8-19-2013

OpenCUDA+MPI

Kenny Ballou
Boise State University

Nilab Mohammad Mousa
Boise State University

OpenCUDA+MPI

A Framework for Heterogeneous GP-GPU Distributed Computing

Kenny Ballou
Nilab Mohammad Mousa
College of Engineering – Department of Computer Science
Boise State University
Alark Joshi, Ph.D

Abstract

The introduction and rise of General Purpose Graphics Computing has significantly impacted parallel and high-performance computing. It has introduced challenges when it comes to distributed computing with GPUs. Current solutions target specifics: specific hardware, specific network topology, a specific level of processing. Those restrictions on GPU computing limit scientists and researchers in various ways. The goal of OpenCUDA+MPI project is to develop a framework that allows researchers and scientists to write a general algorithm without the overhead of worrying about the specifics of the hardware and the cluster it will run against while taking full advantage of parallel and distributed computing on GPUs. As work toward the solution continues to progress, we have proven the need for the framework and expect to find the framework enables scientists to focus on their research.

Keywords: Parallel Computing, Distributed Computing, General Purpose Graphics Processing, High-Performance Computing, Scientific Computing, Frameworks, Software Libraries

1 Introduction

Increasingly, Graphics Processing Units (GPUs) are being used for general purpose computation (General Purpose GPU (GP-GPU)). They have significantly altered the way high-performance computing tasks can be performed today. To accelerate general-purpose scientific applications, computationally intensive portions of the application are passed to GPUs. Once the GPU has completed its task it sends the result back to the Centeral Processing Unit (CPU) where the application code is running. This process can make applications run noticeably faster.

Scientific applications require a large number of floating point number operations, CPUs are not sufficient to carry out such computationally intensive tasks. CPUs are responsible for prioritization and execution of every instruction. Generally, a processor is described by the number of execution cores it owns. Modern CPUs have eight cores while GPUs have hundreds or more cores. More cores grant GPUs the ability to perform more tasks at the same time.

In computer architecture there are two main ways of processing: serial and parallel. CPUs consist of a small number of cores that are best at processing serial data. On the other hand, GPUs consist of thousands of cores that are designed for processing data in parallel. Given a program, we can run the parallel portions of the code on GPUs while the serial portions run on the CPU. The programmable GPU has evolved into a highly parallel, multithreaded, many core processor with tremendous computational power. Compute Unified Device Architecture (CUDA) is an architecture for utilizing and distributing computational tasks onto a computer's GPUs. As a parallel computing platform and programming model, CUDA utilizes the power of a GPU to achieve dramatic increases in computing performance [6]. This fact is well illustrated in figure 1.

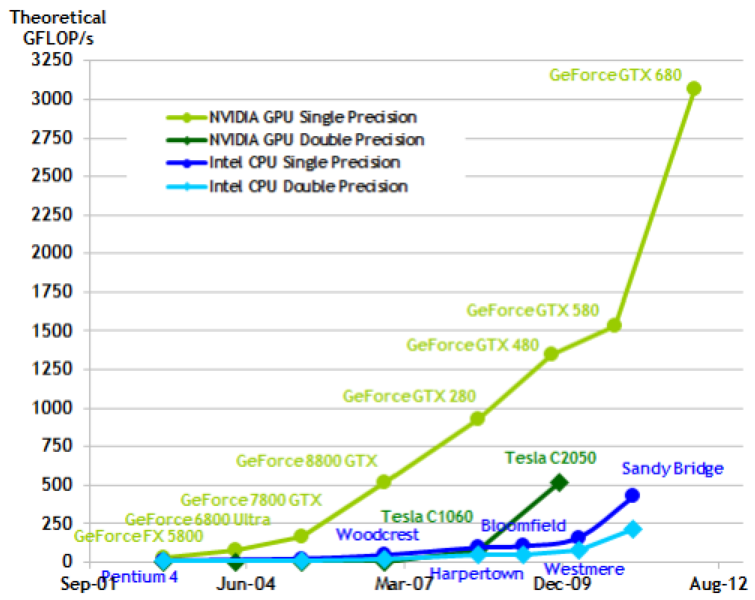

Figure 1: Theoretical Floating-Point Operations per Second for the CPU and GPU [6]

Today, more than one million CUDA-enabled GPUs are used by software developers, scientists and researchers in order to obtain a large performance gain in wide-ranging applications [6]. A major challenge remains in being able to seamlessly integrate multiple workstations to further parallelize the computational tasks [18] [20]. Current approaches provide the ability to parallelize tasks, but they are less focused on optimally utilizing the varied capabilities of heterogeneous graphics cards in a cluster of workstations (across many computer nodes).

We propose to create a framework for using both Message Passing Interface (MPI) and CUDA on a cluster of computers to dynamically and easily assign computationally intensive tasks to the machines participating in the cluster. MPI is a popularly used standardized interface for communication among a group of computers. It facilitates the passing of messages between the nodes in the cluster.

Our framework will be easy to use on a heterogeneous cluster of computers, each containing a CUDA-capable GPU. The framework shall abstract away the difficulties of creating distributed and parallel code against a cluster. The framework shall expose certain debugging and profiling mechanisms as well. Further, we also wish to develop and package together code and configuration to facilitate system administrators with cluster management.

We plan to evaluate the efficacy of our framework using several problems: the N-Body problem and the process of vessel extraction from CT angiography scans, to name a few. Both are computationally intensive and have unique requirements of inter-node communication during execution [17].

2 Significance

To gain a better understanding of the significance of the problem let's examine some pseudo code for computing element-wise vector summation.

2.1 `CUDA` and `MPI` Vector Summation

First, let's look at the pseudo code of the vector summation without the framework. Notice, this code is executed on each node, *rank* is the current node we are using. This simplifies on the MPI side; but we can just as easily do this differently.

Algorithm 1 Algorithm of `CUDA` and `MPI` Element-wise Vector Summation

```
function ADD($slice\_a, slice\_b$)                    ▷ Actual addition is done on GPU
    $slice\_c \leftarrow slice\_a + slice\_b$
    return $slice\_c$
end function
function COMPUTE($card\_max, world\_size, N, a, b$)  ▷ Split Data and Compute Sum  ▷ Ran on all nodes part of the mpi world
    $c \leftarrow empty\_like(a)$
    $M \leftarrow floor((N + card\_max - 1)/card\_max)$
    $m \leftarrow floor((M + world\_size - 1)/world\_size)$
    for $i < m$ do
        $slice\_low \leftarrow (rank * 2 + i) * card\_max$
        $slice\_high \leftarrow (rank * 2 + (i + 1)) * card\_max$
        $c[slice\_low : slice\_high] \leftarrow add(a[slice\_low : slice\_high], b[slice\_low : slice\_high])$
    end for
    return $c$
end function
```

The `ADD` function is uninteresting but defined for completeness. The `COMPUTE` function, on the other hand, has some complexities and other difficulties. Namely, the setting of M and m, the integer number of groups to be computed and the integer number of slices per node to be computed, respectively. You can notice our algorithm (element-wise vector summation) is tightly coupled with the slicing/ division code. Even in such a simple computational problem, setting M and m can be complicated and difficult to get correct, distracting the researcher/developer from the real problem.

2.2 OpenCUDA+MPI Vector Summation

The user code will be run on all minion nodes of the MPI job. Further, because the user code doesn't have to deal with splitting the data or sending the data to other nodes, it is incredibly simple.

Algorithm 2 User code of the framework

```
function VECTOR_ADD($data$)                    ▷ Element-Wise Vector Summation
    $slice\_c \leftarrow data[0] + data[1]$
    return $slice\_c$
end function
```

The framework algorithm can be found in the appendix algorithm 3.

Once the user code is put together with the framework using algorithm 4 we will be given the same result as the first example (that doesn't use the framework). The benefit of the separation is that the user function is easier to manage and isn't coupled to other confounding code. Both of these niceties should grant the benefit of better and easier to understand distributed and parallel code.

3 Related Works

Current approaches to accelerating research computations and engineering applications include parallel computing and distributed computing. A parallel system is when several processing elements work simultaneously to solve a problem faster. The primary consideration is elapsed time, not throughput or sharing of resources at remote locations. A distributed system is a collection of independent computers that appears to its users as a single coherent system. In distributed computing the data is split into smaller parts and subsequently spread across the system. The result is then collected and outputted. Summary of current solutions are listed in table 1 below followed by detailed description of every entry.

Project	**Description**
`Hadoop`	Distributed computation framework [7] [4] [26]
`BOINC`	Volunteer and grid computing project distributed [12]
`GPUDirect`	GPU-to-GPU framework for parallel computing [3]
`MPI`	Message-passing system for parallel computations [28] [23]
`PVM`	Distributed environment and message passing system [5]

Table 1: Summary of Current Approaches and Projects

3.1 Hadoop

Hadoop provides a distributed file system and a framework for the analysis and transformation of very large data sets using the MapReduce paradigm [4] [21] [16] [15]. An important characteristic of Hadoop is the partitioning of data and computation across many (thousands) of hosts (nodes), and executing application computations in parallel close to their data [26]. The MapReduce [24] [8] paradigm which Hadoop implements is characterized by dividing the application into many small fragments of work, each of which may be executed or re-executed on any node in the cluster. The worker node processes the smaller problem, and passes the answer back to its master node. The answers to the subproblems are then combined to form the output. Hadoop is popular model for distributed computations, however, it is does not integrate with CUDA-enabled GPUs.

3.2 BOINC

The Berkeley Open Infrastructure for Network Computputing (BOINC) is an open source middleware system for volunteer and grid computing. The general public can contribute to today's computing power by donating their personal computer's disk space and some percentage of CPU and GPU processing. In other words, BOINC is software that can use the unused CPU and GPU cycles on a computer to do scientific computing [12]. The BOINC project allows for distributed computing using hundreds of millions of personal computers and game consoles belonging to the general public. This paradigm enables previously in-feasible research and scientific super-computing. The framework requires that individuals are connected to the Internet and is supported by various operating systems, including Microsoft Windows, Mac OS X and various Unix-like systems. Although BOINC may not seem relevant to our framework, it represents a different paradigm in the realm of parallel and distributed computing that our framework could use as an example for certain problems.

3.3 GPUDirect

Currently GPU based clusters are becoming more popular. `GPUDirect` is a GPU Remote Direct Memory Access (RDMA) communication specification using Infiniband. `GPUDirect` allows for multiple GPUs to communicate with each other directly by giving GPUs the ability to directly copy memory to another GPU [3]. Prior to GPUDirect, GPU-to-GPU communication involved the CPU. `GPUDirect` performance gain exceed CPU solutions. Direct GPU-to-GPU communication is less computationally expensive since less messages will be sent and received via the host server. An issue with `GPUDirect` is that it requires specific hardware.

3.4 MPI

MPI is the dominant message passing programming paradigm for clusters [1] [2]. MPI is a standardized and portable message-passing system that functions on a wide variety of parallel computers. Although the standard MPI defines the syntax and semantics of a core of library routines in the `C` programming language, many other languages offer Application Programmable Interface (API) call into MPI. `OpenCUDA+MPI` builds upon this model due to the fact that MPI is the dominant model used in high-performance computing [27]. MPI has been implemented for almost every distributed memory architecture and is optimized for the hardware on which it runs.

3.5 PVM

Parallel Virtual Machine (PVM) is a software tool for parallel networking of computers. It is designed to allow a network of heterogeneous Unix and/or Windows machines to be used as a single distributed parallel processor. Thus, large computational problems can be solved more cost effectively by using the aggregate power and memory of many computers. PVM enables users to exploit their existing computer hardware to solve much larger problems at less additional cost [5]. PVM was a step towards modern trends in distributed processing and grid computing but has, since the mid-1990s, largely been supplanted by the much more successful MPI standard for message passing on parallel machines.

3.6 OpenCUDA+MPI

The proposed `OpenCUDA+MPI` framework will take advantage of both MPI and CUDA to perform computations on GPU cards of computers participating in the cluster. `OpenCUDA+MPI`'s goal is to allow a collection of heterogeneous computers each containing a CUDA-capable GPU to be used as a coherent and flexible concurrent computational resource.

4 Methodology

Our framework will be developed using a number of tools and will also build upon a few already developed and mature software libraries. Overall, we plan to use the `Python` programming language, MPI for cross process communication, and CUDA for graphics computing.

4.1 Python

We will use `Python` programming language because of its ease of development and plethora of existing libraries such as `mpi4py` and `pyCUDA`. `Python` also adds some advantages when it comes to usability when needing more performance. For example, if we discover a need for certain aspects of the project to be tuned or otherwise run faster, we can easily switch to C or C++ and write sections of the program in a (more performant) native language.

4.2 MPI

Currently our cluster consists of 16 computers provided to our research lab by the Computer Science department of Boise State University. We will use MPI because it has established itself as the de-facto interface for cross process communication [28]. To allow for interfacing with `Python` [23], we will use `mpi4py` because of the library's maturity and implementation completeness.

4.3 CUDA

We will be using CUDA, and in particular, `pyCUDA`, because of existing knowledge of the library/ framework and also because it is a well established library for GPU computing.

4.4 `OpenCUDA+MPI`

Our frameworks' goal is to make accessible the power of cluster computing without necessarily possessing in-depth knowledge of the complexities of inter-computer related communication. Further, in doing so, we would like to expose functionality that may not be available even in more established cluster libraries and frameworks. Namely, facilities for debugging and profiling.

4.5 Testing

To evaluate the efficacy and accuracy of our framework several tests will be executed. As previously mentioned one of the first algorithms to be tested will be the problem of vessel extraction. Accurately extracting vascular structures from a Computerized Tomographic angiography–also called CT angiography scans–is important for creating oncologic surgery planning tools as well as medical visualization applications [17]. Currently, we use a single GPU to extract vascular structures from a CT angiography scan, which is computationally intensive.

The following test programs will be developed as time allows:

- N-Body Simulation
- Prime Number Searching

Every test program will be evaluated in three categories: CPU-only, CUDA-only, CUDA and MPI and `OpenCUDA+MPI`. Having all of the prior solutions that do *not* use the framework provides a baseline time comparison and provide immense insights into the pains and difficulties we are actually attempting to solve.

5 Results

5.1 Vector Summation

Our first developed test program was a 1 billion element wise vector summation problem. This is a simple and, as we will see, a bad example of using a cluster to speed up the computational time required. Although we did see an increase in performance, the cost of Input/ Output (I/O) far out weighs the benefit. Specifically, to do the computation on one machine (one CPU), it took a wall time of 254 seconds (about 4 minutes) and a CPU time of 13.7 seconds. Further, to do the computation using a single GPU took about 4172 seconds (wall time) or about 70 minutes, 13.83 seconds (CPU time) while the computing the summation of the cluster took about 3177 seconds (wall time) or about 50 minutes, 10.51 seconds (CPU time). Our CPU only implementation took the least amount of elapsed time, it took the second longest CPU time. Our CUDA only implementation took the longest in both wall time and CPU time and our cluster implementation was shortest CPU time but seconds longest elapsed time. The benefit of saving an upwards of 3.3 seconds is not worth the extra incurred cost of 2923 seconds. Not so surprisingly though, running the vector summation problem over the cluster *without* using CUDA does increase the overall elapsed time of the problem; namely, it took 226 seconds wall time (currently the correct CPU time is unable to be collected). Further, increasing the number of nodes part of the program's pool, decreases the wall time for each node.

5.2 N-Body Problem

The `N-Body` problem is a simulation of a system of N bodies and their gravitational (or other force) interaction with each other calculated over a set of time steps. Each time step will calculate and update positions and other attributes of each body in the system.

We have several sizes of the `N-Body` problem that we tested with: $2,001$ particles, $20,000$ particles, $200,000$ particles, $2,000,000$, and $20,000,000$ particles.

The computational times are for a single time step. The method for computing the gravitational potential is an adaptation of the Particle-Particle, Particle-Mesh (P3M) method. The benefit of using this method is we are able to nicely distribute the problem over the cluster and

Method	**Time (s)**	**Total Time (s)**
CPU Only	13.7	254.13
CUDA (Single Node)	13.83	4172
MPI + CUDA (7 nodes)	10.51	(average) 3177
MPI (7 nodes)		(average) 226
MPI (16 slots)		(average) 149

Table 2: Computational Timing Comparison of 10^9 element wise vector summation

/ or over a GPU (because of memory limitations) while maintaining a respectable accuracy for close body interactions. Further, if a grid contains more bodies than a specified threshold (in our case $200,000$), we can further sub-divide the grid to improve performance and maintain accuracy still.

There are other algorithms for computing `N-Body` problems on the CPU that are quite efficient, notably, the Barnes-Hut Tree algorithm[14]; however, using it would distort and confound the comparisons between CPU, GPU, and cluster implementations, not to mention the complexities of implementing such an algorithm on GPUs and over a cluster.

5.2.1 N-Body — CPU

In CPU tests, we were only able to complete several of the problem sizes; the larger sizes are infeasible. Notably, the smaller sizes were computed in a relatively respectable amount of time. While the bigger sizes were time consuming, not even attempted, or aborted. For example, the 2 million body problem was aborted after running for about 2 weeks.

Size	**User (seconds)**	**Sys (seconds)**	**Real (seconds)**
2001	28.81	0.02	30.77
20000	2382.40	2.27	2393
200000	113983	34.45	114349
2 million	aborted	aborted	aborted
20 million	N/A	N/A	N/A

Table 3: CPU N-Body simulation using particle-particle method

5.2.2 N-Body — GPU

As noted above, the GPU (CUDA) implementation uses the same computational method (P3M). Using the GPU, we were able to compute the $20,000,000$ size problem and may be able to compute larger sets within a *reasonable* amount of time. See table 4 for a breakdown of the running times.

Size	User (seconds)	Sys (seconds)	Real (seconds)
2001	10.08	1.52	14.14
20000	22.30	2.46	26.82
200000	44.63	4.84	52.86
2 million	186.59	21.23	217.60
20 million	1289.24	159.29	1510

Table 4: Single GPU (CUDA) N-Body simulation using P3M method

5.2.3 N-Body — 16 Node Cluster

Similar to the other solutions, we are still using the P3M method for computing a single time step of the N-Body problem. Over 16 nodes, we were able to see drastic improvements over CPU and CUDA implementations. Notably, in the $20,000$ problem size, the cluster did nearly 19044% better than the CPU and about 115% better than a single GPU. Further, in the $200,000$ problem size, we see the cluster did about $535,743$% better versus CPU and about 148% better versus GPU. See table 5 for times of each problem size.

Size	User (seconds)	Sys (seconds)	Real (seconds)
2001	N/A	N/A	N/A
20000	0.17	0.032	12.50
200000	0.147	0.028	21.34
2 million	0.15	0.025	97.76
20 million	0.15	0.06	950.045

Table 5: 16 node cluster N-Body simulation using P3M method

6 Conclusions

We have developed some baseline test solutions to a few problems. From the baseline solutions, we can easily tell there are significant performance increases from parallelizing code. However, there is still a complexity cost. The goal of `OpenCUDA+MPI` is to limit this complexity cost and from our really early versions of the framework, the outlook of being able to do just that looks very good.

6.1 Future Work

Work is continuing to progress on the development of the framework, but an early alpha version is nearly complete. As we continue to develop the framework, there are a few objectives we would like to achieve. Namely, we would like to add better debugging and profiling functionality, expose CUDA device initialization to the user, add automatic and/ or configurable checkpointing, and finish node configuration and administrative tasks.

A Appendix

A.1 Algorithms

Algorithm 3 `OpenCUDA+MPI` Framework Pseudo Code

```
function MASTER($size, world\_size$)        ▷ Split data as appropriate and send to nodes
    $card\_max \leftarrow query\_max\_mem()$
    $M \leftarrow floor((N + card\_max - 1)/card\_max)$
    $m \leftarrow floor((M + world\_size - 1)/world\_size)$
    for all $r < world\_size$ do
        $slice\_low \leftarrow (r * 2 + i) * card\_max$
        $slice\_high \leftarrow (r * 2 + m) * card\_max$
        Send indices from low to high to node of rank $r$
    end for
end function
function MINION($data, user\_fn$)        ▷ Receive Indices and Compute Results using user function
    $slice\_low \leftarrow recv()$
    $slice\_high \leftarrow recv()$
    $results \leftarrow user\_fn(data[slice\_low : slice\_high])$
    Send back results or write them to disk
end function
```

Algorithm 4 Combining the framework with user code

```
function MAIN($rank$)
    if $rank == 0$ then
        $master(\dots)$
    else
        $minion(\dots)$
    end if
end function
```

B Reference

[1] MPI: A Message Passing Interface Standard. http://www.mpi-forum.org/, June 1995.

[2] MPI-2: Extensions to the Message Passing Interface. http://www.mpiforum.org/, July 1997.

[3] GPU-Direct Accelerating GPU Cluster Performance. http://www.youtube.com/watch?v=o5Ir93SA8ZE, May 2010.

[4] Apache Hadoop. http://hadoop.apache.org/, April 2013.

[5] Computer Science and Division - PVM: Parallel Virtual Machine. http://www.csm.ornl.gov/pvm/, April 2013.

[6] CUDA C Programming Guide. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, May 2013.

[7] Hadoop Project Description. http://wiki.apache.org/hadoop/ProjectDescription, April 2013.

[8] MapReduce. http://wiki.apache.org/hadoop/MapReduce, April 2013.

[9] Open MPI. http://www.open-mpi.org/, 2013.

[10] TORQUE Resource Manager-Adaptive Computing. http://www.adaptivecomputing.com/products/open-source/torque/, May 2013.

[11] What is CUDA. https://developer.nvidia.com/what-cuda, 2013.

[12] David P Anderson. BOINC: A System for Public-Resource Computing and Storage. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 4–10. IEEE, 2004.

[13] Andreas Kloechner. PyCUDA Documentation, 2008.

[14] Barnes, Josh and Hit, Piet. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 1986.

[15] Dean, Jeffrey and Ghemawat, Sanjay. MapReduce: Simplified Data Processing on Large Clusters, OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004. *S. Dill, R. Kumar, K. McCurley, S. Rajagopalan, D. Sivakumar, ad A. Tomkins, Self-Similarity in the Web, Proc VLDB*, 2004.

[16] Dias, Jonas and Aveleda, Albino. HPC Environment Management: New Challenges in the Petaflop Era. In *High Performance Computing for Computational Science–VECPAR 2010*, pages 293–305. Springer, 2011.

[17] Erdt, Marius and Raspe, Matthias and Suehling, Michael. Automatic hepatic vessel segmentation using graphics hardware. In *Medical Imaging and Augmented Reality*, pages 403–412. Springer, 2008.

[18] Hadri, Bilel and Fahey, Mark and Jones, Nick. Identifying Software Usage at HPC Centers with the Automatic Library Tracking Database. In *Proceedings of the 2010 TeraGrid Conference*, page 8. ACM, 2010.

[19] Hindman, Benjamin and Konwinski, Andy and Zaharia, Matei and Ghodsi, Ali and Joseph, Anthony D and Katz, Randy and Shenker, Scott and Stoica, Ion. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, pages 22–22. USENIX Association, 2011.

[20] Hindman, Benjamin and Konwinski, Andy and Zaharia, Matei and Stoica, Ion. A Common Substrate for Cluster Computing. In *Workshop on Hot Topics in Cloud Computing (HotCloud)*, volume 2009, 2009.

[21] J. Venner. Pro Hadoop. http://www.apress.com/, June 2009.

[22] Johnson, C.R. Biomedical Visual Computing: Case Studies and Challenges. *Computing in Science & Engineering*, 14(1):12–21, 2012.

[23] Lisandro Dalcin. MPI for Python. http://mpi4py.scipy.org/docs/usrman/index.html, Janurary 2012.

[24] Luo, Yuan and Guo, Zhenhua and Sun, Yiming and Plale, Beth and Qiu, Judy and Li, Wilfred W. A Hierarchical Framework for Cross-Domain MapReduce Execution. In *Proceedings of the Second International Workshop on Emerging Computational Methods for the Life Sciences*, pages 15–22. ACM, 2011.

[25] Parker, Christopher and Suleman, Hussein. A Lightweight Web Interface to Grid Scheduling Systems. In *Proceedings of the 2008 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries: Riding the Wave of Technology*, pages 180–187. ACM, 2008.

[26] Shvachko, Konstantin V. Apache Hadoop: The Scalability Update. *;login: The Magazine of USENIX*, 36:7–13, 2011.

[27] Sur, Sayantan and Koop, Matthew J and Panda, Dhabaleswar K. High-performance and scalable MPI over InfiniBand with reduced memory usage: an in-depth performance analysis. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 105. ACM, 2006.

[28] Wes. A Comprehensive MPI Tutorial Reference. http://www.mpitutorial.com/, 2012.

C Acknowledgements

We would like to take this opportunity to thank everyone that helped and made this project a possibility. Thanks to the Boise State University's Student Research Initiative Fellowship for providing the motivation and funding. A special thanks to our mentor, Alark Joshi, PhD., giving us the high-level guidance and inspiration when it mattered most.

Glossary

CPU time time (typically in seconds) spent executing the process. 7

API Application Programmable Interface. 5

BOINC Berkeley Open Infrastructure for Network Computputing. 5

cluster A collection of computers networked together, typically with the goal of combining computational power. 3, 6–9

CPU Centeral Processing Unit. 2, 5, 7–9

CUDA Compute Unified Device Architecture. 2, 3, 5–9

GP-GPU General Purpose GPU. 2

GPU Graphics Processing Unit. 2–9

I/O Input/ Output. 7

Infiniband a networking fabric specification that defines a connection between compute nodes. 5

MPI Message Passing Interface. 3–7

node a computer, member of a cluster. 8

P3M Particle-Particle, Particle-Mesh. 7–9

PVM Parallel Virtual Machine. 6

RDMA Remote Direct Memory Access. 5

slot process space on a node, typically equal to the number of logical CPU cores on the node. 8

wall time all elapsed time (typically in seconds) that passes from start to finish of a program. 7