ELEC 490 Final Report

High Performance Computing with GPUs

Submitted By:

Group 17

Nikita Vakulin *(8nv6)*, Riley Shaw *(8rjs4)*, Timothy Livingston *(8tl13)*

April 5, 2013

Faculty Supervisor:

Dr. Ahmad Afsahi

Table of Contents

Executive Summary

A project was undertaken to improve the performance of a traditional CPU-based sequential program by modifying it for parallel execution in a GPU environment. A speedup of at least 1.5x and the preservation of the program's accuracy and integrity were outlined as the two key goals of the project.

Deal.II, a differential applications analysis library, was selected as the program to be modified. Modifications to the vector multiplication functionality of the program were made which materially improved the execution time of the solve class within deal.II.

After installing a baseline and modified version of the deal.II program on the Electrical and Computer engineering department cluster, multiple runs of a selected benchmark program were conducted in order to gather performance data.

The data obtained provided clear evidence of both a speedup and preservation of program accuracy. An average speedup of **1.59x** was obtained across all measured workloads, with a peak speedup of **2.44x** under optimal conditions. The program's accuracy was verified by comparing the output of the benchmark program with a baseline output.

The project was successful in accomplishing all of its initially stated goals while incurring no cost to the department; all resources were provided free of charge to the group.

1. Introduction

1.1 Purpose

This report is intended to describe and summarize the work and results of the ELEC 490 project undertaken by Group 17. The intended audience is the Course Instructor, our Faculty Supervisor, Dr. Ahmad Afsahi, and future ELEC 490 project groups.

Enough background information and explanation will be given so that the reader can gain a complete understanding of our project and its results.

It is also intended that this report serve as an aid to future groups undertaking the same type of project, as we anticipate they might encounter many of the issues and challenges that we overcame during the course of our work.

1.2 Background and Objectives

Modern Graphics Processing Units (GPUs) serve as highly parallel programmable processors capable of executing far more calculations per second than a Central Processing Unit (CPU) in certain applications.

Traditionally, the majority of a program's execution has been confined to the CPU. CPUs are optimized for latency; they can perform most tasks quickly and efficiently. Their strength is in sequential execution of complex algorithms. Modern CPUs have 2 - 8 cores, and can thus only execute 2 - 8 calculations at once.

GPUs, by contrast, are optimized for throughput. They have hundreds of computing cores and are thus able to execute hundreds of calculations at once. Each computing core on a GPU is significantly less powerful than its CPU counterpart, but the ability to have hundreds of threads running simultaneously outweighs this fact. Performance gains stem from the fact that multiple tasks complete at once; the execution time for these tasks must be added together to find the overall program execution time on a CPU.

With this in mind, our ELEC 490 project sought to identify an application which could be modified to execute in a highly parallel GPU environment. The purpose of these modifications would be to materially and measurably improve the performance of the application over the original CPU-based version while maintaining all of the original program's functionality.

To do so, the group made use of a nVidia Corporation's Compute Unified Device Architecture (CUDA) framework. CUDA provides a toolkit for programmers to implement parallel code on GPU hardware. Queen's University has recently begun offering courses which teach CUDA to students. Our supervisor, Dr. Afsahi, has been at the forefront of the introduction of this new computing paradigm to the Electrical and Computer Engineering department. This project is intended to validate the power of CUDA in increasing existing application performance, as well as to expand upon the general techniques used in parallelizing sequential applications.

An improvement in the performance of our selected application would have broad societal implications, particularly if the application sees wide use in industry and academia. To this end, we selected deal.II, a differential equations analysis library, as the application to be modified. Further discussion on the selection and suitability of deal.II can be seen in Sections 2.1 and 3.1.

1.3 Overview of Project Work

A wide variety of work was completed by the project group over the course of the academic year.

The fall term began with research into potential programs to modify. A list of programs which saw wide use in industry and academia was provided to the group by Dr. Afsahi, and was augmented with further discoveries. A full list of the programs considered, along with their websites, can be found attached to this report as Appendix A. Program research concluded with the selection of deal.II, a differential equations finite element analysis library. The criteria used to select this program are outlined in Section 3.1.

The group then turned to finding specific areas of potential improvement within the deal.II program. Particularly, functions and classes which could be parallelized effectively were noted.

The next order of business was to install deal.II on the department-provided cluster. This was to prove a major stumbling block, as the group was understandably not granted root access; this impeded progress at multiple points over the winter months. During this time, research into the CUDA framework also began. Through a series of test programs, the team familiarized themselves with the techniques and methodologies required for optimized parallelization.

With deal.II eventually installed, coding and modification of the program began in earnest. Having performed extensive research into CUDA over the fall term, the team was able to optimize the chosen deal.II functions.

Once the library was modified and operational, the next step was to rework deal.II's compilation process to accept the parallelized version of the original functions. This proved extremely difficult, but through trial and error the team was successful in compiling a version of deal.II which executed calculations on the GPU.

With the ability to deploy and test modifications on the cluster at will, the group's focus shifted to verifying functionality and fine-tuning performance. A benchmarking program (described in Section 4.1) was selected, and a baseline performance profile was established. All modifications made to deal.II could now be compared to the original sequential application.

Each of the above design process steps are covered in more technical detail in Section 3.

In the end, modifications made to the deal.II program allowed a large portion of the calculations within the application to be performed using the GPU. An average speedup of 1.59x was realized. Despite the significant performance gain, we feel that deal.ii was not as appropriate of a candidate as first postulated. This is detailed in Section 5.3.

2. Background and Motivation

2.1 General

As previously discussed in Section 1.2, the objectives of this project were to realize a measurable performance gain in a CPU-centric application by modifying it to execute in a highly parallel GPU environment.

In selecting an application to improve, we considered multiple factors including how widely the program was used. By choosing a program which had a large number of end users, any improvements in performance would be of great societal benefit to its user base, as numerous daily workflows would be expedited.

After extensive research, deal.II was selected due to its excellent online documentation, use of the C++ programming language, and reliance on vector multiplication to solve finite element analysis systems. Deal.II is a differential equations analysis library which is used in the fields of stress analysis, fluid dynamics, and other engineering applications both in industry and academia. Further explanation of the rationale behind our program selection can be found in Section 3.1.

Deal.II is centered around the manipulation of extremely large sparse matrices to solve systems of differential equations. The bulk of the execution time (see Figure 4) is spent in the solve class of the program, which makes heavy use of vector multiplication.

Vector multiplication is particularly well suited to execution on a GPU. On a traditional CPU, each cell of the final multiplied matrix must be calculated in order, one-by-one. When there are a large amount of elements in the vector, the execution time for this calculation becomes extremely long.

On the other hand, a GPU can calculate hundreds of cells at once, due to the presence of hundreds of computing cores. This greatly improves execution times under the right circumstances.

Our preliminary research into deal.II's source code suggested that a particular class of matrix within the program, `chunk_sparse_matrix`, was well suited to parallelization. We would begin our modification of the program's source code with this class.

2.2 Interface / Performance Specifications

Our two main goals in this project were to measurably improve the performance of deal.II, and to preserve the accuracy and integrity of the results the program generated. Our target speedup at the outset of the project was 1.5 - 2.0x. We felt that this represented a material performance improvement, and would significantly impact the experiences of researchers and scientists using the library for personal or academic use.

Maintaining platform compatibility, accuracy, and integrity within deal.II were also a primary concern throughout the project. All code we wrote would have to compile on multiple platforms and operating systems with various hardware configurations. This meant that we had to verify compilation on Windows and Macintosh platforms in addition to our OpenSUSE Linux cluster.

We also needed to implement our modifications in such a way that deal.II would still execute on a traditional CPU-based system if needed. This meant that we would have to implement a runtime check in our modified functions which would use the “old” code if no CUDA-compatible GPU was detected.

3. Design and Production Approach

3.1 Program Research

The first step in our design process was the selection of an appropriate application for parallelization. In selecting a program to modify, multiple factors were considered, including:

- Availability of documentation
- Open-source, accessible code
- Large proportion of runtime spent in repetitive, independent calculations
- Modular design written in C++
- Wide usage across academic and scientific communities

A chart summarizing each considered application's fit to the above criteria is included in Appendix B. Deal.II was selected for its apparent fit, with FDS and SAMRAI held as backup options. A wealth of online documentation is available for deal.II, and the authors of the program were quick to respond to questions which arose over the duration of the project.

Deal.II appears to be well-suited to parallel execution, as it relies on matrix-vector multiplication for the majority of its workload. A full breakdown of deal.II's execution profile is given in Section 4.2.

By choosing a program which had a large number of end users, any improvements in performance would be of real societal benefit.

For parallelization, nVidia Corporation's CUDA was selected as a robust and transferable GPU programming framework. Using CUDA's architecture, data can be transferred from CPU to GPU and calculations can be run in parallel. When all GPU operations are complete, the final values can be passed back to the CPU to continue program execution. By not only improving the performance of specific functions within a program, but also allowing two devices to operate simultaneously, CUDA allows for excellent performance gains.

3.2 Modification Research

Once deal.II was selected, research began into which classes and functions should be modified to most effectively increase the performance of the program.

Deal.II utilizes numerical techniques to find approximate solutions to partial differential equations. Computational finite element analysis consists of the iterative solving of extremely large sparse matrices. The essence of any finite element analysis program is the creation of tens of thousands to millions of elements that have laws regulating their states and interaction. The program iterates through the elements, solving their local interactions and reaching a solution through sheer brute force. Matrix operations, particularly matrix-vector multiplication, figure prominently in many classes and functions with the program's code.

Matrix multiplication is highly conducive to parallel execution on a GPU. On a traditional CPU, each cell of the final multiplied matrix must be calculated sequentially. When there are a large amount of elements in the vector, the execution time for this calculation becomes extremely long. On the other hand, a GPU can calculate hundreds of cells in parallel due to the presence of hundreds of computing cores. This greatly improves execution times under the right conditions.

As such, batch operations in the linear solver classes were selected as our primary modification target. Specifically, the `chunk_sparse_matrix` class contained the bulk of parallelizable solver functions.

Solving the system from within this class involves iteratively calling one of many solving methods on the system matrix. Any values associated with the system matrix are stored and modified within this class. `chunk_sparse_matrix` initiates large loops that iteratively solve the system one element at a time. The repeated usage of this class within deal.II makes it a prime candidate for optimization.

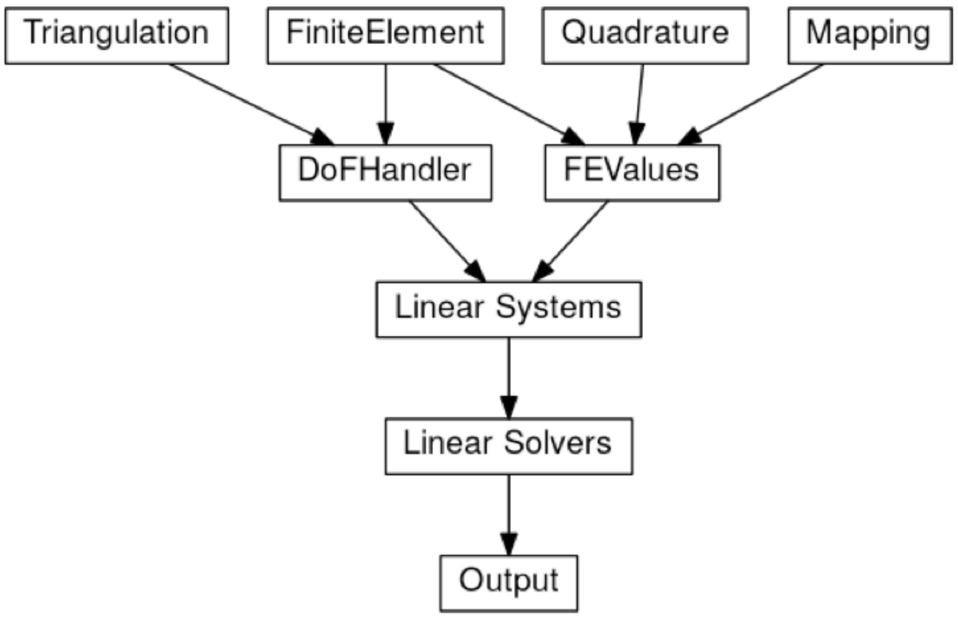


Figure 1 *Showing the structure of the deal.II program.*

From Figure 1, it should be evident that all classes must flow through the linear solvers block of the program, which consists mainly of `chunk_sparse_matrix`, to produce their output. Thus, any improvements made to this class influences the performance of the program as a whole. The bulk of the execution time (see Figure 4) is spent in the solve class of the program.

3.3 Deal.II Installation

In order to evaluate deal.II's performance on both the CPU and GPU, we first needed to install it on the computer cluster provided by the ECE department at Queen's. While this seemed simple at the outset, issues plagued the installation process. The group was understandably not given root access on the cluster; while this prevented us from damaging the systems, it also impeded our progress when attempting to install the programs and modules necessary to complete installation.

Greg McLeod, a computer technician in the ECE department, was extremely helpful and diligent in responding to our requests for package installation, granting of permissions, and other miscellaneous requests. Though Greg was as accommodating as can be expected, there was still an inevitable delay between our requests and their implementation on the cluster.

By having the installation of deal.II delayed, our progress in CUDA development was significantly impeded. The absence of a working install of the library created challenges with program modification and testing, as modified functions could not be compiled or tested. This led to a period of "blind coding", which can encourage disorganized and buggy code. To ensure program integrity, simulation programs such as that seen in Appendix C were created for testing.

3.4 Coding

Having identified `chunk_sparse_matrix` as our primary target for parallelization during the Modification Research portion of our design process, we explored this class further in search of specific functions which could be changed.

The lowest-level batch calculation within `chunk_sparse_matrix` is `chunk_vmult_add`, a function which is called repeatedly during the solve portion of deal.II's execution. To verify that changes to the source code were in fact making it through to the final program, we added print statements to the CPU version of the function and rebuilt the library. After verifying successful modification, our focus turned to rewriting `chunk_vmult_add` in CUDA.

The result of our CUDA rewrite was a new function contained in a file entitled *chunkvmultadd.cu.* This file contained both a "wrapper" function which could be called from the existing `chunk_sparse_matrix` class, and a “kernel” function, which performed the actual calculations on the GPU.

First, the function checks to ensure that a CUDA environment exists on the machine; if not, it will revert back to a sequential algorithm. If the environment is appropriate for parallelization, a passed matrix is multiplied with a source vector and stored in memory at a passed destination pointer. This function uses several parallelized optimization techniques, including tiling and DRAM optimization.

For more detail on the operation of *chunkvmultadd.cu*, consult the downloadable commented source code package whose address is given in Appendix D.

3.5 Compilation

Once modified using CUDA, `chunk_sparse_matrix` would not compile successfully into the final deal.II program. This was problematic, as without the ability to test a final, working version of our project, any results would simply be projections. Thankfully, we were eventually able to link our modified `chunk_sparse_matrix` file into the deal.II program.

To do so, `chunk_vmult_add` was moved to a separate *chunkvmultadd.cu* file and linked to from *chunk_sparse_matrix.templates.h* (both files can be seen in the downloadable Appendix D). The deal.II shared library, now including *chunkvmultadd.cu*, was then compiled and linked together for use in our benchmark program. The linux shell commands used to compile deal.II with CUDA modifications can be seen in Appendix E.

3.6 Benchmarking

The final piece of the design puzzle was to select an appropriate benchmarking program. On the suggestion of deal.II’s authors, we selected an existing tutorial program and modified it to record execution times, both for the overall program and for the various sections of the deal.II program.

The benchmarking methodology used to compile our project results in detailed in Section 4.1.

4. Testing and Results

4.1 Benchmarking Methodology

As stated in Section 2.2, our objective for this project was to achieve a measurable speedup by parallelizing deal.II while maintaining the original functionality and integrity of the program. In order to verify whether we had met our objectives, we selected and ran benchmarking software whose end result and CPU execution time were known.

For benchmarking purposes, we chose a script that was provided in the tutorial section of deal.II, and which was recommended for our use by the program's original author, Wolfgang Bangerth. This program is entitled '*step-6*' and its purpose is to calculate a highly complex solution to the Poisson equation over a domed surface. It is the last "general" tutorial and as such, has all the components of a complete finite element analysis program which one might see in industry or academic research.

The benchmark first breaks down a dome shaped object into a mesh of elements. It then defines the properties of each element and stores the whole system in a very large sparse matrix $\mathbf{M}$. The next step is to iteratively solve $\mathbf{Mx} = \mathbf{y}$ until the solution is achieved.

Finally, the program prints out the result as seen in Figure 2, below.

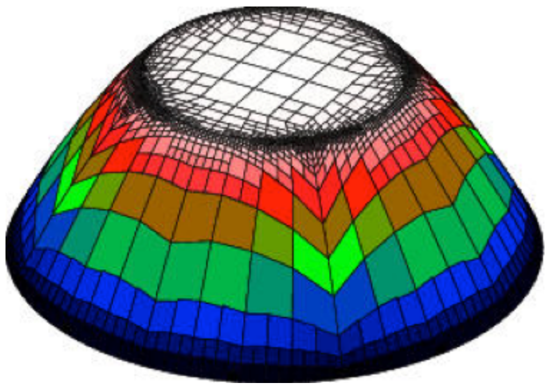


Figure 2 *The end result of the benchmarking program used to measure program performance.*

By observing the final output of the benchmark program after each modification to the deal.II library and comparing it to Figure 2, we were able to verify the program's integrity and ensure consistent and valid performance testing results.

Step-6 not only has all the components of a standard deal.II program, but is also structured such that there are clear transitions between each part of the application. The main method in *step-6* calls sections of the deal.II library called mesh, assemble, solve, and printout in a sequential manner. We initiated a clock at the beginning of the main method and took readings in between function calls to collect execution timing information, printing to the console to record our findings.

Another useful aspect of *step-6* is that after it has finished a round of calculations it refines the mesh imposed on the object and loops through each function again. This loop, which is located in the main class, allowed us to easily scale the computational intensity of the program and gather a range of results under varying workloads.

The modified *step-6* program is included in the downloadable source code appendix, whose address is given in Appendix D.

All results presented in the following sections stem from the execution of the *step-6* benchmark program. The group is confident that *step-6* is a valid representation of the average user's interaction with deal.II, and speedup values gleaned from its execution are representative of those which would be seen by the general public.

4.2 Results

Presented over the following pages are the results gained from execution of our modified deal.II library using our benchmarking software. Our parallelized version of deal.II exhibited a material performance speedup which met the 1.5x threshold outlined in Section 2.2 (Interface / Performance Specifications).

Sample Benchmark Output

A typical run of our benchmark resulted in the output seen in Figure 3 below:

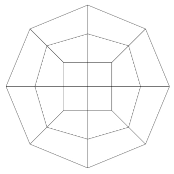
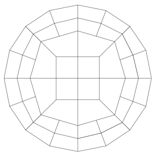
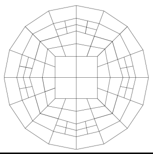
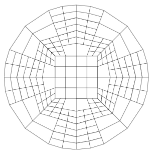
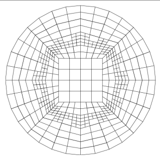
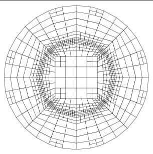
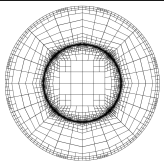
Cycle	Console Output	Graphical Output
0	[100%] Run step-6 with Debug configuration Setup time:0 Cycle 0: Number of active cells: 20 Number of degrees of freedom: 89 Assemble time:10 Solve time:50 Writeout time:50	
1	Cycle 1: Mesh time:10 Number of active cells: 44 Number of degrees of freedom: 209 Assemble time:30 Solve time:270 Writeout time:270	
2	Cycle 2: Mesh time:20 Number of active cells: 92 Number of degrees of freedom: 449 Assemble time:60 Solve time:2320 Writeout time:2320	
3	Cycle 3: Mesh time:50 Number of active cells: 200 Number of degrees of freedom: 921 Assemble time:130 Solve time:12440 Writeout time:12440	
4	Cycle 4: Mesh time:90 Number of active cells: 440 Number of degrees of freedom: 2017 Assemble time:300 Solve time:61220 Writeout time:61230	
5	Cycle 5: Mesh time:190 Number of active cells: 956 Number of degrees of freedom: 4425 Assemble time:750 Solve time:134230 Writeout time:134250	
6	Cycle 6: Mesh time:420 Number of active cells: 1916 Number of degrees of freedom: 8993 Assemble time:1810 Solve time:474630 Writeout time:474680	

Figure 3 *Output of a typical benchmark run - CPU based.*

It is important to note that the outputted *Mesh, Assemble, Solve* and *Writeout* times are cumulative within each cycle. That is to say that in cycle 1, when “Solve time” is given as 270, it has actually taken 270 (the outputted solve time) - 30 (the outputted mesh time) to perform the solve functionality in that cycle.

A video of a sample benchmark run can be seen at http://youtu.be/hojoMRxQE1U. Please note that this video was captured using an early version of our benchmark program, which was far less complex and thus executed in much less time. The output pattern and overall behaviour remain identical, however.

Using the outputted execution times from the CPU-based version of deal.II, we were able to compile an execution profile, given below in Figure 4. This confirmed to us that we were on the right path, as the bulk of the execution time of our benchmark program is spent in the *solve* classes of deal.II. The values below are taken from the benchmark run shown on the previous page.

Cycles	**Execution Time per Section (ms)**				
	Mesh	**Assemble**	**Solve**	**Writeout**	**Total**
0	N/A	20	**30**	0	50
1	10	20	**240**	10	280
2	20	40	**2,260**	10	2,330
3	40	90	**12,280**	10	12,420
4	80	210	**60,890**	10	61,190
5	190	560	**133,500**	30	134,280
6	410	1,370	**472,990**	50	474,820

Figure 4 *Showing the execution profile of the CPU-based deal.II program.*

At any number of cycles above 2, the solve class accounts for almost 99% of the program’s overall execution time. Improvements to this class thus have a significant effect on the program’s overall performance. As we are to see on the following page, modifications to the vector multiplication function within the solve class, particularly within the `chunk_sparse_matrix` data structure, produced a substantial speedup for deal.II.

To compare the performance of the traditional CPU-based program and our final GPU-based program, we performed 5 runs of the *step-6* benchmark program on each device and averaged the results. Our findings are presented below in Figure 5. A printout of the console output for a single benchmark run on the GPU is given in Appendix F. The GPU console output is extremely similar to that given for our sample benchmark output, though the execution times are obviously different.

Cycles	Total Execution Time (ms)		
	CPU	GPU	Speedup
0	50	30	**1.67x**
1	277	180	**1.56x**
2	2,339	1,390	**1.68x**
3	12,411	9,150	**1.36x**
4	61,180	25,060	**2.44x**
5	134,291	95,760	**1.40x**
6	474,793	452,210	**1.05x**

Figure 5 *A comparison of average (n = 5) CPU and GPU execution times for the benchmark program.*

It is evident that the GPU based program enjoys a significant speedup over its traditional CPU counterpart. There is some variation in performance improvement with increased workloads, but on average, a speedup of **1.59x** is achieved. A graphical representation of speedups under various workloads is given below in Figure 6.

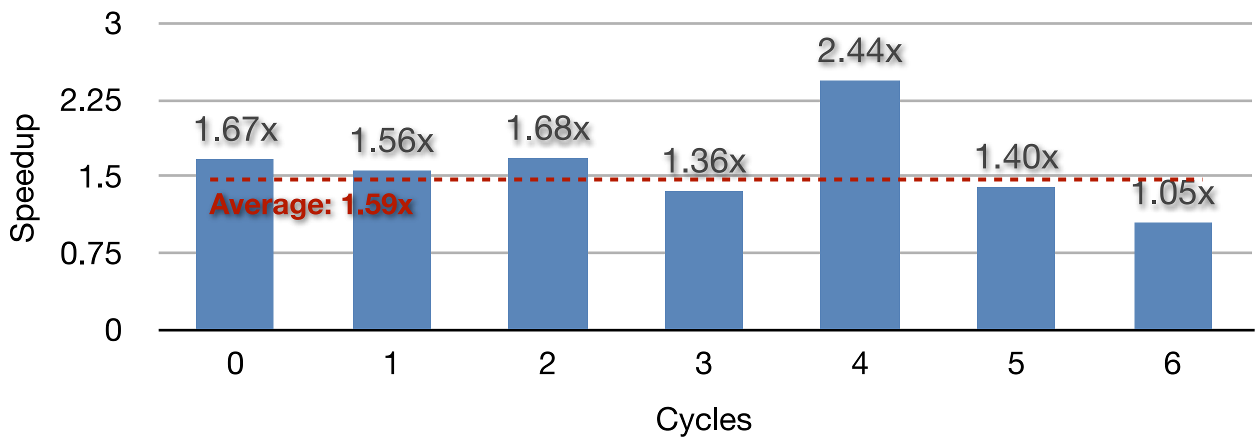


Figure 6 *Showing the speedup achieved under various workloads. Higher cycle counts indicate higher workloads.*

The group was ecstatic to have achieved our goal of a 1.5x speedup when executing our parallelized version of deal.II. Extensive modifications to the `chunk_sparse_matrix` class in addition to a rework of the vector multiplication behaviour within this class permitted us to see a significant reduction in execution time of our benchmark program.

Interestingly, there appears to be an inflection point in our speedups. Cycle counts beyond 4 still experience a speedup, but the speedup decreases with every additional cycle. This could possibly be due to the increased overhead of transferring very large blocks of memory to the GPU outweighing the performance advantage given by performing parallel vector multiplication.

4.3 Comparison With Simulated / Analytic Modeling

A high-resolution theoretical analysis of deal.II performance optimization is impractical on several levels; at its root, the main issue with analytic modeling for deal.II is that it is never run in isolation. Being a library, deal.II is extended from widely varied systems; for each distinct use case, execution flow and runtime distribution will differ.

A very general analytic model can be formed by assuming that almost all of the program's execution is spent in linear solvers and selecting arbitrary values for chunk size, data size, level of convergence, hardware architecture, and a number of other values. Ultimately, such a model is not very useful as it is not representative of any real system.

With this in mind, benchmarking deal.II across a range of applications has proven to be the best way to gauge overall execution time speedups. By doing so, we feel that we have conclusively proved that our modifications have substantially improved the performance of the program.

4.4 Reflections on deal.II

The algorithms implemented in deal.II convert rectangular sparse matrices into ragged matrices before operating on them. Notably, the main linear solver matrix multiplication function, `vmult_add` saves data into a structure with inconsistent row lengths. Though this is optimal for a sequential CPU-based system, it is inoperable with stateless GPU algorithms. Because of this, our parallelization needed to be tacked on to the final, albeit oft-repeated, function in the program, `chunk_vmult_add`. Other issues, such as the need for “doubling up” matrices in memory to prevent access issues and convert data structures, added to the feeling that CUDA was being shoehorned into a still largely sequential library. This was one of the group’s largest challenges, and barring time constraints it might have been prudent to reconsider our selection of deal.II as a vehicle.

5. Evaluation

5.1 Validity of Results

While the results presented in Section 4.2 exhibit a performance improvement over a baseline CPU-centric version of deal.II, the actual speedup that a user will see if they update their deal.II library to our modified version will depend on the type of problem that they are trying to solve. This variation in workload leads us to caution the reader that while a speedup was experienced for our particular benchmarking program, performance may vary in other applications.

We have made all efforts possible to ensure that the benchmark used is a typical finite element analysis problem, and thus indicative of a wide range of use-cases. It includes all steps of a standard analysis script, from initialization to printing results; this is important because although parallelization can reduce the amount of time taken to perform certain calculations, there is also overhead dedicated to other critical sequential tasks. We chose to record the amount of time that the program spends at each stage so that we could more accurately analyze our results.

Furthermore, we have conducted our trials over a wide range of workloads. This has allowed us to observe speedups and comment on how our program scales. To do this, we added a loop to the main function which refines the mesh for each iteration. This divides the object under analysis at greater resolutions for each iteration, increasing the computational intensity. Complexity is indicated by the dimensions of the system matrix; as shown in Section 4.2 we have obtained results for a wide range of parameters.

We conducted several iterations of each trial and averaged the results. We did this because we felt that it would give a more accurate result by removing any random fluctuations between trials. This was mostly unnecessary, as the resulting times were almost identical for each iteration that we performed. This indicates that our program is fairly stable.

5.2 Reliability

Following our initial design goals, we ensured that our modifications would be invisible to an outside observer. We did not change the way that `chunk_sparse_matrix` interfaces with the rest of the library; the edited `chunk_vmult_add` takes the same arguments and returns the same type of data as the original. This enables the function to exist in a "black-box" state, and ensures that the compiler will not throw undue errors.

The parallelized code that we have written can only be executed if the program is running on a machine that has a CUDA compatible GPU as well as all of the required packages. Due to this fact, we added conditional branches to the *chunkvmultadd.cu* wrapper function (line 63, file accessible in the downloadable Appendix D) to select which version of the code to run. This ensures that our modifications will not cause errors for non-GPU users.

5.3 Further Improvements

Below are some improvements which could be implemented to further improve deal.II's performance.

The simplest improvement to our existing modifications would involve expanding parallelization from `chunk_vmult_add` to some of the less-utilized functions within `chunk_sparse_matrix` that are similarly structured. `chunk_vmult_add` performs matrix multiplication on a chunk of the system matrix and adds the resulting vector to the solution. `Tvmult_add` performs the same operations, but on a transposed matrix. `chunk_vmult_subtract` also performs the same operations, but subtracts the partial result from the solution rather than adding it. Clearly, extending implementation to these functions would be trivial; the reason we have not done so for this report is that it was outside the scope of our original hypothesis, and the program calls the alternatives less frequently than `chunk_vmult_add`. Implementing these additional modifications could provide marginal speedup increases.

As mentioned in Section 4.4, proper optimization of the deal.II library implementing parallel best-practices would require a ground-up rewrite. As this was outside the scope of our project, steps were made to circumvent this issue rather than approaching it head-on. A finite element analysis library written with CUDA methodologies in mind from the start would certainly outperform our current build of deal.II.

Finally, it is worth noting that the deal.II library is very extensive and that we have only started to scratch the surface in terms of exploring its full functionality. There are likely areas of marginal improvement that we have not yet discovered; some classes, such as `sparse_matrix`, jump out as frequently called and readily parallelizable. This type of thorough examination is required for commercial production, but is far outside the scope of this project.

5.4 Team Evaluation

During the initial stages of the project, effective task delegation ensured that each team member had his own area of focus in the project. Because of this, work was largely completed independently. Each group member immersed himself in his own delegated area, and became an “expert” in his field. As passion for the project was high, each member took this very seriously. Riley completed two university-level courses on CUDA and parallel programming, and read several textbooks on the subject. Nikita read the deal.II documentation in its entirety and completed their extensive collection of training modules. Tim learned bash shell scripting, the G++ and CUDA compiler architectures, and the deal.II build process from source code download to final execution.

It became evident in later stages of the project that some of this work was misguided. As an example, deal.II’s architecture had some fundamental incompatibilities with parallel porting as discussed in Section 4.4; had Nikita and Riley discussed their work earlier it would have been evident that the ragged matrices of deal.II were incompatible with the stateless rectangular matrices required by CUDA. Similarly, had Tim and Riley worked closer together during the compilation phase, much time could have been saved in the specifics of CUDA file separation.

A lack of communication between group members resulted in considerable amounts of time spent on unneeded research, be it within an unused part of the framework or simply too deep in a hardly-utilized feature.

Despite the inefficiencies suffered from task-separation, the team performed exceptionally as a group. Members were passionate about their assigned tasks and willing to spend late nights working together. A focus on good organization held members accountable for timely deliverables. Each member exceeded expectations regarding their research goals, and because of this the proposed target was handily reached.

The team members are satisfied with their own work, and that of the team as a whole.

6. Social & Environmental Impact of Project

6.1 Social Impacts

As has been alluded to previously, an improvement in the performance of deal.II would bring immediate benefits for thousands of end users in industry, academia, and elsewhere.

By improving the execution time of the most common functions within the deal.II library, we would save users time when running their programs. This would allow them to either increase the complexity of their systems and maintain the same execution time, or leave their programs unchanged and enjoy an extra coffee break or two during the day as they complete their work faster than expected.

The group has engaged the authors of the deal.II program in discussions regarding the performance enhancements achieved. The initial response has been overwhelmingly positive. The authors have requested that the group keep them updated on all significant developments, and have broached the possibility of a future release of deal.II including the modified code which was the end result of this project.

6.2 Legal

Deal.II is an open source program, and is licensed under the Q Public License (QPL). Under this license, the program is free to copy and use, and users have free access to all source code.

However, the program remains the property of the deal.II authors, who have requested that “every publication presenting numerical results obtained with the help of deal.II should cite the reference on the publications page of the deal.II website”. We have provided this publication as a reference, which can be seen in Section 8, line item 1.

Beyond this consideration, there are few foreseen legal ramifications to this project.

6.3 Economic

Deal.II sees wide use in industry for a variety of purposes. Improvements in the application's performance will increase productivity at firms which make use of it in their design and production processes.

Basic economic theory tells us that increases in productivity and technology are the primary method by which a society improves its standard of living. By this reasoning, our project's impact on the wider economy, though comparatively minor, is still beneficial.

6.4 Sustainable Design Strategies

In making our modifications to the deal.II program, we were careful not to change the functionality of the application beyond internal modifications to certain functions to improve their performance.

It is important to conserve as much of a program's interactional structure as possible when making iterative modifications to improve performance. In this way, the accuracy of results and compatibility with other existing functions can be guaranteed. If drastic changes to the way a function was called and returned data were made, issues could cascade through the application as a whole and prevent successful execution. As has been previously mentioned in Section 5.2, conserving the structure of the `chunk_vmult_add` function was a key component in our success.

To be sustainable, program performance improvements must preserve the existing conventions within their parent application.

7. Conclusion

This report has detailed the objectives, design process, and results obtained for the ELEC 490 project undertaken by Group 17.

The project sought to improve the performance of a traditional CPU-based sequential program by modifying it for parallel execution in a GPU environment. A speedup of at least 1.5x and the preservation of the program's accuracy and integrity were outlined as the two key goals of the project.

Work performed over the course of the project began with the selection of deal.II, a differential applications analysis library, as the program to be modified. Research into deal.II's source code showed that the `chunk_sparse_matrix` class was the most viable candidate for parallelization.

After installing a baseline and modified version of the deal.II program on the ECE department cluster, multiple runs of a selected benchmark program were conducted in order to gather performance data.

The data obtained provided clear evidence of both a speedup and preservation of program accuracy. An average speedup of **1.59x** was obtained across all measured workloads, with a peak speedup of **2.44x** under optimal conditions. The program's accuracy was verified by comparing the output of the benchmark program with a baseline output.

Future work to which this project might lead includes the modification of additional classes and functions within deal.II to further improve performance. A more ambitious course of action would be to begin a ground-up redesign of deal.II for execution in a parallel manner. Some vestiges of sequential, CPU-based programming remained in our modified program, largely due to suboptimal data structures used in other areas of the program. A complete overhaul of deal.II would solve these issues, but is well outside the scope of this project.

8. References

1. Bangerth, Wolfgang. Hartmann, Ralf. and Kanschat, Guido. “deal.II — a general-purpose object-oriented finite element library”, ACM Transactions on Mathematical Software, vol. 33, no. 4, article 24. 1 Aug. 2007. Journal.

2. Bangerth, Wolfgang. Email interview. 26 Nov. 2012. Dates 11/26/12 through 03/25/2013.

3. Owens, John. "CS344." Introduction to Parallel Programming. NVIDIA Corporation and Amazon AWS. Udacity, Los Angeles. 4 Feb. 2013. Lecture.

4. Hwu, Wen-mei W. "CUDA Computing." Heterogeneous Parallel Programming. Coursera and the Department of Electrical and Computer Engineering. University of Illinois, Urbana-Champaign. 28 Nov. 2012. Lecture.

5. "CUDA Samples." CUDA Toolkit Documentation. Version 4. NVIDIA Corporation, 1 Jan. 2007. Web. 14 Oct. 2012. <http://docs.nvidia.com/cuda/cuda-samples/index.html>.

6. "Deal.II Library Documentation." deal.II Homepage. Version 1.8.1.2. The deal.ii Group, 28 Feb. 2013. Web. 2 Mar. 2013. <http://www.dealii.org/7.3.0/doxygen/deal.II/index.html>.

7. Hwu, Wen-mei W. GPU computing gems. Jade ed. Burlington, MA: Morgan Kaufmann/Elsevier, 2011. Print.

8. Hwu, Wen-mei W. GPU computing gems. Emerald ed. Burlington, MA: Morgan Kaufmann/Elsevier, 2011. Print.

9. Kirk, David, and Wen-mei W. Hwu. Programming massively parallel processors: a hands-on approach. Second ed. Amsterdam: Elsevier/Morgan Kaufmann, 2013. Print.

10. "Step-by-Step Examples." deal.II Homepage. Version 1.8.1.2. The deal.ii Group, 28 Feb. 2013. Web. 2 Mar. 2013. <http://www.dealii.org/7.3.0/doxygen/tutorial/index.html>.

List of Appendices

Appendix A

Programs Considered for Modification

1. DEAL II - http://www.dealii.org/
2. FDS - http://fire.nist.gov/fds/
3. H.264 Encoder - http://code.google.com/p/h264/
4. LAAMPS - http://lammps.sandia.gov/
5. MM5 - http://www.mmm.ucar.edu/mm5/
6. NAMD - http://www.ks.uiuc.edu/Research/namd/
7. SAMRAI - https://computation.llnl.gov/casc/SAMRAI/
8. ZEUS - http://www.netpurgatory.com/zeusmp.html

Appendix B

Program Suitability Analysis

Program	Language	Comments
DEAL II	C++	• Has already demonstrated scaling to 16K processors • Extensively documented • Small but dedicated online community
FDS	FORTRAN	• Legacy language makes modification difficult • FORTRAN not likely to lend itself well to parallelization
LAAMPS	C++	• Has already been extensively modified for parallel execution on GPUs • Any changes would be minor iterations and likely lead to only slight increases in performance
MM5	FORTRAN	• Legacy language makes modification difficult • FORTRAN not likely to lend itself well to parallelization
NAMD	C++	• Already extensively modified for execution on GPUs • Well suited to GPU execution due to high amount of independent computations
SAMRAI	C++	• Library of programs - large variety to select from • Smaller functions likely easy to modify and parallelize
ZEUS	FORTRAN	• Has been parallelized for multiple CPU cores with some success • Legacy language makes modification difficult

Appendix C

deal.II Simulated CUDA Program

```
        ////////// THIS IS EXAMPLE CODE AND NOT MEANT TO BE RUN!!! //////////
        //                                                                //
        // The following represents example code which was not actually    //
        // debugged for compilation. It is meant to serve as an example of //
        // the methods used for testing when in our period of "blind-      //
        // coding". The following borrows heavily from nVidia              //
        // Corporation's Matrix Multiplication (CUDA Runtime API Version)  //
        // toolkit sample.                                                 //
        //                                                                //
        ////////////////////////////////////////////////////////////////////
        //                                                                //
        // This file was designed to simulate the performance gains of     //
        // integrating CUDA into deal.II without having to rebuild and     //
        // compile the deal.II library. It performs batch matrix           //
        // multiplication on randomized matrices, in groups of varying     //
        // length to simulate the raggedness of deal.II's meta-matrices.   //
        //                                                                //
        ////////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <time.h>

__global__ void matrixMulCUDA(float *C, float *A, float *B, int wA, int wB) {

    //does not play well with padding to reduce code size to be able to fit on a screen

    int bx = blockIdx.x; int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;
    int bdx = blockDim.x; int bdy = blockDim.y;

    int aBegin = wA * bdy * by;
    int aEnd   = aBegin + wA - bdx; //used to be -1
    int aStep  = bdx;

    int bBegin = bdx * bx;
    int bStep  = bdy * wB;

    float Csub = 0.0;

    for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep) {
        __shared__ float As[ty][tx];
        __shared__ float Bs[ty][tx];

        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];
        __syncthreads();

#pragma unroll

        for (int k = 0; k < bdx; k++) //bdy would also work, since square
            Csub += As[ty][k] * Bs[k][tx];
        __syncthreads();
    }
    C[wB * bdy * by * ty + bdx * bx + tx] = Csub;
}

void matrixMulCPU(float *A, float *B, float *C, int ACols, int ARows, int BCols, int BRows) {
    for (int i = 0; i < ARows; i++)
        for (int j = 0; j < BCols; j++){
            float Csub = 0.0;
            for (int k = 0; k < ACols; k++)
                Csub += A[i * ACols + k] * B[k * BCols + j];
    C[i * BCols + j] = Csub;
}

void randInit(float *data, int size) {
    for (int i = 0; i < size; i++)
        data[i] = (float)rand()/(float)RAND_MAX;
}

/**
 * Run a simple test of matrix multiplication using CUDA
 */
int matrixMultiplyGPU(int argc, char **argv, int tile_side, dim3 &dimsA, dim3 &dimsB) {
    // Allocate host memory for matrices A and B
    unsigned int size_A = dimsA.x * dimsA.y;
    unsigned int mem_size_A = sizeof(float) * size_A;
    float *h_A = (float *)malloc(mem_size_A);
    unsigned int size_B = dimsB.x * dimsB.y;
    unsigned int mem_size_B = sizeof(float) * size_B;
    float *h_B = (float *)malloc(mem_size_B);

    // Initialize host memory
    randInit(h_A, size_A);
    randInit(h_B, size_B);

    // Allocate device memory
    float *d_A, *d_B, *d_C;

    // Allocate host matrix C
    dim3 dimsC(dimsB.x, dimsA.y, 1);
    unsigned int mem_size_C = dimsC.x * dimsC.y * sizeof(float);
    float *h_C = (float *) malloc(mem_size_C);

    if (h_C == NULL)
    {
        fprintf(stderr, "Failed to allocate host matrix C!\n");
        exit(EXIT_FAILURE);
    }

    cudaError_t error;

    error = cudaMalloc((void **) &d_A, mem_size_A);

    if (error != cudaSuccess)
    {
        printf("cudaMalloc d_A returned error code %d, line(%d)\n", error, __LINE__);
        exit(EXIT_FAILURE);
    }

    error = cudaMalloc((void **) &d_B, mem_size_B);

    if (error != cudaSuccess)
    {
        printf("cudaMalloc d_B returned error code %d, line(%d)\n", error, __LINE__);
        exit(EXIT_FAILURE);
    }

    error = cudaMalloc((void **) &d_C, mem_size_C);

    if (error != cudaSuccess)
    {
        printf("cudaMalloc d_C returned error code %d, line(%d)\n", error, __LINE__);
        exit(EXIT_FAILURE);
    }
    // copy host memory to device
    error = cudaMemcpy(d_A, h_A, mem_size_A, cudaMemcpyHostToDevice);

    if (error != cudaSuccess)
    {
        printf("cudaMemcpy (d_A,h_A) returned error code %d, line(%d)\n", error, __LINE__);
        exit(EXIT_FAILURE);
    }

    error = cudaMemcpy(d_B, h_B, mem_size_B, cudaMemcpyHostToDevice);

    if (error != cudaSuccess)
    {
        printf("cudaMemcpy (d_B,h_B) returned error code %d, line(%d)\n", error, __LINE__);
        exit(EXIT_FAILURE);
    }

    // Setup execution parameters
    dim3 dimBlock(tile_side, tile_side);
    dim3 dimGrid(dimsB.x / threads.x, dimsA.y / threads.y);

    // Create and start timer
    printf("Computing result using CUDA Kernel...\n");

        matrixMulCUDA<<<dimGrid, dimBlock>>>(d_C, d_A, d_B, dimsA.x, rand() / RAND_MAX * 
dimsB.x);

    printf("done\n");

    cudaDeviceSynchronize();

    // Allocate CUDA events that we'll use for timing
    cudaEvent_t start;
    error = cudaEventCreate(&start);

    if (error != cudaSuccess)
    {
        fprintf(stderr, "Failed to create start event (error code %s)!\n", cudaGetErrorString(er-
ror));
        exit(EXIT_FAILURE);
    }

    cudaEvent_t stop;
    error = cudaEventCreate(&stop);

    if (error != cudaSuccess)
    {
        fprintf(stderr, "Failed to create stop event (error code %s)!\n", cudaGetErrorString(er-
ror));
        exit(EXIT_FAILURE);
    }

    // Record the start event
    error = cudaEventRecord(start, NULL);

    if (error != cudaSuccess)
    {
        fprintf(stderr, "Failed to record start event (error code %s)!\n", cudaGetErrorString(er-
ror));
        exit(EXIT_FAILURE);
    }

    // Execute the kernel
    // using nIter iterations on random b widths to simulate deal.II's sparsity pattern
    int nIter = 300;

    for (int j = 0; j < nIter; j++)
        matrixMulCUDA<<<dimGrid, dimBlock>>>(d_C, d_A, d_B, dimsA.x, rand() / RAND_MAX * 
dimsB.x);

    // Record the stop event
    error = cudaEventRecord(stop, NULL);

    if (error != cudaSuccess)
    {
        fprintf(stderr, "Failed to record stop event (error code %s)!\n", cudaGetErrorString(er-
ror));
        exit(EXIT_FAILURE);
    }

    // Wait for the stop event to complete
    error = cudaEventSynchronize(stop);

    if (error != cudaSuccess)
    {
        fprintf(stderr, "Failed to synchronize on the stop event (error code %s)!\n", cudaGetEr-
rorString(error));
        exit(EXIT_FAILURE);
    }

    float msecTotal = 0.0f;
    error = cudaEventElapsedTime(&msecTotal, start, stop);

    if (error != cudaSuccess)
    {
        fprintf(stderr, "Failed to get time elapsed between events (error code %s)!\n", cu-
daGetErrorString(error));
        exit(EXIT_FAILURE);
    }

    // Compute and print the performance
    float msecPerMatrixMul = msecTotal / nIter;
    double flopsPerMatrixMul = 2.0 * (double)dimsA.x * (double)dimsA.y * (double)dimsB.x;
    double gigaFlops = (flopsPerMatrixMul * 1.0e-9f) / (msecPerMatrixMul / 1000.0f);
    printf(
        "Performance= %.2f GFlop/s, Time= %.3f msec, Size= %.0f Ops, WorkgroupSize= %u threads/
block\n",
        gigaFlops,
        msecPerMatrixMul,
        flopsPerMatrixMul,
        threads.x * threads.y);

    // Copy result from device to host
    error = cudaMemcpy(h_C, d_C, mem_size_C, cudaMemcpyDeviceToHost);

    if (error != cudaSuccess)
    {
        printf("cudaMemcpy (h_C,d_C) returned error code %d, line(%d)\n", error, __LINE__);
        exit(EXIT_FAILURE);
    }

// Checked many times. It works.
/*
    printf("Checking computed result for correctness: ");
    bool correct = true;

    for (int i = 0; i < (int)(dimsC.x * dimsC.y); i++)
    {
        if (fabs(h_C[i] - h_D[i]) > 1e-5)
        {
            printf("Error! Matrix[%05d]=%.8f, ref=%.8f error term is > 1e-5\n", i, h_C[i],
dimsA.x*valB);
            correct = false;
        }
    }

    printf("%s\n", correct ? "OK" : "FAIL");
*/
    // Clean up memory
    free(h_A);
    free(h_B);
    free(h_C);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    cudaDeviceReset();

/*    if (correct)
        return EXIT_SUCCESS;
    else return EXIT_FAILURE;
*/
}

int matrixMultiplyCPU(int argc, char **argv, int tile_side, dim3 &dimsA, dim3 &dimsB) {
    // Allocate host memory for matrices A and B
    unsigned int size_A = dimsA.x * dimsA.y;
    unsigned int mem_size_A = sizeof(float) * size_A;
    float *h_A = (float *)malloc(mem_size_A);
    unsigned int size_B = dimsB.x * dimsB.y;
    unsigned int mem_size_B = sizeof(float) * size_B;
    float *h_B = (float *)malloc(mem_size_B);
    // Create and start timer
    time_t before;
    time_t after;
    double seconds;

    // Initialize host memory
    randInit(h_A, size_A);
    randInit(h_B, size_B);

    // Allocate host matrix C
    dim3 dimsC(dimsB.x, dimsA.y, 1);
    unsigned int mem_size_C = dimsC.x * dimsC.y * sizeof(float);
    float *h_C = (float *) malloc(mem_size_C);

    if (h_C == NULL) {
        fprintf(stderr, "Failed to allocate host matrix C!\n");
        exit(EXIT_FAILURE);
    }


    printf("Computing result using C++ for() loops on the CPU...\n");

        matrixMulCPU(h_A, h_B, h_C, dimsA.x, dimsA.y, dimsB.x, rand() / RAND_MAX * dimsB.y);

    printf("done\n");

    // Record the start time
    time(&before);

    // Execute the kernel
    // using nIter iterations on random B widths to simulate deal.II's sparsity pattern
    int nIter = 300;

    for (int j = 0; j < nIter; j++)
        matrixMulCPU(h_A, h_B, h_C, dimsA.x, dimsA.y, dimsB.x, rand() / RAND_MAX * dimsB.y);

    // Record the stop time
    time(&after);

    seconds = difftime(before,after);
    float msecTotal = (float)seconds * 1000;

    // Compute and print the performance
    float msecPerMatrixMul = msecTotal / nIter;
    double flopsPerMatrixMul = 2.0 * (double)dimsA.x * (double)dimsA.y * (double)dimsB.x;
    double gigaFlops = (flopsPerMatrixMul * 1.0e-9f) / (msecPerMatrixMul / 1000.0f);
    printf(
        "Performance= %.2f GFlop/s, Time= %.3f msec, Size= %.0f Ops, WorkgroupSize= %u threads/
block\n",
        gigaFlops,
        msecPerMatrixMul,
        flopsPerMatrixMul,
        threads.x * threads.y);
    free(h_A);
    free(h_B);
    free(h_C);
}

/**
 * Program main
 */
int main(int argc, char **argv) {
    printf("[Matrix Multiply Using CUDA] - Starting...\n");

    if (checkCmdLineFlag(argc, (const char **)argv, "help") ||
        checkCmdLineFlag(argc, (const char **)argv, "?"))
    {
        printf("Usage -device=n (n >= 0 for deviceID)\n");
        printf("      -wA=WidthA -hA=HeightA (Width x Height of Matrix A)\n");
        printf("      -wB=WidthB -hB=HeightB (Width x Height of Matrix B)\n");
        printf("  Note: Outer matrix dimensions of A & B matrices must be equal.\n");

        exit(EXIT_SUCCESS);
    }

    // By default, we use device 0, otherwise we override the device ID based on what is provided
at the command line
    int devID = 0;

    if (checkCmdLineFlag(argc, (const char **)argv, "device"))
    {
        devID = getCmdLineArgumentInt(argc, (const char **)argv, "device");
        cudaSetDevice(devID);
    }

    cudaError_t error;
    cudaDeviceProp deviceProp;
    error = cudaGetDevice(&devID);

    if (error != cudaSuccess)
    {
        printf("cudaGetDevice returned error code %d, line(%d)\n", error, __LINE__);
    }

    error = cudaGetDeviceProperties(&deviceProp, devID);

    if (deviceProp.computeMode == cudaComputeModeProhibited)
    {
        fprintf(stderr, "Error: device is running in <Compute Mode Prohibited>, no threads can
use ::cudaSetDevice().\n");
        exit(EXIT_SUCCESS);
    }

    if (error != cudaSuccess)
    {
        printf("cudaGetDeviceProperties returned error code %d, line(%d)\n", error, __LINE__);
    }
    else
    {
        printf("GPU Device %d: \"%s\" with compute capability %d.%d\n\n", devID, deviceProp.name,
deviceProp.major, deviceProp.minor);
    }

    // Use a larger block size for Fermi and above
    int tile_side = (deviceProp.major < 2) ? 16 : 32;

    dim3 dimsA(5*2*tile_side, 5*2*tile_side, 1);
    dim3 dimsB(5*4*tile_side, 5*2*tile_side, 1);

    // width of Matrix A
    if (checkCmdLineFlag(argc, (const char **)argv, "wA"))
    {
        dimsA.x = getCmdLineArgumentInt(argc, (const char **)argv, "wA");
    }
    // height of Matrix A
    if (checkCmdLineFlag(argc, (const char **)argv, "hA"))
    {
        dimsA.y = getCmdLineArgumentInt(argc, (const char **)argv, "hA");
    }

    // width of Matrix B
    if (checkCmdLineFlag(argc, (const char **)argv, "wB"))
    {
        dimsB.x = getCmdLineArgumentInt(argc, (const char **)argv, "wB");
    }

    // height of Matrix B
    if (checkCmdLineFlag(argc, (const char **)argv, "hB"))
    {
        dimsB.y = getCmdLineArgumentInt(argc, (const char **)argv, "hB");
    }

    if (dimsA.x != dimsB.y)
    {
        printf("Error: outer matrix dimensions must be equal. (%d != %d)\n",
               dimsA.x, dimsB.y);
        exit(EXIT_FAILURE);
    }

    printf("MatrixA(%d,%d), MatrixB(%d,%d)\n", dimsA.x, dimsA.y, dimsB.x, dimsB.y);

    int matrix_result_gpu = matrixMultiplyGPU(argc, argv, tile_side, dimsA, dimsB);
    int matrix_result_cpu = matrixMultiplyCPU(argc, argv, tile_side, dimsA, dimsB);

    system("pause");

    exit(matrix_result_gpu == EXIT_FAILURE || matrix_result_cpu == EXIT_FAILURE);
}
```

Appendix D

Downloadable Source Code

All source files referenced throughout this report can be downloaded in one convenient zip file from the link below:

http://dl.dropbox.com/u/14032818/Group17490SourceFiles.zip

The contents of this file are as follows:

Within the "Modified" Folder:

1. *chunk_sparse_matrix.templates.h*

   This file contains the call to the parallelized *chunkvmultadd.cu*. In the CPU-based version of deal.II, vector multiplication is performed within this file.

2. *chunkvmultadd.cu*

   A CUDA-compileable file which contains the parallelized implementation of `chunk_vmult_add`. Contains both a wrapper function (called from *chunk_sparse_matrix.templates.h*) and a CUDA kernel, which does the actual vector multiplication.

3. *step-6.cc*

   The benchmark program used to measure performance improvements within deal.II.

Within the "Original" Folder

1. *chunk_sparse_matrix.templates.h*

   This file is the original implementation of `chunk_sparse_matrix` and its vector multiplication function, `chunk_vmult_add` (line 64).

Appendix E

deal.II Compilation Commands

1. Ensure that the updated *chunk_sparse_matrix.templates.h* file is placed in the deal.II/include/lac/ folder.

2. Run the follow shell commands in order:

3. The command below compiles the *chunkvmultadd.cu* file:

```
cd /home/engr/8tl13/final/source/lac/ && nvcc -m64 -arch='compute_20' -c chunkvmultadd.cu
```

4. The command below compiles the updated *chunk_sparse_matrix* class:

```
cd /home/engr/8tl13/final/source/lac && /usr/bin/c++     -DBOOST_NO_HASH -DBOOST_NO_SLIST -
DDEBUG -DTBB_DO_ASSERT=1 -DTBB_DO_DEBUG=1 -march=native -fpic -pedantic -Wall -Wpointer-arith
-Wwrite-strings -Wsynth -Wsign-compare -Wswitch -Wno-deprecated -Wno-deprecated-declarations
-std=c++0x -Wno-parentheses -Wno-long-long -I/home/engr/8tl13/final/source/lac -I/home/engr/
8tl13/deal.II/source/lac  -I/home/engr/8tl13/deal.II/bundled/tbb30_104oss/include  -I/home/
engr/8tl13/deal.II/bundled/boost-1.49.0/include  -I/home/engr/8tl13/deal.II/bundled/
functionparser  -I/home/engr/8tl13/final/include  -I/home/engr/8tl13/deal.II/include  -l/usr/
local/cuda/lib  -lcuda  -lcudart  -O0  -ggdb  -Wa,--compress-debug-sections  -o  CMakeFiles/
obj_lac.debug.dir/chunk_sparse_matrix.cc.o  -c  /home/engr/8tl13/deal.II/source/lac/
chunk_sparse_matrix.cc
```

5. The command below links the all of the deal.II shared library components to one file:

```
cd  ~/final  &&  make  -f  source/CMakeFiles/deal_II.g.dir/build.make  source/CMakeFiles/
deal_II.g.dir/build
```

6. The command below compiles the *step-6* benchmarking program:

```
cd  ~/final/examples/step-6/  &&  /usr/bin/c++  -DBOOST_NO_HASH  -DBOOST_NO_SLIST  -DDEBUG  -
march=native  -fpic  -pedantic  -Wall  -Wpointer-arith  -Wwrite-strings  -Wsynth  -Wsign-compare  -
Wswitch  -std=c++0x  -Wno-parentheses  -Wno-long-long  -O0  -ggdb  -Wa,--compress-debug-sections  -
I/home/engr/8tl13/final/include  -I/home/engr/8tl13/final/include/deal.II  -I/home/engr/8tl13/
final/include/deal.II/bundled  -o  CMakeFiles/step-6.dir/step-6.cc.o  -c  /home/engr/8tl13/final/
examples/step-6/step-6.cc
```

7. The command below links the CUDA *chunkvmultadd.o* object and the *step-6.o* object to create a runnable program:

```
cd  ~/final/examples/step-6/  &&  /usr/bin/c++  -march=native  -fpic  -pedantic  -Wall  -Wpointer-
arith  -Wwrite-strings  -Wsynth  -Wsign-compare  -Wswitch  -std=c++0x  -Wno-parentheses  -Wno-long-
long  -O0  -ggdb  -Wa,--compress-debug-sections  -Wl,--as-needed  -rdynamic  -pthread  -ggdb
CMakeFiles/step-6.dir/step-6.cc.o  /home/engr/8tl13/final/source/lac/chunkvmultadd.o  -o  step-6
-lcuda  -lcudart  -L/usr/local/cuda/lib64  -rdynamic  /home/engr/8tl13/final/source/
libdeal_II.g.so.8.0.pre -lm -ldl -lz -Wl,-rpath,/home/engr/8tl13/final/source
```

Appendix F

GPU Benchmark Console Output

```
Cycle 0:
   Number of active cells:       20
   Number of degrees of freedom: 89
Assemble time:10
Solve time:20
Writeout time:20
Cycle 1:
Mesh time:50
   Number of active cells:       44
   Number of degrees of freedom: 209
Assemble time:70
Solve time:190
Writeout time:190
Cycle 2:
Mesh time:70
   Number of active cells:       86
   Number of degrees of freedom: 417
Assemble time:110
Solve time:1390
Writeout time:1400
Cycle 3:
Mesh time:160
   Number of active cells:       176
   Number of degrees of freedom: 817
Assemble time:230
Solve time:9100
Writeout time:9100
```

```
Cycle 4:

Mesh time:180

   Number of active cells:       374

   Number of degrees of freedom: 1737

Assemble time:370

Solve time:24940

Writeout time:24950

Cycle 5:

Mesh time:370

   Number of active cells:       779

   Number of degrees of freedom: 3693

Assemble time:830

Solve time:93990

Writeout time:94020

Cycle 6:

Mesh time:570

   Number of active cells:       1574

   Number of degrees of freedom: 7515

Assemble time:1650

Solve time:451130

Writeout time:451170
```