Chestnut: A GPU Programming Language for Non-Experts

Andrew Stromme
Computer Science Dept.
Swarthmore College
Swarthmore, PA USA
andrew.stromme@gmail.com

Ryan Carlson
Language Technologies Inst.
Carnegie Mellon University
Pittsburgh PA USA
rcarlson@cs.cmu.edu

Tia Newhall
Computer Science Dept.
Swarthmore College
Swarthmore, PA USA
newhall@cs.swarthmore.edu

ABSTRACT

Graphics processing units (GPUs) are powerful devices capable of rapid parallel computation. GPU programming, however, can be quite difficult, limiting its use to experienced programmers and keeping it out of reach of a large number of potential users. We present Chestnut, a domain-specific GPU parallel programming language for parallel multi-dimensional grid applications. Chestnut is designed to greatly simplify the process of programming on the GPU, making GPU computing accessible to computational scientists who have little or no parallel programming experience, as well as a useful and powerful language for more experienced programmers. In addition, Chestnut has an optional GUI programming interface that makes GPU computing accessible to even novice programmers.

Chestnut is intuitive and easy to use, while still powerful in the types of parallelism it can express. The language provides a single simple parallel construct that allows a Chestnut programmer to "think sequentially" in expressing her Chestnut program; the programmer is freed from having to think about parallelization, data layout, GPU to CPU memory transfers, and synchronization. We demonstrate Chestnut's programmability with example solutions to a variety of parallel applications. Performance results from our prototype implementation of Chestnut show that Chestnut applications perform almost as well as hand-written CUDA code for a set of several parallel applications. In addition, Chestnut code is much simpler and much smaller than hand-written CUDA code.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*concurrent, distributed, and parallel languages*

General Terms

Languages

PMAM '12 February 26 2012, New Orleans, LA, USA

Keywords

GPU, parallel programming language

1. INTRODUCTION

Graphical Processing Units (GPUs) have become an increasingly popular platform for general purpose parallel computing. GPUs have the potential to support large-scale parallel applications on very inexpensive hardware. However, the programming model for GPUs is quite complicated, limiting its accessibility to only very sophisticated programmers. Nvidia's CUDA [16] GPU programming interface, for example, requires programmers to explicitly manage CUDA and CPU memory, to map their parallel computation in terms of blocks of thread groups onto GPU data, and to structure their solution as a set of SIMD-like CUDA kernels that run in parallel on the GPU interspersed with CPU-side synchronization and GPU-CPU data transfers. For simple, embarrassingly parallel applications there are obvious mappings to CUDA kernels, but as the complexity of the parallel computation increases, particularly in the amount of synchronization required, so does the complexity of the CUDA program. Even for expert programmers, CUDA can be a very difficult programming interface to use and debug.

There has been much recent work in new programming languages and libraries designed to simplify GPU programming, but most still require a fair amount of programming expertise, often keeping GPU programming out of reach for computational scientists and novice programmers. Chestnut is our domain-specific GPU programming language that is designed to make GPU programming accessible to a larger set of programmers, particularly targeting the computational scientist who may have little or no parallel programming experience. We additionally provide an optional GUI programming interface on top of a subset of the Chestnut language that is similar in design to Scratch [18], and is specifically designed for novice programmers to easily write parallel GPU applications.

The Chestnut language implements a mostly sequential view of parallel GPU programming. It has a simplified C-style syntax that would be familiar to most scientific programmers who write sequential programs in C, Fortran, R, or other sequential imperative or object-oriented languages.

The Chestnut language is part of our larger system that includes a multi-layered programming model and compilers that translate between each layer. Our fully automated compilation environment goes from a GUI representation of a Chestnut parallel program to an executable file. In addition, we expose all intermediate layers of this translation (GUI to

Chestnut and Chestnut to CUDA-C) to make our system a teaching tool for parallel programming; novice programmers can start with the GUI and use the Chestnut translation to help them learn to program in Chestnut. Most programmers will directly write code in the Chestnut language, but programmers who want hand-optimized CUDA code can use the Chestnut to CUDA-C translation as a starting point, adding in their hand-written CUDA optimizations.

Chestnut is not a general-purpose language but a domain-specific language targeting parallel multi-dimensional grid (array) applications. Multi-dimensional grid applications cover a large subset of parallel scientific applications [9]. Chestnut is particularly good at easily expressing grid stencil pattern parallelism. Stencil patterns include a large class of scientific applications that perform finite discrete modeling or finite solvers for partial differential equations [20]; examples include climate modeling, heat diffusion, and fluid dynamics. Sequential implementations of these computations involve a core operation that is performed many times on each grid element, making them good candidates for GPU parallelization because they map well onto a simple data parallel model where updating individual grid elements can be performed independently.

One difficulty in designing parallel languages is how to easily specify parallelism. The Chestnut parallel programming model allows for a basically sequential view of the program. A Chestnut program has sequential control flow containing parallel blocks, or contexts, that perform computation on grid elements in parallel. A parallel context is expressed using a "sequential" view of its execution—to specify a parallel context, the programmer uses a `foreach` loop based on one or more grids, where one "iteration" of the loop is run for each element in the grid. The `foreach` loop and its loop body look just like sequential code. However, each "iteration" of the `foreach` loop is actually performed by a separate thread in parallel with all of the other "iterations".

In general, Chestnut programs specify their parallelism based on the output grids—one computation for each element in the output grid. For example, matrix multiplication in Chestnut is expressed as a parallel `foreach` loop that "iterates" over each result grid element. The body of the `foreach` loop accesses the necessary row and column of the input grids to perform the computation necessary for each element in the output (result) grid. When executed on the GPU, multiple threads simultaneously and independently execute an "iteration" of the `foreach` loop to compute the result grid elements in parallel.

Chestnut automatically handles data dependencies between the parallel threads executing within a `foreach` context. It ensures that each thread reads the initial state of all grid data as they were at the entry of the `foreach` context. Chestnut also implicitly implements barrier synchronization at the end of each `foreach` context, ensuring that the grid data reflect all the updates from the parallel `foreach` execution; all subsequent grid data accesses read the new values.

Chestnut is easy to use and fully exposed. The programmer can, with a single build command, go from the GUI to an executable that can run on the GPU. Even though our language is designed to be simple to program and use, it is also quite powerful and performs well compared to hand-written CUDA code. We therefore anticipate that even expert programmers will make use of Chestnut.

The rest of our paper is organized as follows: Section 2 discusses related work in programming languages for parallel, specifically GPU, computing; Section 3 discusses the Chestnut programming language; Section 4 presents the full system architecture of Chestnut including the GUI programming interface and compilers; Section 5 presents Chestnut programs and performance results; and Section 6 concludes and discusses future directions of our work.

2. RELATED WORK

The development of general purpose programming interfaces for GPUs, such as CUDA [16], OpenCL [10], StreamSDK [2], and DirectCompute [15], has resulted in a huge increase in the uses of GPUs for parallel computation, particularly for parallel scientific applications. The underlying GPU architecture naturally lends itself to a stream [7] or SPMD model of parallel computing, wherein a parallel kernel of code is run independently and simultaneously across 100s of GPU processors on a stream of data elements. There has been a great deal of work developing interfaces for programming general purpose GPUs, as well as many programming and mathematical libraries ported to GPUs [8, 6], and bindings to many existing programming languages including C, Fortran, Ruby, Java, Python, and Matlab, all helping to simplify GPGPU programming.

There has additionally been a lot of recent work in developing new GPGPU programming languages, libraries, and GPU back-end support to existing parallel languages [17, 21, 14, 13, 5, 12]. This work makes GPGPU programming more accessible to a larger set of users by hiding many of the complications of GPU programing, such as having to think about CPU and GPU memory and having to map parallel computation onto GPU data. However many of these, (hiCuda and Mint, for example), still require explicit memory copies between the host (CPU) and device (GPU).

Many of the new languages are domain specific languages (DSL) that target stencil pattern applications. A large number of parallel applications fit the stencil parallel pattern [9], making these languages very useful for solving a large class of real-world scientific applications. By designing a DSL instead of a more general purpose parallel language, the resulting language is often smaller and easier to learn. In addition, its implementation can often lead to better optimized GPU code that can more easily target optimizations that are specific to stencil patterns [11, 1]. Chestnut is also a DSL, but it targets a larger class of applications that include both stencil pattern and also some more general grid access patterns.

Several of these new languages are implemented as embedded languages. For example, Physis is embedded in C, Mint in C and C++, and Ypnos in Haskell. An advantage to an embedded approach is that by adding parallel language extensions to an existing language, the parallel language gets, for free, all the functionality of the language in which it is embedded. In addition, embedded language development tends to be easier, as it often involves implementing a pre-compilation or pre-processing step to the underlying target language, and can then make use of existing compilers for the full compilation. For programmers who already know the underlying sequential language, embedded languages may be easier to learn because the programmer only needs to learn a set of GPU parallel language extensions.

One drawback of the embedded approach is that the GPU language extension is stuck with all the syntax and features

of the underlying language, many of which may not be useful, or may not be compatible with the parallel language extensions. In addition, for programmers who do not know the underlying language, the parallel language is unnecessarily complicated by the syntax and semantics of underlying language. This approach can lead to the sacrifice of good parallel language design principles [4] as a trade-off for ease of implementation.

Many of these projects use a source-to-source translation approach that simplifies the implementation of the language by making use of existing compilers, libraries and run-time systems. It also allows for more flexibility in the "back-end" of these implementation, as the target language can more easily be changed to take advantage of new libraries or language features at the target language level. In addition, source-to-source translation has the potential for the parallel language implementation to automatically take advantage of improvements and optimizations in updated versions of the underlying target language implementation.

Chestnut uses a source-to-source translation approach, currently targeting CUDA-C and Thrust. However, Chestnut differs from other work in this area in that it is a full language implementation rather than an embedded language. The main goal of our work is to implement a GPU programming language that is very easy to learn and use; by designing our language from scratch we can more easily meet this goal—the Chestnut language is simple, small, and designed to easily express exactly the target applications that our language supports.

3. THE CHESTNUT LANGUAGE

Chestnut is a domain specific language that supports parallel computation on multi-dimensional parallel arrays (grids). The main goal of our language design is to make it simple and easy to use yet able to express a large class of parallel GPU applications. Our language implements a sequential imperative programming model, which allows the programmer to "think sequentially" in expressing her parallel program. Programs in Chestnut consist of sequential code interspersed with parallel loops over parallel array data; to the programmer, a Chestnut program is just a sequence of statements executed in order, some of which execute on the GPU.

Chestnut implements a GPU-centric model of computation where the graphics card is considered the computer as much as possible. Parallel computation in Chestnut is expressed in terms of operations on parallel array data. All parallelism in Chestnut is expressed within extended parallel statements known as parallel contexts. A parallel context (`foreach` loop) contains a set of statements for accessing and modifying elements of parallel arrays. Parallel contexts are executed on the GPU such that the set of statements inside the `foreach` loop body are executed on individual array elements in parallel on the GPU. For example, Listing 1 shows a very simple parallel construct that sets each element in a parallel two-dimensional array to zero.

Because parallel contexts syntactically look like loops over arrays, a programmer can think of the representation of their Chestnut program as being executed sequentially even though parts of it exhibit massive parallelism. Figure 1 shows the high-level flow of a Chestnut program with parallel contexts interspersed among normal statements.

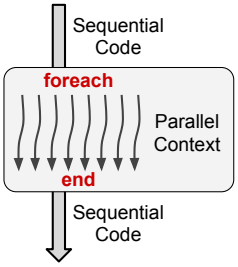

Figure 1: A chestnut program can be expressed as a sequential stream of code consisting of sequential blocks of code interleaved with parallel (foreach) contexts. This general control flow pattern can be repeated over and over in a Chestnut program.

Listing 1: A basic parallel context

```
// parallel 5x5 grid of Int values
IntArray2d array[5, 5];

// runs on GPU, sets each array item to 0
foreach item in array
    item = 0; // body of parallel context
end
```

Chestnut's GPU-centric model frees the programmer from having to explicitly think about CPU and GPU memory, from having to explicitly allocate memory on one or the other, and from having to explicitly copy data between the CPU and the GPU. Parallel arrays are allocated in GPU memory and are only copied to CPU memory when accessed from sequential code in a Chestnut application. In Chestnut, a programmer can only implicitly trigger CPU-GPU data copy by initializing a parallel array to values read in from a file or by printing out parallel array data values to stdout or a file. By hiding GPU and CPU memory from the programmer, Chestnut simplifies programming and can better minimize the amount of data copy between the CPU and GPU.

Chestnut additionally hides from the programmer both GPU parallel execution concepts such as blocks and threads, and the mapping of parallel execution onto GPU data—in Chestnut the tight coupling between the Chestnut parallelism model and the way it internally allocates threads and maps GPU parallelism to parallel data, means that none of the underlying GPU execution model needs to be exposed to the Chestnut programmer. Similarly, synchronization structures (semaphores, mutexes) and explicit thread communication are hidden by the simplicity of the Chestnut parallel programming model. Although this limits the types of parallelism that can be expressed in Chestnut to operations on multi-dimensional parallel arrays, it is a reasonable tradeoff for achieving a concise language that is very easy to learn, and it still leaves the Chestnut language able to express a large class of parallel programs.

Our discussion of the Chestnut language includes its basic and composite types, the parallel context, its scoping model and its support for simultaneous GPU visualization and animation of a parallel application computation.

3.1 Chestnut Types

As a strongly and statically typed language, Chestnut requires type information at the time of a variable's declara-

tion. Currently, Chestnut supports a set of basic built-in types. In the future we plan to extend Chestnut to support user-defined types using a struct or object syntax. Chestnut automatically converts between types, using the same semantics as C, when multiple built-in type operands appear in the same expression.

Chestnut types generally fall into one of three categories: simple types, composite types, and parallel array element types (all shown in Listing 2). Simple types are scalar values or collections of scalar values corresponding to integer, boolean, and floating point values, as well as RGB color values that are used in Chestnut's support of GPU visualizations of Chestnut parallel computation.

The primary composite type is the Parallel Array, described in depth in Section 3.1.1. Chestnut currently has support for 1, 2 and 3 dimensional arrays of any of its basic types. The final category contains Array Element types that consist of their underlying scalar value and attributes for accessing neighboring values. Array Element types are used only within parallel contexts, and provide a localized view of an array element in addition to its value, allowing for stencil computations to be easily expressed in Chestnut.

Listing 2: Types

```
(1) Basic Scalar Types:
 Int     // a 32bit int value
 Real    // a 32bit float point value
 Color   // 4 Reals: red, green, blue,
         //          opacity
 Bool    // a 1bit boolean

(2) Array Types: // 1,2,3 dimensional

 // Example Specific Array Types:
 IntArray2d   // 2-dim array of Int
 RealArray3d  // 3-dim array of Real

 // Array Types have attributes:
 width, height, depth

(3) Array Element Types:
 Int1d, Int2d, Int3d, Real1d, ...

 // Array Element types have attributes
 x, y, z,  east, west, north, south,
 northEast, southEast, southWest, ...
```

3.1.1 Parallel Array Types

The Parallel Array is the main data storage type in Chestnut and the primary model of parallel computation: parallelism is expressed in terms of operations on parallel arrays. Currently, Chestnut supports one, two and three dimensional arrays of the basic types. A specific array type is defined by prefixing `Array` with the basic type (e.g. `Int`) and appending the dimensionality (e.g. `2d`) which in this example forms `IntArray2d` as the canonical type name for a 2 dimensional array of integers. Array types have attributes specifying the dimensions of the array (width, height, and depth). Individual elements in parallel Arrays can be accessed using indexing, or through Array Element variables defined in `foreach` parallel contexts. Listing 5 shows an example that uses both types of access methods.

Arrays can be initialized on the GPU using a parallel construct to set each Array Element value, or they can be initialized to values read in from a file. File initialization is done on the CPU, and Chestnut implicitly copies the array data from the CPU to the GPU once initialized. In the future we plan to support a richer set of sequential (CPU-side) initializations of parallel arrays via sequential loops that contain assignments to parallel array elements. Chestnut would continue to implicitly perform all CPU to GPU data copies.

Array data are initialized from file data using the sequential function `read(filename)`. This function expects to be passed a filename with data in a two part format: the first part is a single line which declares the array's type along with its dimensions; the remainder of the file is a comma separated list of values. All white space characters are ignored, which means that the same list of elements might have multiple valid shapes. The first line in the file describes the shape of the array and avoids ambiguity in how the data read in should be assigned to multi-dimensional array positions.

3.1.2 Parallel Array Element Types

The Array Element types consist of the basic type values stored in each parallel array bucket and a set of attributes that allow for localized neighbor element accesses, and that give each element's (x, y, z) position in the array.

The Array Element type allows for easy expression of stencil pattern accesses via neighbor-specific attributes of the centered element using dot syntax. Chestnut uses attributes of Array Element types instead of indexing syntax (i.e. `array[x-1, y+1]`) to better meet our goal of a language that is easier to read and program. A visual example of an Array Element's neighbor attributes in a two-dimensional array is shown in Figure 2; each element in the array has neighboring points to its `east`, `west`, `north` and `south` as well as corners `northEast`, `southEast`, `southWest` and `northWest`. This can be extended to 3d array access by prefixing either below or above (e.g. `belowSouthEast`, `aboveNorth`) for vertical access. When simplifying to 1d arrays, `left` and `right` are synonyms for `west` and `east`. Chestnut currently automatically supports wrap-around access to points on the edges of arrays. In the future we plan to support constant and function-derived values for edge points.

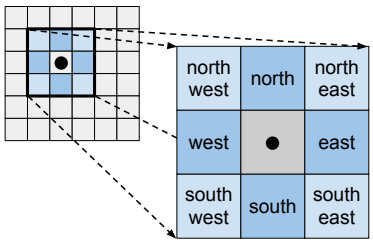

Figure 2: The Neighboring points of An Array Element centered at (2, 2). The Array Element's neighbor attributes allow each array element easy access to each it neighbors' values.

3.2 The Parallel Context

The primary way to express parallel computation is Chestnut is using the parallel context on a set of parallel arrays. A parallel context begins with a `foreach` statement and ends with the keyword `end`. The statements between the start and end of the parallel context specify a set of operations to

be performed in parallel on the parallel arrays, one after another. Local variables defined inside a parallel context go out of scope upon leaving the parallel context. In some sense, parallel contexts can be thought of as anonymous functions on parallel arrays.

In a `foreach` statement, the parallel computation is defined in terms of output arrays. The most basic foreach expression describes a loop over each element in an output array. The body of the parallel context is executed once for each element in the array. An example of the basic context is shown above in Listing 1. Here the anonymous function simply executes the `item=0` statement once for each element in the array `array`, setting its value to zero. Each `item=0` statement runs in parallel and independently from every other statement on the GPU. The `end` keyword ensures that all of these operations have finished before continuing on to the next line of code.

Parallel contexts can contain multiple statements and even function calls. In general, anything that is a valid statement is valid inside of a parallel context. The notable exceptions are that sequential functions cannot be called within parallel contexts, and parallel contexts cannot be nested.

A `foreach` context can be applied to multiple parallel output arrays. Listing 3 shows an example of a foreach context over two parallel arrays that swap values in corresponding positions. Multiple arrays listed in the same `foreach` statement must have the same dimension.

Listing 3: Multiple Arrays in a parallel context

```
IntArray2d a_array[10,10]=read("a.dat");
IntArray2d b_array[10,10]=read("b.dat");

foreach a in a_array, b in b_array
  Int temp = a;
  a = b;
  b = temp;
end
```

The variables declared in a `foreach` statement are Array Elements types with attributes in addition to their values. An element's attributes can be used to access neighboring elements' values as well as getting its (x, y, z) position in the parallel array. In addition, external parallel arrays (those not listed in the `foreach` statement) can be accessed from within the body of a `foreach` block. External access is needed to support parallel computation on arrays of different dimensions.

Listing 4 shows an example of a stencil computation in Chestnut. This example computes one time step of a heat dispersion simulation, where at each time step every array element gets a new value based on its neighbor's values. It shows how to use an Array Element's attributes to obtain its neighbors' values.

The Semantics of Statements Inside Parallel Contexts

The following are the semantic rules for evaluating statements inside `foreach` bodies:

- Array Element values read inside `foreach` blocks (i.e. that appear on the right hand side of assignment statements), get the values as they were immediately before the `foreach` statement.
- There is a localized sequential semantics of multiple statements executed inside a parallel context that involve assignments to the same Array Element variable; Array Elements written to (i.e. that appear on the left hand side of an assignment statement) are modified within the localized view of computation only (i.e. only the specific thread updating this specific Array Element's value will read its new value within the same parallel context).

These rules mean that Chestnut implements a localized sequential semantics of the execution of multiple statements that make assignments to Array Element variables inside a parallel context. The semantics also ensure that when assignment statements inside `foreach` blocks involve functions of neighboring elements, all neighbor values that are read are their value before the `foreach` block. Although when executed on a GPU, the `foreach` body statements are being run in parallel over the array elements, Chestnut ensures that each value read is consistently the array value prior to the parallel context.

As an example, in Listing 4 the first statement sets the localized value of `point` to a new value, so the localized execution of this second statement will use this new value of point in combination with the original values of its neighboring points. In other words, even though each thread is updating an Array Element value in the first statement, in the second statement all threads consistently read their neighboring point values as they were prior to the `foreach` block, and only the particular update to a particular Array Element reads the new value of point when it is used in the second statement.

Listing 4: Heat Dispersion Iteration

```
RealArray2d world[W,H]=read("heat.dat");

Real center = 0.4;
Real other = (1 - center)/4;

foreach point in world
  point  = center*point;
  point  = point +
           other*(point.west + point.east
           + point.north + point.south);
end
```

These semantics mean that the Chestnut programmer doesn't have to think about, or handle, race conditions between parallel updates to individual Array Element values. To support these semantics, Chestnut internally maintains copies of parallel arrays with these types of dependencies; one copy is used for reading values inside a `foreach` loop, another for writing. This way Chestnut ensures consistent state is always read from parallel arrays. Chestnut handles any data copying (or pointer swapping between copies) and any synchronization necessary to support its semantics of parallel context executions.

Listing 5 shows matrix multiply, an example of how to access external parallel arrays in a `foreach` context. In this example, the parallel computation is specified in terms of the output array `output`. For each element in the output array, its x and y coordinate positions are used to index into the two source arrays of the matrix multiplication. Using an Array Element's coordinate attributes to index into external arrays is more complicated than expressing stencil pattern array accesses in Chestnut, but it is still much less

complicated than in CUDA where the programmer has to map parallelism in terms of blocks of thread onto accessing elements from three arrays of three different dimensions.

Listing 5: Matrix multiplication in Chestnut

```
RealArray2d a[10, 5] = read("a.data");
RealArray2d b[5, 8] = read("b.data");
RealArray2d output[10, 8];

foreach e in output
  e = 0;
  for (Int i=0; i < a.height; i++) {
    e = e + a[e.x, i] * b[i, e.y];
  }
end

write(output, "output.data");
```

3.3 Functions

Chestnut supports two types of user-defined functions: sequential functions run on the CPU, and parallel functions run on the GPU. Both types of functions follow C-like syntax and semantics. Parallel functions in Chestnut are prefixed with the keyword `parallel`, and sequential functions with the keyword `sequential`. Although this prefix annotation to functions appears to violate one of our goals of hiding "thinking about parallelism" from the programmer, we use it to help the programmer more easily remember the differences between how parallel and sequential functions can be called and defined in a Chestnut program.

Sequential functions can be passed scalar types and Parallel Arrays, and they can contain parallel `foreach` constructs. They cannot, however, be passed individual parallel Array Element values. Parallel functions are functions that can only be called from within parallel contexts. They are primarily to support better modular design and code reuse of sub-functionality in the bodies of parallel constructs. Listing 6 shows an example of a parallel function definition and use. The following are the set of constraints associated with using and calling Chestnut parallel functions:

- A parallel function can be passed Array Element, Array, or scalar types. Typically, a parallel function is called on an output array element.
- A parallel function can only be called from within a parallel context or from another parallel function.
- A parallel context cannot be inside of a parallel function. This follows from the restriction that parallel contexts can't be nested and because a parallel function is by definition called from a parallel context.
- A parallel function can't issue `read()` or `write()` calls to read or write data in from disk.

3.4 Built-in Functions

Chestnut currently supports three built-in functions. The `random` function is a parallel function that can be called from within a parallel context. It returns a random Real value between 0.0 and 1.0.

The other two built-in functions can be applied to parallel data, but are called outside of parallel constructs. They perform operations on parallel data that cannot be easily expressed using Chestnut's basic `foreach` parallel context. One of these is the `display` function, described in Section 3.5, for visualizing and animating Chestnut computations. The second, `reduce`, performs a parallel reduction. It takes a parallel array parameter and returns a scalar value result of the reduction operation applied over all the values in the array. Internally, Chestnut implements the reduction operation in parallel on the GPU, but the programmer needs only to make a simple reduction function call to invoke this parallel operation. Listing 6 shows an example that calls the `reduce` built-in function.

Currently, Chestnut supports only a sum reduction. We plan to extend Chestnut's reduction functionality by adding a second parameter that would take a custom written reduce operation that could be applied over the parallel array. For example, a max function could be passed as the second argument to perform a max reduction instead of a sum reduction.

Listing 6: User-defined and Built-in Parallel Functions

```
IntArray2d array[500,500]=read("in.dat");

parallel Int square(Int2d value) {
    return value * value;
}

foreach x in array
    x = square(x);
    x = x * 2;
end

Int sum = reduce(array);
```

3.5 Support for GPU Visualization

Chestnut provides built in support for visualizing parallel array computation as it is executed on the GPU. Basic pixel values, using the `Color` type, can be associated with values in parallel arrays. To visualize Chestnut parallel array data, a programmer makes a call to a special Chestnut built-in function `display`. This function takes a parallel array of data values to visualize, and an optional second argument. The second argument is a parallel color conversion function that specifies how to set a `Color` value based on an Array Element's value. Internally, Chestnut applies the passed color conversion function (or uses the default conversion function) to every element in the parallel array and then displays the result. The visualization computation is done in parallel on the GPU and the display stays on the GPU (i.e. it is not copied to and from the CPU for display).

Listing 7: Visualizing a simple gradient

```
parallel Color green(Real2d input) {
  Color c;
  c.red     = 0;   c.green   = input;
  c.blue    = 0;   c.opacity = 1;
  return c;
}

RealArray2d gradient[720, 480];

foreach pixel in gradient
  pixel = pixel.y / gradient.height;
end

display(gradient, green);
```

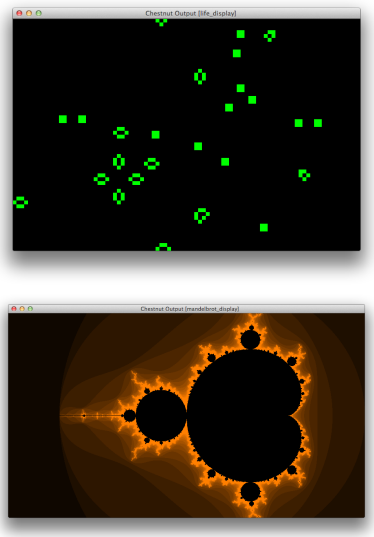

Figure 3: Chestnut GPU Animation of Conway's Game of Life, and of Mandelbrot Fractal.

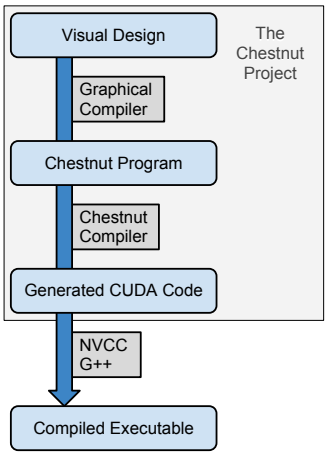

Figure 4: The Chestnut System: A GUI representation of a Chestnut program represented using the Visual Designer is translated to Chestnut source code; a program expressed in the Chestnut language is compiled to CUDA-C source, which is in turn compiled to an executable file.

Listing 7 shows an example using a custom color function passed to the `display` function. Calling the display function repeatedly with the same array causes that display to be updated rather than new displays being created, which provides for a simple and efficient way to animate the computation and create other dynamic displays of the data. Figure 3 shows examples of Chestnut's on-GPU visualizations of a zoomed-in portion of Conway's Game of Life, and Mandelbrot fractal pattern.

With the basic types and constructs provided by Chestnut, a programmer can easily express stencil pattern parallelism and, without much more expertise, an even larger set of parallel computations on multi-dimensional grids. The Appendix lists a complete Chestnut program, showing interspersed sequential and parallel constructs and parallel functions.

4. CHESTNUT SYSTEM ARCHITECTURE

The Chestnut language is part of a larger, multi-level Chestnut GPU programming environment including multiple programming interfaces and compilers. Figure 4 shows the architecture of the Chestnut system. At the top level is the Chestnut Designer, which is an optional GUI programming interface to Chestnut. The Chestnut Designer is very easy to use, and is designed to be an interface for novice programmers or programmers first learning the Chestnut language. The graphical compiler translates the GUI representation of a parallel program to Chestnut source code.

At the next level is the Chestnut language. Programmers can directly write GPU programs at this level using the Chestnut language, or they can indirectly generate Chestnut programs using the Chestnut Designer and the graphical compiler. The Chestnut compiler translates programs written in the Chestnut language to C++ source code with calls to Walnut library functions. Walnut is a C++ library of routines that provides an abstraction on top of CUDA and Thrust. It greatly simplifies the implementation of the Chestnut compiler's C++ back-end because the Walnut API is much closer to the Chestnut model than the CUDA API is. It also allows for the Chestnut back-end to be more easily ported to target other GPU programming interfaces and libraries. The Chestnut system then makes use of NVCC g++ to compile its C++ CUDA/Thrust translation to an executable that can run on the underlying system. The Chestnut system provides a simple, single build command that can go from GUI to executable, automatically translating from each layer to the next—a programmer can go from a build button on the GUI to an executable without having to issue a command line to build. In addition, the Chestnut system exports the translation results at each level so that a programmer can see the underlying Chestnut and C++ CUDA/Thrust translations of the program.

4.1 Chestnut Designer

The Chestnut Designer is a GUI interface to the Chestnut language. The Designer is composed of a canvas on which program objects can be placed and connected together to represent the data-flow of a Chestnut program. Different types of program components are dragged from the left sidebar onto the canvas and connected together to create the flow of data in the program. It provides Chestnut's basic and array types, and function objects that operate on some combination of these types. Currently, Chestnut Designer supports two functions map and reduce. The map function is translated to a Chestnut `foreach` block and the reduce function to Chestnut's built-in parallel `reduce` function. As in the Chestnut Language, functions in the designer are well specified and strongly typed; for example, the built-in map function requires two inputs: one `Array` and either another `Array` or a basic type. The map function then outputs a result `Array`. Functions (such as `map` and `reduce`) can optionally accept operation in the form of a function which takes two basic types and combines them (for example + or $\star$). With a basic map-reduce data-flow paradigm, many parallel applications can be elegantly expressed in the Chestnut Designer.

The main design goal of the Chestnut Designer is to provide an easy-to-use, discoverable interface. We've taken a number of steps towards this end. First, each class of object

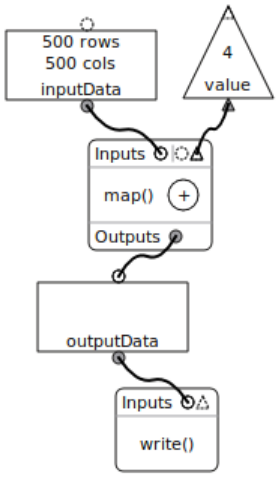

Figure 5: A sample GPU program expressed using Chestnut Designer. It adds 4 to every element of a 500x500 `array` and then writes the result to a file.

has a consistent and unique shape. `Functions` are rounded rectangles, `Arrays` are sharp-cornered rectangles, and basic types are triangles. This gives the user a visual reference to the type of the object he or she is using. Objects can have inputs and outputs, known as `sources` and `sinks`. A sink corresponds to an output, and it can accept data from a source. A source is like a fountain of data, produced by some object and available to connect to an arbitrary number of sinks. Secondly, sources and sinks have shapes to represent how they can be connected together; circle objects can only be connected to circle shaped connectors, triangles to triangle connectors and so on (see Figure 5.) Thirdly, sinks are differentiated from sources by a darker interior color. A sink can not be connected to another sink, nor can a source be connected to another source. It is possible to have a sink accept multiple different types of sources; for example, the print function can take either an `Array` or a basic type. This is represented in the GUI by having both a triangle and a circle connector next to each other. To help the user create acceptable connections between objects, Chestnut Designer prevents the user from doing things that do not correspond to valid data control flow, such as connecting a source and a sink of incompatible types or connecting two sinks to one source.

The drag and drop paradigm is central to the design. Objects are placed on the canvas by dragging them from the left toolbar, connections are made and destroyed by drags from the respective sinks and sources, and objects can be rearranged by dragging them around the canvas.

Our current prototype implementation of the Chestnut Designer supports only a subset of programs that can be expressed in the Chestnut language. We plan to further extend it to include a GUI interface for expressing a richer set of parallel functions, for explicitly representing Chestnut's `foreach` parallel construct, and for representing sequential loops. It is possible, however, that even in its final design, the Chestnut Designer may not support the full Chestnut Language in order to keep its interface very easy to use.

4.1.1 The Graphical Compiler

The Chestnut Graphical Compiler translates a GUI program constructed using the Chestnut Designer to Chestnut source code. The compiler generates Chestnut code using a traversal algorithm on the underlying graph representation of the GUI program. Each node contains information about sources and sinks connected to it, and it contains a `flatten` function that is used to generate Chestnut code specific to that node type. Thus, using a basically breadth-first traversal of the nodes, invoking their `flatten` functions in the correct order in relation to flattening their sources and sinks, produces a Chestnut source code output that correctly matches the data flow represented by the GUI program.

4.2 The Chestnut Compiler

A programmer can directly write parallel programs in the Chestnut language, or start with the Chestnut Designer and automatically generate Chestnut source code. In either case, the resulting Chestnut source code is compiled by the Chestnut Compiler, which is written in Python and uses the LEPL [3] library for parsing. The front-end of the compiler parses Chestnut source code into an abstract syntax tree (AST). The back-end currently generates C++ and Walnut Library source code from the AST.

The Walnut library can be thought of as part of the Chestnut Compiler back-end. It is an interface on top of underlying GPU interface and library functions and code. Currently, Walnut is implemented on top of CUDA and Thrust. Its purpose is to simplify the back-end by providing a target language that more closely conforms to Chestnut code.

5. EXAMPLES AND RESULTS

We evaluate Chestnut by comparing three applications each written in Chestnut, CUDA-C, and sequential C: Matrix Multiply; Conway's Game of Life (GOL); and Heat Dispersion. We compare different versions of each applications qualitatively on their "ease of programming" and quantitatively on their performance. The three applications illustrate different types of parallel grid access patterns: Matrix Multiply uses indexing and Array Element accesses; and GOL and Heat Dispersion use stencil pattern accesses. The CUDA versions of the applications are either taken directly from Nvidia's "CUDA by Example" book [19], or they are based on example code from this book. They all use 16x16 thread blocks, and the matrix multiply example additionally uses shared memory to speed up part of the computation. The full Chestnut code for GOL is listed in the Appendix. Parts of Heat Dispersion and Matrix Multiply are listed in Section 3. Full versions of all three programs are available on a web page [1] and are not included here due to space limitations.

We analyze Chestnut both qualitatively and quantitatively in comparison to sequential C and hand-written CUDA. A qualitative code comparison allows us to evaluate our main goal of Chestnut: a very easy to learn and use programming language specifically designed for non-expert programmers. The quantitative analysis compares the performance of Chestnut versions of applications to their hand-written CUDA and sequential counterparts. This allows us to show that Chestnut makes the power of GPU programming accessible to sequential programmers, and that it also is comparable in performance to hand-written CUDA code.

[1] `http://andrewstromme.com/chestnut/pmam2012`

Listing 8: CUDA GOL Initialization

```
//////  CUDA GOL initialization  //////
__global__ void  initboard(int *grid,
                           float percent)
{
  // a parallel thread needs to map
  // itself onto a GPU grid cell
  int offset, row, col;
  row=threadIdx.y+blockIdx.y*blockDim.y;
  col=threadIdx.x+blockIdx.x*blockDim.x;

  // map 2D view to its 1D index
  offset = col + row*M;

  // not all threads map to a valid
  // grid location
  if(row < N && col < M) {
    float val=curand_uniform(rand_state)
    grid[offset]= val < percent ? 1: 0;
  }
}

int main (int argc, char *argv[])  {

 int *grid, *copy;

 // allocate memory space on GPU
 // for grid and grid copy:
 cudaMalloc((void**)&grid,
             sizeof(int)*N*M);
 cudaMalloc((void**)&copy,
             sizeof(int)*N*M);

 // define blocks of 16x16 threads
 dim3 threads(16,16);
 dim3 blocks((M+15)/16, (N+15)/16);

 // initialize the board on the GPU
 initboard<<<blocks,threads>>>(grid,0.25)
```

Listing 9: Chestnut GOL Initialization

```
//////  Chestnut Initialization  //////
foreach cell in grid
  cell = (random() < 0.25);
end
```

5.1 Qualitative Comparisons

We use the Game of Life program to demonstrate the readability and ease of writing in Chestnut. The results of our quantitative analysis of GOL is very similar across the three example applications.

In comparing the number of lines of code across the three versions of all applications, we found that Chestnut versions average about 15% of the length of the CUDA version (40 vs 300 lines for Game of Life, for example), and that they are about the same length, or slightly shorter than equivalent sequential versions. Chestnut GOL, for example, is smaller than its sequential counterpart due to it not requiring nested loops, having a more concise way of initializing the grid, not having to explicitly maintain a copy, and having built-in support for wrap-around neighbor access for edge points.

More important than a comparison of the number of lines of code, is a comparison of the "ease of expressing parallelism" in Chestnut vs. CUDA. To evaluate this, we compare the initialization portion of the CUDA version of GOL to the Chestnut version.

Listing 8 shows the main part of the initialization of the CUDA version of GOL. All error detection and handling code has been removed to make the listing easier to read, also making it shorter than it is in the full implementation. We also removed code that initializes random state needed by Nvidia's CURAND library function `curand_uniform`, which is used to initialize the GOL grid cell values. This further simplifies and shortens the CUDA example from its full implementation.

In general, CUDA GPU data can be initialized on the CPU first, followed by a call to `cudaMemcpy` to explicitly copy CPU data to a CUDA GPU allocated array, or it can be initialized on the GPU using a CUDA kernel. We show the latter of these in our example because it matches the way we initialize the grid in the Chestnut version of GOL.

Comparing the CUDA version of GOL initialization to the Chestnut version (shown in Listing 9) illustrates how much more concisely this can be expressed in Chestnut. More importantly, however, it shows that in Chestnut the programmer does not need to think about GPU vs. CPU memory, does not need to explicitly allocate GPU memory, nor copy data between GPU and CPU. Additionally, a Chestnut programmer does not need to explicitly specify a parallel execution model (the `threads` and `blocks` definitions in the CUDA version), nor does a programmer need to map the parallel execution model in terms of blocks of threads onto accessing individual GPU data. In the CUDA `initboard` kernel function, each thread needs to calculate a row and column value for the data element it is accessing based on its id within a block and thread group. It also needs to check that its mapping maps onto a valid part of the CUDA data array (for some sized data there may be extra threads that do not participate in changing grid values). In addition, a programmer's two-dimensional view of the grid has to be mapped onto a one-dimensional CUDA representation of the data to obtain its offset.

In Chestnut, the initialization code is a very simple single line statement inside a `foreach` loop over the elements in the parallel grid—it is even simpler than the sequential version's initialization code. Here the programmer only needs to "think sequentially" to initialize GPU data. All parts of the underlying parallel computation model (i.e. blocks and threads) are hidden from the programmer.

Comparing the CUDA and Chestnut expressions of playing the game of life (executing multiple rounds of simulating one time-step change in the game-of-life grid), the two differ in much the same way as they do in the initialization code. The Chestnut version (listed in the Appendix) involves the programmer simply writing a loop that iterates for the total number of time-steps, calling a `foreach` statement that calls a parallel helper function to change each element based on its neighboring values. The bulk of the code in the parallel function looks identical to the code in a sequential version that computes the new value. The code to get the neighboring values is trivial compared to the CUDA version and the sequential version. The Chestnut version uses an Array Element's attribute values to read its neighboring values, and takes advantage of Chestnut's support for automatically doing wrap-around for edge points. In a sequential version, the

Benchmark	Sequential	CUDA	Chestnut	Speed-up
GOL	371.0 s	0.8 s	1.7 s	217
MatrixMult	398.6 s	0.2 s	1.1 s	347
Heat	349.5 s	1.7 s	5.4 s	65

Table 1: Average Run Time of Sequential, CUDA, and Chestnut versions of the three applications. *The data show the time in seconds. The speed-up values are the speed-up of the Chestnut version over the sequential version.*

Benchmark	CUDA	Chestnut	Slow-down
GOL	6.8 secs	15.1	2.2
MatrixMult	11.9 secs	84.9	7.1
Heat	15.4 secs	51.7	3.4

Table 2: Average Run Time of Larger versions of the CUDA and Chestnut versions of the three applications. *The data show the time in seconds. The slow-down values are Chestnut's slow-down over the CUDA version.*

programmer would explicitly have to handle wrap-around with the edge values. In the CUDA version, the programmer has to also explicitly program wrap-around for edge points, but additionally must map a parallel thread onto a portion of the grid data using its block and thread ids, and must handle maintaining a copy of the grid data that can be read from while the threads simultaneous write to a different copy the new value for each cell.

5.2 Quantitative Comparisons

We compare Chestnut implementations to their sequential and CUDA counterparts both in terms of number of lines of code, and in terms of their execution times. Since the main goal of Chestnut is making GPU programming accessible to a much larger set of programmers, its main advantage is its speed-up over an equivalent sequential version of the code. However, we also want Chestnut to perform comparably well to hand-written CUDA code, so that Chestnut programmers achieve speed-ups that are close to those they would get from programming in CUDA without the difficulty of having to program in CUDA.

We ran performance tests of sequential, Chestnut, and CUDA versions of our three example programs. The results are shown in Table 1. These data show the total execution time of each version of the three programs [2]. These data show huge improvements in the Chestnut version over the sequential version of each benchmark. Matrix Multiply shows the largest speedup value of 347 (1.1 seconds vs. 398.6 seconds), and Heat Dispersion the smallest speed-up value of 65 (5.4 seconds vs. 349.5 seconds). Part of the reason for the smaller speed-up with heat dispersion is that only four neighbor values are read to update each point value vs. eight in GOL. Matrix multiply is surprisingly fast given that each point update still requires sequential accesses to a single row and a single column. One explanation for the huge improvement in the GPU versions of matrix multiply could be a low cache hit rate for the sequential version due to the large size matrices. Because GPU memory is organized differently than CPU memory, and because of its parallelism, the GPU version does not suffer from a similar performance degradation.

In Table 2, we show larger runs of only the CUDA and Chestnut versions of the three benchmark programs. These data show that the hand-written CUDA versions are faster across all benchmarks, but that Chestnut is comparable. The largest slow-down of Chestnut over CUDA is 7.1 for Matrix Multiply, and the smallest is 2.2 for GOL. The CUDA version of Matrix Multiply makes use of CUDA's support for shared memory and synchronization between threads, and this accounts for a large part of its improvement over Chestnut.

To help explain why the longer runs of the Chestnut versions of the benchmarks are slower than the hand-written CUDA versions, we ran versions with timing code added to measure different parts of each of their executions. Our results show that the performance difference is not due to extra CPU-side overhead in the Chestnut versions, but instead points to Thrust as being the cause of much of the performance difference between the two.

Our current implementation of Chestnut has focused on implementing and designing the language, using a fairly straight forward back-end implementation that generates Thrust code. In the future we plan to investigate generating better optimized code, and anticipate that Chestnut will then perform similarly to hand written CUDA code. However, even using our current back-end implementation, Chestnut produces GPU code that is comparable to hand-written CUDA code.

Our quantitative results further support Chestnut as an easy to use and powerful GPU programming language designed to make GPU programming accessible to non-experts.

6. CONCLUSION AND FUTURE WORK

We have demonstrated that Chestnut is designed well to satisfy its main goal of making GPU programming easy and accessible to programmers with little to no prior parallel programming experience. Chestnut is designed from scratch, making it small and very easy to learn and use. A Chestnut programmer does not have to think about CPU or GPU memory, about parallel or sequential computation, or about synchronization; a Chestnut programmer can "think sequentially" about simple operations on arrays of data. With Chestnut, a sequential programmer can easily parallelize her code to take advantage of huge speed-ups in computation over a similarly structured sequential version of her application. In addition, our preliminary results show that even though generating optimized GPU code is a secondary goal of our language, Chestnut does very well compared to hand written CUDA code. Additionally, the design of the Chestnut compiler allows it to be fairly easy to port to target different CUDA libraries and to take advantage of other work in GPU optimization for grid and stencil based applications.

Future Chestnut work includes: extending the prototype implementation of the Visual Designer to support a larger set of the Chestnut programming language that can be expressed with the GUI programming interface; adding support to the Chestnut language for user-defined structured data types and for multi-dimensional parallel array of user-defined types; further work on the Chestnut back-end to generate better optimized GPU code, which may include adding support for generation of CULA or other GPU library code;

[2]Experiments were run on a system with NVidia GeForce GTX card with 480 CUDA cores running at 1.4GHz and 1535MB of memory.

adding more language-level support for sequential functions and calling sequential library code that can be interspersed with Chestnut parallel constructs, which would allow for user-defined CPU-side initialization functions; adding support for specifying parallel array sub-slices that can be iterated over in a `foreach` statement or accessed sequentially, which would allow a programmer to more easily specify distinct operations over parts of parallel arrays including initializing parts from different files; and finally, implementing and testing more parallel applications written in Chestnut to evaluate the expressiveness of our language, and to evaluate the performance of the GPU code it generates.

APPENDIX

We show the complete Chestnut Game of Life program, including all data declarations, parallel function definitions, array initializations and parallel `foreach` contexts interspersed with sequential code. Game of Life demonstrates initialization inside a parallel context, support for stencil pattern applications via easy neighbor value accesses, and an example of using parallel functions.

Listing 10: Chestnut Game of Life

```
// 1000x500 parallel grid (world)
IntArray2d life_data[1000, 500];

////////////////////////////////
// parallel function: update grid element
parallel Int game_of_life(Int2d e) {

  Int neighbor_count = e.northWest
        + e.north + e.northEast + e.west
        + e.east + e.southWest + e.south
        + e.southEast;

  Int state = 1;  // start w/live result
  if (e == 1) {   // if cell is alive
    if (neighbor_count <= 1) {
      state = 0; // dies from loneliness
    } else if (neighbor_count >= 4) {
      state = 0; // dies from overpopul
    }
  } else { // if cell is dead
    if (neighbor_count != 3) {
      state = 0; // stays dead
    }
  }
  return state;
}
////////////////////////////////
// main program control flow:

// parallel initialization:
// set 25% of elements to 1, rest to 0
foreach cell in life_data
  cell = (random() < 0.25);
end

// run 10000 rounds of GOL
for(Int i=0; i < 10000; i++) {
  // update each cell's value
  foreach cell in life_data
    cell = game_of_life(cell);
  end
}
write(life_data, "outfile.txt")
```

A. REFERENCES

[1] An auto-tuning framework for parallel multicore stencil computations. In *2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)* (2010).

[2] AMD. AMD Accelerated Parallel Processing (APP) SDK. http://developer.amd.com/sdks/AMDAPPSDK/.

[3] Andrew Cooke. Lepl. http://www.acooke.org/lepl/, 2011.

[4] Chamberlain, B. L., Callahan, D., and Zima, H. P. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications 21*, 3 (August 2007), 291–312.

[5] Han, T. D., and Abdelrahman, T. S. hiCUDA: A High-level Directive-based Language for GPU Programming. *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units 383* (March 2009), 52–61.

[6] Hoberock, J., and Bell, N. Thrust: A parallel template library, 2011. Version 1.4.

[7] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. In *SIGGRAPH* (2004).

[8] J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini, E. J. Kelmelis. CULA: Hybrid GPU Accelerated Linear Algebra Routines. In *SPIE Defense and Security Symposium (DSS)* (April 2010).

[9] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. D. Kubiatowicz, E. A. Lee, N. Morgan, G. Necula, D. A. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. A. Yelick. The Parallel Computing Laboratory at U.C. Berkeley: A Research Agenda Based on the Berkeley View. "EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-23", 2008.

[10] Khronos OpenCL Working Group, A. Munshi, Editor. The OpenCL Specification 1.2. http://www.khronos.org/registry/cl, 2011.

[11] Kjolstad, F. B., and Snir, M. Ghost cell pattern. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns* (2010), ParaPLoP '10, ACM.

[12] KlÃűckne, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., and Fasih, A. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Computing* (2011).

[13] Lee, S., Min, S.-J., and Eigenmann, R. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming* (2009).

[14] Maruyama, N., Sato, K., Matsuoka, S., and Nomura, T. Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers Categories and Subject Descriptors.

[15] Microsoft. DirectCompute. msdn.microsoft.com//directx/, 2009.

[16] NVIDIA. NVIDIA CUDA Compute Unified Device Architecture. http://www.nvidia.com, 2007.

[17] Orchard, D. A., Bolingbroke, M., and Mycroft, A. Ypnos: declarative, parallel structured grid programming. In *Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming* (2010).

[18] Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., and Kafai, Y. Scratch: Programming for All. *Communications of the ACM 52*, 11 (November 2009), 60–67.

[19] Sanders, J., and Kandrot, E. CUDA by Example: An Introduction to General-Purpose GPU Programming . Pearson Education, Publisher, 2011.

[20] Strikwerda, J. C. Finite difference schemes and partial differential equations, 2nd edition. *SIAM* (2004).

[21] Unat, D., Cai, X., and Baden, S. Mint: Realizing CUDA performance in 3D stencil methods with annotated C. In *Proceedings of the ACM International Conference on Supercomputing ICS* (2011).