Chestnut: Simplifying General Purpose Graphics Processing

Andrew Stromme & Ryan Carlson

May 10, 2010

Abstract

Graphics processing units (GPUs) are powerful devices capable of rapid parallel computation, but using them for general computing tasks can be a difficult and tedious endeavor. We have created a graphical language called Chestnut that simplifies the process of programming on the GPU. The graphical environment is supported by a readable backend language that can be translated into GPU-ready code. Our language is intuitive, discoverable, and supports common operations used in parallel computing. Runtime tests demonstrate that our code is much faster than sequential code run on the central processing unit (CPU) and is often comparable to optimized code written specifically for the GPU.

1 Introduction

There are a number of advantages to using a GPU for general purpose computation. Since GPUs were designed to process large amounts of data in parallel very quickly, a user can greatly speed up his or her runtimes by translating sequential code into code run on the GPU. Certain problems lend themselves to such a conversion, commonly called *embarrassingly parallel problems*. In this class of programs, one generally wants to perform the same operation on every element of large chunks of the data. For example, a user might want to double every element in an array, find the sum of the elements, or sort the array. Unfortunately, this translation process can be prohibitively difficult for many amateur programmers. The tools used to program on the GPU require experience with C or C++, a solid grounding in memory management, and an understanding of the interactions between the CPU and GPU.

Chestnut is our solution to these problems. It provides a graphical environment in which users can drag and drop blocks of data along with the functions that operate on them. Clearly marked inputs and outputs allow the user to connect data to functions. Thus at the graphical level we enforce the paradigm required for general purpose programming on graphics processing units (GPGPU) which pushes large blocks of data through functions that operate on each element and output the result as more blocks of data. The graphical frontend is then translated into a text-based backend language which we call *Chestnut code*. Chestnut code reinforces the data-centric model, is easy to read and write, and has a simple syntax. We finally translate the Chestnut code into an existing language called *Thrust* which is compiled and run on the GPU.

The rest of the paper is organized as follows. Section 2 discusses work related to introductory programming languages and GPGPU languages. In section 3 we detail the graphics processing unit and describe the tools used to program on it. We give a broad overview of Chestnut in section 4 before delving into the graphical frontend (section 5) and the text-based backend (section 6). Qualitative and quantitative experiments are presented in section 7 showing our code is more readable than the code constructed using current tools and provides comparable speedup benefits. Finally, we add some concluding remarks in section 8 and discuss the future of the project in section 9.

2 Related Works

We can break up designing this project into two broad categories: the pedagogy of programming languages and the paradigm presented by programming on the GPU. In other words, we need to equip ourselves with the tools to design *a* language, and then we need to figure out what is necessary to design *the* language. Since our language aims in part to be a teaching language, we look towards *Scratch*, a graphical programming language developed at the Massachusetts Institute of Technology (MIT), for guidance. Scratch pro-

vides an appealing graphical environment aimed at introducing programming to 8-16 year olds. Familiar programming constructs like conditionals and loops are present, but are presented in an intuitive manner. The authors hold that the "computational thinking" programming affords is a valuable tool that should be encouraged but is currently lacking in non-Computer Science curriculum. Most children's digital fluency is defined by "reading" but not "writing." The article discusses some of the design decisions that went into the language. To make it appeal to their target audience, the authors wanted Scratch to be more tinkerable, meaningful, and social. Additionally, we can think of any programming language as having a "low floor" (easy to begin), "high ceiling" (ability to become more complex), and "wide walls" (support for a diverse range of projects). Scratch appropriately focuses on the low floor and wide walls, arguing that a higher ceiling is outside the scope of a learning language, and if students are interesting in "raising the ceiling" they can and should look to more established languages like C or Python (Resnick et al., 2009).

Robertson and Lee (1995) draw parallels between teaching a second natural language and teaching a first programming language. The authors claim that these processes are not as different as one might expect. While much research has gone into how best to communicate the syntax and semantics contained in a natural language, teaching programming languages is sorely in need of such instruction. In traditional programming classes, there is a concentration on writing code over reading it. But a large and important part of learning a natural language is an ability to read, digest, and internalize the information being presented. So, the authors write, we must teach our students to read the code as well as write it. We note that this comes as a contrast to the MIT Scratch approach, which focuses on creation. Clearly, both aspects of a language are important. We need a language that is both intuitive to write and readable for others to interpret.

We now turn to work regarding Compute Unified Device Architecture (CUDA) (see section 3.3 for more details). Most of an undergraduate's computer science training is involved with writing single-threaded, sequential code. Since single-core CPUs are reaching limits on a hardware level, a shift is underway towards multi-core machines. Tran (2010) describes how to harness a GPU for general purpose computing using CUDA and how one might implement this into a college curriculum. A high-level overview of the CUDA model is presented along with a sample algorithm that illustrates the usefulness of teaching this new paradigm. The most useful part of this paper was the overview of the process that goes into writing a CUDA program. Given a bird's-eye view of the CUDA model, we began to get an idea of how to shape our own language. For example, in CUDA the programmer must decide how many threads will be executed. We have hidden this detail from the user since it is much too low level.

Finally, we need to consider the paradigm of actually programming for a graphics processor. Converting existing programs from a singly threaded model to the CUDA model can take significant time and experience, partly because of the mechanical conversions needed such as computing array references instead of using loops. hiCUDA is an example of an OpenMP style C preprocessor language that allows the programmer to wrap his singly threaded loops to allow them to be automatically parallelized by CUDA. It was found that examples written with the prototype hiCUDA implementation did not perform significantly worse than the same examples written directly with CUDA. Additionally, the hiCUDA examples were more concise and better fit into existing loop-based computation models that are often present in single-threaded programs. However, the language is still based around C using compiler directives and as such do not reflect or emphasize the models that one must internalize to understand GPGPU programming. Still, hiCUDA shows a similar abstraction-minded attitude and gave us an idea of what can be accomplished when building on top of existing GPU languages (Han & Abdelrahman, 2009).

3 General Purpose Graphics Processing

In this section we describe the structure of the GPU and the paradigms associated with GPGPU programming. We then introduce two tools for programming on the GPU: CUDA and Thrust.

3.1 Structure of the Graphics Processing Unit

A graphics processing unit is a dedicated piece of hardware separate from the CPU. It was originally designed for graphics processing where each pixel is

run through a series of transformations and filters before it is shown on the screen. However, the massively parallel architecture of the GPU is now being used for more general purpose programming. As with conventional multi-CPU parallel programming this is considerably different from sequential programming, and in many cases requires some knowledge of the underlying system to get good performance benefits.

Modern GPUs are structured with dozens of individual processing units organized as blocks. Blocks themselves are arranged as a grid of threads. With CUDA the programmer has to specify the grid layout of both the blocks and the threads and then from that information has to know how to access various parts of memory so that each process can do the right thing. This is difficult in part because each of these blocks has access to a small amount of fast block-shared memory (similar in speed to the cache on a CPU) and the vastly slower but much larger GPU main memory. Unlike CPUs, GPUs can not assign and run different threads of computation simultaneously on their many processing units. Instead, they run a single process (called a *kernel*) on every unit. These kernels are able to access both types of memory and perform their computations in parallel. The programmer is responsible for writing this kernel, as well as for designing the way in how threads will be assigned data. This can be a complex task that requires some low-level knowledge of how the GPU is designed as well as knowledge of special extensions to a programming language (e.g. CUDA or OpenCL) that can tell the GPU what to do.

3.2 Data Flow

GPUs lend themselves to certain types of programming which can differ from what traditional parallel models offer. For Chestnut we have focused on a single paradigm that allows for significant and semi-automatic parallelization. This model is data oriented and focuses on the interactions between functions and data. The model is also fairly common in present CUDA example code, such as fluid computation demos or vector shading. Large chunks of data are assumed, such as large 2D arrays where each element in the array is a scalar value. Functions are written to perform the same operation on each element of data with no ordering constraints. This means that the operation can be applied to each element in any order, and more importantly means that the operations can be applied in parallel. This model of programming maps quite nicely onto the core parts of GPU programming. A single program can be written as a kernel and each running instance of that kernel can operate on a small chunk of the overall data. We think that this is a decent model to support given our target audience of computational scientists that need to perform similar calculations on very large data sets (see section 4.2 for more details). Similarly, we think that it is a relatively simple paradigm to understand and internalize which should help with the target audience of people who are just learning how to program in parallel.

3.3 CUDA

Compute Unified Device Architecture (CUDA)[1] is a system developed by NVidia for performing general purpose computations on the GPU. CUDA is a series of extensions to C along with a specialized compiler. This results in a very low-level language where the programmer must know about the GPU to program for it. Because of the extensions to C and amount of fine-grained control that CUDA offers code written in it ends up being verbose and difficult to understand for someone who has not learned about how graphics cards work. This creates an enormous learning curve for aspiring GPU programmers. In CUDA the programmer is responsible for things such as allocating memory space on both the CPU and GPU (separately), copying data back and forth from these two different memories, organizing computations into thread sized chunks and allocating the correct number of threads per block and blocks per application. One example of the complexity is the concept of blocks and threads. As mentioned before, a single kernel is run many times simultaneously in the form of many threads. Instead of having a linear addressing scheme for these threads CUDA chooses to expose the underlying hardware which is organized into a 3D structure. Many threads are organized into a block, and the programmer has to choose how these threads are organized. One level higher, the blocks can also be allocated in three dimensions. This is more useful than allocating in one dimension because there are limits on how many threads can be in a block and how many blocks can be run at the same time. For example, only 65,535 blocks can be allocated in one dimension, but if the programmer uses two dimensions 65,535x65,535 blocks can be used. In Chestnut the number of blocks is determined automatically (using Thrust) so that we don't have to

[1] `http://www.nvidia.com/object/cuda_home_new.html`

expose this implementation detail to the user.

CUDA code is also verbose. This verbosity comes from all of these additional things that the programmer needs to specify but also from the format of the CUDA language itself. There are additional reserved keywords and code structures required to write a CUDA program. Everything has to be run through nvcc which requires buildsystem modifications, and there are issues with using complex C++-specific constructions in code that nvcc processes, regardless of if the GPU will or won't be running the code natively. Debugging and timing are more complicated because of asynchronous kernel execution and because of how nvcc modifies the code prior to compilation. All of this adds up to an extremely powerful and extensible language but not one that is easily learned, especially for non-C programmers.

3.4 Thrust

Thrust[2] is a C++ template library that provides convenient, optimized abstractions for many common CUDA operations. Where CUDA is very low-level, verbose, and C-like, Thrust operates on a higher level, is more succinct, and much more closely resembles C++ code. Thrust makes an introduction to GPU programming much more feasible compared to CUDA. However, Thrust code is still more verbose than we would like, is specific to C++ – a python coder would be lost – and requires knowledge of the GPU. For example, a Thrust programmer still needs to know that copying memory between CPU and GPU is very slow. Using this knowledge, there are some optimizations that one can use in Thrust to speed up the execution of code. We want to use many of the paradigms that Thrust expresses while simplifying the process such that a user will not need a deep understanding of the GPU framework. Specifically, we want to capture the essence of applying a function to a dataset and the simplification of memory access hidden by the libraries (Hoberock & Bell, 2009).

3.4.1 Vectors

There are two types of vectors in Thrust. The first is the `thrust::host_vector`[3], which is stored in the CPU memory. The other is the `thrust::device_vector`, living on the GPU. In general, a user can only initialize a vector on the CPU and then must transfer it over to the GPU. These vectors are very similar to the STL C++ vectors, and as such are inherently one-dimensional. In Chestnut we view our data in two-dimensions, but we abstract this difference away so the user never concerns himself or herself with the conversion.

If the user wants to fill a vector full of the same value, Thrust offers the `thrust::fill` function, which fills values directly into the device vector without the need to copy data over. `Fill` calls a kernel on the GPU that sets each index in the vector to the specified value. This is significantly faster than copying from CPU to GPU. There also exists a special type of iterator called a `thrust::constant_iterator` that takes only one unit of space. The user "fills" it with some constant, and a query at any index returns that value. For example, suppose a user created a `constant_iterator` called `iter` with value 2. Making the query `iter[0]` would return 2, as would `iter[100]` or any other index. Since constant space is reserved for any "size" vector, using the `constant_iterator` is a great optimization that we have automated so Chestnut users don't have to learn the quirks of a language like Thrust.

3.4.2 Functions

Thrust offers a large library of default functions. For this project we have chosen to focus on three in particular: `thrust::transform`, `thrust::reduce`, and `thrust::sort`. These three functions lay the foundation for manipulating data in Chestnut. We briefly introduce them here before delving into the translation from Chestnut code to Thrust code in section 6.4. The `transform` function convolves two vectors using a specified operator. `Reduce` consolidates a vector with a given operator. `Sort` orders the data according to a given comparator. Each of these operators and comparators can be chosen from one of several provided by Thrust or they can be customized. At present, Chestnut uses just the default operators and comparators. For a discussion specifying how Chestnut implements these functions, see section 6.2.1.

[2]The Thrust C++ Library is used throughout Chestnut and is currently the only backend that Chestnut code can be compiled into. Thrust performs its operations using CUDA `http://thrust.googlecode.com/`

[3]We use the `thrust::` notation when introducing a Thrust object to distinguish Thrust code from Chestnut code.

4 Chestnut

In this section we give a high-level overview of Chestnut. We discuss our target audience and the general applications of our language. Additionally, we state the specific goals for our language.

4.1 Overview

Chestnut is a graphical environment for parallel programming. It is composed of a graphical frontend, an intermediate language and a compiler that translates the intermediate language into Thrust C++ code. The main point of entry for users is the Chestnut GUI, with the simple code underlying this to be a small but more complicated step towards actual Thrust code. The GUI was developed in tandem with the Chestnut language and both are based on the data-oriented programming model that was explained earlier. The language is an extremely simple interface to the Thrust code and vastly simplifies the common cases while hopefully continuing to provide some flexibility. The conceptual models are enforced through the GUI and through the limited syntax offered by the underlying code. When possible the GUI restricts the actions that the user can perform to try and prevent coding mistakes. For example, the text entry box that allows the user to rename variables only accepts characters in an order that translates cleanly to C-style variable names (only alphanumeric characters and can not start with a number). These components work together to provide an interface for a style of parallel programming that can be mapped easily into lower level Thrust and CUDA code that can use the GPU.

4.2 Target Audience

Chestnut is meant to be a relatively simple introduction to parallel GPU programming. We expect little to no experience with C++ and see two categories of users with this type of knowledge. The first group is computer science/programming students who have not had any exposure or introduction to GPU programming before. The second group is non computer science programmers who could benefit from parallelizing their computations. This might include natural science researchers or computational scientists who have written number-crunching programs with parallelizable algorithms in single-treaded scripting languages.

With both cases the learning curve can be too steep to encourage these programmers to learn about GPU programming. With the underlying Chestnut code we are interested in having a similar syntax to Python or JavaScript, where the code is similar to C but without some of the extra syntax such as namespaces, headers versus source files, includes and other complexities. We hope that a person with some knowledge of programming and with a parallel problem could use Chestnut to write a basic solution to this problem.

4.3 Target Applications

At this point Chestnut is targeted towards embarrassingly parallel problems. We reason that there are a vast number of embarrassingly parallel problems out there that have not been written to take advantage of the GPUs because of the learning curve associated with GPU programming. Those are our target problems. Additionally, we are not convinced that the extra complexity of fine-grained synchronization, kernel specification and block/thread allocation is necessary. This is mainly because our target audience is not the advanced computer science programmers who would want to understand enough of those complexities to take advantage of them.

4.4 Goals

With both the target audience and the target applications in mind we have identified a few important goals for Chestnut. Primarily it should expose the data-focused model that can be translated to the GPU easily. This can be done by imposing limitations on the GUI and on the core language syntax. Secondly we want Chestnut to be modular. This means separating the GUI interface from the underlying language, and providing a compiler for this underlying language that could be changed in the future to not depend on Thrust or CUDA should either become eclipsed by other GPU programming environments. Chestnut needs to be discoverable. The GUI frontend and its visual and drag and drop interface is a direct result of this, where the user can see how things are supposed to be connected. Lastly we are still interested in achieving a speedup when compared to the CPU. Although Chestnut is not about the fastest possible runtime on the GPU compared to other GPU programs we still want it to be faster than running the same computations on the CPU because even with

chestnut there is still a learning curve to working on the GPU.

5 Chestnut Frontend

The Chestnut frontend is designed with the data-centric programming model in mind. It is composed of a canvas upon which objects can be placed and connected together to represent the dataflow within a program. Components can be dragged from the left sidebar onto the canvas and then can be connected to create the flow of data in the program. From the GUI the program can be translated into Chestnut code and then run – which implies compiling into Thrust, building that Thrust code, and running it on the GPU then feeding back the response – without having to drop into the command line.

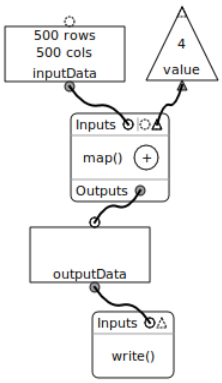

Figure 1: A sample program in the chestnut GUI that adds 4 to every element of a 500x500 `DataBlock` and then writes the result to disk.

5.1 Primitives

The frontend has three main types of objects, of which two are data containers. The simplest data container is a `Value`, composed of just a single scalar. `Values` are used primarily for the second input to a map function or the result calculated by a reduction function. `DataBlocks` are data containers that contain multidimensional data. Currently, only 2D arrays of arbitrary dimensions are implemented, but this could be expanded to three dimensions. Lastly there are `Functions` which operate on the two data containers. `Functions` are well specified and strongly typed; for example, the map function requires two inputs: one `DataBlock` and either another `DataBlock` or a `Value`. The map function then outputs a `DataBlock`. Functions (such as `map` and `reduce`) can optionally accept an operation type which takes two scalars and combines them (for example `+` or `*`). With these three types of primitive objects and the connections between them we think many embarrassingly parallel problems can be elegantly expressed in Chestnut code.

5.2 Interface Concepts

With Chestnut one of the main goals was to have a discoverable interface. We've taken a number of steps towards this end. Firstly, each class of objects has a consistent and unique shape. `Functions` are rounded rectangles, `DataBlocks` are sharp-cornered rectangles and `Values` are triangles. This gives the user a visual reference to the type of the object he or she is looking at rather than having to parse and understand some text. Objects can have inputs and outputs, known as `sinks` and `sources`. A sink corresponds to an input. It can accept data from some source. A `source` is like a fountain of data, produced by some object and available to connect to an arbitrary number of sinks. Continuing with the theme of a discoverable interface, sources and sinks have shapes to represent how they can be connected; circles can only be connected to other circles, triangles to triangles and so on. Similarly, 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 a DataBlock or a Value. This is represented by having both a triangle and a circle next to each other. To help the user create acceptable connections between objects, the Chestnut frontend specifically prevents the user from doing things that are impossible, such as connecting a source and a sink of incompatible types or connecting two sources to one sink.

The drag and drop paradigm is central to the Chestnut GUI. 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. The Chestnut frontend forces the data-centric GPU model that we illustrated earlier because of the limitations enforced with connections and the high level atomic functions such as `sort`, `reduce`, and `map`.

5.3 Translating GUI to Chestnut Code

In order to translate the graphical representation of a given program to Chestnut code that we can translate into Thrust code we need a robust method of traversing the graph and constructing correct code. It is difficult to decipher a "starting point" in a graphical program, so we start by choosing a node at random from the graph. This node may be data or it may be a function. Each node knows its contribution to the code and also knows what sinks and sources it is connected to. Additionally, every node has a `flatten` function that builds up the Chestnut code. Thus, once we have selected our starting node, we call its `flatten` function which in turn calls the `flatten` function of all its sinks, then provides its contribution to the Chestnut code, and finally calls the `flatten` function of all its sources. Since all sinks are "flattened" before the node's contribution is added and all sources are flattened after that point, we ensure that the order of commands in the resulting Chestnut code is as the GUI layout intended.

We have so far been vague about what a node does to provide its contribution. This of course depends on the node's type and its specific function. But first we need to discuss the structure of Chestnut code as it is being built up. Every Chestnut program can be broken up into variable declarations and function calls (for Chestnut code specifications see section 6.2) and so we can create two lists of strings, one for declarations and another for calls, that will eventually be written out to disk in the Chestnut code file. Thus as a node contributes to the list of strings, it adds to the declarations or calls.

So, when a `DataBlock` or `Value` is encountered its contribution will be to the variable declaration list. The node uses the context provided by a presence or lack of sinks to decide whether or not it has been initialized and adds an appropriate variable declaration (see section 6.2.2 for a description of variable declarations in Chestnut code). If `flatten` is called on a `Function`, the contribution will be to the list of function calls. As with the `DataBlock` the node can gain context from its sinks and sources, allowing it to add the appropriate parameters to the Chestnut code string.

In choosing a random node from the graph and traversing it as we are, we make two crucial assumptions. The first is that any connections a `Function` has will either be a `DataBlock` or `Value`. Similarly, any connections to or from a variable must be a `Function`. This decision reinforces the data-centric model of pushing data through functions and getting data back. We can enforce this at the graphical level by not allowing two `Functions` or two variables to be connected. The second assumption is that the graph is connected – i.e. there are no disjoint subgraphs. Since we choose a node at random and traverse the graph from there, we can only translate code that is reachable from that initial node. Thus we don't guarantee any correctness if disjoint subgraphs (including single, unconnected nodes) exist.

6 Chestnut Backend

Once the GUI to Chestnut translation is finished, we have Chestnut code. In this section we describe the backend language we have created and discuss the process that brings our Chestnut code to executable object code.

6.1 Process

The translation from Chestnut to executable is a four step process. First, we use Flex[4] to tokenize the Chestnut code. For each valid word in the language, a token is created. For example, if Flex reads the word `map` it outputs the `TOKMAP` token. While Flex can return just a token without any extra context, it also has the ability to recognize and then save the input in a string or integer and pass that information along as well. In this case, one might have the variable `var1` which gets mapped to the *identifier* token `ID` and additionally saves the string `"var1"` to pass along to the next stage. Every valid word or character in the file (including braces, parentheses, etc.) must be tokenized to be accepted and passed on. If a token is not found, Flex will exit with a syntax error, providing a simple first level of error checking.

As Flex tokenizes the file, it passes each token along to Bison[5] for processing. Bison generates an LALR(1) (one token Look-Ahead read Left to right producing a Rightmost derivation) or a GLR (generalized LR) parser. Using very simple regular expressions, Bison generates code that analyzes each token and matches it to predefined acceptable sequences of tokens. For example, suppose the input code for a reduction was `"result = reduce(+,data)"`[6]. The resulting sequence of acceptable tokens to check for

[4] `http://flex.sourceforge.net/`
[5] `http://gnu.org/software/bison/`
[6] See section 6.2 for a full description of syntax.

would be "ID ASSIGN TOKREDUCE LPAREN OP COMMA ID RPAREN"[7]. Once a valid sequence of tokens is identified, we gather the appropriate information (variable names, operators, parameters) and pass them into a utility function (part of a utility class) that takes care of writing all the necessary Thrust code to disk. Once Thrust code has been written it can be compiled to object code using the CUDA compiler, `nvcc`.

The utility class knows what files we eventually want to write to, keeps a symbol table, and has all the necessary functions to write appropriate Thrust code. We have broken each file into four general regions: headers (`#includes`), function declarations, the `main` function, and function definitions. Each region is represented as an STL vector of strings. Thus, any header files we want to include in a given file or code we want to write to `main` get stored as entries in their respective vectors until the Chestnut code is finished being parsed. Once our parser reaches the end of the file, we execute a single write to disk.

The symbol table is implemented as an STL vector of `symbol_entry` structs. A `symbol_entry` is a simple container for four fields. The first is the name of the function or variable. The second is that object's type, either `int` or `float`. The third field is the category of the entry. It can be either a `FUNCTION` or two different variable types. If the variable is a single value, we assign the category `VARIABLE_SCALAR`. If the variable is a vector (or `DataBlock`, in the frontend terminology) we assign it `VARIABLE_VECTOR`. Finally, each entry has a scope. This field is not presently being used and is by default set to zero.

6.2 Specifications

We currently support two basic variable types, two variable categories, and seven default functions. Here we discuss them and specify each of their uses. This section will introduce the reader to basic Chestnut syntax and give a sense of the level of abstraction used.

At this early stage, the only data types we support are standard C++ `int` and `float`. Considering our target audience and what we anticipate them using Chestnut for, we believe that in an overwhelming number of cases these basic number types will be sufficient. Users will be acting primarily on large numerical datasets. In the future, we would like to implement more default types and allow users to define their own types, described in section 9.

Before a function is called, variables used by that function must be defined. As discussed earlier, variables can be either vectors or scalars. If a vector is going to be assigned data before being used in a function, it must be initialized using a `foreach` loop or a `read` command, discussed in section 6.2.2. Otherwise, if data is going to be used as the result of a function, we must still declare it, but need not specify anything except its type, name, and category. To declare a variable in this way, the general syntax is

```
[type] variableName [category];
```
[8]

Chestnut currently has seven functions that operate on data: `map`, `reduce`, and `sort` all manipulate the data in some way; `foreach`, `read`, `write`, and `print` comprise the Input/Output interface of our language. The general syntax for a function call in Chestnut is

```
result = functionName(
        param_1, ..., param_k,
        inputData);
```

The result of the operation is always stored in a variable (either a scalar or vector) to be used later. Every function also operates on some block of data. These two specifications enforce the data-centric model of programming. Note that it is easy to modify data in place by setting both the result and input to the same variable. Each $param_i$ is an extra parameter that specifies some option of the function. There can be zero or more such parameters.

6.2.1 Manipulator Functions

Let's consider the `map` function, the most complex of the default functions. A `map` takes a block of data and applies the same modification to every element. For example, we could add 2 to every element in an array. Our `map` also works as a convolution operator, applying an operator to corresponding elements in two arrays. Formally, given two 2-dimensional arrays `A,B` convolved into a resulting array `C`, each with indices `i,j`, the resulting array would have the property `C[i][j] = A[i][j] + B[i][j]` for all `i,j`. The result of a `map` is a vector. The first parameter is the operator (e.g. `"+"`). The second parameter can either

[7] `OP` is short-hand for "operator," satisfied here by `"+"`

[8] Bracket notation is used to indicate there a defined set of choices for the given identifier. Here, `[type]` could be `int` or `float`, `[category]` could be `vector` or `scalar`.

be a single value (e.g. `"2"`) or a `DataBlock`. The last parameter is, as always, the input data that will be mapped over. So, if the end-user wants to modify a data block `inputData` by adding 2 to every element the function call would be

```
inputData = map(+, 2, inputData);
```

If instead the user wants to multiply two blocks of data, `input1` and `input2`, and store the result in `output`, the call would be

```
output = map(*, input1, input2);
```

The `reduce` function accumulates every element of a data block using a specified operator and stores the value in a scalar. For example, a user could use the function to take the sum of every element in an array. The function takes only two parameters, an operator and an input vector. The output is a scalar. Using the example from before, to sum over a data block `input` and store the result in the scalar `reduced`, we have the syntax

```
reduced = reduce(+, input);
```

The `sort` function sorts a block of data according to the given comparator. The output is always a data block. The first parameter is the comparator to use (e.g. `"<"`) while the second is the data to be sorted. Suppose a user wanted to sort a block of data `input` in descending order and wanted to store the result in that same variable, the syntax is

```
input = sort(>, input);
```

We see that each of the manipulator functions Chestnut offers takes a vector as input, performs an operation on every element and returns some value to be stored in another variable. We note that these functions can be strung together into long sequences of maps, reduces, and sorts. Indeed, we anticipate much of the value of Chestnut to arise from the ability to sequence functions in a pipeline to efficiently compute results.

6.2.2 Input/Output Functions

To initialize a vector, Chestnut offers two options, one of which is the `foreach` construct. This is a simplified for-loop construction that traverses every element of the vector and allows the user to create a formula to dictate what value is stored in each cell. The general syntax for a `foreach` declaration is

```
[type] variableName numRows numCols
    foreach (ForeachExpression);
```

Every *ForeachExpression* starts with `value =` and then some right-hand side expression. The `value` keyword is shorthand for the current index of the data block in the for loop. Thus if the progress of the loop had reached *currentRow*, *currentCol*, then `value` would correspond to `array[`*currentRow*`][`*currentCol*`]`.

The right-hand side of the equation can be any arithmetic expression (e.g. 2+4) combined with a set of reserved words. Note that these words are not reserved outside of this context, so a user oblivious of these rules would not be in danger of name conflict errors. The user can reference the current row or column using the `row` and `col` keywords, respectively. To reference the total rows or total columns a vector we use the `maxrows` and `maxcols` keywords, respectively. Finally, users can use the keyword `rand` to generate a random number. Since this function invokes the C++ *rand*() function, this will be an integer between 0 and the maximum random number. This offers the highest level of customization for the user's random number. So, to fill a 10x10 vector of random real numbers between 0 and 1, the full statement would be

```
float data 10 10
    foreach (rand/RAND_MAX);
```

Chestnut also allows its users to read data from disk and write data to disk. We use a very clear, simple syntax for the files that are read from and written to. The first line contains the number rows in the data, then the number of columns. After this the data is listed in a space-separated format. Recall from above that data reads are called only when variables are declared, so the current syntax follows the declaration pattern. In the future we may want to allow data reads at any time. To read from a file `"infile"` and store it in a vector `data` of integers, we have the syntax

```
int data read("infile");
```

The output functions have a different syntax from other functions. To write a block of data, `outdata`, to a file, `outfile`, the code in Chestnut would be

```
write(outdata, "outfile");
```

Similarly, Chestnut provides a function to print the contents of a data block to standard output. The output is formatted into a two-dimensional array format, printing a newline character before starting each new row. To print a data block `outdata`, one uses the syntax

```
print outdata;
```

The input and output functions Chestnut provides allows the user to view progress in between a series of computations and to read blocks of data from disk and later write them out for storage. When dealing with enormous data, it seems that reading and writing to disk will be most useful. Still, for benchmarking and various other operations, the `foreach` construct is convenient.

6.3 Sample Program

We have provided in-depth specifications of Chestnut code but have not yet given an example of a full program written in the language. Here we take the opportunity to detail a program written in Chestnut code from start to finish, given in Code Snippet 1.

First, we initialize `input1` and `input2` with a random number and data from file, respectively. We then need to declare two vectors, `sorted1` and `sorted2`. Below the declarations, we sort the initial vectors into their sorted analogues, the first in ascending order, the second in descending order. To get a peek at the computation in progress, we print out both vectors.

Once the data are sorted, we want to convolve over the vectors using the `"*"` operator and store them in `mapped`, which we then write out to the file `"outputdata"`. Finally, we reduce over a sum and print out the result, which is stored in `reduced`. This sample program shows the ease with which a user can quickly populate and operate on sets of data. Our programming model is clearly at work, moving data through functions and storing the results in more data.

6.4 Translating Chestnut Code to Thrust

The process of converting Chestnut Code to Thrust code consists of expanding single lines into many lines of code, declaring extra variables that are encoded implicitly in Chestnut, and making some assumptions that a Thrust programmer would not need to. In certain circumstances, we can also optimize some commands. In this section we discuss Thrust code that is produced from samples of Chestnut code. We also use this as a vehicle to discuss other design decisions and assumptions we made regarding the translation.

Code Snippet 1 Sample Chestnut code for a program that adds two sorted vectors together, then sums over their mapped result. Note that line numbers do not include blank lines or wraparound. We do this to focus on the lines of actual code rather than the code's organization.

```
1   float input1 100 100 foreach
            (value = rand/RAND_MAX);
    float input2 read ("inputdata");

    float sorted1 vector;
    float sorted2 vector;
5   sorted1 = sort(<, input1);
    sorted2 = sort(>, input2);

    print sorted1;
    print sorted2;

    float mapped vector;
10  mapped = map(*, sorted1, sorted2);
    write (mapped, "outputdata");

    float reduced scalar;
    reduced = reduce(+, mapped);

15  print reduced;
```

6.4.1 Variable Declarations

Any time a variable is declared in Chestnut there are up to five possible variables that may need to be declared in Thrust. Consider a vector defined using the syntax `int data vector;` which just creates a vector for later use. This causes a utility function to create a host vector and a device vector along with variables to contain the number of rows and columns of the object. None of these variables are initialized immediately, but they must be declared to be referenced later in the program. The Chestnut code `int data scalar;` corresponds to just a single variable in Thrust code.

When translating a `foreach` construct, we must create variables for the host, device, rows, and columns, but we still have two options after those declarations. If the user specifies a row, column, or the random keyword in the $foreachExpression$, then we must create a double for-loop that iterates over every row and column and fill the host vector, then copy the host vector to the device. But if the user makes no reference to those reserved words we

can optimize the process by using the `thrust::fill` function (described in section 3.4.1). Since all other expressions (those containing `maxrows` and `maxcols` and all arithmetic expressions) are constants, we can simply pass the expression to the function and have it move straight to the GPU. Since there is no need to copy data from CPU to GPU, this operation is much faster.

When the user calls the `read` function, we create the usual four variables and then must also reference a special `DataInfo` struct. In the Thrust code, we call a templated library function we wrote that reads the data into a host vector, fills out row and column information and packs them into a special struct. This is then returned to the Thrust file where it gets unpacked.

With so many variables in the Thrust code, we need a convenient naming scheme. The base of every Thrust variable name is the name given from Chestnut. Then some descriptive suffix is added to enable us to reference the variable and to ensure there are no naming collisions. Thus, if a Chestnut vector was `input`, the associated host vector in Thrust would be `input_host` and the number of rows would be `input_rows`.

6.4.2 Manipulator Functions

Where name collisions caused conflicts when declaring variables, in-place operations cause conflicts when calling functions in Chestnut. Every function in Chestnut has an input and an output, and unless those two are the same object we guarantee that the function will not modify the input object. But some functions in Thrust modify the data in place. For example, if a user writes the Chestnut code `output = sort(<, input)`, we must copy the contents of `input` to `output` and then run an in-place `thrust::sort` on `output`.

Similarly, when using the `map` function, a conflict arises when the user sets both the modifier and the output as the same object, as in `modifier = map(+, modifier, input)`. Since we want to guarantee `input` is not modified, we generally set the destination (here, `modifier`) to an empty vector of the correct size. Thus, left unmodified, the translation results in `modifier` being overridden and then simply filled with the `input`. We need only switch the order of the input and modifier to resolve the problem, because then both input and destination will be the same object and our algorithm will treat this as an in-place modification. These problems are indicative of the level of abstraction we strive for. General purpose graphics processing requires a different way of thinking about the data, and to really establish and enforce this paradigm requires some bridges between existing code structures and our abstractions.

6.4.3 Output Functions

With the output functions `write` and `print` we come to one of the most crucial assumptions. To write a data block out to disk or to print an object, the data must be on the CPU memory. Since the Chestnut programmer is not informed about the CPU/GPU boundary and by design has no way of affecting the location of the data, we must assume that the most recent copy of the data is on the GPU and thus needs to be copied over to the CPU on every output function call. Thus every print and write, even if they are consecutive calls, necessitates a slow copy. Future work could look at optimizing this limitation out.

7 Experiments

Experiments with Chestnut take two forms: qualitative and quantitative. The first allows us to evaluate the readability of code. We compare the Chestnut code automatically translated from the graphical interface against equivalent Thrust code and CUDA code. We use this evidence to argue that our language is much more accessible than the alternatives. The quantitative experiments help to validate the usefulness of Chestnut. We show that Chestnut code is significantly faster than sequential code executed on the CPU and is approaching the speed of handwritten CUDA code.

7.1 Qualitative

To demonstrate the readability and ease of writing in Chestnut, we present a simple example involving a single `map` operation and show how the verbosity increases dramatically as we translate the code into equivalent Thrust and then into CUDA code. We see in Code Snippet 2 that in three commands the user can define an array, fill it with a value, add one to every element in that array, and print out the result. The commands are clear, succinct, and immediately accessible to a new user.

Let us contrast the Chestnut code with equivalent Thrust code. Code Snippet 3 contains the result of

Code Snippet 2 Chestnut code mapping over each element of a 10x10 array, adding 1 to each element, and printing out the result.

```
float data 10 10 foreach (value = 2);
data = map(+, 1, data);
print data;
```

automatic translation from Chestnut to Thrust code, formatted slightly to fit the page. The first feature to notice is how many more lines of code are necessary to do the same thing. In the Chestnut code, we need only declare the variable and use it without worrying about about whether its contents reside on the CPU or GPU. That abstraction disappears when we begin to program in Thrust. First host and device vectors must be declared and populated. Then a verbose `transform` function must be called wherein the user must worry about bounds and optimizations like the `constant_iterator` (discussed in section 3.4.1). Finally, the data on the GPU must be copied to the CPU and printed out.

But Thrust is still a strong improvement in terms of usability over the low-level CUDA code. In Code Snippet 4 we see handwritten CUDA code equivalent to the previous two programs. While there are not markedly more lines of code in the CUDA program compared to the Thrust program, each line is nearly indecipherable for a user not well versed in C programming. Among the things a CUDA programmer needs to worry about are pointers, using CUDA's set of memory allocation tools, and running a kernel that uses obscure syntax. Recall the simple task of adding 1 to a small array. We believe that such an easy to understand concept should have accompanying code that is both easy to understand and easy to program. Chestnut makes this a reality.

Code Snippet 3 Thrust code mapping over each element of a 100 element array, adding 1 to each element, and printing out the result.

```
1   int main() {
      int data_rows = 10;
      int data_cols = 10;

      // Memory on host and device
5     host_vector<float> data_host;
      device_vector<float> data_dev
          (data_rows*data_cols);

      // populate data
      thrust::fill(data_dev.begin(),
          data_dev.end(),  2);

      /* Begin Map Function */
10    thrust::transform(data_dev.begin(),
          data_dev.end(),
          thrust::make_constant_iterator(1),
          data_dev.begin(),
          thrust::plus<float>());
      /* End Map Function */

      // print out data
      data_host = data_dev;
      for (int r=0; r<data_rows; r++){
15      for (int c=0; c<data_cols; c++){
          std::cout
            << data_host[r*data_cols+c]
            << " ";
        }
        std::cout << "\n";
      }
20    std::cout << "\n";

      return 0;
22  }
```

7.2 Quantitative

An important step in determining whether Chestnut is a viable alternative to other general purpose graphics processing languages is benchmarking. We ran our automatically generated Thrust code against handwritten sequential C++ code running on the CPU and optimized, handwritten CUDA code. We tested `Map`, `Reduce`, and `Sort`. The first two operations were performed on arrays containing 8192x8192 = 67,108,864 elements. The sorting operation ran on an array of size 1024x1024 = 1,048,576 elements. In all cases, arrays were initialized with random floats between 0 and 1. Each operation was executed 100 times per run. Experiments were conducted using an NVidia GeForce GT 230M card with 48 CUDA cores and 1GB of memory running at 1100 MHz. The average runtime per operation over five runs is presented in Table 1. Note that the times below reflect the runtime of a single operation, not all 100 operations. Additionally, speedups between sequential and Thrust code and between Thrust and

Code Snippet 4 CUDA code mapping over each element of a 100 element array, adding one to each element, and printing out the result

```
int main() {
  int* host;
  int* dev;
  int N = 100;

  cudaMallocHost((void*)&host,
     N*sizeof(int));
  cudaMalloc((void*)&dev,
     N*sizeof(int));

  for (int i=0; i<N; i++){
    host[i] = 2;
  }

  // Copy the array host to dev
  cudaMemcpy(dev, host,
     N*sizeof(int),
     cudaMemcpyHostToDevice);

  // run map kernel on device
  map<<<1, dim3(N)>>>(dev, N);

  // copy back dev to host
  cudaMemcpy(host, dev,
     N*sizeof(int),
     cudaMemcpyDeviceToHost);

  for (int i=0; i<N; i++){
    printf("%d ", host[i]);
  }
  printf("\n");

  return 0;
}

__global__ void map
   (int* array, int cols){
  int row = threadIdx.x;
  int col = threadIdx.y;

  array[row*cols + col] += 1;
}
```

CUDA code are featured.

We see from the results that in all cases we have excellent speedup compared to sequential code. Our generated Thrust code is anywhere from 13 to 23 times faster than handwritten sequential code. When we look at the Thrust code against optimized CUDA code, we (in general) fare respectably. When considering the map and sort operations, our code is at most three times slower. The reduction operation, however, is almost sixty times slower than the CUDA code. It would appear that the default Thrust `reduce` operation is not nearly as optimized as it could be. We know that the `reduce` function requires copying the result from GPU to CPU memory and it may be the case that the optimized CUDA is getting around that. While we are satisfied with the `map` and `sort` speedups, we need to look more carefully about how to handle reductions to get the best runtimes.

A few aspects of our experimental methods should be made explicit. Since Thrust function calls are asynchronous, it is difficult to time the actual executions of our operations. Thus our timer was the UNIX time[9] utility. To isolate the function runtimes from the overhead of copying memory from the CPU to the GPU, we timed the average execution of that copy without any actual function call and subtracted that time away from the runs when we issued function calls. The overhead, however, was nontrivial. Since we are using Thrust vectors instead of C-style pointers, the CUDA code spent less time allocating space than the Thrust code. Specifically, in the `map` experiments, memory allocation using Thrust took (on average) a full 71.9 percent of the runtime compared to only 30.4 percent using CUDA. So while each operation is comparable, CUDA definitely beats Thrust on memory transfer. We don't view this as a big problem because we expect the average use case of our language to involve allocating the data and then running many consecutive computationally expensive tasks. As computation increases, the significance of memory transfer decreases.

The CUDA and sequential code was a mix of code we wrote and precompiled executables to which we did not have access to the source. All sequential code was handwritten, and since sequential maps, reductions, and sorts are very straightforward, we find this reasonable. The C++ STL `sort` function was used. We wrote the CUDA `map` function, which consists of a single kernel call (as shown in Code Snippet 4). The CUDA `reduce` and `sort` were precompiled and thus we were limited to the given array sizes.

[9] `http://www.opengroup.org/onlinepubs/9699919799/utilities/time.html`

	Sequential to Thrust Speedup	Thrust to CUDA Speedup	Sequential (ms)	Thrust (ms)	CUDA (ms)
`Map`	**23.1**	**1.01**	545	24.3	24.1
`Reduce`	**19.2**	**58.0**	266	13.9	0.24
`Sort`	**13.7**	**3.32**	409	29.8	8.97

Table 1: Runtime statistics comparing automatically generated Thrust code to Sequential code run on the CPU and optimized CUDA code. Map and Reduce were run on arrays containing 67,108,864 elements while Sort was run on 1,048,576 elements. Each operation was executed 100 times per run and the average runtime per operation is presented in milliseconds. Speedup in the first column is calculated as $\frac{sequential\ time}{Thrust\ time}$, the second column is $\frac{Thrust\ time}{CUDA\ time}$.

8 Discussion

In this paper we have presented our graphical language Chestnut aimed at simplifying general purpose graphics processing. Taking advantage of the GPU can greatly speed up runtimes but, with the tools available now, require more knowledge and overhead than many users are willing or able to put in. With Chestnut we provide a simple, discoverable, intuitive interface that allows amateur programmers or scientists perform many common data-centric tasks without learning about the details of the GPU. Chestnut can also serve as a teaching tool to introduce programmers to the mindset of parallel programming. Our language is much faster than sequential code running on the CPU and is in most cases comparable to handwritten, optimized CUDA code. While our language is limited in this early stage and will never be as fast as well-tuned CUDA code, the resulting Chestnut code is much more readable and easier to follow than the alternatives.

9 Future Work

As we have acknowledged, Chestnut is still in its early stages. We have a working solution to many of the problems that make GPGPU programming difficult, but there is still much room for expansion and improvement. In this section we present some options for the future of this project.

9.1 Custom Functions

A significant limitation with the current Chestnut implementation is that it doesn't support the creation of custom functions from within Chestnut code or from the GUI. This complicates the use of Chestnut for applications with some data inter-dependence. One example of such an application is Conway's Game of Life[10] which requires knowledge of the eight neighboring squares to calculate the game state for any given square. Custom functions would need significant limits on how they could access memory to ensure that the data-centric GPU model is followed, but we think that they are possible. A custom function would have a limited syntax and would be able to assign the values to each bucket based on the value at that bucket and the values of other nearby or arbitrary buckets. A sample syntax for custom functions is presented in Code Snippet 5 where the specific keywords `above, below, left, right` correspond to the values in the array at those positions relative to the current bucket and the keyword `value` represents the value in the output array of the current bucket.

Code Snippet 5 Proposed syntax for a custom function in Chestnut code.

```
function average(Input in, Output out) {
    value = above + below + left + right;
    value = value / 4;
}
```

9.2 More Functions and Datatypes

Chestnut currently supports integers and real values (floats); both are very primitive datatypes. Users would benefit from a larger array of types to choose from, including doubles and strings as well as a `pair` type capable of combining two values into a single datatype. We would also like to expand the list of default functions and expand the functionality of those we currently provide. For example, if a user had control of a `pair` or some arbitrary n-tuple, he or

[10] `http://en.wikipedia.org/wiki/Conway's_Game_of_Life`

she could run different `sort` functions along different subfields. We would also like to provide a mechanism by which users could define their own functions and data types. This would greatly increase the functionality and usefulness of Chestnut. The syntax for these functions is something we need to carefully consider before implementing. We need to stick to our goals of being intuitive and discoverable while making the process as configurable as possible.

9.3 Refine GUI

As features are added to the Chestnut language it is important to continue to find ways to have them cleanly map into the GUI. Custom functions should appear alongside predefined functions and should follow the same semantics for advertising their available inputs and outputs as sinks and sources. The GUI also currently lacks any way of saving or loading a workspace. It would additionally be important to streamline the build process. Currently it takes quite a few steps and it should be closer to a one click process if the user is unfamiliar with the different parts that make up the Chestnut system.

9.4 For Loops

In the current language, there is no looping structure to execute multiple function calls without explicitly laying each one out. For example, to run 100 maps a user needs to place 100 `map` calls and all the necessary intermediate connections. This is of course infeasible in its current state, especially in the graphical environment. In the future we would like to provide a `for` loop that specifies the number of times to execute a section of code. To implement this functionality, we first need to get the backend to recognize the construct and figure out where data needs to be copied to provide reliable and well-defined results. Then we need to implement the appropriate frontend support and decide on the most intuitive way to present a looping construct to the user.

References

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

Hoberock, J., & Bell, N. (2009). *Thrust: A parallel template library.* Available from `http://www.meganewtons.com/` (Version 1.2)

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., et al. (2009, November). Scratch: Programming for All. *Communications of the ACM*, *52*(11), 60–67.

Robertson, S. A., & Lee, M. P. (1995, December). The Application of Second Natural Language Acquisition Pedagogy to the Teaching of Programming Languages – A Research Agenda. *SIGCSE Bulletin*, *27*(4), 9–12.

Tran, Q.-N. (2010, April). Teaching Design & Analysis of Multi-Core Parallel Algorithms Using CUDA. *Journal of Computing Sciences in Colleges*, *25*(4), 7–14.