: A 256-element array decomposed into a grid of 2 128-element blocks

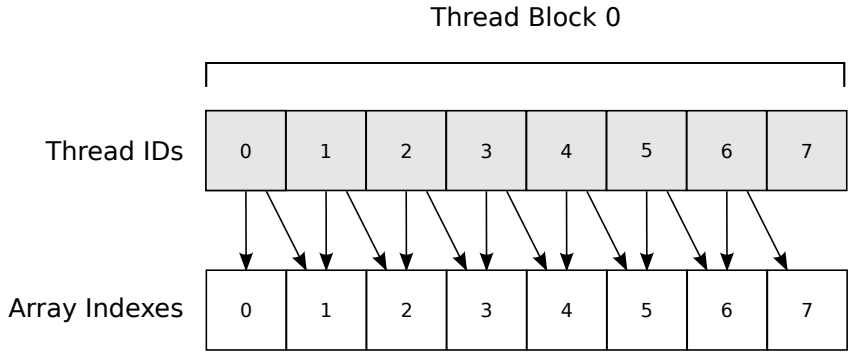


Figure 1.3: The memory access pattern of the input array for the forward difference routine. With the exception of the first and last array elements, each element is read twice.

wise to L1 cache in a CPU, but differs sharply in the fact that shared memory caching must be done by hand by the programmer.[2]

### 1.2.5 Example: Forward Difference

A more complicated element-wise transformation demonstrates the use of shared memory. The forward difference operator applied to a one-dimensional array is given by the equation

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

where $x$ and $x+1$ are valid indexes for $\mathrm{arr}$. This operator is an archetypal tool used in finite difference methods for numerical approximation of differential equations.

A naive CUDA implementation of this routine would have each thread read two elements from the input array residing in device memory. Analyzing the memory access patterns of this routine, we see that with the exception of the first and last elements of the input array, each input element would be read by two threads. This is shown in Figure 1.3. By using the on-chip shared memory of a GPU multiprocessor, only one read, rather than two, would be required.

Our strategy for using shared memory in this example is as follows. First, each thread reads a single element of the input array if it is in bounds, storing it in shared memory. Second, each thread synchronizes with the others in its block, necessary to make the writes

[2]Beginning with CUDA 3.0, implicit L1- and L2-caching is performed with the latest NVIDIA GPU architectures; however, NVIDIA still suggests explicit use of shared memory for best performance.

```
__global__ void
forward_diff_kernel (const float *in, float *out, const unsigned len)
{
    // Declare block-shared memory.
    extern __shared__ float scratch[];

    // Get the output index assigned to this thread.
    const unsigned i = blockIdx.x * blockDim.x + threadIdx.x;

    // Load one element into shared memory & synchronize.
    if (i < len)
        scratch[threadIdx.x] = in[i];
    // Synchronize within the thread block to make writes visible.
    __syncthreads();

    // Compute the difference.  Threads on the end of a block
    // have to read from device memory.
    if (i < len - 1)
        // If the executing thread is at the rightmost edge of a
        // thread block, read from device memory, otherwise read from
        // shared memory.
        out[i] = threadIdx.x == blockDim.x - 1
               ? in[i + 1] - scratch[threadIdx.x];
               : scratch[threadIdx.x + 1] - scratch[threadIdx.x];
    }
}
```

Listing 1.3: A CUDA kernel to compute the forward difference of an array

visible. Third, the threads responsible for in-bound elements compute the forward difference. At this step, threads that are at the rightmost edge of a thread block need to read the forward value from device memory rather than shared memory, as it will not reside in shared memory. Finally, each in-bound thread writes its result to the output array. The kernel and the wrapper are shown in Listings A.1 and A.2. The wrapper code is nearly identical to the wrapper code for the previous example, except for the additional kernel parameter `smem_size` that specifies the amount of shared memory to dynamically allocate for the kernel launch.

We see more CUDA extensions to C++ in this example. The `__shared__` storage specifier in a kernel indicates that the variable resides in the block-shared memory on the GPU. Recall that this is fast memory available to all threads of a thread block. The `__syncthreads()` primitive implements a barrier synchronization within a thread block. Due to the memory access pattern of this example and because shared memory is shared

```
void
forward_diff_wrapper(const float *in, float *out, const unsigned len)
{
    // Allocate device memory for the arrays.
    float *in_d;
    float *out_d;

    const size_t in_size = len * sizeof(float);
    // The output is one element shorter than input.
    const size_t out_size = (len - 1) * sizeof(float);

    cudaMalloc((void **)&in_d, in_size);
    cudaMalloc((void **)&out_d, out_size);

    // Copy input array into device memory.
    cudaMemcpy(in_d, in, in_size, cudaMemcpyHostToDevice);

    // Set up and invoke the kernel.
    const dim3 block_dim(128, 1, 1);
    const dim3 grid_dim((unsigned)ceilf((float)len / 128f), 1, 1);
    // How much shared memory to allocate per block?
    const size_t smem_size = 128 * sizeof(float);

    forward_diff_kernel<<<grid_dim, block_dim, smem_size>>>
        (in_d, out_d, len);

    // Copy results back into CPU memory.
    cudaMemcpy(out, out_d, out_size, cudaMemcpyDeviceToHost);

    // Release the device memory.
    cudaFree(out_d);
    cudaFree(in_d);
}
```

Listing 1.4: A CUDA wrapper procedure to invoke the kernel of Listing A.1

only within a thread block and there is no way to synchronize among thread blocks, threads with `threadIdx.x` of 127 must perform two reads from device memory.

### 1.2.6 Coalesced Memory Access

When certain conditions hold, accesses of device memory can be *coalesced* so that 16 apparently independent memory accesses can be combined into one or two larger memory accesses. This can have a dramatic impact on overall performance.

In older NVIDIA GPUs, the following conditions are necessary in order for memory coalescing to occur: [3]

1. The 16 accesses are for either 32-, 64-, or 128-bit words;
2. All 16 words lie in the same memory segment;
3. Threads access the words in sequence;
4. Arrays are indexed using one of a few common indexing patterns.

In order to aid in memory coalescing, it is important that device allocations be properly aligned. Ensuring that the necessary and sufficient conditions for memory coalescing are met is a subtle matter, particularly for higher dimensional arrays.

### 1.2.7 Example: Array Maximum

The previous two examples have computed element-wise transformations of an array. An example that is *not* an element-wise transformation is determination of the maximum value of an array. To do this efficiently on a GPU using CUDA, we must use a parallel algorithm. Recall that CUDA imposes a two-level decomposition of the problem into a grid of thread blocks. Because of this constraint, we will perform a two-phase computation: have each thread block compute its local maximum, and then compute the maximum of the local maxima.

[3] With newer NVIDIA GPUs, the details of these conditions are different.

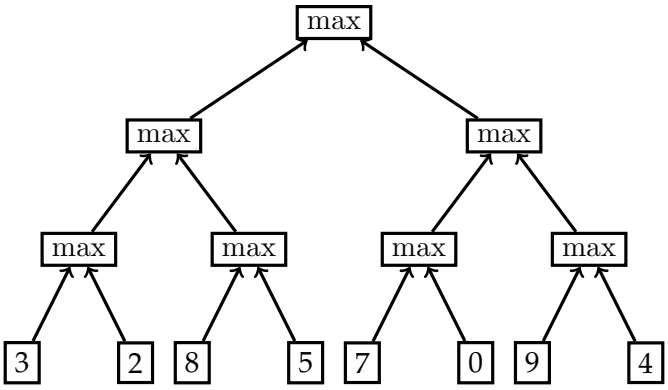


Figure 1.4: Logarithmic fan-in computation of the maximum element. A CUDA implementation of array maximum has each thread block compute its local maximum using a logarithmic fan-in.

Each thread block will compute its local maximum through a logarithmic fan-in using shared memory. For a thread block with $n$ elements, there will be $n - 1$ calls to `max` done in $\log_2 n$ iterations. This is shown schematically in Figure 1.4.

The kernel code for a naive implementation of the local maximum computation is given in Listing 1.5. It takes as arguments the input array `in` for which the maximum value is desired, an output array in which to write the results for the thread block-local maxima, and the length of the input array. It begins by declaring and loading the shared memory that will be used when performing the logarithmic fan-in. Once shared memory has been loaded, the fan-in is performed: in the first iteration, half the threads do work; in the second, a quarter do work; in the third, an eighth do work, and so on. Finally, the thread with the lowest index—the 'leader' thread—writes the local maximum for the block to the output array.

The second phase of the computation of the maximum is performed in the wrapper code, given in Listing 1.6. The code here is similar to the wrapper code for the forward difference operation in Listing A.2, with some additions: an intermediate array is required for the results from the computation of block-local maxima, and after invoking the kernel, the maximum of local maxima is computed by the CPU.

```
__global__ void
maximum_kernel(const float *in, float *out, const unsigned len)
{
    extern __shared__ float scratch[];

    // Each thread loads one element from device mem -> shared mem.
    const unsigned i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < len)
        scratch[threadIdx.x] = in[i];
    // Wait for loading to be complete.
    __syncthreads();

    // Determine block maximum using shared memory.
    for (unsigned s = 1; s < blockDim.x; s *= 2)
    {
        const unsigned idx = 2 * s * threadIdx.x;
        if (idx + s < blockDim.x && idx + s < len)
            scratch[idx] = max(scratch[idx], scratch[idx + s]);
        // Must synchronize after writing to shared memory to make
        // changes visible to other threads in the block.
        __syncthreads();
    }

    // Leader thread writes results for the block.
    if (threadIdx.x == 0)
        out[blockIdx.x] = scratch[0];
}
```

Listing 1.5: A CUDA kernel to compute the block-local maxima

```
float
maximum_wrapper (const float *in, const unsigned len)
{
    const dim3 block_dim (128, 1, 1);
    const dim3 grid_dim ((unsigned)ceilf((float)len / 128f), 1, 1);

    // The number of elements in the intermediate array.
    const unsigned intermediate_len = grid_dim.x;
    // The size, in bytes, of the intermediate array.
    const size_t intermediate_size = intermediate_len * sizeof(float);

    const size_t in_size = len * sizeof (float);

    // How much shared memory to allocate per block?
    const size_t smem_size = 128 * sizeof (float);

    // Allocate arrays in device memory.
    float *in_d;
    float *intermediate_d;

    cudaMalloc ((void **)&in_d, in_size);
    cudaMalloc ((void **)&intermediate_d, intermediate_size);

    // Load up the device input array.
    cudaMemcpy (in_d, in, in_size, cudaMemcpyHostToDevice);

    // Invoke the kernel to determine block maximums.
    maximum_gpu_kernel<<<grid_dim, block_dim, smem_size>>>
        (in_d, intermediate_d, len);

    // Compute the maximum of all the blocks on the CPU.
    float *intermediate = new float[intermediate_len];

    cudaMemcpy (intermediate, intermediate_d, intermediate_size,
                cudaMemcpyDeviceToHost);

    float result = intermediate[0];
    for (unsigned i = 1; i < intermediate_len; ++i)
        result = max(result, intermediate[i]);

    delete [] intermediate;

    cudaFree (intermediate_d);
    cudaFree (in_d);

    return result;
}
```

Listing 1.6: CUDA wrapper to compute the maximum element of an array

## 1.3 Toward Easier General-Purpose GPU Programming?

The implementation of the array maximum just given is already significantly more complicated than the previous examples due to the use of a non-trivial parallel algorithm. Unfortunately, it is also very bad code: a significantly longer and more complex implementation can be more than an order of magnitude faster [Harris, 2008].

CUDA code is low-level. Compared to programming in sequential C++ for the CPU, CUDA (*a*) has additional address spaces and data transfers to worry about; (*b*) does not automatically cache data, except with the latest GPUs; (*c*) involves more complicated array indexing; (*d*) requires non-trivial parallel algorithms and synchronization in many realistic applications; (*e*) involves much more boilerplate code; and (*f*) requires significant care and expertise in order to get good performance. Use of higher-dimensional arrays further complicates array indexing. It is easy to write CUDA code that is either incorrect or inefficient. It would be a great boon if application-level GPU programming could be made simpler.

# CHAPTER 2

# RELATED WORK

An obvious way to program GPUs at a higher level than CUDA is to create a new general-purpose programming language with GPU support. Examples of this approach include Brook, a C-like language that treats the GPU as a stream processor [Buck et al., 2004], and BSGP, which applies the bulk-synchronous programming model for GPU programming [Hou et al., 2008].

Creating a new general-purpose language is a huge undertaking. The design space for general-purpose programming languages is enormous. Furthermore, significant infrastructure is required: not only a compiler or interpreter need be created, but also the debugger, profiler, and numerous libraries that one expects from a practical general-purpose programming language. Existing languages have been extended with GPU programming support, which requires less implementation effort than creating a new language. CUDA itself takes this approach, being a dialect of C++. Other examples include PyCUDA, which allows CUDA programming from within Python and adds some higher-level support [Kloeckner et al., 2009]; Copperhead [Catanzaro et al., 2010] and Theano [Bergstra et al., 2010], systems for Python that can compile subsets of Python code to GPU code; a system to translate OpenMP-annotated code into GPU code [Lee et al., 2009b]; and Accelerator, which adds data-parallel constructs to .NET languages to allow GPU programming [Tarditi et al., 2006].

Several array libraries for GPU programming have also been created. The library ap-

proach has a low implementation cost and has the advantage of not requiring new programming languages or tools. However, in most programming languages, compile-time optimization at the library level is either not possible, limited, or awkward [Veldhuizen and Gannon, 1998, Veldhuizen, 1998], so the library approach is more suitable for cases where fast development time is valued above performance. An example is the Mars system, which implements a MapReduce framework in C++ for GPUs [He et al., 2008, Dean and Ghemawat, 2008].

In contrast to a general-purpose language designed to handle a wide variety of problems, a *domain-specific* language is designed for a specific, constrained problem domain. Examples are numerous and are discussed by both Bentley [1986] and van Deursen et al. [2000].

Often, domain-specific languages have been implemented from scratch and have their own syntax, semantics, and interpreters or compilers. Because they are special-purpose, they can provide a very high level of abstraction. Because they are implemented as languages rather than libraries, they have control over the optimizations applied and the code that is generated, rather than relying on an existing language's compiler to generate efficient code.

Although such languages are special-purpose, implementation effort may be high if they are written as stand-alone tools. Furthermore, as a stand-alone domain-specific language evolves, it is tempting to add general-purpose features, which can destroy the simplicity and elegance of a little language.

The temptation to add general-purpose features and implementation effort can both be reduced by *embedding* a domain-specific language within another language [Hudak, 1996, 1998]. Example uses include drawing and animation [Kamin and Hyatt, 1997, Elliott et al., 2003], hardware design [Bjesse et al., 1999], defining parsers [Hutton and Meijer, 1998, Niebler, 2007], and deriving high-performance numerical solvers for partial differential equations from mathematical descriptions [Dinesh et al., 2000].

Several domain-specific embedded languages for GPU programming have been created. An early example is Sh, a shader and stream programming language implemented in C++, making heavy use of the template mechanism and preprocessor [McCool et al.,

2002, 2004].

More recently and most similar to the work presented in this thesis are array languages embedded within Haskell. These array languages feature *collective operations*, such as element-wise transformations and prefix scans, as their fundamental array primitives. These collective operations are sufficient to express a wide variety of problems and have well-known parallel implementations [Blelloch, 1989, 1996]. Accelerate [Lee et al., 2009a] and Nikola [Mainland and Morrisett, 2010] are two examples. Accelerate provides an imperative array language that is dynamically compiled into CUDA code implementing the array program. Nikola provides an applicative array language that is similarly dynamically compiled into CUDA code.

# CHAPTER 3

# THE BARRACUDA LANGUAGE

Barracuda is a domain-specific language for array processing that emphasizes simplicity and efficient code generation. Barracuda is a purely functional language for defining array operations, embedded within the functional programming language Haskell [Peyton Jones et al., 2003].

Barracuda was designed to be used to generate GPU implementations of array operations, specifically for the following use case: the user (i.e., programmer) wants to accelerate parts of a C++ application by running certain array operations on a GPU. The user expresses the operations in Barracuda, then uses the Barracuda compiler to generate procedures that hide most of the details of working with a GPU.

We begin this chapter with an introduction in tutorial form. This tutorial assumes that the reader is not familiar with Haskell, and attempts to explain enough so that the Barracuda programs will be understandable. After the tutorial, the Barracuda language is described in detail, followed by a detailed discussion of the optimization and compilation processes. We conclude this chapter with a comparison of Barracuda to related work.

## 3.1 A Tutorial Introduction

**Getting Started** Barracuda is special-purpose and has no notion of input or output, so it is not possible to write Hello World. Instead, we'll write a simple example discussed earlier: element-wise computation of the sine of an array.

A program in Barracuda is in fact a Haskell program. We begin by importing the module that defines the Barracuda language

```
import Language.Barracuda
```

We then write our function to compute the sine of each element. We will be writing a Haskell function that uses the Barracuda constructs. The first line is its type signature

```
arraySine :: AExp Float -> AExp Float
```

This says that the value `arraySine` is a Haskell function that takes an `AExp Float` and returns an `AExp Float`. The type `AExp Float` is a type defined by Barracuda, representing a floating-point array that resides on the GPU. In other words, `arraySine` is a function that takes a floating-point array and returns a floating-point array. Next comes the function definition

```
arraySine x = amap sin x
```

`arraySine` takes an argument `x`, and returns the expressions `amap sin x`, which applies the `sin` function element-wise to `x`. `sin` is a function from the standard libraries in Haskell, and has been overloaded to work with Barracuda types. `amap` is a function defined in Barracuda analogous to the standard `map` function found in Haskell and many other languages that operates on lists, and is one of the fundamental constructs by which Barracuda makes use of the GPU. `amap` takes a unary scalar function as its first argument, and an appropriately-typed array as its second argument. It is compiled into code that applies the function to every element of the array in parallel.

Because Barracuda is used to generate CUDA code that the user will call from a larger C++ application, the user will likely want explicit control over the names given to the generated procedures and arguments. We next annotate the `arraySine` function with name information using the `Function` data type defined by Barracuda

```
arraySineFun :: Function
arraySineFun = Function arraySine "array_sine" ["x"] "result"
```

```
import Language.Barracuda

arraySine :: AExp Float -> AExp Float
arraySine x = amap sin x

arraySineFun :: Function
arraySineFun = Function arraySine "array_sine" ["x"] "result"

main :: IO ()
main = compile defaultCudaConfig files [arraySineFun]
  where files = ("array_sine_cuda.cuh", "array_sine_cuda.cu")
```

Listing 3.1: The complete Barracuda array sine program

This says that the value `arraySineFun` has type `Function`, and is equal to the right-hand side. The `Function` constructor takes four arguments: a Barracuda function, a name to give the generated procedure, a list of names for the input arguments, and a name for the output argument.

Finally, we write the entry point into the Haskell code

```
main :: IO ()
main = compile defaultCudaConfig files [arraySineFun]
  where files = ("array_sine_cuda.cuh", "array_sine_cuda.cu")
```

The entry point to a Haskell program is named `main` and has type `IO ()`. That is, it is an IO action that returns `()`, which is the Haskell analog of a `void` type in C++. We see in the definition of `main` the use of local definitions in Haskell using the `where` form: `files` is equal to the pair of file names given in its definition. One could equivalently replace the occurrence of `files` in `main` with its definition. This Haskell program simply compiles the array sine function into CUDA code using Barracuda's `compile` function with the `defaultCudaConfig` compiler configuration, generating a CUDA header file and a CUDA source file with the given names.

The complete example is shown in Listing 3.1. With Barracuda properly installed, assuming the example was saved to a file named `array_sine_cuda.hs`, the code could be compiled with

```
> runhaskell array_sine_cuda.hs
writing array_sine_cuda.cuh
writing array_sine_cuda.cu
```

The generated code is written to the indicated files. The user would then include the header (i.e., `array_sine_cuda.cuh`) in his or her application, which exposes a procedure with the signature

```
void
array_sine (const gpu_float_array &x, gpu_float_array_view result);
```

This procedure implements the array sine transformation on the GPU. We won't go into more detail about the generated code at this point, but this should give a taste of what programming with Barracuda is like.[1] In general, the Barracuda workflow is this:

1. Write Barracuda functions for the desired array operations.
2. Annotate the functions with naming information.
3. Use Barracuda's `compile` function to generate CUDA code.
4. Use the generated procedures from C++.

The details of the CUDA code are hidden from the user; the user only needs to be aware that the `gpu_*` types indicate values that reside on the GPU.

### 3.1.1 Mapping over Multiple Arrays

A more complicated array operation is SAXPY—single-precision alpha x plus y—from BLAS Level 1 [Lawson et al., 1979]. This operation takes a floating-point scalar $\alpha$ and two floating-point arrays $x$ and $y$ and computes the expression $\alpha * x + y$, which is a floating-point array.

A Barracuda implementation is shown in Listing 3.2. Like the previous example, we begin by importing the module for the Barracuda language. We then write the Barracuda version of SAXPY

```
saxpy :: SExp Float -> AExp Float -> AExp Float -> AExp Float
saxpy alpha x y = azipWith (+) (amap (* alpha) x) y
```

[1] For a complete example of a Barracuda program, the generated code, and a handwritten implementation, see Appendix A.

```
import Language.Barracuda

saxpy :: SExp Float -> AExp Float -> AExp Float -> AExp Float
saxpy alpha x y = azipWith (+) (amap (* alpha) x) y

saxpyFun :: Function
saxpyFun = Function saxpy "saxpy" ["alpha", "x", "y"] "result"

main :: IO ()
main = compile defaultCudaConfig files [saxpyFun]
  where files = ("saxpy.cuh", "saxpy.cu")
```

Listing 3.2: An implementation of SAXPY in Barracuda

We see some new things here. First, `saxpy` is a function that takes *three* inputs: the first of type `SExp Float` and the latter two of type `AExp Float`. `SExp Float` is the Barracuda type for floating-point scalars. `SExp` stands for 'scalar expression' and `AExp` stands for 'array expression', and both are parameterized by the element type.

Second, the definition of `saxpy` uses the `azipWith` function provided by Barracuda. This function takes three arguments: a binary scalar function and two appropriately-typed array arguments. `azipWith` is an extension of `amap` to two input arrays instead of a single input array. Similarly, it is compiled into code that executes the transformation on each array element in parallel.[2]

The third new thing we see is use of Haskell's section syntax. The first argument to `azipWith` is `(+)`. Putting parentheses around an infix function treats it as a prefix function. That is, `7 + 35` is equivalent to `(+) 7 35`. The first argument to `azipWith` is a binary function, so in this case we are supplying the addition function as the first argument.

The fourth new thing we see in this example is nested use of Barracuda's array primitives. The second argument of `azipWith` is `(amap (* alpha) x)`. Recall that `amap` takes two arguments, a unary scalar function and an array, and returns an array. In this case, we give the 'times-alpha' function as the first argument, and the array `x` as the second argument. This is another example of Haskell's section syntax, as well as partial function

[2]The name `azipWith` was chosen to mirror Haskell's `zipWith` function on lists. In dynamically typed languages, these are usually both called 'map'. However, it is difficult to ascribe a type to a variadic map function in most statically typed languages, which is the main reason why the `map` and `zipWith` functions have different names, rather than having a generic function that encompasses both.

application: `(* 7) 6` is equivalent to `6 * 7`, which is equivalent to `(6 *) 7`.

One might worry about the use of nested array operations like this, as in a naive implementation, temporary arrays would be introduced to compute each nested array subexpression. Wary of this prospect, one might instead write SAXPY in Barracuda as follows:

```
saxpy2 :: SExp Float -> AExp Float -> AExp Float -> AExp Float
saxpy2 alpha x y = azipWith f x y
  where f x' y' = alpha * x' + y'
```

This definition passes the local function `f` as the first argument to `azipWith`, which computes both the multiplication and the addition. However, this is unnecessary: the Barracuda compiler fuses nested use of these array primitives, and the code generated from compiling either `saxpy` or `saxpy2` is identical, and temporary arrays are allocated in neither.

### 3.1.2 Reducing Arrays

Computing the sum of an array, for example, is not an element-wise transformation. To express this operation, we need to make use of a second array primitive that Barracuda provides: array reduction. The Barracuda function `areduce` takes three arguments: an associative binary function, an initial scalar value, and an array. Given the associative binary function $\oplus$, the initial scalar value $i$, and the array $a = [a_0, a_1, a_2 \ldots a_n]$, `areduce` $\oplus\, i\, a$ computes the scalar value

$$i \oplus a_0 \oplus a_1 \oplus a_2 \oplus \ldots \oplus a_n.$$

Reductions can be efficiently computed in parallel using a tree reduction. Summation is expressed as an array reduction with the addition function and the identity for addition:

```
asum :: AExp Float -> SExp Float
asum x = areduce (+) 0 x
```

The function takes a floating-point array and returns a floating-point scalar. We see one more new thing here: scalar literals. Number literals can be overloaded in Haskell, and here the value 0 represents a constant value of type `SExp Float`.

```
import Language.Barracuda

rmse :: AExp Float -> AExp Float -> SExp Float
rmse x y = sqrt (sumDiff / len)
  where
    len = intToFloat (alength x)
    sumDiff = areduce (+) 0 (amap (^2) (azipWith (-) x y))

rmseFun :: Function
rmseFun = Function rmse "rmse" ["input1", "input2"] "result"

main :: IO ()
main = compile defaultCudaConfig files [rmseFun]
  where files = ("rmse.cuh", "rmse.cu")
```

Listing 3.3: An implementation of RMSE in Barracuda

The array reduction primitive is useful for a variety of operations, e.g., summation, maximum, or logical conjunction. This array primitive is compiled by the Barracuda compiler into the efficient, two-phase logarithmic fan-in reduction for the GPU discussed in Chapter 1.

### 3.1.3 Putting It All Together

We now show a more complicated example that uses the constructs introduced so far. Root mean squared error is a commonly used measure of error, defined as

$$\mathrm{RMSE}(\theta_1, \theta_2) = \sqrt{\frac{\sum_{i=1}^{n}(x_{1,i} - x_{2,i})^2}{n}}.$$

A Barracuda implementation is given in Listing 3.3. This example uses `amap`, `azipWith`, `areduce`, as well as several new functions. The Barracuda function `alength` gives the length of an array. `intToFloat` casts a value of type `SExp Int` to a value of type `SExp Float`; Both Haskell and Barracuda are strongly and statically typed, and neither performs implicit type conversions.

Again, one might be worried about the nested use of `azipWith` within `amap`, and the nested use of that within `areduce`. Again, such worry would be misplaced, because the Barracuda compiler fuses these nested array operations into a complicated array reduction. In the generated code, a CUDA procedure with the following signature is exported

```
void rmse (const gpu_float_array &input1,
           const gpu_float_array &input2,
           float &result);
```

We will not look closer into the generated code at this point, more than observing that the Barracuda function was compiled into more than 150 source lines of CUDA code.

## 3.2 Language Definition

Barracuda is an embedded domain-specific language for array processing, implemented in Haskell, and designed for generating efficient CUDA code to be called from a larger application. Like much of the related work, Barracuda's programming model is based on collective array operations that can be straightforwardly compiled to a GPU. In the particular case of Barracuda, these primitive operations are *map*, *reduce*, and *slice*. In this section we describe the interface exposed to a programmer using Barracuda.

Barracuda is purely functional; it has no notion of assignment or other side effects. A Barracuda program is a Haskell function operating on Barracuda expressions. Barracuda supports scalars, arrays, and matrices. `SExp`, `AExp`, and `MExp` are the types for scalar, array, and matrix expressions.

```
data SExp a
data AExp a
data MExp a
```

These data types are abstract, i.e., their constructors are not exported.

Each of the three expression type constructors is parameterized by its element type. Only a limited set of types can be used as element types; such types are members of a special type class.

```
class (Eq a, Ord a, Show a, Typeable a) => Scalar a where
  -- definition omitted

instance Scalar Int
instance Scalar Float
instance Scalar Double
instance Scalar Bool
```

This set of allowed element types is very limited, but it maps naturally onto the set of types supported in GPU code in CUDA, which is Barracuda's primary target language.

Notice that the type of Barracuda expressions is determined by both expression rank and element type. Although we omit formal typing rules in this thesis, Barracuda is strongly statically typed, and trying to express a type-incorrect Barracuda program results in a Haskell type error.

### 3.2.1 Scalar Expressions

Numeric scalar expressions in Barracuda are instances of the appropriate Haskell type classes:

```
instance Num (SExp Int)
instance Num (SExp Float)
instance Num (SExp Double)

instance Fractional (SExp Float)
instance Fractional (SExp Double)

instance Floating (SExp Float)
instance Floating (SExp Double)
```

`Num`, `Fractional`, and `Floating` are standard Haskell type classes for overloading many common numeric operations. Making the Barracuda scalar expressions instances of these classes allows one to write scalar literals and use arithmetic operations and many floating point operations on scalars. Integer division and modulus, and a cast from integer to float are provided specially, rather than implementing instances for the built-in Haskell type class `Integral` to avoid having to define incomplete instances.

```
idiv, imod :: SExp Int -> SExp Int -> SExp Int
intToFloat :: SExp Int -> SExp Float
```

Equality tests are provided for Barracuda scalar expressions through a new typeclass.

```
class (Scalar a) => ExprEq a where
  (==*) :: SExp a -> SExp a -> SExp Bool
  (/=*) :: SExp a -> SExp a -> SExp Bool
```

```
infix 4 ==*, /=*

instance ExprEq Int
instance ExprEq Float
instance ExprEq Double
instance ExprEq Bool
```

We cannot use Haskell's built-in `Eq` typeclass for equality testing in Barracuda, since its methods have the wrong type.

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

Specifically, the methods return a regular `Bool`, and not `SExp Bool`. For built-in infix Haskell functions that are not easily overloaded, Barracuda uses the names appended with an asterisk.

Comparison operators are also provided for the numeric scalar expressions.

```
class (Scalar a) => ExprOrd a where
  (>*)  :: SExp a -> SExp a -> SExp Bool
  (>=*) :: SExp a -> SExp a -> SExp Bool
  (<*)  :: SExp a -> SExp a -> SExp Bool
  (<=*) :: SExp a -> SExp a -> SExp Bool

infix 4 >*, >=*, <*, <=*

instance ExprOrd Int
instance ExprOrd Float
instance ExprOrd Double

maxVal, minVal :: (ExprOrd a) => SExp a -> SExp a -> SExp a
```

Similarly to the equality testing functions, the methods of Haskell's built-in `Ord` class have the wrong type, so Barracuda provides a new type class. The `maxVal` and `minVal` functions return the maximum or the minimum of two scalar expressions, respectively.

Boolean literals are written `true` and `false`. Boolean expressions can be combined using conjunction, disjunction, and negation.[3]

[3]The terrible name 'snot'—short for 'scalar not'—was chosen instead of simply 'not' so that the Prelude function would not be shadowed.

```
true, false  :: SExp Bool
(&&*), (||*) :: SExp Bool -> SExp Bool -> SExp Bool
snot         :: SExp Bool -> SExp Bool
```

A Boolean expression can be used to write a scalar conditional:

```
scond
  :: (Scalar a)
  => SExp Bool                          -- predicate
  -> SExp a                             -- value when true
  -> SExp a                             -- value when false
  -> SExp a
```

This function acts like an if-then-else in Haskell, or the ternary operator in C.

Finally, explicit sharing can be expressed using the `slet` function:

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

This is the Barracuda analogue of a let-expression in Haskell, allowing one to indicate that the value of a named subexpression should be computed once and reused.

### 3.2.2 Array Expressions

There are no terminal array expressions in Barracuda. Hence, the only place a term of type `AExp a` can appear is in the definition of a Haskell function that takes an argument of type `AExp a`.

The length of an array can be determined using `alength`:

```
alength :: AExp a -> SExp Int
```

An element-wise transformation of one, two, or three arrays can be expressed using the array mapping functions.

```
amap
  :: (Scalar a, Scalar b)
  => (SExp a -> SExp b)
```

```
  -> AExp a
  -> AExp b

azipWith
  :: (Scalar a, Scalar b, Scalar c)
  => (SExp a -> SExp b -> SExp c)
  -> AExp a
  -> AExp b
  -> AExp c

azipWith3
  :: (Scalar a, Scalar b, Scalar c, Scalar d)
  => (SExp a -> SExp b -> SExp c -> SExp d)
  -> AExp a
  -> AExp b
  -> AExp c
  -> AExp d
```

An array can be reduced to a scalar value using the array reduction function `areduce`, which takes a reducing function, an initial value, and an array expression as arguments:

```
areduce
  :: (Scalar a)
  => (SExp a -> SExp a -> SExp a)    -- reducing function
  -> SExp a                          -- initial value
  -> AExp a                          -- array to reduce
  -> SExp a
```

The value of an array reduction expression is only well-defined if the reducing function is associative. This is a requirement that cannot be captured in Haskell's type system.[4]

Finally, a slice of an array can be extracted.

```
type SliceInfo = (SExp Int, SExp Int, SExp Int)

aslice :: (Scalar a) => AExp a -> SliceInfo -> AExp a
```

Slice information is represented as a triple of integer expressions, denoting the start index, stop index, and stride. In a value of type `SliceInfo`, the stride must be non-zero and the stop index greater than the start. Furthermore, both the start and stop indexes must be less

[4]Many floating-point operations are neither associative nor commutative. However, given that manipulating floating-point arrays is the primary use of GPUs, perhaps the non-associativity and non-commutativity of floating-point operations combined with parallel algorithms is not so much a problem in practice.

than the extent of array dimension that the slice information corresponds to. The result of an array slice expression is an array consisting of the elements from the original array at indexes $[i_{start}, i_{start+stride} \ldots i_{stop}]$.

### 3.2.3 Matrix Expressions

Like arrays, there are no terminal matrix expressions in Barracuda, and the only place a term of type `MExp a` can appear is in the definition of a Haskell function that takes an argument of type `MExp a`.

The number of rows and number of columns can be retrieved using the `mrows` and `mcols` functions, both of type `(Scalar a) => MExp a -> SExp Int`.

An element-wise transformation of one or two matrices can be expressed using the matrix mapping functions.

```
mmap
  :: (Scalar a, Scalar b)
  => (SExp a -> SExp b)
  -> MExp a
  -> MExp b

mzipWith
  :: (Scalar a, Scalar b, Scalar c)
  => (SExp a -> SExp b -> SExp c)
  -> MExp a
  -> MExp b
  -> MExp c
```

A matrix can be reduced to a scalar value using the matrix reduction function `mreduce`, which takes a reducing function, an initial value, and a matrix expression as arguments:

```
mreduce
  :: (Scalar a)
  => (SExp a -> SExp a -> SExp a)    -- reducing function
  -> SExp a                          -- initial value
  -> MExp a                          -- matrix to reduce
  -> SExp a
```

As in the case with array reduction, the value of the matrix reduction expression is only well-defined if the reducing function is associative.

Finally, a sub-matrix can be extracted using `mslice`:

```
mslice
  :: (Scalar a)
  => MExp a                          -- matrix to slice
  -> SliceInfo                       -- row slice
  -> SliceInfo                       -- column slice
  -> MExp a
```

Slicing a matrix requires slice information for both the rows and columns, in that order.

## 3.3 Representation of Barracuda Programs

The functions listed in the previous section are *smart constructors* that build up abstract syntax trees representing the Barracuda functions. Following Elliott et al. [2003], these smart constructors perform optimizations from the bottom up, including constant folding and certain algebraic simplifications.

The abstract syntax trees in Barracuda comprise a family of data types, `SExp a` for scalar expressions of type `a`, `AExp a` for array expressions with element type `a`, and `MExp a` for matrix expressions with element type `a`. These types are abstract and not exposed to Barracuda programmers. The type constructors `SExp`, `AExp`, and `MExp` are defined as GADTs, which allows the type system of the object language to be embedded within the type system of the metalanguage, making ill-typed object language expressions impossible to construct [Xi et al., 2003, Rhiger, 2003].

Definitions of these type constructors are shown in Listings 3.4, 3.5, and 3.6. Most of these have analogous smart constructors, described in the previous section, but some, such as `SVar`, `AVar`, and `MVar` are only used internally and have no direct smart constructor analog.

Barracuda functions are represented using higher-order abstract syntax [Pfenning and Elliott, 1988]. This permits the use of functions in the metalanguage (i.e., Haskell in this case) to represent functions in the object language, 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.

```
data SExp :: * -> * where
  -- scalar terms: constants and variables
  SCFloat :: Float -> SExp Float
  SCInt   :: Int   -> SExp Int
  SCBool  :: Bool  -> SExp Bool
  SVar    :: (Scalar a) => String -> SExp a

  -- let-expressions
  SLet :: (Scalar a, Scalar b)
       => SExp a                              -- expr. to bind
       -> (SExp a -> SExp b)                  -- body
       -> SExp b

  -- primitive scalar operations (e.g. addition) omitted
    ...

  -- array / matrix reductions
  AReduce :: (Scalar a)
          => (SExp a -> SExp a -> SExp a)     -- reducer
          -> SExp a                           -- initial value
          -> AExp a                           -- array to reduce
          -> SExp a
  MReduce :: (Scalar a)
          => (SExp a -> SExp a -> SExp a)     -- reducer
          -> SExp a                           -- initial value
          -> MExp a                           -- matrix to reduce
          -> SExp a

  -- array / matrix indexing
  AIdx    :: (Scalar a)
          => String                           -- array variable
          -> SExp Int                         -- index value
          -> SExp a
  MIdx    :: (Scalar a)
          => String                           -- matrix variable
          -> SExp Int                         -- row index
          -> SExp Int                         -- column index
          -> SExp a

  -- array / matrix accessors
  ALen    :: (Scalar a) => AExp a -> SExp Int
  MRows   :: (Scalar a) => MExp a -> SExp Int
  MCols   :: (Scalar a) => MExp a -> SExp Int
```

Listing 3.4: Representation of scalar expressions in Barracuda.

```
data ASliceInfo = ...  -- definition omitted

data AExp :: * -> * where
  -- array variables
  AVar :: (Scalar a) => String -> AExp a

  -- array slices
  ASliceVar :: (Scalar a) => ASliceInfo -> String -> AExp a

  -- array map expressions
  AMap   :: (Scalar a, Scalar b)
         => (SExp a -> SExp b)
         -> AExp a
         -> AExp b
  AMap2  :: (Scalar a, Scalar b, Scalar c)
         => (SExp a -> SExp b -> SExp c)
         -> AExp a
         -> AExp b
         -> AExp c
  AMap3  :: ...  -- definition omitted
```

Listing 3.5: Representation of array expressions in Barracuda.

```
data MSliceInfo = ...  -- definition omitted

data MExp :: * -> * where
  -- matrix variables
  MVar :: (Scalar a) => String -> MExp a

  -- matrix slices
  MSliceVar :: (Scalar a) => MSliceInfo -> String -> MExp a

  -- matrix map expressions
  MMap   :: (Scalar a, Scalar b)
         => (SExp a -> SExp b)
         -> MExp a
         -> MExp b
  MMap2  :: (Scalar a, Scalar b, Scalar c)
         => (SExp a -> SExp b -> SExp c)
         -> MExp a
         -> MExp b
         -> MExp c
```

Listing 3.6: Representation of matrix expressions in Barracuda.

## 3.4 Compilation Details

### 3.4.1 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 3.4.8 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 elementwise. 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 the 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.

### 3.4.2 Specifying Names for the Generated Code

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 mentioned in the tutorial section of this chapter:

```
data Function :: * where
  Function
    :: (Instantiable a)
    => a                          -- a Barracuda function
    -> Id                         -- the name for the function
    -> [Id]                       -- the names for inputs
    -> Id                         -- the name for the output
    -> Function
```

`Function` is a GADT whose sole constructor takes a Barracuda function (i.e., something that is `Instantiable`, a type class for things that can be compiled, described shortly) and wraps it up with name information that is used when compiling.

### 3.4.3 The `compile` Function

Barracuda's `compile` function is the entry point into the compiler. It has the following type:

```
compile
  :: CompileConfig                -- compiler options
  -> (FilePath, FilePath)         -- output filenames
  -> [Function]                   -- annotated functions
  -> IO ()
```

It takes configuration information (e.g., optimization settings), a pair of file names, a list of name-annotated Barracuda functions, compiling the list of functions and writing the results to a header file and source file specified by the file names.

### 3.4.4 Runtime Representation

Barracuda functions are compiled into C++ procedures that invoke generated CUDA kernels. The goal with Barracuda is to generate procedures to be called from a C++ application. Barracuda defines C++ types corresponding to each of the Barracuda array and matrix types.

This mapping is fairly straightforward. For scalar expressions, `SExp Int` in Barracuda maps to `int` in C++; `SExp Float` maps to `float` in C++; and `SExp Bool` maps to `bool` in C++. For array and matrix expressions, the C++ component of Barracuda defines the template classes:

```
template <class T>
class gpu_array;

template <class T>
class gpu_matrix;
```

Barracuda expressions of type `AExp Int` are mapped to `gpu_array<int>`; `AExp Float` is mapped to `gpu_array<float>`; and `AExp Bool` is mapped to `gpu_array<bool>`. Matrix expressions are mapped similarly.

A Barracuda function is either a Haskell value of type `(Scalar a) => SExp a`, `(Scalar a) => AExp a`, `(Scalar a) => MExp a`, or a Haskell function that takes one of these types as an argument and returns a Barracuda function. This recursive definition is encoded by the `Instantiable` type class:

```
class Instantiable a where
  ...  -- methods omitted

instance (Scalar a) => Instantiable (SExp a)
instance (Scalar a) => Instantiable (AExp a)
instance (Scalar a) => Instantiable (MExp a)

instance (Scalar a, Instantiable r) =>
    Instantiable (SExp a -> r)
instance (Scalar a, Instantiable r) =>
    Instantiable (AExp a -> r)
instance (Scalar a, Instantiable r) =>
    Instantiable (MExp a -> r)
```

For example, Haskell functions of type `SExp Int -> SExp Int` or `MExp Float -> MExp Float -> MExp Float` are Barracuda functions, but Haskell functions of type `Int -> SExp Int` or `SExp Float -> AExp Float -> Float` are not.

In Barracuda-generated code, outputs are passed by reference to the generated procedures. Inputs, in contrast, are passed by constant reference. This convention eliminates the need to do allocations for output variables within Barracuda-generated procedures at the expense of placing the responsibility for memory management of output arguments on the user of such code.

### 3.4.5 Closure Conversion

A function in Barracuda closes over its lexical environment. This presents a challenge for compilation, as the target (i.e., CUDA) has no notion of closures. For instance, consider a Barracuda function that adds a given scalar value to each element of an array:

```
addValue :: SExp Float -> AExp Float -> AExp Float
addValue x xs = amap (x +) xs
```

The mapping function `(x +)` contains the closed-over variable `x`. Lambda lifting [Johnsson, 1985], or closure conversion, is used to transform the mapping function into a function with no closed-over variables. While in the general case lambda lifting may need to be run to fixed point, in Barracuda, only a single iteration is needed.

### 3.4.6 Array Fusion

The array primitives in Barracuda can be freely nested. For instance, in the root mean square code, `azipWith` (array map on two arrays) occurs within `amap`, which occurs within `asum` (an array reduction):

```
rmse :: AExp Float -> AExp Float -> SExp Float
rmse x y = sqrt (sumDiff / len)
  where
    len = intToFloat (alength x)
    sumDiff = asum (amap (^2) (azipWith (-) x y))
```

Naive compilation of this expression would generate code that evaluates from the most nested outward, using two temporary arrays 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 an array with some constants:

```
mulAdd :: AExp Float -> AExp Float
mulAdd xs = amap (+ 1) (amap (* 2) xs)
```

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

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].

### 3.4.7 Automatic Use of Shared Memory Cache

Recall from Chapter 1 that graphics processors possess a limited on-chip shared memory space that can be accessed much more quickly than device memory. When an array operation at each element is computed using that element's value and the value of neighboring elements, i.e., the operation is a *stencil computation*, large performance gains can often be realized by generating CUDA code that uses the GPU's shared memory cache rather than accessing device memory repeatedly. The forward difference function is a simple example:

```
forwardDiff :: AExp Float -> AExp Float
forwardDiff x = azipWith (-) x'' x'
```

```
  where x'  = aslice x (0, alength x - 1, 1)
        x'' = aslice x (1, alength x, 1)
```

The forward difference is computed by slicing the input array twice, with the two slices offset by one element, and performing an element-wise subtraction on the two slices. In this example, the value of each element of the output is computed using two adjacent values of the input array.

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. It does this as follows:

1. Inspect the function being compiled, collecting all uses of array variables that are used in two or more *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, and if the size of the overlap falls within a certain range (determined by the CUDA thread block dimensions used in the generated code).

Consider the forward difference code given above. 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 difference slice contexts in this function (named by `xs'` and `xs''`). Furthermore, these contexts are statically known to overlap, as long as `xs` contains more than two elements. In this case, 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 beginning or end of a thread block, or at the beginning or end of the array.

### 3.4.8 Nested Data Parallelism

The array primitives in Barracuda are compositional, 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 *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.

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

```
addSum :: AExp Float -> AExp Float -> AExp Float
addSum arr1 arr2 = amap f arr2
  where f y = asum arr1 + y
```

Here, `asum`, an array reduction, is used within the mapping function given to `amap`. In this case, the subexpression containing the nested array operation, `asum 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 :: AExp Float -> AExp Float -> AExp Float
ndp arr1 arr2 = amap f arr2
  where f v = asum (amap (\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 such 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 be as 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.

# CHAPTER 4

# EVALUATION

Barracuda allows declarative, type-safe programming of graphics processors at a much higher level of abstraction than CUDA for a subset of array-based operations. It hides the details of CUDA programming (except for the fact that arrays can reside in main memory or in device memory). A Barracuda programmer need not worry about block and grid dimensions, making use of shared memory, or array indexing within kernel code.

Because the primary motivation of using graphics processors for non-graphics uses is performance, it is important that programs written in Barracuda are compiled into efficient GPU code. To test this, the performance of Barracuda-generated GPU code was measured in several benchmarks. The Barracuda-generated code for these benchmarks is competitive with handwritten CUDA code, running no more than 50 percent slower, and in two cases running faster. Benchmarks were also used to measure the impact of array fusion and shared memory optimization, both of which can affect performance by large factors. The remainder of this chapter contains these benchmark results and further discussion.

## 4.1 Experimental Methodology

I ran several commonly used benchmarks, such as operations from the BLAS library [Lawson et al., 1979] and Black-Scholes option pricing, as well as several stencil operations to demonstrate the basic viability of Barracuda and to show the performance impact of array fusion and shared memory optimization. In most of these experiments, code generated

by Barracuda was compared with handwritten CUDA code. 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 8.10 and CUDA 2.3.

In all the benchmarks, only the computation time is counted, not the time to copy data between main memory and device memory. This is justified because the copying costs will be the same for handwritten and Barracuda implementations. The purpose of these benchmarks is to compare the performance of generated code and of other implementations and to show the impact of different optimizations; the overhead incurred by copying is not germane.

These benchmarks ran within microseconds even for very large arrays. This complicated the 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. By increasing the measurement time for the benchmarks, the average execution time per run could be easily computed.

The benchmarks that work with one-dimensional arrays were run with arrays of power-of-two sizes. The benchmarks that work with two-dimensional arrays were run with square matrices with power-of-two dimensions. This was done for simplicity and also because power-of-two arrays and matrices satisfy the memory alignment properties necessary for good CUDA performance. Note, however, that the Barracuda-generated code is not restricted to run with power-of-two arrays or square matrices, but is general.

### 4.1.1 Standard Benchmarks

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, 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 operations chosen were SAXPY and SDOT. SAXPY takes a single-precision float $\alpha$ and two single-precision float arrays $X$ and $Y$ and stores the array result $\alpha * X + Y$ in $Y$, and is expressed using a map operation in Barracuda. SDOT computes the dot product of two single-precision floating-point arrays, and is expressed using a map and reduction in Barracuda. Due to array fusion, the entire operation is compiled into a single array reduction by the Barracuda compiler, and so this benchmark demonstrates the efficiency of array reduction.

Results of these benchmarks are shown in Figure 4.1. For the SAXPY benchmark, we see that the Barracuda version runs faster than the cuBLAS version for all tested array sizes, which is a surprising result. There are two likely causes: (1) Barracuda and cuBLAS use different thread block dimensions, and (2) the Barracuda-generated code has each thread compute the value of a single output element, whereas the cuBLAS implementation has each thread compute the value of multiple output elements.[1]

We see in the SDOT results that Barracuda performs slightly faster than cuBLAS in all but one case; it is difficult to say why, especially without access to the source code for the cuBLAS implementation. The performance profile for this benchmark looks strange, particularly for array sizes between $2^6$ and $2^{16}$. This benchmark was re-run several times, and the performance profile consistently appears this way.

A Barracuda version of Black-Scholes option pricing was compared to the implementation found in the CUDA 2.3 SDK examples. Black-Scholes option pricing is another example of an element-wise array computation, but with a much more complex and expensive transformation 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 arrays are large enough to warrant running on a GPU, the performance of the Barracuda version is within ten percent of the handwritten version from the CUDA SDK.

The Black-Scholes benchmark points out a deficiency in Barracuda. The handwritten

[1]The source code for the cuBLAS implementation is not available, but these facts about what it is doing are able to be determined using CUDA's profiling facilities.

```
rmse :: AExp Float -> AExp Float -> SExp Float
rmse x y = sqrt (sumDiff / len)
  where
    len = intToFloat (alength x)
    sumDiff = asum (amap (^2) (azipWith (-) x y))
```

Listing 4.1: Root mean square error in Barracuda

version can evaluate both call and put option prices within a single pass, and work can be shared between the two evaluations. In other words, two output arrays are returned. However, only a single result can be returned by Barracuda programs, and so one would have to execute two different Barracuda-generated procedures to evaluate both call and put option prices. This would require two passes over the input arrays and would eliminate the possibility of sharing computation between the two evaluations. Barracuda could be made more generally usable by allowing functions to return multiple results.

### 4.1.2 Root Mean Square Error

To test the impact of array fusion, a benchmark from the RMSE program of Listing 4.1 was created, both with and without array fusion.[2] The results are shown in Figure 4.2. The difference is most dramatic when the input arrays are between $2^{10}$ and $2^{16}$ elements, and tapers off when the arrays are larger. This unusual performance curve shows that for medium-size arrays especially, the impact of array fusion is significant.

### 4.1.3 Shared Memory Optimization Benchmarks

To test the impact of shared memory optimization, I ran three stencil 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 out arrays or matrices that are larger than the inputs.

**Forward difference** The forward difference function given in Chapter 3 and again in Appendix A was tested with and without shared memory. We see in the results that once

[2]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.

```
wmAvg
  :: [Float]          -- weights
  -> AExp Float       -- input array
  -> AExp Float
wmAvg ws xs = foldr1 (azipWith (+)) slices
  where
    slices :: [AExp Float]
    slices = zipWith weight [0..] ws

    weight :: SExp Int -> Float -> AExp Float
    weight i w = amap (* float w) slice
      where
        slice = aslice xs (i, sliceLen + i, 1)
        sliceLen = alength xs - int (length ws)
```

Listing 4.2: Weighted moving average in Barracuda

the arrays 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 array (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 code from Listing 4.2 was given the weights for Spencer's 15-point moving average. When the arrays become large, shared memory optimization decreases runtime by a factor of eight.

**Jacobi iterative solver** Solving a two-dimensional discretization of Poisson's equation is a more complicated demonstration of a stencil computation than forward difference. The two Barracuda implementations were also compared to two handwritten implementations. We see from the results that when the matrices become large enough, use of shared memory improves performance by roughly a factor of two. The Barracuda versions perform worse than the handwritten versions, but are no more than forty percent slower. This is likely because the array indexing code is slightly more complicated in the Barracuda

versions compared to the handwritten versions; more care in code generation for two-dimensional array operations might eliminate this performance difference.

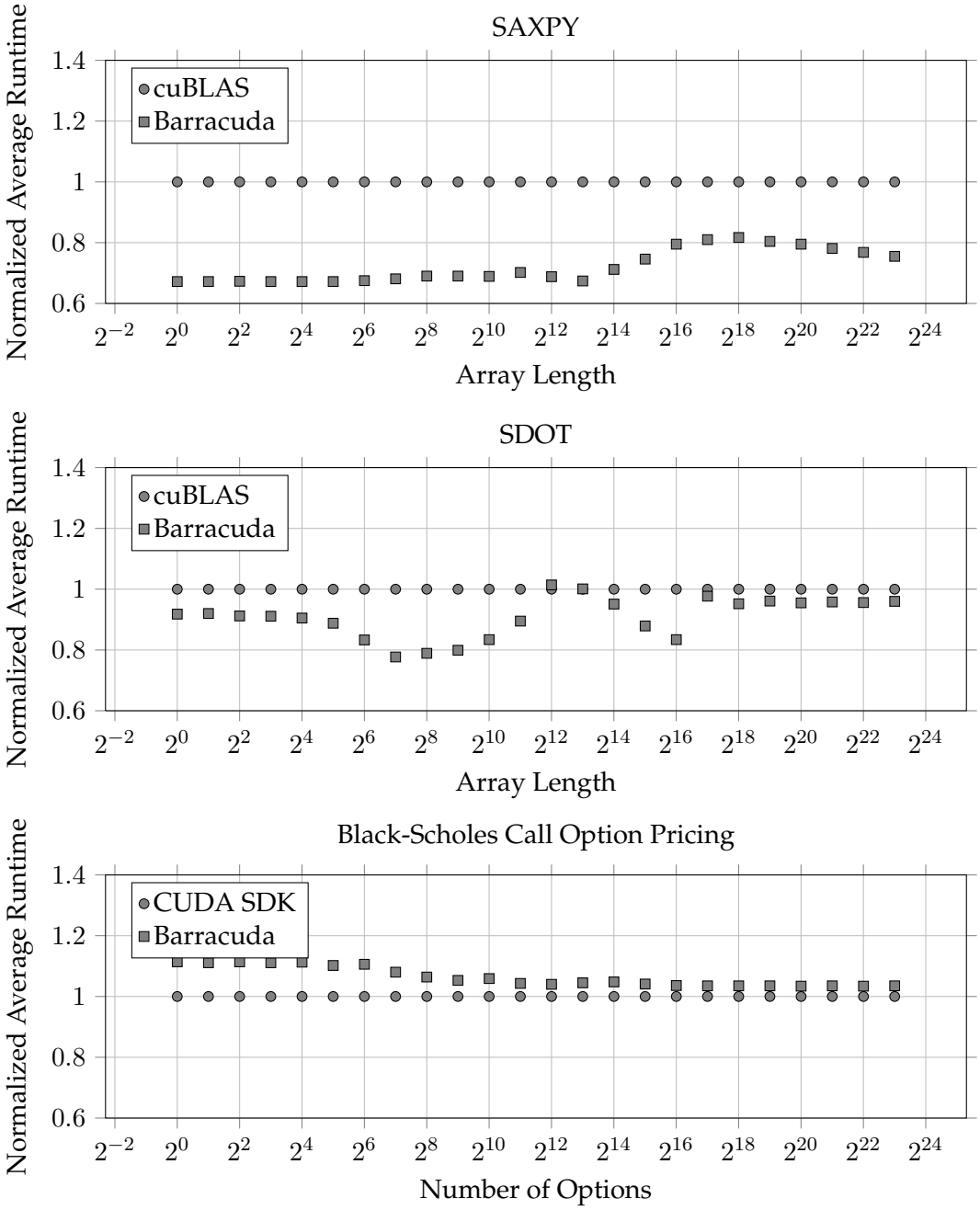


Figure 4.1: Performance results from semi-standard benchmarks. These benchmarks demonstrate the basic viability of an array programming system.

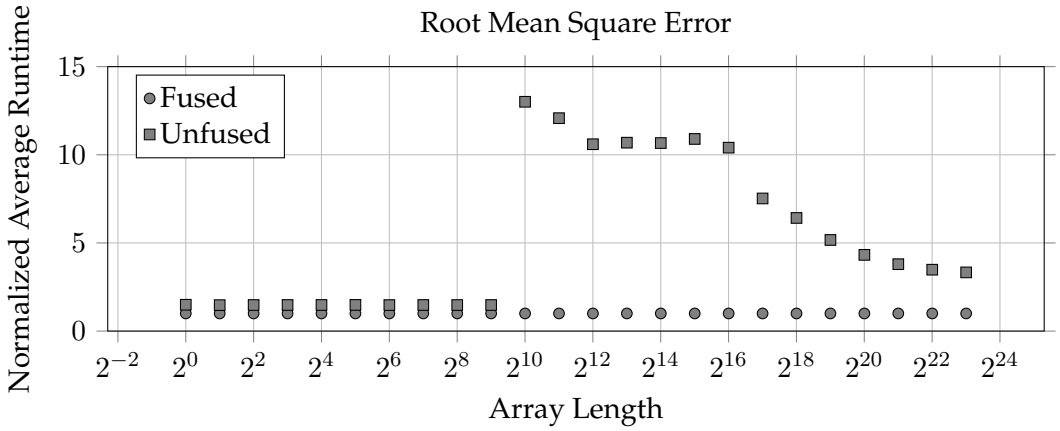


Figure 4.2: Benchmark results for RMSE

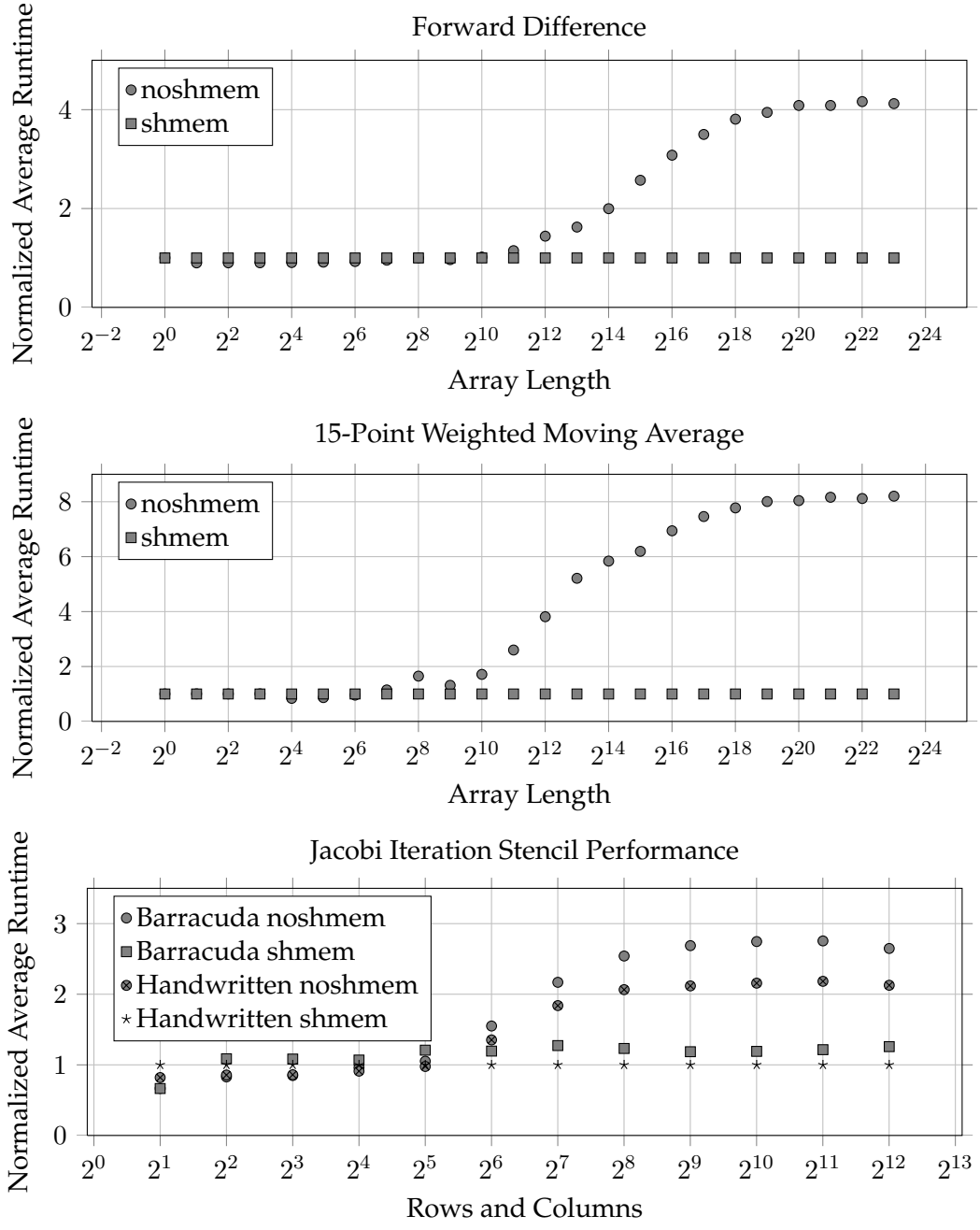


Figure 4.3: Shared memory benchmark results

# CHAPTER 5

# CONCLUSION

We have seen the difficulty of programming graphics processors correctly and efficiently using CUDA. We have seen that a higher-level approach—namely, using an array-based domain-specific language—can hide most of the complexity yet can still have performance competitive with that of handwritten code.

**Embedded Language Challenges** Several challenges with embedded languages were encountered during the numerous design iterations of Barracuda. First, the metalanguage constrains the possible concrete syntax of an object language. With Haskell as a metalanguage these constraints are bearable, as Haskell supports definition of new infix operators and overloading through type classes. However, one wishes for a more syntactically liberal metalanguage: Haskell does not allow overloading of several built-in constructs (e.g. conditional expressions, let-expressions, and lambda expressions), nor does it allow definition of mixfix functions.

Second, it is difficult to enforce certain static semantics of an object language statically in the metalanguage. Given a typed metalanguage with a sufficiently powerful type system, it is possible to embed the typing discipline of the object language into the type system of the metalanguage, causing object language type errors to be type errors in the metalanguage. (This is an issue that has been examined formally, e.g., by Rhiger [2003].) In the case of this thesis, Haskell 98 with the addition of GADTs has a type system powerful enough to embed Barracuda's typing discipline.

The difficulty lies in static semantics not implied by the typing discipline of the object language. For example, it would be desirable to statically guarantee against expression of troublesome nested data parallel functions in Barracuda, yet not restrict the language in other ways. Several attempts were made to encode this property into the Haskell types for Barracuda functions, but all restricted the language in other undesirable ways. While a metalanguage with a more expressive type system (e.g., full dependent types) could allow embedding of more complicated static semantics, it would be a great boon to be able to specify static semantics of an object language in a more explicit way than through the type system of the metalanguage.

Third, it would be helpful for usability to be able to specify the error messages to be reported when the static semantics of an object language are violated. If an object language's static semantics are enforced through the type system of the metalanguage, object language-level errors are reported as type errors, which obscures the higher-level cause of the error.

**Future Work** Barracuda is a very 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 be further improved by allowing multiple results to be returned from functions, i.e., by adding product types to the language. Furthermore, the language could be 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 both Accelerate [Lee et al., 2009a] and Nikola [Mainland and Morrisett, 2010] allow.

There are several ways in which the generated code could be improved. For instance,

elimination of unnecessary array bounds checking and specialization of kernels for array arguments of given dimensions could both improve performance.

Finally, although Barracuda is compiled into CUDA code, a more useful array-centric language might have multiple targets, e.g., CPUs, GPUs, the Cell processor, and other high-performance architectures.

# LIST OF REFERENCES


Jon Bentley. Programming pearls: little languages. *Communications of the ACM*, 29(8):711–721, 1986. ISSN 0001-0782.

James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010.

Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in Haskell. *ACM SIGPLAN Notices*, 34(1):174–184, January 1999. ISSN 0362-1340.

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

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

Guy E. Blelloch, Jonathan C. Hardwick, Siddhartha Chatterjee, Jay Sipelstein, and Marco 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.

Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat 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.

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

Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. ISSN 0001-0782.

T. B. Dinesh, Magne Haveraaen, and Jan Heering. An algebraic programming style for

numerical software and its optimization. *Scientific Programming*, 8(4):247–259, 2000. ISSN 1058-9244.

Stefan Edelkamp, Damian Sulewski, and Cengizhan Yücel. Perfect hashing for state space exploration on the GPU. In Ronen I. Brafman, Hector Geffner, Jörg Hoffmann, and Henry 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.

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

Erich Elsen, Mike Houston, V. Vishal, Eric Darve, Pat Hanrahan, and Vijay 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.

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

Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: a MapReduce framework on graphics processors. In *PACT '08: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 260–269, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-282-5.

Qiming Hou, Kun Zhou, and Baining Guo. BSGP: bulk-synchronous GPU programming. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–12, New York, NY, USA, 2008. ACM. ISBN 978-1-4503-0112-1.

Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28 (4):196, December 1996. ISSN 0360-0300.

Paul Hudak. Modular domain specific languages and tools. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, pages 134–142, Washington, DC, USA, June 1998. IEEE Computer Society. ISBN 0-8186-8377-5.

Graham Hutton and Erik Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4), July 1998.

Kenneth 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.

Thomas 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.

Samuel N. Kamin and David Hyatt. A special-purpose language for picture-drawing. In

*Proceedings of the Conference on Domain-Specific Languages*, pages 297–310. USENIX Association, October 1997.

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

A. Kloeckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA: GPU run-time code generation for high-performance computing. Technical Report 2009-40, Scientific Computing Group, Brown University, Providence, RI, USA, November 2009.

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.

Sean Lee, Manual M. T. Chakravarty, Vinod Grover, and Gabriele Keller. GPU kernels as data-parallel array computations in Haskell. *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods*, 2009a.

Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. *SIGPLAN Notices*, 44(4):101–110, April 2009b. ISSN 0362-1340.

Geoffrey Mainland and Greg 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.

Panagiotis Manolios and Yimin Zhang. Implementing survey propagation on graphics processing units. In Armin Biere and Carla 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.

Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. Shader metaprogramming. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware*, pages 57–68, Aire-la-Ville, Switzerland, 2002. Eurographics Association. ISBN 1-58113-580-7.

Michael D. McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 787–795, New York, NY, USA, 2004. ACM.

Eric Niebler. Proto: a compiler construction toolkit for DSELs. In *LCSD '07: Proceedings of the 2007 Symposium on Library-Centric Software Design*, pages 42–51, New York, NY, USA, 2007. ACM. ISBN 978-1-60558-086-9.

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

Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):i–xii,1–255, January 2003.

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

Morten Rhiger. A foundation for embedded languages. *ACM Transactions on Programming Languages and Systems*, 25(3):291–315, 2003. ISSN 0164-0925.

Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John 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.

David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. *SIGOPS Operating Systems Review*, 40(5):325–335, December 2006. ISSN 0163-5980.

Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.

Todd L. Veldhuizen. Arrays in Blitz++. In Denis Caromel, Rodney Oldehoeft, and Marydell 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.

Todd L. Veldhuizen and Dennis Gannon. Active libraries: Rethinking the roles of compilers and libraries. In Michael E. Henderson, Christopher R. Anderson, and Stephen L. Lyons, editors, *Object Oriented Methods for Interoperable Scientific and Engineering Computing: Proceedings of the 1998 SIAM Workshop*, pages 286–295. SIAM Press, October 1998.

Hongwei Xi, Chiyan Chen, and Gang 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.

# APPENDIX A

# FORWARD DIFFERENCE IMPLEMENTATION

This appendix contains two implementations of the forward difference operator, an example from Chapter 1: the handwritten CUDA version and the Barracuda-generated version, both using shared memory. Recall the definition of the forward difference operator for an array:

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

where $x$ and $x+1$ are valid indexes for $\mathrm{arr}$. In this operation, with the exception of the first and last array elements, each element is read twice, which makes the use of shared memory beneficial.

The handwritten implementation is given in Figures A.1 and A.2. This is the same implementation given in Chapter 1, and uses GPU shared memory to reduce device memory traffic. The kernel, shown in Figure A.1, begins with a loading phase of shared memory, where each thread loads one array element from device memory into thread block-shared memory. After loading is complete, the actual forward difference computation occurs: if the thread executing the kernel is within bounds, it writes the result for one output array element. If the executing thread is at the right edge of its thread block, it needs to read from device memory, as one of the input array element values that it requires is not loaded into shared memory. Otherwise, the thread can get both of its required values from thread

```
__global__ void
forward_diff_kernel (const float *in, float *out, const unsigned len)
{
    // Declare block-shared memory.
    extern __shared__ float scratch[];

    // Get the output index assigned to this thread.
    const unsigned i = blockIdx.x * blockDim.x + threadIdx.x;

    // Load one element into shared memory & synchronize.
    if (i < len)
        scratch[threadIdx.x] = in[i];
    // Synchronize within the thread block to make writes visible.
    __syncthreads();

    // Compute the difference.  Threads on the end of a block
    // have to read from device memory.
    if (i < len - 1)
        // If the executing thread is at the rightmost edge of a
        // thread block, read from device memory, otherwise read from
        // shared memory.
        out[i] = threadIdx.x == blockDim.x - 1
               ? in[i + 1] - scratch[threadIdx.x];
               : scratch[threadIdx.x + 1] - scratch[threadIdx.x];
    }
}
```

Listing A.1: A handwritten forward difference CUDA kernel

block-shared memory that was loaded previously in the kernel. The wrapper procedure, shown in Figure A.2, contains the necessary memory management and kernel invocation code to actually run the kernel on the GPU. Note that in this handwritten wrapper procedure, the `in` and `out` arguments point to locations in the system's main memory, not into GPU device memory. To run the operation on the GPU, then, requires copying the input array before the kernel execution and copying the output array after kernel execution.

A Barracuda implementation of the forward difference is shown in Figure A.3. This is a complete Haskell program that generates a header file and the source file shown in Figure A.4. The generated code has similar overall structure to the handwritten version, where a wrapper procedure invokes a kernel that does the actual work. The structure of the kernel is also similar to the handwritten kernel from Figure A.1: first each thread participates in a loading phase, in which an element of the input array in device memory is copied into shared memory. Then, each thread that is within bounds computes the

```
void
forward_diff_wrapper(const float *in, float *out, const unsigned len)
{
    // Allocate device memory for the arrays.
    float *in_d;
    float *out_d;

    const size_t in_size = len * sizeof(float);
    // The output is one element shorter than input.
    const size_t out_size = (len - 1) * sizeof(float);

    cudaMalloc((void **)&in_d, in_size);
    cudaMalloc((void **)&out_d, out_size);

    // Copy input array into device memory.
    cudaMemcpy(in_d, in, in_size, cudaMemcpyHostToDevice);

    // Set up and invoke the kernel.
    const dim3 block_dim(128, 1, 1);
    const dim3 grid_dim((unsigned)ceilf((float)len / 128f), 1, 1);
    // How much shared memory to allocate per block?
    const size_t smem_size = 128 * sizeof(float);

    forward_diff_kernel<<<grid_dim, block_dim, smem_size>>>
        (in_d, out_d, len);

    // Copy results back into CPU memory.
    cudaMemcpy(out, out_d, out_size, cudaMemcpyDeviceToHost);

    // Release the device memory.
    cudaFree(out_d);
    cudaFree(in_d);
}
```

Listing A.2: A handwritten forward difference wrapper procedure

```
import Language.Barracuda

forwardDiff :: AExp Float -> AExp Float
forwardDiff x = azipWith (-) x'' x'
  where x'  = aslice x (0, alength x - 1, 1)
        x'' = aslice x (1, alength x, 1)

forwardDiffFun :: Function
forwardDiffFun = Function forwardDiff "forward_diff" ["x"] "result"

main :: IO ()
main = compile shmemCudaConfig files [forwardDiffFun]
  where
    files = ("forward_diff.cuh", "forward_diff.cu")
```

Listing A.3: Forward difference in Barracuda

forward difference for a single output array element. The conditional code and indexing code in this Barracuda-generated kernel is more complex than that found in the handwritten kernel; however, the CUDA compiler seems to do a good job optimizing the generated code, judging from the benchmark results seen in Chapter 4.[1] Note that the Barracuda-generated wrapper procedure takes input and output array parameters that already reside on the GPU, unlike the handwritten version that expected arguments to arrays in the system's main memory. This is done for the sake of performance: moving data between main memory and device memory is the costliest GPU operation, and so Barracuda places that responsibility on the caller.

[1] No handwritten implementation was used in the forward difference benchmarks, as forward difference was used only to show the impact of shared memory optimization. Other benchmarks, such as Black-Scholes call option pricing, show that the runtime of Barracuda implementations is very close to that of handwritten implementations.

```
// begin forward_diff.cu

#include "forward_diff.cuh"
#include "barracuda.hpp"

static __global__ void
kernel3 (const kernel_float_array x,
         kernel_float_array_view result)
{
    __shared__ float x_shared [256];
    if (blockIdx.x * 256 + threadIdx.x < x.length())
    {
        x_shared[threadIdx.x] = x[blockIdx.x * 256 + threadIdx.x];
    }
    __syncthreads();
    int i2 = blockIdx.x * 256 + threadIdx.x;
    if (i2 < (x.length() - 1))
    {
        result[i2] = (blockIdx.x * 256 <= 1 + i2 &&
                      1 + i2 < 256 * (blockIdx.x + 1)
                        ? x_shared[1 + i2 - blockIdx.x * 256]
                        : x[1 + i2])
                     -
                     (blockIdx.x * 256 <= i2 &&
                      i2 < 256 * (blockIdx.x + 1)
                        ? x_shared[i2 - blockIdx.x * 256]
                        : x[i2]);
    }
}

void forward_diff (const gpu_float_array &x,
                   gpu_float_array_view result)
{
    const dim3 blockDims4 (256, 1, 1);
    const dim3 gridDims5 (((x.length() - 1) + 255) / 256, 1, 1);
    kernel3<<<gridDims5, blockDims4>>>(
        kernel_float_array (x),
        kernel_float_array_view (result));
}

// end forward_diff.cu
```

Listing A.4: A Barracuda-generated shared memory forward difference implementation