CUDA Web API
Remote Execution of CUDA Kernels using Web Services

A Thesis
Presented To
The Faculty of California Polytechnic State University
San Luis Obispo

In Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science

by
Massimo J. Becker

June 14, 2012
Committee

Title: CUDA Web API
Author: Massimo J. Becker
Date Submitted: June 2012
Committee Chair: Christopher Lupo, Ph.D
Committee Member: Alex Dekhtyar, Ph.D
Committee Member: Gene Fisher, Ph.D
Abstract

CUDA Web API

Remote Execution of CUDA Kernels using Web Services

Massimo J. Becker

Massively parallel programming is an increasingly growing field with the recent introduction of general purpose GPU computing. Modern graphics processors from NVIDIA and AMD have massively parallel architectures that can be used for such applications as 3D rendering, financial analysis, physics simulations, and biomedical analysis. These massively parallel systems are exposed to programmers through interfaces such as NVIDIA’s CUDA, OpenCL, and Microsoft’s C++ AMP. These frameworks expose functionality using primarily either C or C++. In order to use these massively parallel frameworks, programs being implemented must be run on machines equipped with massively parallel hardware. These requirements limit the flexibility of new massively parallel systems. This paper explores the possibility that massively parallel systems can be exposed through web services in order to facilitate using these architectures from remote systems written in other languages. To explore this possibility, an architecture is put forth with requirements and high level design for building a web service that can overcome limitations of existing tools and frameworks. The CUDA Web API is built using Python, PyCUDA, NumPy, JSON, and Django to meet the requirements set forth. Additionally, a client application, CUDA Cloud, is built and serves as an example web service client. The CUDA Web API’s performance and its functionality is validated using a common matrix multiplication algorithm implemented using different languages and tools. Performance tests show runtime improvements for larger datasets using the CUDA Web API for remote CUDA kernel execution over serial implementations. This paper concludes that existing limitations associated with GPGPU usage can be overcome with the specified architecture.
Acknowledgements

I would like to acknowledge all the people who helped make this thesis possible. If it were not for their support, this would not have been possible.

I would like to thank Stephan Holsapple for his help learning the intricacies of Python when I was swimming in the deep end as well as Andrew Guenther for his work showing that GPU JSON deserialization is possible.

Many thanks go to my family. My mother, Michelle, who set the bar high, and my grandparents, Jo and Bill for their endless encouragement. I especially want to thank my father, Dr. Theodore J. Becker for his help revising this document, his encouragement which keeps me going through good and bad, and his endless support in everything I do.

Thanks to my committee members, Dr. Alex Dekhtyar and Dr. Gene Fisher, for their great feedback and encouragement on this project. I could not ask for greater teachers and mentors than those I have encountered at Cal Poly.

A very big thank you goes out to my advisor, Dr. Chris Lupo, for his support, encouragement, and guidance, which have kept me moving at all times, invoating in the right areas, and from overfrosting an already sweet cake. It has been a privilege.
# Contents

List of Tables

List of Figures

1 Introduction 1

2 Background 4

2.1 Massively Parallel Hardware and Architectures 4

2.1.1 NVIDIA 5

2.1.2 AMD 6

2.2 Massively Parallel Frameworks 7

2.2.1 NVIDIA CUDA 7

2.2.2 OpenCL 8

2.2.3 Microsoft C++ AMP 9

3 Related Work 10

3.1 Folding@Home 10

3.2 Globus Toolkit and Infrastructure as a Service 11

3.3 GridCuda 12
3.4 PyCUDA ......................................................... 13
3.5 SGC Ruby CUDA ............................................. 14
3.6 BOINC ......................................................... 14
3.7 Hadoop MapReduce ........................................... 15

4 Architecture .................................................. 17

4.1 Analysis ...................................................... 17
  4.1.1 Overview ................................................. 18
  4.1.2 Programming Language ................................. 18
  4.1.3 Device Locality ......................................... 19
  4.1.4 Utilization ............................................... 20
  4.1.5 Costs .................................................... 21

4.2 Requirements ................................................. 21

4.3 High Level Design ........................................... 24
  4.3.1 Client .................................................... 26
  4.3.2 Server ................................................... 27
  4.3.3 Worker ................................................... 27
    4.3.3.1 Single-Instance .................................... 28
    4.3.3.2 Multi-Instance ..................................... 28
  4.3.4 Work Dispatch / Handling ............................... 28
  4.3.5 Kernel Execution ....................................... 29
  4.3.6 Cleanup ................................................ 30

4.4 Request Structure .......................................... 30

4.5 Response Structure .......................................... 32
4.6 Summary ........................................... 33

5 Implementation ...................................... 35

5.1 Server ............................................ 35

5.1.1 Web Service Paths ............................. 36

5.1.2 Request Handling .............................. 37

5.1.3 Data Serialization .............................. 38

5.1.4 Work Queue .................................... 41

5.1.5 Execution ...................................... 42

5.1.6 Result Handling ............................... 45

5.2 Workers .......................................... 46

5.2.1 Tracking ....................................... 47

5.2.2 Request Dispatching ......................... 47

5.3 Client / User Interface .......................... 49

6 Tutorial ............................................ 51

6.1 Requirements ................................... 51

6.2 Installation ...................................... 52

6.3 Server Startup ................................... 52

6.4 Worker Startup .................................. 53

6.5 Operational Modes ............................. 54

6.6 API .............................................. 56

6.7 Request Format .................................. 56

6.8 Result Format ................................... 57

6.9 Errors ........................................... 58
List of Tables

4.1 Request Dictionary Structure ........................................ 31
4.2 Parameter Dictionary Structure ..................................... 31
4.3 Successful Response Dictionary Structure ....................... 33
4.4 Error Response Dictionary Structure ............................... 33

5.1 List of Server Paths .................................................. 37
5.2 List of Server Paths - Cont. ........................................ 38
5.3 Request States .......................................................... 42
5.4 Worker Information .................................................... 48

6.1 Worker POST Dictionary Structure ................................. 53
6.2 Configuration File - Operational Mode Settings ................ 55
6.3 Error Codes and Definitions ......................................... 59

7.1 Evaluation Implementations .......................................... 62
# List of Figures

4.1 High Level Design .................................................. 25  
4.2 System Diagram .................................................. 25  
4.3 Client Communication ............................................. 27  
4.4 Worker and Work Handler Communication ......................... 29  
4.5 Example of JSON Encoded Request .................................. 31  
4.6 Example of JSON Encoded Request .................................. 32  
5.1 CUDA Cloud Screenshot ............................................. 50  
6.1 CUDA Cloud Worker\Compute Server Management .................. 54  
6.2 CUDA Cloud Configuration .......................................... 56  
6.3 Valid JSON Encoded Request without Kernel Code .................. 57  
6.4 Valid JSON Encoded Response ........................................ 58  
7.1 Maximum value difference between GPU and CPU matrix multiplication . 64  
7.2 Matrix Multiplication Runtime using Python, PyCUDA, and CUDA Web  
    API ................................................................. 68  
7.3 Matrix Multiplication Runtime using C and CUDA C ................. 69  
D.1 Scalar Parameter - type: int ........................................ 83
D.2  Scalar Parameter - type: float ............................................. 83
D.3  Array Parameter - type: 1D array of floats .............................. 83
D.4  Array Parameter - type: 2D array of floats .............................. 83
D.5  Array Struct Parameter - type: 2D array of structs ................. 84
D.6  Output Parameter - type: 2D array of floats ........................... 84
D.7  Empty Output Parameter - type: 1D array of floats ................. 84
D.8  Empty Output Parameter - type: 2D array of floats ................. 84
Chapter 1

Introduction

Advances in GPU hardware over the last several years have given rise to the ability to use GPUs for what has become known as general purpose GPU computing. General purpose GPU computing (GPGPU) is the ability to use graphics hardware for general computation in software for financial analysis, physics simulation, alternative graphics rendering, bioinformatics, and other applications that can benefit from a large-scale parallel computing platform.

In the early days of GPGPU computing there were no advanced frameworks that could be applied to general computational problems. The only APIs available to programmers were those used for graphics programming. Due to this limitation, programmers would try to manipulate their computational needs to resemble that of graphics problems. Once the representation of the problem had been manipulated into a new form, the programmer could then use a GPU to accelerate the runtime of their application. This technique added additional complexity and development time for the programmer. [15]
The next logical evolution in GPGPU computing is a framework which allows the programmer to directly represent their problem as one that is solvable utilizing parallel hardware. This level of abstraction is shown in frameworks such as OpenCL, CUDA, and C++ AMP. [12, 17, 23] These frameworks allow the programmer to avoid additional complexities when trying to represent problems as graphics problems and instead, represent them in their native form. An added benefit is the generalized representation of the graphics hardware that the framework presents to the programmer.

While the evolution in GPGPUs has developed to a point that makes their use easier for the programmer, there are still some limitations with the existing frameworks for use with GPGPUs. As I will point out in the background section, each framework is limited to a few programming languages. Additionally, each framework requires that the hardware it will run on be directly connected to a GPGPU over a high-speed bus. This second limitation is constricting and will be discussed in later scenarios.

The first scenario that shows limitations is one in which the programmer has executed the application on a machine with a GPGPU and now wishes to execute it remotely from another machine. This is to say that CPU piece of the program will be executed on the local machine but the GPU piece of the program should be executed on a remote machine that contains a GPGPU. This scenario might occur given that within a small group of machines, there is one machine which contains a GPGPU and many more which do not. Providing that the GPGPU on the single machine is not being fully utilized, it may be beneficial to allow the other machines access to the single GPGPU to increase utilization of the GPGPU.

The second scenario is one in which a programmer has an operation that could be
parallelized on a GPGPU device but where the runtime improvement does not justify the purchase of GPGPU. In this event it would be beneficial for the programmer to be able to dispatch this parallelizable operation to a remote service or device that contains a GPGPU that can perform the operation. This would eliminate the need for the machine performing the operation to have one or more costly GPGPU cards.
Chapter 2

Background

GPGPUs are available with different architectures from different vendors. GPGPU architectures share some common features but each may or not share the same programming paradigms or frameworks. This chapter explores GPGPU architectures as a basis for showing some of the benefits and recent developments of GPGPUs. Frameworks for GPUGPU are also discussed in this chapter to show how GPGPUs are utilized for general computation by programmers.

2.1 Massively Parallel Hardware and Architectures

GPGPUs are currently available from two different manufactures NVIDIA and AMD. Each of these manufacturers uses a different architecture and framework to allow programmers to use GPUs for general computation. This section discusses the architectures from each manufacturer and their use.
2.1.1 NVIDIA

NVIDIA’s first GPGPU was the GeForce 8800 GTX that used their GF80 chip architecture [15]. This was the first chip that had a set of programmable shader’s that had read and write access to memory and a managed cached shared memory. These features were added to allow for GPGPU computing using the CUDA framework. NVIDIA continues to produce CUDA capable architectures and currently produces the Fermi architecture which adds double precision floating point capabilities to GPGPUs.

One of the unique aspects of CUDA is its ability to allow the programmer to choose the type of memory for data storage and access [25]. This gives the programmer the ability to adapt their problems to fit the architecture of the GPU and gain additional run time speedup by utilizing faster parts of the GPU memory.

In addition to its line of GPGPUs, NVIDIA offers clustered solution GPGPU cards in it’s Tesla compute cluster [20]. Tesla clusters consist of multiple host machines containing one or more GPGPU cards in each machine. These machines are visible to each other over a local network and provide a method for programmers to expand from one GPGPU to multiple GPGPUs within a datacenter setting. A Tesla cluster is exposed to the programmer in the same way that multiple cards within a single machine would be exposed. Thus, this allows the programmer to dispatch specific operations to specific GPGPUs within the cluster.
2.1.2 AMD

AMD’s GPGPU computing platform started with the Radeon line of graphics cards, with each card having multiple processing cores called stream processors. These cards with parallel architectures are exposed to the programmer through OpenCL and now through Microsoft’s DirectX implementation of C++ AMP. [12, 21]

One of the limitations of early GPU architectures is the need for the program data to exist on the GPU memory before the GPU computation can begin. This is because GPUs traditionally only have access to the memory that is on the card. In order to get data onto the GPU memory, it must be transferred from the CPU memory over a graphics bus to the GPU. The added overhead of this transfer is large for operations with a short runtime on the GPU, but for operations with longer runtimes, this overhead becomes negligible. As a result of this added transfer time, AMD began producing an APU that has both CPU and GPU like hardware on the same chip and gives both architectures direct access to the same system memory. [9] These newer APUs are also exposed to the programmer through the same OpenCL and C++ AMP frameworks.

This thesis does not explore AMD’s GPGPU architecture in depth, but acknowledges them to show that any added benefit provided by exposing CUDA through web services, could potentially be applied to frameworks that operate on AMD’s architectures.
2.2 Massively Parallel Frameworks

In order to use the general computation power of GPUs, there must be some framework that allows the programmer to make use of the GPGPUs from each GPU manufacturer. This sections reviews the frameworks available for use with GPGPUs and how they accomplish their respective tasks.

2.2.1 NVIDIA CUDA

Compute Unified Device Architecture is a framework offered by NVIDIA to enable its GPUs to perform general purpose computation [25]. CUDA exposes GPGPU capabilities to programmers through an extension to the standard C and C++ language. Special syntax is added when using CUDA that allows the NVCC compiler to recognize GPU specific code. The compiler can then recognize segments of a program that are intended to run on the CPU and also which segments are intended to run on the GPU. Code that will be executed is then compiled into a format that the GPU can recognize and linked with the CPU code before an executable is produced.

CUDA allows the programmer to execute their function in multiple threads to be executed on the GPU device. Threads in CUDA are organized differently than on the CPU where multiple threads can share the time of one CPU core. Using CUDA, one thread will execute for each processing core. Threads are then organized into a block and multiple blocks organized into a grid.

As part of the process of executing a CUDA program, data that will be operated upon must be copied to the GPU for execution, and then resulting data must be copied back after execution. CUDA provides the programmer a unique method for
addressing memory on the GPU by giving each thread an index that can be used as a basis for indexing memory on the GPU. This indexing is the basis for dividing large sets of data to be operated on by CUDA’s massively parallel architecture.

One of the beneficial features of a GPU is its fast memory, that allows it to perform faster computation than the CPU on large data sets. CUDA exposes special types of memory on GPU that when utilized to its full potential results in faster runtimes than use of the GPU without special types of memory.

CUDA exposes very powerful graphics hardware with the ability to process large datasets and utilize fast access memory to speed up runtime. In order to utilize the benefits of CUDA, an NVIDIA GPU must be present in the host computer where the program will be running. The other requirement of CUDA is that code must be written in C or C++ and compiled using the NVCC compiler. A faster program runtime can be realized using CUDA providing that these requirements are met.

### 2.2.2 OpenCL

OpenCL is a GPGPU framework that is similar to CUDA in that it allows a programmer to utilize the massively parallel architectures of a GPU [14, 17]. The framework is exposed to the programmer as an extension to the C programming language. Programs that make use of OpenCL must have a GPU present in the host machine that will be used for execution, however, it is more flexible because it can make use of both NVIDIA and AMD GPUs. OpenCL adds flexibility to the programmer due to its ability to function across platforms, but is still limited by its language requirement and requirement for a directly connected GPU.
2.2.3 Microsoft C++ AMP

Microsoft has also recently released a framework as part of DirectX that exposes parallel and massively parallel GPU architectures for GPGPU computing called C++ Accelerated Massive Parallelism [12]. C++ AMP exposes GPUs in a way that hides architectural hardware details from the programmer. Abstracting the details of the hardware away from the programmer allows for easier development and better integration with C++. Creating a layer of abstraction also allows the hardware layer to change without needing to change the software layer. Programs written for existing architectures can then benefit from future improvements in hardware. C++ AMP also makes use of parallel CPU architectures with multiple cores, and special instruction sets for processing multiple pieces of data simultaneously. CUDA and C++ AMP are similar in the way that they allow special indexing methods for addressing data but with C++ AMP there is no guarantee in the way each operation will be dispatched to the hardware. There is also no ability with C++ AMP for the programmer to access specialized memory to gain additional speedup in the way that CUDA does.
Chapter 3

Related Work

GPGPUs have already seen wide adoption in different fields to accelerate general computation. Some of the projects that make use of GPUs are applications that work with large amounts of data, while other applications are new frameworks that build off of existing frameworks to add flexibility or new features. This chapter discusses some of the many projects using GPGPUs today and how they add new functionality to overcome some limitations associated with GPGPUs.

3.1 Folding@Home

Folding@Home is an example of a distributed system that makes use of both CPU and GPGPU resources for computation [7]. Folding@Home simulates the folding of proteins in order to better understand how proteins fold or misfold leading to diseases. Simulating protein folding requires simulating the interaction of each atom that comprise the protein’s atomic structure. At the scale of 1ns for a 10ms simulation of a whole protein, it requires significant computation. A serial implementation of
the folding algorithm would require years of computation on current hardware. In order to perform this complex simulation in a realistic amount of time, Folding@Home chose to use a distributed systems approach. By doing this, Folding@Home has been able to build a distributed network of systems that can run 10,000 years of current CPU time in approximately 1 year.

Initially Folding@Home built its distributed network using a CPU based client that allowed users to process segments of data. The system has since expanded in order to increase the number of clients and processing power available by introducing a GPU client. By integrating the use of GPGPUs into their system, Folding@Home was able to realize more floating point operations per second with fewer clients than with a CPU implementation. This use shows that it is possible to dispatch workloads to remote systems with attached GPGPUs. As a result, it is possible to realize the same computational benefits using remote systems with GPGPUs as seen in systems with directly connected GPGPUs.

3.2 Globus Toolkit and Infrastructure as a Service

The Globus Toolkit is a set of software libraries that allow programmers to expose existing infrastructure to build new systems [10]. Building software systems requires building the infrastructure to support those applications such as underlying file storage, messaging servers, and computational servers. A programmer must specify the functionality of each of these underlying infrastructures in order to function as part of the whole system. Utilizing existing infrastructures as a service eliminates some of the redundancy of building new systems, but a programmer must still specify the
more intricate details of how that infrastructure fits into the whole. The Globus Toolkit allows the programmer to specify this level of detail that allows the system to make use of remote resources in a more general way. The benefit that this provides is a method for programmers to specify the functionality of remote infrastructure resources.

### 3.3 GridCuda

GridCuda is a grid computing architecture that is specific to CUDA capable GPGPUs [18]. The grid consists of multiple host machines each with any number of CUDA devices installed. In the traditional CUDA runtime API, function calls are made to allocate and copy data to the GPU, a kernel launch is then called which executes the kernel function, and then additional function calls are made to copy resultant data back from the GPU. Using GridCuda each of these calls is intercepted by the GridCuda API before they are sent to the CUDA device. The calls are then sent over a network to a remote device and executed on that device. This provides the illusion to the programmer and host code that the remote CUDA devices are installed locally and that the programmer has access to all CUDA devices in the grid as if they were part of the host machine which originated the CUDA calls.

By creating this solution, GridCuda created a layer of abstraction that allows the programmer to utilize CUDA GPGPUs without the requirement of having them installed locally. The results from GridCuda also show that faster runtime for complex programs is possible, even with the added overhead of network communication. The requirement of using the C programming language still remains with GridCuda but
allows existing CUDA programmers to utilize more resources without having to learn the functionality of another framework.

### 3.4 PyCUDA

All three GPGPU frameworks covered in this paper, C++ AMP, CUDA, and OpenCL, require the programmer to develop their entire program using either C or C++ [16]. Using these languages allows faster computational time, but does not allow the programmer to use more flexible or more appropriate higher level languages. PyCUDA eliminates this problems by exposing CUDA through Python.

PyCUDA is able to expose CUDA C in Python by taking a representation of the CUDA kernel that will be run on the GPU, passing to the NVCC compiler for compilation and wrapping the resulting output in a Python module. Defining the kernel function with PyCUDA can either be done by using the string representation of the C kernel function or by building the syntax tree for the kernel function. Runtime code generation (RTCG) is one of the added benefits that allows programmers to make better use of the GPU. The ability to change the kernel module during runtime allows a program to dynamically change its use of the GPU.

One of the difficulties of writing CUDA programs is the need for handling how data is transferred to and from the GPU. PyCUDA makes this task easier by providing access to functions that will examine the data to be transferred to the GPU and then implicitly allocate the correct amount of memory.

The restriction to the C language is partly overcome by PyCUDA. Using PyCUDA allows a programmer to structure most of the program in Python and adds the ability
to build the kernel with a syntax tree in addition to CUDA C. The result of PyCUDA is a more flexible framework for the use of CUDA.

### 3.5 SGC Ruby CUDA

Ruby CUDA is very similar to PyCUDA. It exposes CUDA functionality in the Ruby programming language in a way that allows programmers to utilize the massive parallelism of CUDA and the functional language features of Ruby [26]. Ruby CUDA lets programmers create kernel functions by writing CUDA C code which maintains the same runtime API and driver API expected in CUDA C. There are no other points of integration in Ruby CUDA at this time like the syntax tree building ability in PyCUDA but that does not deter from the benefits of Ruby CUDA. The result of using this technology is essentially similar to PyCUDA in that it allows the programmer to work in another language which alleviates part of the C or C++ requirement of CUDA.

### 3.6 BOINC

BOINC is a system that allows users to contribute their computing resources to an open project[5]. Scientists can create a BOINC project with a central web URL and have their data computed by volunteers rather than purchasing expensive compute infrastructure to perform computation. Each BOINC project contains an application and a version that allows BOINC to track which application a client has and what type of computation will be performed. The creator of the BOINC project can provide
source code that a client can compile, or a ready application that the client can run. Each client that has volunteered to contribute resources is tracked by a database associated with the BOINC project. Work units are dispatched to clients with the appropriate client application where computation is performed and resulting data is returned. Transfer of work units is accomplished using HTTP file transfer and web services using XML. In order to make use of special hardware architectures, clients can compile their own version of the application that targets their own architecture to provide better performance.

This type of system allows work to be done on remote workers and allows the project to track and change the remote work that is done. Several different projects already make use of BOINC including Folding@Home and SETI@Home. This system allows clients to perform remote execution on distributed data, and allows project owners to distribute new applications that specify the work that will be performed. However, the amount of time it takes to distribute a new version of an application is unknown and while a new application may be available, it is the client’s responsibility to update their system with the latest version. BOINC does show however, that it is possible to use web services to manage remote workers and dispatch work units. Additionally, we see that using BOINC provides researchers with added computational power without the need for it to be part of an expensive in house infrastructure.

3.7 Hadoop MapReduce

Hadoop is a project well known for its implementation of MapReduce[2, 3, 4]. The MapReduce process allows large sets of data to be operated on and then reduced to a
smaller dataset before the process is repeated. After a sufficient number of iterations of the MapReduce cycle, the input dataset will converge to a smaller resulting dataset which is then returned at the end of computation. Hadoop MapReduce provides an API for programmers to specify the operation that is mapped to the dataset and how the dataset will be reduced. Requests are submitted to the MapReduce system where they are queued for execution. Clients can poll for results or can provide an optional callback where data will be sent upon completion.

Hadoop MapReduce is able to compute on large sets of data very quickly by reducing the dataset after each mapping phase. Additionally, the mapping function can be specified at runtime allowing custom operations to be performed on the data. MapReduce is currently used in many large scale applications to handle extremely large datasets that would not be feasible for computation using single CPU compute device. MapReduce shows that it is possible for programmers to dispatch large datasets to remote resources where specified computation can be performed faster than on a local system. Some datasets are well suited for computation using the MapReduce pattern, however, there could be some cases in which MapReduce is not suited to the problem domain. As such, we can see that one of the more powerful aspects of MapReduce is its ability to perform runtime defined remote computation.
Chapter 4

Architecture

The CUDA Web API architecture is designed to overcome the existing limitations of current GPU tools such as those discussed in the related work chapter. This chapter puts forth an architecture for the CUDA Web API by analysing limitations of existing tools and using those to derive a set of requirements for a new architecture. Once a set of requirements has been defined, we can build a high level design that meets the requirements and lays the foundation for implementation.

4.1 Analysis

In order to build the requirements for a new tool for use in GPU computing, we analyze the tools reviewed in the background and related work chapters. This analysis discusses limitations and why those limitations should be overcome to build a more flexible tool.
4.1.1 Overview

We have covered many different frameworks, tools, and implementations that relate to the GPGPU computing. Each framework has exposed a set of highly parallel architectures that, when utilized in the scope of programs that require SIMD computation, can result in runtime speedups greater than 100%. These benefits outweigh most of the costs and requirements associated with GPGPU programming but does not remove them from the picture. It is possible, however, that with added tools and further work we can begin to reduce some of those costs and requirements.

4.1.2 Programming Language

Requiring use of C or C++ to implement GPGPU code limits the flexibility of programmers to use many features offered by higher level languages. While C and C++ are considered to be very fast languages, they are not considered to be the easiest to use for development. They do not include many modern paradigms such as functional programming and reflection. Two of the tools addressed in this paper have managed to eliminate the language requirement of GPGPU programming to a certain extent, but have not eliminated it completely. The ability to write CUDA code and use it in Python or Ruby should help to ease the development of GPGPU programming while promoting further adoption. Ultimately these tools allow the programmer to work in another language that may better suit their problem domain.
4.1.3 Device Locality

The GPGPU frameworks covered in this section assume that both GPU and CPU code segments of a program will be executed within the same machine. This requires that the GPU and CPU be collocated in the same machine under control of the same operating system. These frameworks do not describe any scenarios by which the GPU would be located within a second machine where GPU code segments could be sent for execution.

Folding@Home works around this limitation by requiring the user to install a GPU client that runs specific GPU code. Workloads from the system are dispatched in a 3-step process. First, data is dispatched from an assignment server to the GPU client. Next, the client simulates protein folding on the GPU using a preset kernel function. Finally, data is returned to a collection server to complete the process. While this process comes close to overcoming the locality requirement, it does not overcome it completely. The process allows work to be dispatched to remote GPGPUs but the function performed by the device is fixed to what has been specified in the client program.

We can illustrate this further by defining static and dynamic GPU kernel functions in the context of this paper. A static function is one which is written by the programmer, compiled and does not change for the life of the program. A dynamic function in this case is one in which the function can be specified or changed many times throughout the life of the program. Folding@Home only allows a static kernel function to be specified in its GPU client. In order for the kernel operation to be changed, a new client must be written, compiled, and re-distributed out to each
client. This is possible, but when considering the many GPU clients that are part of
the system, the cost of re-distribution begins to grow.

Using a framework like PyCUDA allows a program to change its kernel function
during runtime. By integrating PyCUDA into a system like Folding@Home, the cost
of changing the kernel function on remote systems could be reduced. This would allow
dynamic kernel functions and data to be dispatched to remote systems containing
GPGPUs.

4.1.4 Utilization

In single CPU systems, the processor may not be under full load at all times but
it will be performing some operation as part of the idle process of the operating
system. In systems containing a CPU and one or more CUDA GPUs, the CPU will
experience similar loading but the GPU will have different characteristics. A CUDA
GPU will only perform operations during data copy and when a kernel function has
been launched. This means that unless kernel functions are being continually launched
or are running, the GPU will not experience full utilization when compared to the
CPU. Essentially, while the host is turned on, there will likely be time that the GPU
is not utilized for computation.

With the additional costs associated with running GPGPU hardware, it is most cost
effective to utilize the GPU hardware at the highest possible level of utilization. This
allows operators to make the best use of expensive hardware.
4.1.5 Costs

There is an essential cost of entry to performing CUDA GPGPU computation. The simplest cost is the addition of a single CUDA capable NVIDIA card. While this cost varies depending on the performance of the CUDA card, it is still a fixed cost card that may be performing varying amounts of computation.

As with web servers, there may be burst times of computation for which large numbers of CUDA-capable cards would be beneficial but which may be unneeded during periods of low utilization. The pricing model for CUDA GPUs is simple but may not be the best model to suit users with bursting needs for CUDA computation. A similar pattern can be seen with the rise of cloud services that provide infrastructure as a server. Cloud services allow the user to pay for more resources when needed and then scale back when not needed. As such, a similar model for GPUs may prove beneficial and more cost effective for bursting GPU usage.

4.2 Requirements

After analyzing the current requirements, limitations and uses of GPGPUs, we can begin to see the need for a single solution to alleviate these restraints. A set of new requirements for this solution can be derived from the previous analysis.

**Dynamic Kernels.** The first requirement is for the solution to allow a programmer to specify dynamic kernel functions during runtime. This allows the behavior of the program to change without re-compilation. Meeting this requirement may allow systems like Folding@Home to reduce the costs associated with changing kernel
functions.

Cross Platform Functionality. A second requirement stems from the GPGPU frameworks covered earlier. CUDA will only work with CUDA capable devices from NVIDIA but is available across operating systems. OpenCL will work across capable devices from AMD and NVIDIA and functions across operating systems. Finally, C++ AMP works across AMD and NVIDIA GPGPUs and some newer Fusion CPUs but is only available on the Windows operating system. All three of these frameworks have some ability to work across either hardware or software platforms. As such, a full solution is required to allow functionality across multiple software and hardware architectures. Fulfilling this requirement is most important on the host computer because it will allow users to execute programs from a wider variety of systems. This also allows the programmer to work in a broader range of languages which may fit the problem domain better than C or C++.

Remote Execution. The third requirement derives from uses of the Globus Toolkit and Folding@Home. The Globus Toolkit allows programmers to utilize existing remote infrastructure for computation. Folding@Home allows a system to make use of remote GPGPU resources. From this we see that a solution must allow remote execution GPGPU kernel functions. Fulfilling this requirement will allow programmers to send GPGPU workloads to any existing infrastructure with GPGPU capabilities. Creating this layer of separation also allows programmers to build solutions without needing in depth detail of each system where the solution will be run. By allowing code to be executed on remote GPGPUs, the solution is set to allow any GPGPU
to be targeted or swapped out in place of a different GPGPU. Essentially, this sets the GPGPU up to be exposed as a service in the same way that the Globus Toolkit exposes infrastructure as a service.

Some older requirements must be maintained from existing frameworks in the creation of a new solution. The following requirements from existing tools maintain functionality that will allow a solution to enhance existing GPGPU solutions but not redefine it’s functionality.

**Complex Data Types.** Fourth, a programmer must be able to specify complex data types, such as structs, that kernel functions can use during kernel execution. In combination with requiring remote GPGPU execution, this will allow the programmer to send data structures to the remote GPGPU without requiring any prior knowledge of the structure.

**Error Reporting.** Finally, executing a kernel function on a given set of data must result in success or failure so that the caller can process the resulting data appropriately. This last requirement coincides with the remote execution of GPGPU kernel functions. Executing on remote hardware necessitates the ability to determine whether execution was successful, failed, or lost. This knowledge in a program dictates the next steps to take during runtime.

With this set of requirements, we can specify a new solution that will expand the existing capabilities and limitations of GPGPU computing.
4.3 High Level Design

Given the requirements derived from the problem analysis, we can now begin designing a new system for use in GPU computing. This system exposes the CUDA framework through a web service while meeting the requirements laid out previously. Using CUDA allows programmers to work with an existing GPGPU solution with known functionality and documentation. CUDA is restricted to working with NVIDIA GPUs but the remote nature of this solution will allow kernel functions to be executed from any host client computer.

In order to facilitate the remote execution of GPGPU kernel functions, a web service is used to provide remote communication. A web service for this solution works in the following manner:

1. In order to provide the necessary functionality, this system builds on and uses the features exposed by the CUDA Driver API but not the CUDA Runtime API.

2. The CUDA Runtime API makes use of CUDA easier but can only be used if the complete code base is compiled before runtime with a CUDA compiler.

3. Functionality exposed by the CUDA Driver API is available without use of a CUDA compiler and makes it possible to receive kernel functions and execute them at runtime.
Figure 4.1: High Level Design

Figure 4.2: System Diagram
4.3.1 Client

First, the client machine builds up a request with a specified structure that is encoded using a serialization library. Because CUDA is best suited to process large amounts of data, the serialization library has to produce minimal output. For this reason, data is serialized using JSON. Each request consists of a dictionary which contains the kernel code to be executed, a list of the data and its structure definition that are passed as parameters to the kernel, and the grid and block sizes of the kernel. The client specifies the kernel function code by sending its standard CUDA C representation and data formats, such as C struct definitions, to the web service. The structure definition acts as data contract that specifies what data types will be sent to the web service and which data types will be expected in return. This data contract, used specifically during the deserialization and serialization phases of execution, allows the web service to correctly map the input data to the kernel function and map results back as output data.

The client then establishes a connection to the web service that is connected to a GPGPU or set of workers with GPGPUs and submits the request for execution. Workers are discussed in further detail in the Worker subsection that follows. Depending on the implementation, the client either blocks until the request has been finished and a response is returned, or immediately receives an identifier for the request which can be used to poll for the response of the request. Alternatively, the client may also submit a path to another web server where a callback can be made giving the response upon completion of the request.
4.3.2 Server

Once a request is received, the web server validates the request data, ensuring that the JSON structure is correct. During validation, the server immediately notifies the client if any part of the request is invalid or if the JSON format is incorrect. Depending on implementation, the web server follows one of two paths. The first path is a single-instance mode that is simpler and does not make use of a work queue. The second is a multi-instance mode that uses a work queue and dispatches workloads from the queue to each worker.

4.3.3 Worker

The server can be configured to distribute requests to remote workers in the event that server experiences heavy loads or additional compute power is needed. A remote worker is required to have a CUDA capable GPU and the same CUDA Web API implementation as the server. By having, the same implementation as the server, a worker can perform the same execution as the server would when operating without the optional workers.
4.3.3.1 Single-Instance

If the server is running alone, in a single instance mode without additional workers, then the server will execute the request and return the result to the client immediately without being stored.

4.3.3.2 Multi-Instance

When the server is running in multi-instance mode with workers, a more complex flow is followed to dispatch received requests to workers. The same request validation is performed after receiving a request as in the single-instance mode. At this point, the request is stored in a work queue to be executed by the workers. A separate process is started at this point to handle dispatching the received request from the work queue and storing the result for later pickup. After the server has placed the request in the work queue and the separate process is running, the server returns a generated ID for the request to the client that can be used when polling for the result.

4.3.4 Work Dispatch / Handling

Multi-instance mode requires the use of a separate process or work queue handler to allow the web server to operate asynchronously and handle web service requests without being interrupted by other tasks. The work handler is responsible for pulling unprocessed jobs from the work queue and sending them to a worker for execution. After pulling requests from the work queue, the work handler serializes them in a format with additional data for parameter deserialization by the worker.
4.3.5 Kernel Execution

The execution phase is the same for both stand alone servers and worker execution. Execution begins by reading the request parameters and deserializing each one using the structure definition so that the data is correctly laid out in GPU memory. The structure definition contains a direction parameter that specifies what action must be taken to put the data in memory. Parameters specified as In or InOut must be copied to the GPU before execution while Out parameters only need memory space allocated for resulting data. The kernel code is then compiled using a CUDA compiler and the resulting binary is set up on the GPU for execution. The grid and block size is retrieved from the serialized request data and used to specify the grid and block size for the kernel execution. The kernel name that is specified in request data is then executed once all request data has been loaded into its respective place. After execution has finished all parameters marked as Out parameters are copied from the device and serialized. The serialized result is then sent back to the request sender. In the single-instance case, the sender will be a client and in the multi-instance case, the
sender is the work handler responsible for dispatch. This process can be done using PyCUDA or another CUDA binding library to specify kernel functions and copy data to the GPU.

### 4.3.6 Cleanup

When the kernel has finished executing, the web service will respond to the host client with a successful result including data or a failure indication. The requirement for remote execution can be met following this process by using a web service in conjunction with GPGPUs.

### 4.4 Request Structure

The request sent from the client to the server is encoded using JSON to minimize the markup sent to the server and decrease server load. JSON allows for dynamic types, but requests in this case must maintain a strict high-level structure while items lower in the structure can be more loosely defined. The top-level item sent in the request is a JSON serialized dictionary containing five keys with a specific data structure mapped to each key. See Table 4.1 for a list of the keys and their types.

A request with empty code and kernel name blocks and without any parameters, grid dimensions, or block dimensions is represented as shown in Figure 4.5.

The parameters list is a list of dictionary items that represent each parameter and provide some information for the server about how to handle the parameters. See Table 4.2 for a list of keys and their types.

The serialized representation of a parameter’s data is a list if a vector or array
<table>
<thead>
<tr>
<th>Key Name</th>
<th>Data Types</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>string</td>
<td>CUDA Kernel Code \CUDA C Code defining the kernel function(s) and structs used within kernel functions</td>
</tr>
<tr>
<td>kernel name</td>
<td>string</td>
<td>The name of the global CUDA kernel to execute (some code sections will have multiple kernels specified, this will decide which kernel to execute)</td>
</tr>
<tr>
<td>grid</td>
<td>dictionary</td>
<td>A dictionary that contains the grid dimensions for the kernel execution</td>
</tr>
<tr>
<td>block</td>
<td>dictionary</td>
<td>A dictionary that contains the block dimensions for the kernel execution</td>
</tr>
<tr>
<td>params</td>
<td>list</td>
<td>A list of dictionary objects with each dictionary representing a parameter and its information</td>
</tr>
</tbody>
</table>

Table 4.1: Request Dictionary Structure

```json
{"code": "", "kernelname": "", "grid": {"x": null, "y": null}, "block": {"x": null, "y": null, "z": null}, "params": []}
```

Figure 4.5: Example of JSON Encoded Request

<table>
<thead>
<tr>
<th>Key Name</th>
<th>Data Types</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>string</td>
<td>A JSON serialized representation of the data for the parameter</td>
</tr>
<tr>
<td>struct</td>
<td>string</td>
<td>A structure definition defining the type of the data represented in the parameter data field. Similar to a C struct</td>
</tr>
<tr>
<td>direction</td>
<td>string</td>
<td>Acceptable values: ”in”, ”out”, ”inout”. Describes whether the parameter is input to the function or output from the function.</td>
</tr>
</tbody>
</table>

Table 4.2: Parameter Dictionary Structure

of data is given. Scalar values are represented as a simple string representation of the value. Data representing an instance of a struct is specified as a list and as such, an array of structs is represented as a list of lists. Multi-dimensional data
can be represented as lists of lists, but it may be better for requests to maintain one dimensional collections of data for simplicity. See Appendix D for examples of parameters and structure definitions. CUDA can operate on both integer and floating point numbers and as a result, structure definitions are limited to combinations of type "int" and type "float". Complex structs are specified as a comma (,) separated string of these types. An example of a complex struct definition can be seen in Figure 4.6.

```json
{"direction": "in",
 "data": [[[1, 2.0], [3, 4.0]], [[5, 6.0], [7, 8.0]]],
 "struct": "int32, float32"}
```

Figure 4.6: Example of JSON Encoded Request

### 4.5 Response Structure

Responses from the server are also represented using JSON serialization to minimize data returned to the client. All responses are represented as a dictionary object at the highest level. The response structure depends on the successful execution of the request. Successful execution of the request gives a dictionary with multiple keys representing the resulting data and any meta data regarding the execution. Resulting data is returned in a JSON list with each item in the list representing a resulting data set. Each data set within this list has its own representation based on the struct specification given in the request structure. Output datasets are created depending on which parameters’ directions are specified as "out" or "inout". See Table 4.3 for
a list of the keys in a successful response.

<table>
<thead>
<tr>
<th>Key Name</th>
<th>Data Types</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>list</td>
<td>A list of each resulting data set from execution</td>
</tr>
<tr>
<td>runtime</td>
<td>int</td>
<td>The runtime of the kernel execution</td>
</tr>
</tbody>
</table>

Table 4.3: Successful Response Dictionary Structure

Requests that encounter an error during the validation, handling, or execution phases generate an error response that is similar to the successful response structure. Error responses are serialized using JSON and are represented as a dictionary object at the highest level. The keys within the error dictionary map to a message which provides information about the error and an error code. The error code primarily functions to give additional information that can help determine where the error occurred. See Table 4.4 for a list of the keys in an error response.

<table>
<thead>
<tr>
<th>Key Name</th>
<th>Data Types</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>message</td>
<td>string</td>
<td>A message explaining the error that occurred</td>
</tr>
<tr>
<td>errorcode</td>
<td>int</td>
<td>A code that helps locate the source of the error</td>
</tr>
</tbody>
</table>

Table 4.4: Error Response Dictionary Structure

4.6 Summary

This architecture is constructed in way such that multiple host clients have access to it and can send a request to the web service that will provide the data for execution and specify which kernel to execute. The process outlined here can be repeated with multiple host clients. Doing this may allow requests to be queued within the web
service so that utilization of GPU hardware can be increased. Connecting multiple host clients to one remote system also eliminates the need to have a GPGPU device within each host client. This design meets the requirements as specified and lays out an architecture for building a solution to the known limitations of GPGPU computing.
Chapter 5

Implementation

With the architecture specified in the previous chapter for the CUD Web API, a full implementation of the CUDA Web API can be built that satisfies the specified requirements and builds on the high level design found in the architecture. This chapter explains the implementation details of the server used to host the CUDA Web API and how it operates alone or in a distributed environment, as well as how a client is built to use the CUDA Web API.

5.1 Server

Implementation of the CUDA Web API requires a language, web framework, and access to the CUDA Driver API that are able to communicate easily. The web framework must be able to communicate with the CUDA Driver API in order to execute kernel functions when web requests are received. Additionally, the language used must be able to deserialize JSON and place it in a GPU memory as it would when using the CUDA runtime API. For this implementation, Python, Django, NumPy, and
PyCUDA are chosen as the frameworks to build on because they allow their respective features to be used easily and are interoperable. Django allows easy creation of a restful web service that connects to multiple types of databases. The database allows the web service to maintain a stateful work queue and a list of available workers and their states when in multi-instance mode. This implementation uses a SQLite3 database but other DBMS systems could be used with Django. PyCUDA allows direct access to the CUDA Driver API and useful features for moving data to and from the GPU. Python does not have native arrays that are laid out in memory in the same fashion as C and CUDA C. NumPy is able to merge the gap between Python’s representation of data, and how C and CUDA C represent data in memory by laying out data in memory in the same manner as C and CUDA C. The gap is then bridged by providing access functionality with Python. A discussion of this gap in further detail is provided in the data serialization section of this chapter. Other frameworks may be used for implementation as well depending on their ability to integrate these components.

5.1.1 Web Service Paths

The web server implementation exposes a set of paths to the web service, providing functionality to host clients. See Table 5.1 and Table 5.2 for a full list of server paths and their functionality. Paths not listed in the figure have no function and are redirected to the root path “/”.
<table>
<thead>
<tr>
<th>Path</th>
<th>HTTP Method</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>GET</td>
<td>Serves as the default page and contains links to the other parts of the web server. Attempts to access non-existent paths are redirected here.</td>
</tr>
<tr>
<td>/cuda/</td>
<td>GET</td>
<td>Requests that have been submitted to this path using the POST method can be polled for status or for a response at this path. This path only has results when the server is set in multi-instance mode using a work queue.</td>
</tr>
<tr>
<td></td>
<td>POST</td>
<td>Receives requests containing the JSON encoded request format.</td>
</tr>
<tr>
<td>/execute/</td>
<td>POST</td>
<td>Exposed by workers in multi-instance mode for work handlers to submit workloads for execution. The JSON format submitted to this path is similar to the request JSON format but contains additional data to assist in deserializing parameters. Client hosts should not submit any data to this path. JSON data is assumed to be pre-validated by the web server.</td>
</tr>
<tr>
<td>/status/</td>
<td>GET</td>
<td>Used by work handlers when in the worker role to send a simple get request to determine if the worker is available at a basic level and able to complete web requests. This path returns an http response with the string &quot;Up&quot;.</td>
</tr>
<tr>
<td>/server/</td>
<td>POST</td>
<td>Used by worker machines to report to the web server on startup by posting data describing how the worker can be contacted. Workers post their URI, path for execution, and path for checking status.</td>
</tr>
<tr>
<td>/servers/</td>
<td>GET</td>
<td>Returns the worker management page where remote workers\compute servers can be added, removed, and updated.</td>
</tr>
<tr>
<td></td>
<td>POST</td>
<td>Receives requests containing the JSON encoded request format.</td>
</tr>
</tbody>
</table>

Table 5.1: List of Server Paths

5.1.2 Request Handling

Each web request is received at the "/cuda/" path of the web server where the body of the web request is retrieved and assumed to be the JSON encoded kernel request. If the request body cannot be deserialized or its structure is invalid, then a JSON
Table 5.2: List of Server Paths - Cont.

<table>
<thead>
<tr>
<th>Path</th>
<th>HTTP Method</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>/configuration/</td>
<td>GET</td>
<td>Returns a configuration page where the server can be switched between single-instance and multi-instance mode.</td>
</tr>
<tr>
<td></td>
<td>POST</td>
<td>Used by the configuration page to post changes to the mode. Returns the configuration page after processing POST data.</td>
</tr>
<tr>
<td>/client/</td>
<td>GET</td>
<td>Returns a simple web UI that can be used to submit kernel requests.</td>
</tr>
<tr>
<td>/tutorial/</td>
<td>GET</td>
<td>Returns a page containing a simple set of documentation describing the operation and use of the implementation of the CUDA Web API.</td>
</tr>
</tbody>
</table>

decoded error response is created and returned to the client host. After deserializing the kernel request, the structure is validated ensuring that each field of the request dictionary is present and is of the correct type. Analysis is done on the request to determine if the parameters contain a collection of data, a scalar value, or are a complex struct type. The validated request is then added to the work queue database as a Django model.

5.1.3 Data Serialization

Serialization and deserialization of request data is critical to the successful implementation of the CUDA Web API. Traditional CUDA programs are written using C and therefore their data layout in memory is determined by the implementation of the C compiler. Data that is operated on by a CUDA kernel must be memory copied from the host to device and back again and as such, CUDA GPUs have data laid out in memory in the same manner as traditional C programs.

One of the difficulties in implementing the CUDA Web API lies in taking the
parameter data represented in JSON, and its structure definition string, and deserializing it such that the data is ultimately laid out in memory in the same way that it would be with a traditional C program. In order for a C compiler to lay out memory correctly, the C compiler must read in the structure at compile time and derive a scheme for aligning the data correctly in memory for access. Traditionally, serializing C data structs could be done using functions tailored for the data struct to perform serialization and deserialization. However, with the CUDA Web API, the data structure must be read at runtime and deserialization and serialization of data must happen with a dynamic function that can use the runtime provided structure definition as a key to determine how data will be laid out in memory. This cannot be accomplished with the C compiler alone.

Python is able to deserialize JSON strings into dynamic objects as required for this implementation. However, since Python does not lay out data in memory in the same way as C and CUDA, it is not possible to memory copy data from Python data structures to CUDA device memory. Python represents linear data structures as a list rather than in an array. NumPy is able to lay out data in memory as it would by the C compiler and can be easily accessed within Python. PyCUDA makes use of NumPy to layout data in host memory before copying it to device memory. Since JSON data can be easily converted to and from Python representations and data can be easily moved between NumPy representation and device memory, there is a clear path for serialization and deserialization. Inbound parameter data thus flows from JSON to Python to NumPy to CUDA device, and resulting data flows in reverse from CUDA device to NumPy to Python to JSON.

Libraries exist for transforming data between JSON and Python, and between
CUDA and NumPy, and data should be easily transferable between Python and NumPy representation. However, this implementation encountered several problems trying to move data through both data flows to and from the CUDA device. NumPy has libraries that allow it to convert data from array layout to Python list layout and back. It accomplishes this by converting an array of a simple data type such as an "int" or "float" to a Python list of values. Converting data back is easily done by telling NumPy the type for each value it interprets. The difficulty in this transformation lies in transforming structured data. NumPy takes an array of structured data such as a struct with two "int"s and two "float"s and represents them in Python as a list of tuples, where each value in the tuple is in the same order as the fields listed in the NumPy data type. Giving this data to JSON creates problems because JSON does not distinguish between tuples and lists and represents both as lists. When the Python representation of the data makes a round trip through JSON representation, a list of lists is given. This presents a problem for NumPy which expects the data from structs to be given in a tuple. In order to overcome this obstacle, this implementation makes use of a function that converts the innermost list within a recursive list structure back into tuples. This function is only performed if the data type specified with the data parameter is a struct consisting of multiple data rather than a scalar data type. In order to make this distinction, a special analysis step is performed at the end of data validation when receiving a kernel request that serves to tell NumPy how to layout the Python represented data in memory. Performing these steps to move data from JSON to the CUDA device requires many steps and could be simplified as discussed in the future work section. The data serialization process only performs a shallow copy of the serialized data given with the request, and therefore, copying
pointers to structures or arrays of pointers to structures results in copying pointers which do not point to valid data. In order to copy a pointer to a structure or an array of pointers to structures and the underlying structure data, a method for following the pointers to their source data and copying it is required. This type of deep copying is not provided in this implementation, and therefore pointers to structure and arrays of pointers to structures are not supported at this time.

5.1.4 Work Queue

A work queue is required when operating in multi-instance mode that tracks requests and acts as a stateful FIFO data structure. The stateful nature of the work queue requires that it exist outside of the web service due to the stateless nature of the web service. As a result, the work queue must be stored in a persistent data store. This implementation uses the Django model framework on top of a SQLite3 database in order to maintain state outside of the web service. Using the Django model framework allows easy integration with the web service and the rest of the CUDA Web API. In order to mimic the behavior of a FIFO queue data structure, the time at which the request is received is stored with the request. Each time a work item is pulled from the queue, the handler selects the request with the oldest received time to enforce the FIFO rule. This method ensures that no requests are left in the queue for an undefined amount of time. Handlers change the state of each item as they are processing the request and do not mark it as completed until the request is successfully completed or an error occurs. See Table 5.3 for a list of request states.
<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queued</td>
<td>Request has been placed in the work queue and is awaiting execution</td>
</tr>
<tr>
<td>Inflight</td>
<td>A work handler has pulled the request from the work queue but has not yet found an available worker to process it. Inflight acts as a lock to prevent multiple execution of a single request</td>
</tr>
<tr>
<td>Executing</td>
<td>Request has been pulled from the work queue and is being executed by a worker.</td>
</tr>
<tr>
<td>Completed</td>
<td>Request has been executed by a remote worker and the resulting success or error is ready to be picked up by the remote client.</td>
</tr>
</tbody>
</table>

Table 5.3: Request States

5.1.5 Execution

All workers and the servers have the same method for execution. This allows the web service to operate in its single or multi-instance modes. This implementation uses PyCUDA in order to interact with the CUDA Driver API and CUDA compiler from Python. PyCUDA provides the necessary wrappers for CUDA and also handles memory copying data to and from the GPU device.

The first step in execution is to create a device context. The device context specifies which CUDA capable device on the executing system will be used for kernel execution. Only one device is used for each request per worker. However, it may be possible for a worker to handle multiple kernel executions at one time depending on the number of CUDA capable devices available to the worker. Any errors encountered while selecting a CUDA device cause an error response to be sent back to the sender detailing the cause of the error. It is possible to use a PyCUDA auto initialization to create the device context under normal circumstances but not when being used in conjunction with Django. Testing of the implementation across linux and OS X platforms reveals different errors, relating to the management of the CUDA context.
stack being either non-empty upon completion of execution or not containing the expected context references. As such, device contexts are handled manually to avoid errors.

Once CUDA device selection finishes, the request data, including grid and block size as well as kernel parameters, is retrieved from the request structure and stored. Each kernel parameter has already been pre-processed and given additional meta data to allow faster data deserialization. Each parameter is deserialized into a NumPy array or NumPy scalar type using the meta data. The system can construct arrays of scalar types limited to integers and single precision floating point numbers as well as arrays of simple structures also consisting only of integers and single precision floating point numbers as well as simple scalar types. Because of limitations encountered using NumPy, it is not possible to create single instances of structure types and as such passing structure types to the kernel function must be done using an array of size one. Using the array instead of a simple structure type does not create significant additional overhead for the system but does place a limitation on the programmer. Additionally, data specified as "out" or "inout" must be an array. As in typical C function calling, any parameter that will be modified by the function must be given using a pointer so that changes to the data persist after the function has been called. This implementation makes use of pointers to scalar types in this implementation and as such data that will be modified must be given as an array. Although structure types that will be modified by the kernel function must be given in an array, within the kernel code, the programmer can still address the structure type as if it were a point to the structure type rather than a pointer to an array of that structure type.

After data has been loaded and set up for kernel execution, the kernel code is JIT
compiled and loaded onto the GPU. In order to set up the kernel, the NVCC compiler is invoked, producing a cubin file. Cubin files are a special binary output that when loaded on a GPU, provides the operations for the GPU to perform and are analogous to binary files output from other compilers. The resulting cubin file is loaded into system memory and then onto the CUDA device. Using JIT compilation with the NVCC compiler allows the kernel code to be sent in a form that is identical to any CUDA kernel compiled and executed using a local CUDA device. The kernel name from the request is used to specify which kernel function with the `__global__` attribute is the starting point in the kernel code similar to a kernel launch using the `<<<grid,block>>>` notation used in the CUDA Runtime API. This functionality is accomplished by using the CUDA Driver API calls `cuModuleLoad()` and `cuModuleGetFunction()`. The kernel is then launched with the parameters loaded into GPU memory using the `cuLaunchKernel()` call from the CUDA Driver API. In order to accomplish these steps in python, this implementation uses PyCUDA which wraps these steps and function call into two lines.

```
SourceModule()
```

wraps the compilation and module loading steps and returns the module in a python wrapper. The module represented in python then has a function
get_function()

which retrieves a function that wraps the kernel launch function. This implementation then uses the function provided by PyCUDA from the

get_function()

call with the appropriate deserialized NumPy data parameters to launch the CUDA kernel. After execution, parameters wrapped with the PyCUDA "out" or "inout" types are copied back from the GPU into CPU memory and accessible from their NumPy arrays. The resulting set of CUDA Driver API function calls follows the same flow as that shown in the CUDA C Programming Guide. [5]

Data retrieved from the GPU is collected into a list of all response data and then serialized into a response structure using JSON. Runtime data is also collected and added to the response, giving the programmer additional metadata to measure performance increases in their kernel. The CUDA device context is cleaned up at this stage to free the device for later kernel executions. Finally, the JSON serialized response is returned and execution is complete.

5.1.6 Result Handling

Results are handled differently at the end of computation depending on mode of operation. In single-instance mode, results are returned to the client immediately after execution without being stored in the database. Under multi-instance mode, results from execution are stored in the database and associated with their respective stored request. Client hosts are able to retrieve requests by issuing a GET request to the web service at the appropriate path with the ID of the request. The request
ID is given to the client host during the request submittal phase in multi-instance mode. Clients can get the result of their request from the web service when request execution has finished. If a request has not been executed or is still in progress, the client is given a status update indicating if the request is waiting for an able worker or being executed. Clients are not notified when requests have been executed and are required to poll the web service to get the response. It is possible to add callback functionality to the web service as discussed in the future work section but is not part of the web service at this time.

5.2 Workers

Operating the web service in multi-instance mode requires the use of workers to handle requests. In multi-instance mode, the server hosting the web service does not execute requests directly. Instead, the server adds the request to a work queue from which requests are selected and dispatched to workers. Each worker is then responsible for the compilation of kernel code included in the request, deserialization of parameters, execution of the request, and return of the response back to the work dispatcher. Workers are required to have a CUDA capable device in order to handle requests but workers are not required to have identical hardware. Additionally, workers must have the NVCC compiler in order to perform JIT compilation using PyCUDA. Execution on each worker is performed in the same way as on the server when operating in single-instance mode. Workers run the same application as the server and expose the same web service paths as the server. However, workers are only contacted using the execute and status paths of the web service. Other paths of the worker should not
be used and each worker should not be treated as host for the public web service in the same way as the server hosting the web service. Using a worker paths outside of execute and status path can create several problems such as: discrepancies in the data store of the worker which should not exist, interrupted request workflows resulting in failed request execution, or possibly user confusion as a result of looking for data stored in the server on a worker machine. Using alternate worker paths can also result in multiple data stores containing completed requests and subsequently causing further confusion when debugging. The same application is used for both server and worker roles in order to maintain a simple system with a common code base.

5.2.1 Tracking

The list of available workers is kept in a database. Each worker is tracked by its URI, and additional metadata regarding web service paths where the worker can be contacted and the last time the worker was contacted. Workers are chosen by the request dispatchers by selecting the available worker with the oldest update time. Choosing the worker with the longest idle time allows the database to act as a LILO data structure and ensures that all workers are used equally. See Table 5.4 for a list of information stored for each worker.

5.2.2 Request Dispatching

Requests are dispatched to workers by work handlers that are responsible for pulling unexecuted requests from the work queue and sending them to an available worker. Work handlers operate in their own process and are initialized each time a new request
<table>
<thead>
<tr>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>URI</td>
<td>The URI used to contact the worker. Can be an IP address or domain name. Concatenated with status path and execute path to contact the worker.</td>
</tr>
<tr>
<td>Status Path</td>
<td>Web service used by work dispatcher to determine if worker can be contacted.</td>
</tr>
<tr>
<td>Execute Path</td>
<td>Web service path where request data can be sent in specified internal format for execution. Different from the path used by client hosts to access public API.</td>
</tr>
<tr>
<td>status</td>
<td>Available for request handling or busy while executing request.</td>
</tr>
<tr>
<td>updated</td>
<td>Time stamp from the last action involving the worker. Includes: startup, request dispatch, response received</td>
</tr>
</tbody>
</table>

Table 5.4: Worker Information

comes in, or a new worker is added to the list of workers. Each handler remains active as long as there is work in the work queue and an available server to perform execution. Handlers that start and do not find work in the work queue, or do not find an available server will cease functioning and leave further handling to other handlers. This ensures that at least one handler is active while there are requests in the work queue and an available worker. Handlers are responsible for updating the status of the request they are processing and the status of the worker that is executing a request.

When selecting a request, the handler immediately updates the status of the request to prevent other handlers from selecting the same request and duplicating execution. The status of the worker is updated by the handler immediately after finding an available worker to prevent overloading a single worker. Once the worker and request have been selected from their respected queues, the handler submits the request to the worker and waits for the response. Handlers send data to workers using a JSON structure similar to the structure listed in Table 4.1 with additional
metadata from the stored request that simplifies the deserialization process.

After the worker has finished executing the request and responded to the handler, the handler stores the successful or error response in the database for retrieval by the client host. The handler then releases the worker by updating its status as available and changing its last updated time. Releasing the workers after execution helps to distribute requests among all available workers. Handlers repeat this process until there are no requests remaining in the work queue or there are no available workers.

5.3 Client / User Interface

In this implementation, a simple client application is used in order to demonstrate how an application can connect to the CUDA web service, build requests and send them using JSON. This application is built using HTML and Javascript for data processing, sending, and handling. Client hosts are not required to use this application to submit data, but it demonstrates the ability to use CUDA from a domain that is not typically associated with CUDA. This application is titled CUDA Cloud and allows a user to enter kernel code in the browser as well as JSON formatted parameters, submit them and see the result. CUDA Cloud is not meant as a front end for doing intense computation but could be useful for demonstrating how to write CUDA kernels and to quickly test or prototype kernels. This application uses the CodeMirror syntax highlighting text area for the kernel code text area and parameter text areas. C syntax highlighting and JSON syntax highlighting allow users to more easily work with CUDA kernels and parameter data. Request data is validated and structured using javascript. Each request is submitted to the web service using AJAX post.
Resulting data is displayed without needing to refresh the interface once the AJAX call has returned. See Figure 5.1 for a screenshot of the CUDA Cloud web client application.

Figure 5.1: CUDA Cloud Screenshot
Chapter 6

Tutorial

This chapter provides instructions for deployment, setup, and use of this implementation of the CUDA Web API.

6.1 Requirements

This implementation is built on and has the following requirements for operation. All operating systems are supported provided that they have the tools listed installed. This implementation requires specific versions of the tools listed. Newer versions of these tools may exist but have not been tested. The system running the web service or workers must contain a CUDA capable device with compute capability 1.0 or higher depending on requirements of the kernels submitted. Although this implementation is built using the CUDA 4.2 toolkit, it may be possible to use an older toolkit but this has not been tested. This list of prerequisite tools for this implementation is as follows:
• Python 2.7
• CUDA Toolkit 4.2 or higher
• Django 1.3.1 or 1.4
• PyCUDA 2011.2.2
• NumPy 1.6.1
• httplib2 0.7.2

6.2 Installation

To begin installation, obtain a copy of the web service implementation source code. This serves as the basis for both the web service and the compute workers when operating in multi-instance mode.

6.3 Server Startup

Startup of the system is performed by running the following command in a command line terminal. This is the same command used to start other Django applications. Set the current directory to the root directory of the application where the manage.py file is located. Run the following command where 1.1.1.1 is the ip address desired for operation and 80 is any port desired.

```
python manage.py runserver 1.1.1.1:80
```
Deployment is only supported using the Django development web server at this time. However, it could be possible to use other web servers in the future.

### 6.4 Worker Startup

When operating in multi-instance mode, workers are tracked using the persistent worker queue in the database. Workers can be added to the web server’s worker queue by using the Web UI or by making an HTTP POST to the web server. In order to add a worker to the system at runtime using the HTTP POST method, an HTTP POST must be made with a JSON encoded dictionary containing information about how to contact the worker. This dictionary must include the worker’s URI, web service path used to check status, and web service path used to post requests for execution. See Table 6.1 for a description of the JSON dictionary structure used with HTTP POST method.

<table>
<thead>
<tr>
<th>Key Name</th>
<th>Data Types</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>uri</td>
<td>string</td>
<td>The URI of the worker. Must be reachable by the web server\handlers.</td>
</tr>
<tr>
<td>statuspath</td>
<td>string</td>
<td>Web service path on the worker that returns a small HTTP response to check functionality.</td>
</tr>
<tr>
<td>executepath</td>
<td>string</td>
<td>Web service path on the worker where requests can be sent for execution using an HTTP POST.</td>
</tr>
</tbody>
</table>

Table 6.1: Worker POST Dictionary Structure

Additionally, new workers can be added to the web servers worker queue by entering them manually using the worker/compute server management page as shown in Figure 6.1 of this section. The server’s management page shows all workers/compute
servers registered in the system and their current status.

![CUDA Web Service](Image)

**Available Servers**

<table>
<thead>
<tr>
<th>ID</th>
<th>URI</th>
<th>Status Path</th>
<th>Execute Path</th>
<th>Status</th>
<th>Last Updated</th>
<th>Update</th>
<th>Delete</th>
</tr>
</thead>
</table>

**Add New Server**

<table>
<thead>
<tr>
<th>Server URI</th>
<th>Server Status Path</th>
<th>Server Execute Path</th>
<th>Add Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>127.0.0.1</td>
<td>/status/</td>
<td>/execute/</td>
<td>Add Server</td>
</tr>
</tbody>
</table>

Figure 6.1: CUDA Cloud Worker\Compute Server Management

To remove or change the status of a worker, the web UI has buttons associated with each worker/compute server to perform each action respectively.

### 6.5 Operational Modes

The server can be configured to host the web service in either a single-instance mode or multi-instance mode as discussed in the implementation chapter. The default mode of operation is single-instance but can be changed before or during runtime. The operational mode can be changed in two ways. The first way to change the mode is by using the configuration file in the root directory of the cudawebapiproj project folder, named `settings.py`. The default setting for operational mode can be found in the configuration file as follows:
SINGLE_INSTANCE = True

In order to change the operational mode of the server to mult-instance mode, the value associated with SINGLE_INSTANCE must be changed to False. Changing the configuration file yields the following statement. Additionally, see Table 6.2 for a simplified list of operational modes and their corresponding values within the configuration file.

SINGLE_INSTANCE = False

<table>
<thead>
<tr>
<th>Operational Mode</th>
<th>SINGLE_INSTANCE value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Instance</td>
<td>True</td>
</tr>
<tr>
<td>Multi Instance</td>
<td>False</td>
</tr>
</tbody>
</table>

Table 6.2: Configuration File - Operational Mode Settings

The mode can also be changed at runtime using the web UI. The web UI for configuration can be found at the \configuration\ path of the server as listed in the path Table 5.2 in the implementations section. A screenshot of the configuration page can be seen in Figure 6.2 of this section.

Toggling the operational mode between single-instance and multi-instance at runtime is not recommended by the Django documentation, because the value toggled is part of the configuration file. Changing the operational mode at runtime does not permanently modify the configuration file, and subsequently the server will start up in whichever mode is specified in the configuration file. Since the server persists requests in multi-instance mode and not in single-instance mode, all requests are removed from
the database when switching the server from multi-instance mode to single-instance at runtime.

6.6 API

Client applications must submit all requests to the web server at the \cuda\ path using the HTTP POST method. When operating in multi-instance mode, the client can retrieve a processed request from the \cuda\ path by using and HTTP get request with the id of their request.

6.7 Request Format

A strict request format must be used to submit a request to the web service for execution. The example in Figure 4.1 shows the structure of a valid request used for matrix multiplication. See Appendix D for examples of the data types that can be passed as parameters to the request.
6.8 Result Format

Successful results from the web service as a result of request execution are returned in a very strict format. These result format must be correctly handled by the originating client host to retrieve computed data. A complete valid response is shown in Figure 6.4. Output data is returned in a list. Figure 6.4 shows one set of output data that is a 2 dimensional array of floats. In order to handle only the response data, data must be retrieved from the result dictionary using the output key and then the matrix must be pulled from the output list of data. Since the output data list only contains one item, the first and only item from the list is the resulting data. It is important to retrieve data from the output list only and not move deeper into any lists contained in the output data list. This is an important step during use which could easily be overlooked during use resulting in difficult bugs. Traversing the data deeper than the first list to retrieve complete data sets could corrupt data during retrieval. Should a user traverse the data structure incorrectly, debugging could be very difficult depending on the structure and type of the output data. As an example,
incorrect traversal of the datastructure could result in interpreting a two dimensional matrix of floats as a two dimensional matrix of structures consisting of multiple floats.

{"totaltime": 1111, "output": [[[2.0, 3.0], [6.0, 11.0]]], "runtime": 1111}

Figure 6.4: Valid JSON Encoded Response

6.9 Errors

In order to determine where errors occur during execution, the server provides an error code in error responses. Each error code has a specific meaning and origin that assists debugging on both client and server sides of a request. See Table 6.3 for a list of the error codes, meanings, and next step actions for users.
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
<th>Programmer Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Invalid JSON</td>
<td>Check JSON formatting to verify conformity to JSON standard.</td>
</tr>
<tr>
<td>2</td>
<td>Invalid Request Structure</td>
<td>Check request dictionary for required keys, key names and data types.</td>
</tr>
<tr>
<td>3</td>
<td>Invalid Parameter Structure</td>
<td>Check parameter dictionary as specified in error message for required keys, key names, and data types.</td>
</tr>
<tr>
<td>4</td>
<td>Deserializing Error</td>
<td>Check parameter data to verify conformity to JSON standard. Verify complex structs are represented as JSON comma (,) separated lists.</td>
</tr>
<tr>
<td>5</td>
<td>Parameter Error (direction)</td>
<td>Check parameter direction to verify value of ”in”, ”out”, or ”inout”.</td>
</tr>
<tr>
<td>6</td>
<td>Kernel Code Error</td>
<td>Check kernel code for syntax errors. Read error message for NVCC compiler output.</td>
</tr>
<tr>
<td>7</td>
<td>CUDA Execution Error</td>
<td>Read error message for information regarding error.</td>
</tr>
<tr>
<td>8</td>
<td>Serialization Error</td>
<td>Check return data type specification and kernel for mutual conformity.</td>
</tr>
</tbody>
</table>

Table 6.3: Error Codes and Definitions
Chapter 7

Validation

The purpose of exposing CUDA through a web service was to expand the capabilities and uses of CUDA and GPGPU computing. Current GPGPU frameworks have been beneficial but are limited by their platform and locality requirements. We can now measure the ability of the CUDA web service to expand those limitations in order to validate its use. The following metrics are used to measure this.

1. Functionality

2. Usability

3. Performance

Functionality will be determined by the web services ability to match the function of a standard CUDA program. In detail, this means for a given kernel function and set of input data, the output data of the operation using the web service is the same as the data from using CUDA with a locally attached CUDA device.
Usability is measured by the capability of a programmer to use the web service. There are two cases to assess for usability. First, can an existing CUDA kernel and set of data be transferred to the web service without difficulty. Second, can a programmer utilize CUDA from an environment without a GPU or the CUDA toolkit with the same result as using standard CUDA with a locally attached CUDA device.

Performance is calculated by running an existing function with a serial implementation, parallelization with standard CUDA, and parallelization with the CUDA web service to determine runtimes. Runtime for the CUDA web service should show noticeable improvements over the serial implementation. Additionally, a difference between the standard CUDA implementation and CUDA web service implementation should be enough to account for network overhead but still be minimal. Some fixed overhead is also expected when using the CUDA web service, as with memcpy overhead encountered with standard CUDA.

7.1 Evaluation Method

These measurements determine the capabilities of the CUDA web service to perform as specified. In order to gather thesis measurements, as part of validation, several applications are used to gather quantitative data. A simple matrix multiplication algorithm was utilized as specified for performance validation and functionality validation respectively. Matrix multiplication using square matrices is a well-known algorithm with resultant data that is not interdependent on itself, and therefore benefits from CUDA parallelization. Five different implementations for matrix multiplication are created using combinations of Python, C, and CUDA as shown in table 7.1.
<table>
<thead>
<tr>
<th>Implementation</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>CPU</td>
</tr>
<tr>
<td>CUDA C</td>
<td>GPU</td>
</tr>
<tr>
<td>Python</td>
<td>CPU</td>
</tr>
<tr>
<td>Python with PyCUDA</td>
<td>GPU</td>
</tr>
<tr>
<td>Python with CUDA Web API</td>
<td>GPU</td>
</tr>
</tbody>
</table>

Table 7.1: Evaluation Implementations

The first two implementations are simple serial implementations using both Python and C to set a benchmark for performance and correctness. Two additional implementations using Python and CUDA are built using Python with PyCUDA and Python with the CUDA web service. All Python implementations have data stored using NumPy arrays that lay out data in memory in the same way as C. Finally, the last implementation is written using C and traditional CUDA C. The CUDA C implementation provides a benchmark for standard CUDA performance.

Data is collected with these implementations using square matrices starting with two dimensional matrices of size 5 in each dimension (25 elements) and increasing by 5 in each dimension, \((5^2)\) elements) up to a a maximum of 120 in each dimension for Python implementations and 250 in each dimension for C implementations. The end matrix size is different for each implementation as a result of the performance differences between using C and Python. The goal of this section is to show the point at which the data to be operated on is large enough or requires sufficient computation to realize a performance increase using a GPU. In this case, matricies for size 120 and 250 in each dimension are sufficient to show the performance increases from using GPUs with Python and C respectively. Data is collected for each input matrix size ten times.
The CUDA Cloud web client application is also used in order to assess usability and general viability of the web service as a platform for CUDA computation.

All tests are run on a system using an Intel Core i7 processor at 2.66ghz and an Nvidia GeForce GT 330M. The NVCC compiler is used for all C and CUDA C implementations. Tests running a remote client host and web service server separately used the same Intel Core i7 system as the server and a system with an Intel Core 2 Duo at 2ghz as the client with data sent over a gigabit ethernet network.

### 7.2 Functionality

Functionality is measured by the web service’s ability to produce the same output from the same input data using matrix multiplication. Each input matrix has preset data with values starting from 1 and incrementing by 1 to a maximum value of $N^2$ where $N^2$ is the size of the square matrix. Resulting matrices computed using the CPU and GPU implementations are equal up to size 30 in each dimension with Python implementations and up to size 60 in each dimension with C implementations. After these points, maximum value differences grow as the maximum value input matrices grows with matrix size. This data can be seen in Figure 7.1. Differences are seen between the C and CUDA C implementation as well as between Python and PyCUDA \CUDA Web API implementations. Values from PyCUDA and CUDA Web API implementations are equal in all cases as expected since this implementation of the CUDA Web API uses PyCUDA. These differences in values can be explained by the loss in precision with floating point values due to different ALU architectures between the CPU and GPU. Although the CUDA Web API yields different values
with large numbers when performing floating point arithmetic, matrices of smaller size yield constantly equal results. This suggests that the CUDA Web API does not alter CUDA kernel functionality and performs in the same way as traditional CUDA C kernels.

![Figure 7.1: Maximum value difference between GPU and CPU matrix multiplication](image)

7.3 Usability

The GPU matrix multiplication implementations are able to produce resulting matrices with equal results as shown in the functionality validation. Each of the GPU
implementations has one CUDA kernel that performs the matrix multiplication algorithm. All of these kernels have the same CUDA kernel code which can be seen in the Appendix. As a result, we see that it is possible to use the same CUDA kernel with the CUDA web service as is used in a CUDA C implementation. This fulfills the requirements laid out in the architecture for the CUDA Web API and shows that transition from local CUDA to use of the CUDA web service is a viable option for using CUDA.

The CUDA web service requires that data be sent in a specific JSON encoded format. This format allows the web service to correctly handle CUDA data and perform remote compilation and execution. The CUDA Web API implementation of the matrix multiplication program is able to meet this requirement without the need to communicate directly with the GPU or the CUDA toolkit. This shows that it is possible to make use of CUDA without the need to be directly connected to GPGPU resources. Using this system, programmers can make use of CUDA and achieve equal results as they would with local GPGPU resources.

7.4 Performance

Performance is measured by the web service’s runtime when compared to CPU and standard GPU implementations. CUDA kernels have greatly improved runtimes over CPU implementations but have a fixed runtime overhead associated with GPU memory handling. GPU memory overhead is a result of the time necessary to allocate GPU memory, copy data from CPU memory to GPU memory, copy data back from GPU to CPU memory, and free GPU memory for future use. The added overhead
from GPU memory handling creates a point at which it is more efficient to perform computation using the CPU rather than the GPU. The CUDA web service has additional overhead associated due to the time required to deserialize and manipulate request data. In order for the CUDA web service to be effective, there must be a point at which computation is completed faster using the CUDA web service than with local CPU computation.

In the case of CPU implementations, runtime is calculated as the time it takes to multiply two matrices but not the time to set the data within the matrices. CUDA C runtime is calculated as the time it takes to allocate memory on the GPU, copy input matrices onto the GPU, perform the GPU matrix multiplication, copy the resulting matrix from the GPU, and free GPU memory. For the PyCUDA implementation, runtime is measured as the time it takes CUDA to compile the kernel code in addition to the operations performed using CUDA C. Runtime of the CUDA web service is calculated from a host client that submits a request for matrix multiplication accompanied by the input matrices as the time it takes for the request to be created and sent, and for a response to be received and deserialized.

The data from these implementations can be seen in Figures 7.2 and 7.3. Looking first at the C and CUDA C implementations in Figure 7.3, we see that GPU kernel execution time is faster than CPU execution time in all cases. However, total runtime including memory operations for the GPU is slower than the C implementation for small matrices and faster for large matrices. This shows that the overhead associated with GPU memory operations is overcome if the workload is large enough. The transition point for efficiency in this case happens with two dimensional matrices of approximately 85 elements in each dimension in size.
Data from the Python implementations show a similar pattern in Figure 7.2. We see that the runtimes of the CPU implementation of matrix multiplication with Python increase exponentially as input matrix sizes increase just as with the C CPU implementation. The local PyCUDA implementation of matrix multiplication shows only a slight increase in runtime with an increase in input matrix size. The execution phase of the CUDA web service shows nearly identical runtime as the local PyCUDA implementation as would be expected due to the CUDA web service’s use of PyCUDA during the execution phase. More interestingly, runtime for the CUDA web service is significantly greater in all cases than the PyCUDA implementations but shows the same increase in runtime as the input matrices increase in size. The increase in runtime in this case shows how much overhead is added by the need to serialize the request, send it to the web service, deserialize the request, execute the kernel and follow the same process backwards to the client host. Performance tests done using the gigabit ethernet connection between separate server and client systems show approximately the same runtimes as those done locally where the server and client system are colocated within the same machine.

Overhead involved in using the CUDA web service results in slower runtimes than the CPU implementation with small input matrices. However, the quadratic growth of the CPU implementation results in faster runtime using the CUDA web service than the CPU implementation with larger input matrices. A crossover is seen in Figure 7.2 with two dimensional input matrices with size of approximately 30 in each dimension. This shows that in cases with large amounts of computation to perform, that the CUDA web service provides significant performance increase over using a CPU implementation. Although network traffic was minimal, the tests
using separate server and client systems show that the added overhead associated with moving data over the gigabit network is negligible. As such the CUDA web service meets architectural performance requirements shown earlier. Additionally, we see that using web services to perform remote execution of CUDA kernels is viable from a performance perspective.

Figure 7.2: Matrix Multiplication Runtime using Python, PyCUDA, and CUDA Web API
Figure 7.3: Matrix Multiplication Runtime using C and CUDA C
Chapter 8

Conclusion

The results of this analysis show that it is possible to create a system which enhances the GPGPU’s abilities and minimizes their requirements. Building such a system shows that it is possible to reduce costs associated with GPGPU programming by increasing utilization and decreasing hardware requirements. Using existing CUDA and PyCUDA technologies allow the programmers to use the system without adding significant additional complexities or barriers to entry. There is a need for better tools to use in conjunction with GPGPU programming and this architecture should present itself as a viable alternative to existing tools and frameworks.
Chapter 9

Future Work

The CUDA Web API is a viable option for performing remote CUDA kernel execution as shown by the validation data. However, there is some work that can be done in order to improve the performance and functionality of the implementation.

9.1 GPU Data Serialization

This implementation of the CUDA web service uses NumPy, Python JSON libraries, and some manipulation in order to deserialize and serialize data coming into and out of the CUDA web service. Since NumPy is not available for all other languages, this implementation cannot be copied directly in another language. Ideally, it should be possible to implement the architecture put forth in any environment or language that has bindings to the CUDA Driver API. In order to do this without rewriting significant pieces of code, a common library for data serialization is needed.

Existing JSON libraries use the CPU in order to read serialized data and lay it out correctly in memory. Datasets used for GPU computation tend to consist of large
sets of individual values with the same format. These types of datasets could be deserialized on the GPU using a CUDA kernel in order to achieve better runtimes with the CUDA web service. A CUDA kernel used for deserialization would receive the serialized string representation of data used for computation and deserialize each element of the dataset using an individual thread, while determining the actual datatype using a map that provides the type of each element in the dataset. Using a CUDA kernel for deserialization would allow the CUDA web service to bypass other data handling libraries. Additionally, using a CUDA kernel for serialization and deserialization would allow greater portability of each implementation of the CUDA Web API.

9.2 Response Call Back

Operating in multi-instance mode with this implementation of the CUDA web service requires a client to submit a request and then poll the web service for a response until it is ready. In order to minimize unneeded traffic talking to the web service, it would be better if the web service was built using a mechanism that allowed the web service to report results to the client host when processing is finished. This could be accomplished by allowing the client to provide the URI of a web service within the request, that would be used by the CUDA web service to submit the response upon completion of the request.
9.3 Performance Improvement

This implementation of the CUDA Web API shows benefits in performance and ability to simplify barriers to entry in using CUDA. Although the performance tests show a benefit from use, it may be possible to lower the overall runtime associated with using the CUDA Web API. This may be done by streamlining data serialization, request processing, using a more intelligent work handler algorithm, or using multiple GPUs on the same worker to process requests in parallel.

9.4 Alternative Implementation

The CUDA Web API architecture puts forth an interface for using CUDA with a well-defined web service. While the implementation presented here meets the requirements set forth, it is not necessary for this implementation to be written using a specific language or CUDA Toolkit. In order to meet the requirements of the architecture, it is necessary to use a language and CUDA binding, that bridge the gap between the web service interface with its data serialization format and the GPU with its memory layout. This could be accomplished with another language and toolkit provided that data serialization can be done in a way that is language independent as with the CUDA kernel discussed earlier in the data serialization future work section. Using another language or toolkit may provide better runtime performance and reduce the overhead associated with using the CUDA web service. Reduced overhead could decrease the input data size needed to realize performance improvements from using the CUDA web service instead of CPU computation. An alternative implementation
would not change the forward facing interface of the web service and as such not require any changes by the client host.
Bibliography


Appendix A

Matrix Multiplication - C

```c
void long multiply(float * mat1, float * mat2, float * mat3, int size)
{
    int x, y, i;
    float sum;
    for(x = 0; x < size; x++){
        for(y = 0; y < size; y++){
            sum = 0.0;
            for(i = 0; i < size; i++){
                sum += mat1[i + y * size] * mat2[x + i * size];
            }
            mat3[x + y * size] = sum;
        }
    }
}
```
Appendix B

Matrix Multiplication - Python

# mat1, mat2, and mat3 are 2-dimension numpy arrays
def multiply(mat1, mat2, mat3, size):
    for x in range(size):
        for y in range(size):
            sum = 0.0
            for i in range(size):
                sum = sum + (mat1[x][i] * mat2[i][y])
            mat3[x][y] = sum
Appendix C

Matrix Multiplication - CUDA

```c
__global__ void multiply(float *array1, float *array2, float *array3, int size){
    for(int tid = threadIdx.x + blockIdx.x * blockDim.x;
        tid < size * size; tid += blockDim.x * gridDim.x){
        int x = tid % size;
        int y = tid / size;
        float sum = 0;
        int i;
        for(i = 0; i < size; i++){
            float a = ;
            float b = array2[x + i * size];
            sum += array1[i + y * size] * array2[x + i * size];
        }
        array3[tid] = sum;
    }
}
```
Appendix D

JSON Encoded Parameter examples

{"direction": "in", "data": 9, "struct": "int32"},

Figure D.1: Scalar Parameter - type: int

{"direction": "in", "data": 9.0, "struct": "float32"}

Figure D.2: Scalar Parameter - type: float

{"direction": "in", "data": [1.0, 2.0], "struct": "float32"}

Figure D.3: Array Parameter - type: 1D array of floats

{"direction": "in", "data": [[1.0, 2.0], [3.0, 4.0]], "struct": "float32"}

Figure D.4: Array Parameter - type: 2D array of floats
Figure D.5: Array Struct Parameter - type: 2D array of structs

Figure D.6: Output Parameter - type: 2D array of floats

Figure D.7: Empty Output Parameter - type: 1D array of floats

Figure D.8: Empty Output Parameter - type: 2D array of floats