Spring 2016

Performance Portable High Performance Conjugate Gradients Benchmark

Zachary Bookey
College of Saint Benedict/Saint John's University, zabookey@gmail.com

Performance Portable High Performance Conjugate Gradients Benchmark

AN ALL COLLEGE THESIS

College of St. Benedict/St. John's University

In Partial Fulfillment

of the Requirements for Distinction

In the Departments of Computer Science and Mathematics

by

Zachary Bookey

April, 2016

PROJECT TITLE: Performance Portability and High Performance Conjugate Gradients

Approved By:

Michael Heroux
Scientist in Residence of Computer Science

Imad Rahal
Associate Professor of Computer Science

Robert Hesse
Associate Professor of Mathematics

Imad Rahal
Chair, Department of Computer Science

Robert Hesse
Chair, Department of Mathematics

Emily Esch
Director, All College Thesis Program

Abstract

The High Performance Conjugate Gradient Benchmark (HPCG) is an international project to create a more appropriate benchmark test for the world's most powerful computers. The current LINPACK benchmark, which is the standard for measuring the performance of the top 500 fastest computers in the world, is moving computers in a direction that is no longer beneficial to many important parallel applications. HPCG is designed to exercise computations and data access patterns more commonly found in applications. The reference version of HPCG exploits only some parallelism available on existing supercomputers and the main focus of this work was to create a performance portable version of HPCG that gives reasonable performance on hybrid architectures.

Contents

1 Introduction

The High Performance Conjugate Gradients (HPCG) is an international project to create a more appropriate benchmark for the world's largest computers. The current High Performance Linpack (HPL) benchmark, which is the standard for measuring performance of the top 500 fastest computers in the world [2], is directing changes in a way that is only partially beneficial to many important parallel applications. HPCG was designed to complement HPL and move changes in a desirable direction. HPCG uses the preconditioned conjugate gradients method to solve a system of equations, that executes both dense computations with both high and low computational intensity such as sparse matrix-matrix multiplications and matrix-vector multiplications. The reference version of HPCG exploits only some parallelism available on existing supercomputers and the main focus of this work is to create a portable version of HPCG that gives reasonable performance on hybrid architectures.

The Trilinos project is an effort to facilitate the design, development, integration, and ongoing support of mathematical libraries for the solution of large scale, complex, physics and scientific problems [9]. While there are many packages in Trilinos to accomplish its goal, this project focuses specifically on the Kokkos package.

1.1 An Introduction to Iterative Methods

One of the most frequently encountered problems in scientific computing is solving systems of linear equations [12].

$$Ax = b \tag{1}$$

The most straightforward approach to solving the system is to do Gaussian elimination but this faces its own set of problems. On large problem sizes Gaussian elimination becomes slow and susceptible to roundoff error. Iterative methods can address these problems.

Iterative methods generate a sequence of solution vectors x so that the error between each x and the exact solution x^* is hopefully smaller than the previous x. There are many examples of this ranging from Jacobi, to Gauss Seidel, to successive over-relaxation method, to conjugate gradients. Iterative methods do not guarantee the finding of the exact solution but will guarantee finding a solution with an error within a user specified tolerance. Since calculating the actual error between x_i and x^* involves actually knowing the solution it is not a readily available measure of error. To account for this the idea of residuals, $r_i = b - Ax_i$, is used instead since if $x_i = x^* \rightarrow r_i = 0$. Thus in practice, iterative methods aim to reduce the norm of each residual in a way so that it converges to 0. Iterative methods that guarantee finding an exact solution are called direct methods, conjugate gradients falls under this categorization but is more often used as an iterative method due to it's fast convergence rate. Since this is such a broad topic, this thesis focuses on matrices A that are symmetric and positive definite. Thus the matrix satisfies the properties in Equation 2. These properties allow solutions to be found using the methods steepest descent, and conjugate gradients.

$$A = A^T \text{ and } \forall x \neq 0 \rightarrow x^T A x > 0 \tag{2}$$

2 Contributions to this Topic

This thesis provides a general introduction into iterative methods to solve a linear system of equations namely, steepest descent and conjugate gradients.

Secondly it demonstrates why conjugate gradients is a more optimal method than steepest descent. Thirdly it introduces the programming model created by the Kokkos package in the Trilinos Software Library. The contribution to the field provided by this thesis is a version of HPCG that utilized Kokkos to create a more performance portable reference version and compares it's performance to the reference version of HPCG.

3 Problem Statement

Computer architectures are evolving faster than ever and it has become increasingly difficult to write optimal code across the multitude of setups. To address this issue, Sandia National Laboratories has been working to develop the Kokkos software package to allow for one version of code to be written that performs optimally across multiple architectures. By successfully doing this it becomes quicker and simpler to test code implementations and computer architectures.

To accurately compare new architectures to existing ones a unified way of testing needs to be used. HPCG is the benchmark of choice but the reference version consists of minimal parallelism using OpenMP and MPI. The aim of this thesis is to mesh HPCG and Kokkos to create a performance portable benchmark that can be quickly used to test various architectures. Finally since HPCG uses the conjugate gradients method, we want to explore the efficiency of a variety of iterative methods and compare speed up in various parallelizations of the symmetric Gauss-Seidel preconditioner.

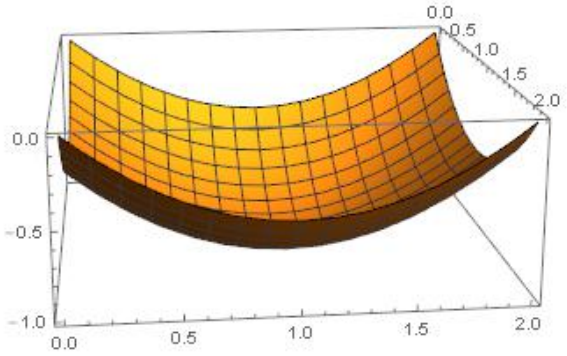


Figure 1: Plot of quadric form of a symmetric positive definite system

4 Background

4.1 Conjugate Gradients Method

The Conjugate Gradients Method is an iterative method where, given a symmetric and positive definite matrix A and a vector b it can find the solution vector x to Equation 1 which is done by minimizing the quadric form of the system:

$$f(x) = \frac{1}{2}x^T Ax - x^T b \tag{3}$$

Since A is symmetric and positive definite the solution to our system can be found at the minimum value of Equation 3 because:

$$\nabla f(x) = Ax - b \tag{4}$$

To help clarify if A is 2x2 matrix, Equation 3 looks like a 3-dimensional paraboloid, as seen in Figure 1, and the solution is at the very bottom of the paraboloid [16]. To best understand how Conjugate Gradients works it helps to first understand the method of Steepest Descent.

The Steepest Descent algorithm works like dropping a magical marble that

ignores friction into the paraboloid created by Equation 3. The marble will travel down the steepest path and then the next steepest path as it gets continually closer to the bottom but will take quite some time to stop if it ever completely stops. Steepest Descent starts with an initial guess x_0 and then aims to improve the guess by proceeding in the direction where $f(x)$ appears to change most rapidly which is equal to $-\nabla f(x_0)$. Notice how this is equal to the residual at x_0, $r_0 = b - Ax = -\nabla f(x_0)$. The only piece remaining is how far to travel in the direction of r_0 so $x_1 = x_0 + \alpha r_0$ so that x_1 reduces the error between itself and the exact solution and consequentially reduces the residual. From the exact line search framework [6] we get that:

$$\alpha = \frac{r_0^T r_0}{r_0^T A r_0} \tag{5}$$

Now all the pieces are in place and Algorithm 1 results. Steepest Descent is guaranteed convergence, however the rate at which it achieves this is often too slow [15]. Each residual vector is orthogonal to one another and thus create a basis for $\mathbb{R}^n$ and the exact answer should be the initial guess plus a linear combination of the residuals. Steepest Descent's downfall is that it uses scalars of the same residual vectors more than once. It would be more beneficial to instead only travel the exact distance needed in a direction and never travel that direction again. This is what the Conjugate Gradients Method achieves.

Algorithm 1 Steepest Descent

1: Choose the initial guess x_0

2: $k = 1$

3: **while** not close enough to solution **do**

4: $\quad r_{k-1} = b - Ax_{k-1}$

5: $\quad \alpha_k = \frac{r_{k-1}^T r_{k-1}}{r_{k-1}^T A r_{k-1}}$

6: $\quad x_k = x_{k-1} + \alpha_k r_{k-1}$

7: $\quad k = k + 1$

The idea behind Conjugate Gradients is similar to that of Steepest Descent. It still aims to minimize Equation 3, but instead of using the steepest gradient for a search direction it follows a set of n linearly independent directional vectors $p_1, p_2, p_3, ...$ traveling no further in a direction than needed to find the solution to Equation 1. Now the question remains of how to find these vectors? The answer is to find a set of n A-orthogonal vectors which means

$$p_i^T A p_j = 0 \text{ if } i \neq j \tag{6}$$

and this set of A-orthogonal vectors is linearly independent and thus form a basis for $\mathbb{R}^n$ due to the fact that A is positive definite. With this set of vectors the actual solution x_* can be found by

$$x_* = x_0 + \alpha_1 p_1 + \alpha_2 p_2 + ... + \alpha_n p_n \tag{7}$$

Again r_i is the residual vector at step i and the values for α_i can be found similar to the steepest descent but instead of using the residual vector as the direction, use the next vector from the A-orthogonal set.

$$\alpha_i = \frac{p_i^T r_{i-1}}{p_i^T A p_i} \tag{8}$$

Thus resulting in Algorithm 2 which assuming no rounding errors $Ax_n = b$ [5] will provide the exact answer to the solution after using each directional vector.

Algorithm 2 Orthogonal Directions

1: Choose the initial guess x_0

2: $k = 1$

3: **while** not close enough to solution **do**

4: $\quad r_{k-1} = b - Ax_{k-1}$

5: $\quad \alpha_k = \frac{p_k^T r_{k-1}}{p_k^T A p_k}$

6: $\quad x_k = x_{k-1} + \alpha_k p_k$

7: $\quad k = k + 1$

It's important to note that in computing, storing n n-dimensional direction vectors is expensive, especially for large values of n. So storing n directional A-orthogonal vectors is unwise and advised against. The conjugate gradients method remedies this by using the first residual vector as a direction and then generating the next A-orthogonal direction vector based on the fact that all the previous residual vectors are linearly independent. The Gram Schmit conjugation method can be used to find the set of A-Orthogonal vectors [16].

$$p_{k+1} = r_k + \frac{r_k^T r_k}{r_{k-1}^T r_{k-1}} p_k \tag{9}$$

This means it is no longer necessary to store all n direction vectors. In fact it is only necessary to store the previous residual vector if we remember $r_k^T r_k$ between iterations. Finally we get Algorithm 3 for the Conjugate Gradients method.

Algorithm 3 Conjugate Gradients

1: Choose the initial guess x_0

2: $r_0 = b - Ax_0$

3: $p_1 = r_0$

4: $k = 1$

5: **while** not close enough to solution **do**

6: $\quad \alpha_k = \frac{r_{k-1}^T r_{k-1}}{p_k^T A p_k}$

7: $\quad x_k = x_{k-1} + \alpha_k p_k$

8: $\quad r_k = r_{k-1} - \alpha_k A p_k$

9: $\quad p_{k+1} = r_k + \frac{r_k^T r_k}{r_{k-1}^T r_{k-1}} p_k$

10: $\quad k = k + 1$

Even though CG can be used to find the exact solution to a system of equation it converges fast enough that it sees most of it's use as an iterative method to approximate a solution. Due to this if the matrix A has a high condition number it becomes highly susceptible to error. To remedy this the matrix A can be preconditioned by a matrix C^{-1} so that we solve the system:

$$C^{-1}A(C^{-1})^T C^T x = C^{-1}b \tag{10}$$

Solving this system gives the solution $C^T x$ and can be brought back to our solution x by multiplying by $(C^{-1})^T$. This is only beneficial if the matrix $C^{-1}A(C^{-1})^T$ has a much smaller condition number than A. Consider Equation 10 as $\widetilde{A}\widetilde{x} = \widetilde{b}$ where the solution $\widetilde{x} = C^T x$. This results in the CG iteration performing the following:

$$\alpha = \frac{\widetilde{r}_k^T \widetilde{r}_k}{\widetilde{p}_k^T \widetilde{A} \widetilde{p}_k}$$

$$\widetilde{x}_{k+1} = \widetilde{x}_k - \alpha \widetilde{p}_k$$

$$\widetilde{r}_{k+1} = \widetilde{r}_k + \alpha \widetilde{A} \widetilde{p}_k$$

$$\beta = \frac{\widetilde{r}_{k+1}^T \widetilde{r}_{k+1}}{\widetilde{r}_k^T \widetilde{r}_k}$$

$$\widetilde{p}_{k+1} = \widetilde{r}_k + \beta \widetilde{p}_k$$

Now let $M = C^2$ and notice that $\widetilde{r}_k = \widetilde{b} - \widetilde{A}\widetilde{x}_k = C^{-1}(b - Ax_k) = C^{-1}r_k$ which leads to another rendition of the CG iterations:

$$\alpha = \frac{r_k^T M^{-1} r_k}{(C^{-1}\widetilde{p}_k)^T \widetilde{A} (C^{-1}\widetilde{p}_k)}$$

$$Cx_{k+1} = Cx_k - \alpha \widetilde{p}_k$$

$$C^{-1}r_{k+1} = C^{-1}r_k + \alpha C^{-1} A C^{-1} \widetilde{p}_k$$

$$\beta = \frac{r_{k+1}^T M^{-1} r_{k+1}}{r_k^T M^{-1} r_k}$$

$$\widetilde{p}_{k+1} = C^{-1}r_k + \beta \widetilde{p}_k$$

Finally by defining $p_k = C^{-1}\widetilde{p}_k$ and $z_k = M^{-1}r_k$ this becomes the preconditioned conjugate gradients algorithm which can be found in Algorithm 4 [6].

Algorithm 4 Preconditioned Conjugate Gradients

1: Choose the initial guess x_0

2: $r_0 = b - Ax_0$

3: Solve $Mz_0 = r_0$

4: $p_1 = z_0$

5: $k = 1$

6: **while** not close enough to solution **do**

7: $\quad \alpha_k = \frac{r_{k-1}^T z_{k-1}}{p_k^T A p_k}$

8: $\quad x_k = x_{k-1} + \alpha_k p_k$

9: $\quad r_k = r_{k-1} - \alpha_k A p_k$

10: $\quad p_{k+1} = z_k + \frac{r_k^T z_k}{r_{k-1}^T z_{k-1}} p_k$

11: $\quad k = k + 1$

12: $\quad$ Solve $Mz_k = r_k$

HPCG implements the preconditioning by solving the system $Az = r$, where A is the original matrix of the system and r is the residual vector. This is completed by using a single forward sweep symmetric Gauss-Seidel followed by a single backward sweep of symmetric Gauss-Seidel.

4.2 HPCG

HPCG was designed to complement the High Performance Linpack (HPL) benchmark test and better represent the needs of todays applications. It works by generating a sparse linear system that is mathematically similar to a finite element, finite volume or finite difference discretization of a three-dimensional heat diffusion problem on a semi-regular grid.[11] The system is then solved using a preconditioned conjugate gradient method where each sub-domain is preconditioned with a single symmetric Gauss-Seidel sweep. It's important to mention that HPCG uses some multi-grid to precondition our problem but this

will not be touched upon in this thesis.

HPCG has eight distinct phases:

1. **Setup Problem:** HPCG generates the matrix and vector structures for the system of equations $Ax = b$. Specifically this phase generates the matrix A, the vectors b and x. HPCG knows the exact solution to the system it generates so it creates the predetermined solution vector as a means to check accuracy later. Next it generates the information needed for the multi-grid component of preconditioning. For each matrix level it generates a smaller matrix than the previous level and its corresponding x and b vectors. Before HPCG 3.0 was released this segment of code was illegal to modify and not counted against the results generated in the end. Now with 3.0 the user is allowed to optimize this code and the amount of time it takes to generate the problem is counted against the end results. Changes to this section are only valid if the generated matrices and vectors pass the test found in "CheckProblem.cpp."

2. **Time Reference SpMV + MG:** HPCG times the reference versions of the function that computes sparse matrix vector multiplication (SPMV) and symmetric Gauss-Seidel (SYMGS). The results gathered from running the reference versions of these functions are included in the output report to help the user determine which kernel to focus on for optimizations as these kernels should be the most computationally intensive and thus reduce performance the most when not optimized.

3. **Time Reference CG:** This phase runs fifty iterations of the reference version of the conjugate gradients method (CG). The reference of CG and the optimized version of CG are the same except reference CG calls the reference version of all the compute kernels used where as the optimized version calls the optimized compute kernels. HPCG records the final re-

duction in residual reached after the fifty iterations and requires that the optimized version of CG reach the same reduction in residual even if that means more iterations are required.

4. **Optimize Problem:** This section of code calls the user specified `OptimizeProblem` function. Within this function the user is allowed to perform changes to the sparse matrix data structure to enable more optimal memory access patterns. It also allows for permutation of the system to better allow for parallelism in kernels like SYMGS. A provided optimization found in HPCG 3.0 will color the vertices of the matrix A which is beneficial in the parallelization of the SYMGS kernel.

5. **Test Validity:** This calls two separate validation functions that make sure that the user optimizations found in both SYMGS and SPMV produce valid results. The first of the tests is `TestCG`, this method replaces the diagonal of the matrix A to make much more diagonally dominant. With this change, it should only take a few iterations to CG to converge to a solution and even less with preconditioning. To pass this test the optimized CG code must converge in as few as twelve iterations without preconditioning and two with preconditioning. The second of the tests is `TestSymmetry` which, as the title suggests, confirms that our system stayed symmetric after optimization and that the optimized SPMV and SYMGS respect symmetry. To check SPMV it creates two randomly filled vectors x and y and verifies that $x^T Ay - y^T Ax$, which should be zero, is within some tolerance due to potential roundoff error. To check SYMGS it calls `ComputeSYMGS` to find $M^{-1}y$ and $M^{-1}x$ and then calculates $xM^{-1}y - yM^{-1}x$ to verify that it is also with in the same tolerance.

6. **Setup Optimized CG:** Here HPCG runs a single set of optimized CG to determine how many iterations are required to reach the reduction in

residual previously recorded in the "Reference CG Timing" phase. This phase also determines whether or not the optimized version of CG will converge and how many sets of CG are required to fill the user specified amount of time alloted for benchmark. In order for the results to be considered for submission HPCG must run for an hour. However as of HPCG 3.0 with proper permissions this is no longer the case.

7. **Time Optimized CG:** At this point in HPCG runtime, the timed analysis actually occurs. It runs as many optimized CG sets as determined in the previous phase and times how long it takes to complete the sets. HPCG also collects the residual norm of each set and later computes the mean and variance of all the norms to verify that both are within a tolerance. If not HPCG considers the result as not reproducible and therefore invalid.

8. **Report Results:** Finally HPCG produces an output YAML file and finalizes the ongoing creation of the log files used for debugging. This final phase is also responsible for deallocating any and all memory used by HPCG.

The HPCG output outlines specifically where the benchmark performed well and where it's bottlenecks were. For each major computational kernel, ie. sparse matrix vector product, HPCG will tell you how many giga floating point operations per second (GFLOP/s) were achieved, this is very handy for optimizing the code as it will tell you which algorithms to focus on and which are performing optimally.

4.3 Kokkos

Kokkos is a package in the Trilinos software repository that is designed to accommodate the multitude of computer architectures that exist in modern high

performance computing. Modern supercomputers may consist of many nodes and these nodes may carry a variety of different components. For example the node could contain multi-core processors that focus on powerful serial performance, it could contain many-core processors which sacrifices serial performance for more cores to run simultaneously, it could contain graphics processors which function like smaller many-core processors designed to handle long latencies, or finally it could contain a combination of these. Each of these components uses different parallel techniques found in various software packages to achieve maximum performance. Kokkos was created to abstract the data structures and parallel kernels so that the programmer could write one version of the code that would run at peak performance for the desired architecture specified at compile time. [17]

Kokkos has two main features: `Kokkos::View`, hereby referred to as `View`, polymorphic multidimensional arrays and parallel dispatch. At compile time the user specifies which `execution space` to run their parallel kernels from and which `memory space` to store data for `View`. Currently Kokkos supports the following `execution spaces`:

- `Serial`
- `PThreads` [1]
- `OpenMP` [14]
- `Cuda` [13]

4.3.1 `View`

`View` is a wrapper around an array of data that gives you option to specify which `memory space` to store the data on and to choose memory access traits to decide how this data will be accessed. `Views` handle their own memory management

via reference counting so that the structures automatically deallocate themselves when all the variables that reference them go out of scope, thus making memory management across multiple devices simpler for the programmer.

Since `View` allows the user to specify the `memory space` where it's data are located and which `execution space` to run parallel kernels from, the programmer needs to be wary of where in during runtime the data are accessed. If the `View` is stored on the host then its data is inaccessible from the device and similarly if the `View` is stored on the device it is inaccessible from the host. Kokkos addresses this issue through the use of `Kokkos::View::HostMirror`, hereby referred to as `HostMirror`, which is a `View` that has the same properties as the `View` it is derived from but now it's data will be stored on the host. The easiest way to create a `HostMirror` is to use the built-in function `Kokkos::create_mirror` to return the `HostMirror` of the lone `View` input. To get the data from the `View` to its `HostMirror` the function `Kokkos::deep_copy` needs to be called which does exactly what the name implies, it copies all the data from one `View` to the other. Kokkos never calls `Kokkos::deep_copy` implicitly and thus must be called by the programmer to copy date from one `View` to another, this gives the programmer greater control when it comes to managing data [3]. Nvidia solves this problem through Unified Virtual Memory (UVM) which Kokkos can use but was discovered to be too inefficient for our work.

4.3.2 Parallel Dispatch

Kokkos has three different parallel kernels, `parallel_for`, `parallel_reduce`, and `parallel_scan`. These kernels are all launched from either a functor or a lambda, however the latter can only be used if using the C++11 standard and is not fully supported with Cuda as it currently does not support lambdas from the host to be run on the device, Nvidia has stated that this will be a feature in later versions of Cuda. Kokkos allows for all of these kernels to be launched

nested within each other by creating a league of teams of threads.

`parallel_for` is a generic for loop that runs the whole context of the loop in parallel, each thread gets a range of iterations and runs until it completes it's iteration. To avoid race conditions, the loop must not have iterations that depend on results from other iterations. This type of parallel kernel is suited for algorithms like vector-vector addition by simultaneously adding corresponding entries with one another and storing the result in its proper location in the resulting vector.

`parallel_reduce` is like `parallel_for` but allows for each simultaneous iteration of the loop to update a single global value. Kokkos does not guarantee the order in which the iterations are processed so to avoid race conditions its best not to rely on a certain order of the iterations. Kokkos does however guarantee that the end result of the value being updated is deterministic if race conditions are not present. This kernel works well for algorithms like dot product by simultaneously multiplying the corresponding entries in the two vectors and adding the product to our global result.

`parallel_scan` is a kernel that keeps a running sum of values in an array and stores the sums in the array. For example, a scan over the array [1, 2, 3, 4] becomes [1, 3, 6, 10] or [0, 1, 3, 6] depending whether or not the scan is inclusive or exclusive, with inclusive resulting in the former. This kernel is useful in setting up sparse matrices by turning an array that carries the number of non-zero values per row into a map that tells what entries belong to which rows.

Kokkos permits for parallel kernels to be nested greatly increasing the potential for parallelism. This allows the programmer to run up to three levels of parallelism inside a single kernel. This works by creating a league of teams so that each team owns a specified number of threads. This can be furthered by

taking advantage of the vector lanes in each thread to run those in parallel as well. In essence this nested parallelism looks like three separate loops, the outer loop is run in parallel by the teams in the league, the middle loop is run by the threads in each team and finally the inner-most loop is run asynchronously by the vector lanes in the threads. Depending on the type of problem where this hierarchical is introduced, performance can significantly improve over just running the outer most loop in parallel, but at the same time if the algorithm is not suited for multileveled parallelism the increase in parallelism may not outweigh the increase in overhead.

5 Methods

5.1 Solving a System using Iterative Methods

Earlier, this thesis introduced three distinct iterative methods for solving a system of equations given that A is symmetric and positive definite. It was also pointed out that Conjugate Directions and Conjugate Gradients are actually direct methods for solving and that absent any round off error will produce the exact solution to the system in n = length of A iterations, whereas Steepest Descent is an iterative method that only guarantees convergence. To demonstrate this section will solve Equation 11 using Steepest Descent, Conjugate Directions, and Conjugate Gradients.

$$Ax = \begin{pmatrix} 4 & 1 & 1 & 0 & 1 \\ 1 & 3 & 1 & 1 & 0 \\ 1 & 1 & 5 & -1 & -1 \\ 0 & 1 & -1 & 4 & 0 \\ 1 & 0 & -1 & 0 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 6 \\ 6 \\ 6 \\ 6 \\ 6 \end{pmatrix} = b \tag{11}$$

Using Matlab the exact solution to the system in Equation 11 is found in Equation 12. Matlab code for all the given algorithms can be found in the Appendix.

$$x^* = \begin{pmatrix} 0.4516 \\ 0.7097 \\ 1.6774 \\ 1.7419 \\ 1.8065 \end{pmatrix} \tag{12}$$

5.1.1 Steepest Descent

The steepest descent algorithm requires nothing more than the system of equations from Equation 11 and an initial guess x_0 for our purposes the zero vector will do. Resulting in our residual vector $r_0 = b - Ax_0$.

$$x_0 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad r_0 = b - Ax_0 = \begin{pmatrix} 6 \\ 6 \\ 6 \\ 6 \\ 6 \end{pmatrix}$$

This is all the information needed to calculate α_0

$$\alpha_0 = \frac{r_0^T r_0}{r_0^T A r_0} = 0.192308$$

Putting these pieces together results with:

$$x_1 = x_0 + \alpha_0 r_0 = \begin{pmatrix} 1.1538 \\ 1.1538 \\ 1.1538 \\ 1.1538 \\ 1.1538 \end{pmatrix}$$

A second iteration through Steepest Descent results in the following:

$$r_1 = \begin{pmatrix} -2.0769 \\ -0.9231 \\ 0.2308 \\ 1.3846 \\ 1.3846 \end{pmatrix} \alpha_1 = 0.320151$$

and

$$x_2 = x_1 + \alpha_1 r_1 = \begin{pmatrix} 0.4889 \\ 0.8583 \\ 1.2277 \\ 1.5971 \\ 1.5971 \end{pmatrix}$$

Continuing as long as the norm of the residual in each iteration is within a tolerance of .00001, this algorithm needs 26 iterations to produce a result within the desired tolerance. Below are the recorded values for x_{14} and x_{26} to try and get a feel for how the rate of convergence slows.

$$x_{14} = \begin{pmatrix} 0.4515 \\ 0.7098 \\ 1.6772 \\ 1.7418 \\ 1.8064 \end{pmatrix} x_{26} = \begin{pmatrix} 0.4516 \\ 0.7097 \\ 1.6744 \\ 1.7419 \\ 1.8065 \end{pmatrix}$$

Clearly not the most efficient algorithm for this system of equations since both Conjugate Directions and Conjugate Gradients are guaranteed to produce the exact solution in five iterations for this particular system.

5.1.2 Conjugate Directions

Conjugate Directions is almost computationally identical to steepest descent but instead of traveling down the larges gradient it uses a set of A-Orthogonal vectors to determine which direction to travel and calculates exactly how far it needs to travel. Again using the system found in Equation 11, the zero vector can be used as an initial guess x_0, and with a bit of work the set $P = \{p_1, p_2, p_3, p_4, p_5\}$ can be found where each p_i is A-Orthogonal to p_j when $i \neq j$.

$$p_1 = \begin{pmatrix} -0.5955 \\ -0.7545 \\ 2.1364 \\ 0.7227 \\ 1 \end{pmatrix} p_2 = \begin{pmatrix} -3 \\ 2 \\ -2 \\ -1 \\ 12 \end{pmatrix} p_3 = \begin{pmatrix} 1 \\ -4 \\ 0 \\ 1 \\ 0 \end{pmatrix} p_4 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} p_5 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Given the initial conditions, the first iteration can be run using p_1 by first

easily finding r_0 and α_0. Then putting all the pieces together to get x_1

$$r_0 = b - Ax_0 = \begin{pmatrix} 6 \\ 6 \\ 6 \\ 6 \\ 6 \end{pmatrix} \text{ and } \alpha_0 = \frac{p_1^T r_0}{p_1^T A p_1} = 0.859084$$

$$x_1 = x_0 + \alpha_0 p_1 = \begin{pmatrix} -0.5115 \\ -0.6482 \\ 1.8353 \\ 0.6209 \\ 0.8591 \end{pmatrix}$$

It is simple to continue through the second iteration to find the following:

$$\alpha_1 = 0.859084 \text{ then } x_2 = \begin{pmatrix} -0.7484 \\ -0.4903 \\ 1.6774 \\ 0.2419 \\ 1.8065 \end{pmatrix}$$

Notice that by comparing this to the exact solution found in Equation 12 that the value in the third and fifth row are exactly where they should be. This goes back to how Conjugate directions only travels as far as it needs to in each search direction and the remaining search directions don't move orthogonal to those

axes, thus these values are exact. To further see this look at x_3 below.

$$x_3 = \begin{pmatrix} -1.0484 \\ 0.7097 \\ 1.6774 \\ 0.2419 \\ 1.8065 \end{pmatrix}$$

Since the only remaining search directions are along two different unit vectors every value with the exception of those units are exact and the next iteration will solve its search directions corresponding unit. The final two produced vectors are:

$$x_4 = \begin{pmatrix} -1.0484 \\ 0.7097 \\ 1.6774 \\ 1.7419 \\ 1.8065 \end{pmatrix} \quad x_5 = \begin{pmatrix} 0.4516 \\ 0.7097 \\ 1.6774 \\ 1.7419 \\ 1.8065 \end{pmatrix}$$

Since Conjugate Directions is a direct method $x_5 = x^*$ as long as there is no round-off error. Conjugate Directions is not used as an iterative method because if the search directions are not chosen carefully it may not converge at all until the final search direction has been used. This method is also highly impractical for large systems since it requires a set of A-Orthogonal vectors to be known in advance which may be difficult to do and, if the size of the system is large enough, too expensive to store in memory. It's interesting to note that with this algorithm, if you guess the correct search direction right away this method could converge on the first iteration.

5.1.3 Conjugate Gradients

Conjugate gradients remedies the issues raised in Conjugate Directions by using the residuals to compute the next search direction which avoids storing all the directional vector and allows for early convergence. Once the current directional vector is created the operations are the same as conjugate directions. So again it begins with the initial guess x_0 as the zero vector and the initial residual vector $r_0 = b - Ax_0$.

The initial search direction p_1 is chosen to be r_0 since there are no other search directions to be A-Orthogonal to. Now α_0 and x_1 are calculated as shown below.

$$\alpha_0 = \frac{r_0^T r_0}{p_1^T A p_1} = 0.192308 \text{ and } x_1 = x_0 + \alpha_0 p_1 = \begin{pmatrix} 1.1538 \\ 1.1538 \\ 1.1538 \\ 1.1538 \\ 1.1538 \end{pmatrix}$$

At this stage in each iteration CG calculates the next residual as such:

$$r_1 = r_0 - \alpha_0 A p_1 = \begin{pmatrix} -2.0769 \\ -0.9231 \\ -.2308 \\ 1.3846 \\ 1.3846 \end{pmatrix}$$

Notice that the residual is not calculated in the traditional fashion of $r_1 = b - Ax_1$ but the two equations can easily be shown to be equivalent. The interpretation shown is preferred since Ap_i needs to be calculated earlier in the iteration and by reusing Ap_i, CG can avoid doing unnecessary matrix-vector

multiplications.

The iterations that proceed after the first iteration are slightly different since CG needs to calculate a new directional search vector. To do this CG uses the fact that each residual vector is linearly independent to one another and the Gram Schmidt Conjugation to produce the next A-Orthogonal vector from the set of linearly independent residual vectors [16]. Giving the following computations.

$$\beta_1 = \frac{r_1^T r_1}{r_0^T r_0} = 0.0503 \text{ and } p_2 = r_1 + \beta_1 p_1 = \begin{pmatrix} -1.7751 \\ -0.6213 \\ 0.5325 \\ 1.6864 \\ 1.6864 \end{pmatrix}$$

With p_2 in place the rest of the iteration follows as the previous one.

$$\alpha_1 = 0.349407 \text{ and } x_2 = \begin{pmatrix} 0.5336 \\ 0.9368 \\ 1.3399 \\ 1.7431 \\ 1.7431 \end{pmatrix} \text{ and } r_2 = \begin{pmatrix} -0.1542 \\ -0.4269 \\ 1.3162 \\ -0.5692 \\ -0.1660 \end{pmatrix}$$

The rest of the iterations proceed in the exact same manner as this and in the provided example, CG requires all five iterations to converge to a solution. However since CG is also a direct method the fifth iteration gives us the exact solution if round off error is not present. Below are the remaining results of

each iteration and the search vector used to find it.

$$p_3 = \begin{pmatrix} -0.6031 \\ -0.5840 \\ 1.4509 \\ -0.1426 \\ 0.2605 \end{pmatrix} \text{ and } x_3 = \begin{pmatrix} 0.3972 \\ 0.8047 \\ 1.6680 \\ 1.7108 \\ 1.8020 \end{pmatrix}$$

$$p_4 = \begin{pmatrix} 0.1207 \\ -0.2054 \\ 0.0089 \\ 0.0162 \\ 0.0696 \end{pmatrix} \text{ and } x_4 = \begin{pmatrix} 0.4415 \\ 0.7292 \\ 1.6713 \\ 1.7168 \\ 1.8276 \end{pmatrix}$$

$$p_5 = \begin{pmatrix} 0.0314 \\ -0.0609 \\ 0.0192 \\ 0.0783 \\ -0.0658 \end{pmatrix} \text{ and } x_5 = \begin{pmatrix} 0.4516 \\ 0.7097 \\ 1.6774 \\ 1.7419 \\ 1.8065 \end{pmatrix}$$

This system is small, thus it is unsurprising that it took all five iterations to converge to a solution using CG. However this can be improved further by implementing the same preconditioning as found in HPCG. By using the Preconditioned Conjugate Gradients method found in HPCG this system converges in only four iterations with our tolerance of .000001.

The Matlab code for all the iterative methods introduced can be found in the appendix.

5.2 Merging Kokkos and HPCG

The goal for this project is to create a version of HPCG that produces valid results across many architectures without sacrificing performance. The Kokkos library was decided to be the best available tool to provide performance portability for the C++ code. Thus it was chosen to re-factor HPCG.

General Strategy for Kokkos re-factoring includes:

- Replace custom data types with Kokkos multidimensional arrays;
- Replace the parallel loops with Kokkos parallel dispatch;
- Code optimizations to improve usage of co-processors.

5.2.1 Replacing Custom Data Types

Prior to HPCG 3.0 the changes made to HPCG involved replacing every array that HPCG created with a `Kokkos::View`. However this was considered an illegal optimization and in HPCG 3.0 the correct optimizations were made by creating two new data structures: `Optimatrix` and `Optivector`. As the names suggest these new structures held the Kokkos information needed for our optimized matrix and vector respectively.

`Optimatrix` is arguable the more interesting of the two since it contains a new implementation of the sparse matrix created by HPCG. The original HPCG sparse matrix held data in a 2-dimensional array in which accessing `matrixValues[i][j]` would grab the jth non-zero value in the ith row. To inquire what column entry this value has you would have to access another array with the identical i and j values. The Kokkos implementation, `Kokkos::CrsMatrix`, of this array removed the 2-dimensional implementation and used a 1-dimensional approach. The `CrsMatrix` consists of three 1-dimensional `Views` namely values, entries, and row_map. Values holds all the non-zero values found in the array

sorted by row and then columns, so the value in position [2,5] appears before [3,2] but after [1,8]. Entries stores all of the corresponding column positions for values. Finally row_map stores where each row begins in both entries and values. So `row_map[i]` returns the starting position of row i in values and entries. `Optimatrix` also holds relative information needed for various optimizations to the symmetric Gauss-Seidel optimizations. Less interestingly `Optivector` contains a 1-dimensional `View` that stores a copy of the original HPCG vector data.

To fill the newly created data structures, HPCG had to first generate it's own data structures per legality requirements. After those had been created we were allowed to use the information stored in these structures to create the new structures, however since the data trying to be copied is stored on the host and the new structures may be on the device, Kokkos parallel dispatch was unavailable to move data in parallel and thus is done in serial. It is possible to exploit other types of parallel dispatch to parallelize this chunk of code but that is outside the scope of this project.

5.2.2 Replacing Parallel Kernels

HPCG has a specific set of compute kernels that can be optimized, some of these include `ComputeSPMV`, `ComputeDotProduct`, etc... Reference HPCG has given these functions basic parallelism through the use of OpenMP and this part of the project was to replace this basic parallelism with Kokkos parallel dispatch. A basic example of a for loop being converted to a Kokkos kernel is presented in Figure 2. The outer loop is replaced with `Kokkos::parallel_for` and the inner workings of the loop are moved into the functor.

Most compute kernels were straightforward to replace with Kokkos and most of our efforts were spent on replacing the `ComputeSPMV` and `ComputeSYMGS` kernels. By nature of these kernels, HPCG spends most of it's runtime in one

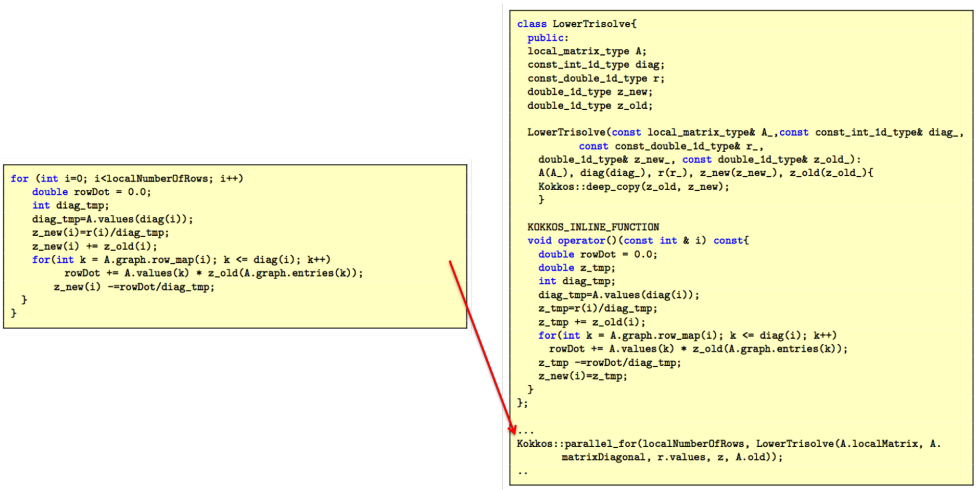


Figure 2: An example of "nested loop to Kokkos kernel" conversion.

or the other, with a preference towards `ComputeSYMGS`. Since HPCG spends most of it's time here it made sense to focus on optimizing these kernels to best improve performance. `ComputeSPMV` was easy to transform into Kokkos as there is already a sparse matrix vector multiplication function built into Kokkos, since this is already highly optimized for Kokkos it is used instead of my own interpretation. `ComputeSYMGS` is inherently serial as the algorithm:

$$x_i^k = (b_i - \sum_{j=1}^{i-1} a_{ij} x_j^k - \sum_{j=i+1}^{n} a_{ij} x_j^{k-1})$$

has forward dependencies depending on which row of the matrix A we are working on.[8] Thus in order to parallelize, the matrix had to be broken into chunks that that could safely be processed without running into dependencies within each chunk. This resulted in two options *level solve* and *color solve*.

5.2.3 Code Optimizations

This subsection will use the following matrix to demonstrate how each optimization breaks the matrix into chunks.

$$A = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} \tag{13}$$

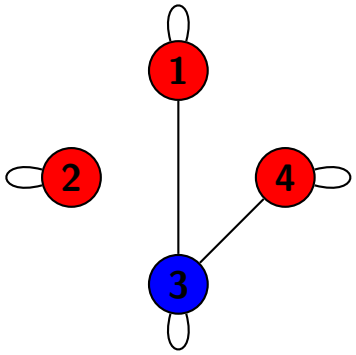
The level solve algorithm works by splitting our matrix A into parts L and U where L and U are the lower and upper diagonal matrices that include the diagonal. For this algorithm to work we introduce another data structure `Levels` inside of `Optimatrix` which stores all the information needed to break our matrix A into distinct levels. When optimizing the problem we find dependencies in solving the system $Lx = b$ and sort the rows based upon those dependencies such that a row will be contained in level i if and only if all of it's dependencies are in level j where $j < i$. In order to maintain symmetry level solve breaks A into another set of distinct levels this time based on the dependences in the system $Ux = b$. Using our matrix in Equation 13, we can find our set of levels from L with the following logic:

- Row 1 has no dependencies so it gets *level 1*
- Row 2 has no dependencies so it gets *level 1*
- Row 3 has a dependency on row 1 so it gets *level 2*
- Row 4 has a dependency on row 3 so it gets *level 3*

With the matrix A broken into its chunks for both L and U we can parallelize `ComputeSYMGS` running each chunk in parallel and waiting for the chunk to finish

before running on the next chunk. To maintain symmetry we first do this with the chunks from L and then with the chunks from U. In the end we get a version of `ComputeSYMGS` that has some parallelism depending on the denseness of A and with HPCG, the larger problem size, the sparser matrix.

The color solve algorithm works by creating an adjacency matrix out of A where any non zero entry is denoted as an edge between its row and column. With the adjacency matrix we use a greedy color algorithm to color the matrix as quickly as possible and ensuring that the coloring is correct. While the coloring is always valid it is not deterministic and there is no guarantee that the same rows are always in the same color. The following is what the matrix in Equation 13 could look like colored:



Similar to the level solve algorithm we can parallelize `ComputeSYMGS` by running each color in parallel and waiting for that color to finish before starting the next. To maintain symmetry we run the colors in ascending order and then again in descending order. While coloring greatly increases symmetry in sparse matrices it is again lacking in dense matrices. It's also important to note that unlike level solve, since we are not guaranteeing that each row's dependencies have completed before computing on the row, the output of this kernel is nondeterministic and generally causes an increase in the overall number of iterations required by CG to converge. This is a trade-off of higher parallelism

than levels but more CG iterations that HPCG will not count towards the final result.

6 Results

This thesis attempts to compare how Kokkos + HPCG with no optimizations performs in comparison to reference HPCG and could be used as a basis for a new reference implementation, and how the various preconditioning parallelizations perform in the new Kokkos + HPCG implementation across both OpenMP and Cuda while attempting to analyze what causes the fluctuation in performance. All the data gathered for the analysis comes from running Kokkos + HPCG and reference HPCG on the test bed Shannon with access provided by Sandia National Laboratories. The specifications for Shannon can be found in Table 1. The tests were all completed using the Stella nodes of Shannon which contain an Intel Xeon processor and a Tesla K80 Co-Processor.

6.1 KHPCG vs. HPCG

The initial goal was to create a reference version of HPCG that used Kokkos data structures and Kokkos parallel dispatch. Performance wise we hoped to see a version of HPCG that performed on par with reference HPCG with a slight slow down due to the necessity of creating the new data structures. Each variation of the code was run three separate times and the harmonic mean of the results is plotted in Figures 3, 4, 5, 6, and 7. However to much surprise, there is an overall increase in performance and performance remains relatively similar across various problem sizes whereas HPCG showed a reduction in GFLOP/s (Giga Floating Point Operations per Second) as problem size increased.

Name	Shannon
Nodes	32
CPU	2 Intel E5-2670 HT-off
Co-Processor	K20x ECC on
Memory	128 GB
Interconnect	QDR IB
OS	RedHat 6.2
Compiler	GCC 4.8.2+CUDA 6.5
MPI	MVAPICH2 1.9
Other	Driver: 319.23

Table 1: Configurations of Shannon testbed

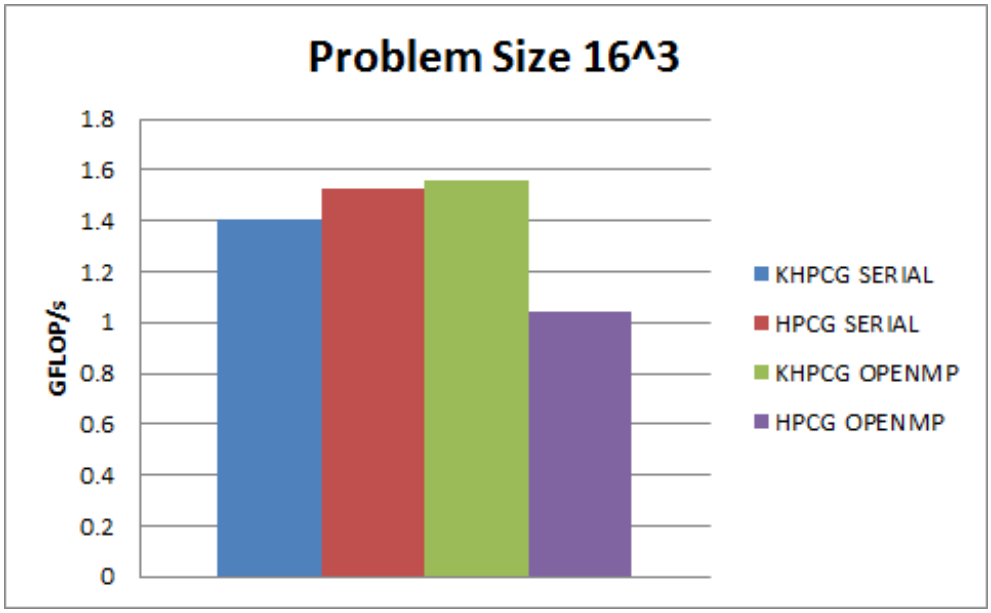


Figure 3: GFLOP/s for all reference variations of HPCG and KHPCG on problem size 16^3.

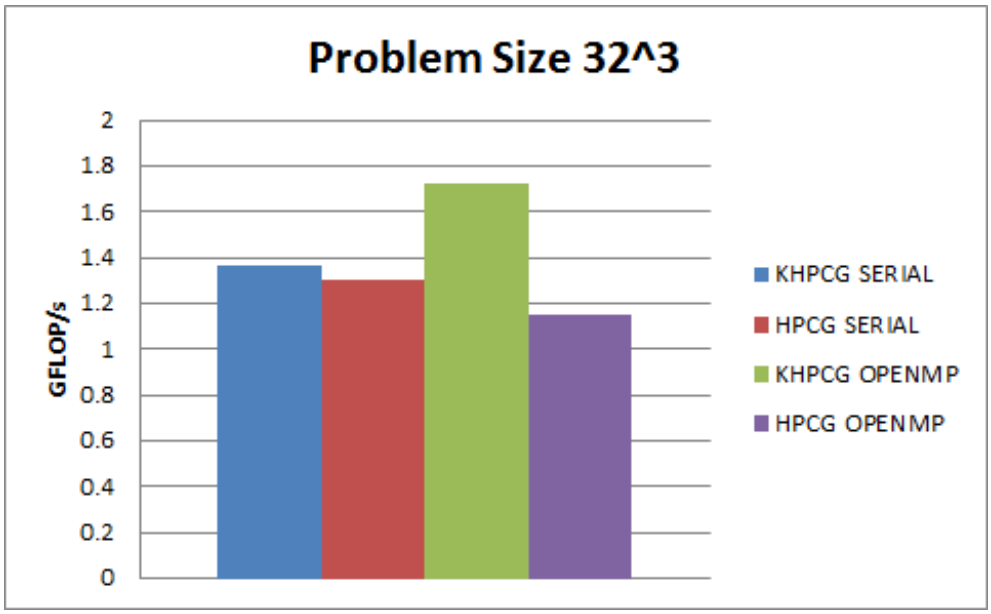


Figure 4: GFLOP/s for all reference variations of HPCG and KHPCG on problem size 32^3.

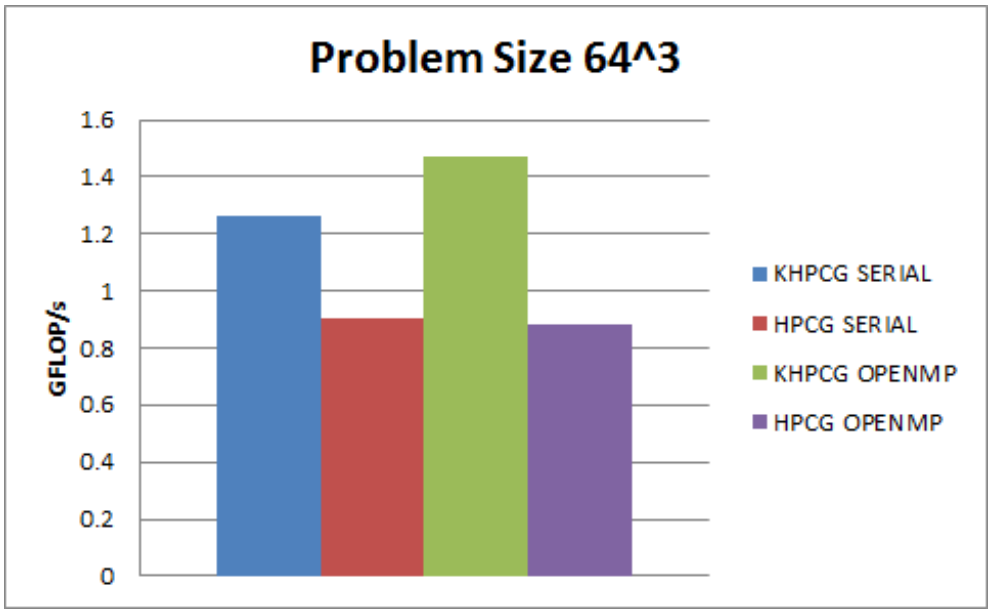


Figure 5: GFLOP/s for all reference variations of HPCG and KHPCG on problem size 64^3.

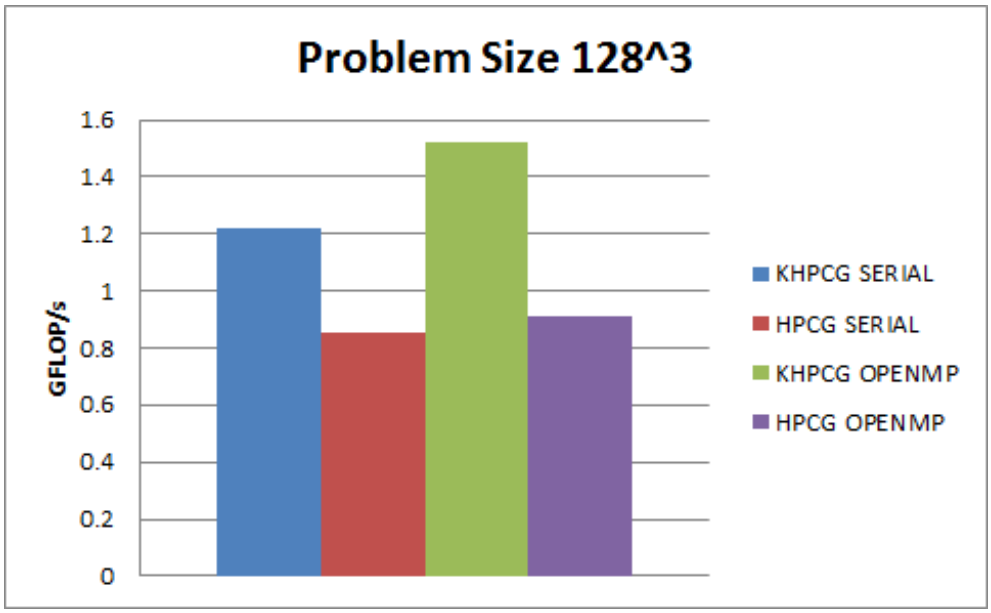


Figure 6: GFLOP/s for all reference variations of HPCG and KHPCG on problem size 128^3.

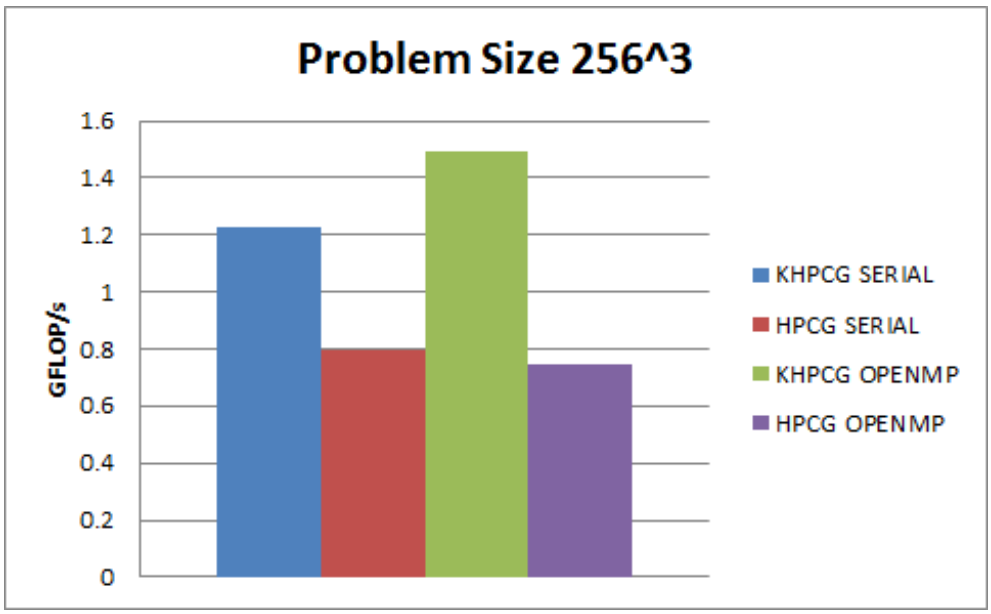


Figure 7: GFLOP/s for all reference variations of HPCG and KHPCG on problem size 256^3.

At first the overall increase in performance along with the stable production through various problem sizes was tough to explain. However going back to the previous change made during HPCG 2.4, the reason for the change in results across implementations arose when we replaced HPCGs implementation of a sparse matrix with the Kokkos implementation. Once things were moved to HPCG 3.0 the Kokkos "wrapper" around the HPCG sparse matrix was deprecated due to the illegal way it was set up. Since the most expensive kernels are the ones that need to access all the information in a sparse matrix the change from a two dimensional implementation to a one dimensional implementation cuts the amount of time spent accessing the memory. Instead of each data access requiring a memory access that points to another array in memory the one dimensional implementation gathers which section of the array to look in and then directly accesses the data in memory, which may already be loaded in cache. It would be interesting to recreate the Kokkos "wrapper" of the HPCG sparse matrix implementation and the various Kokkos Kernels and compare the results between the three implementations.

6.2 Comparing Preconditioning Algorithms

Once the reference version of the code was complete, attempts to optimize the code began. The main focus of the optimizations were in the symmetric Gauss-Seidel preconditioning kernel of HPCG. The optimizations were described in detail earlier and now we discuss the results of each optimization. Both optimizations use the same solve kernels and the way the matrix is broken into chunks is the only difference between the two. All runs were tested with either OpenMP or Cuda and either hierarchical parallelism or single level parallelism. Both implementations of the algorithm showed a similar distribution of performance among the combinations of tests and problem sizes. Each test was run

three times on each problem size and the harmonic means of the results are plotted in figures 8, 9, 10, 11, and 12.

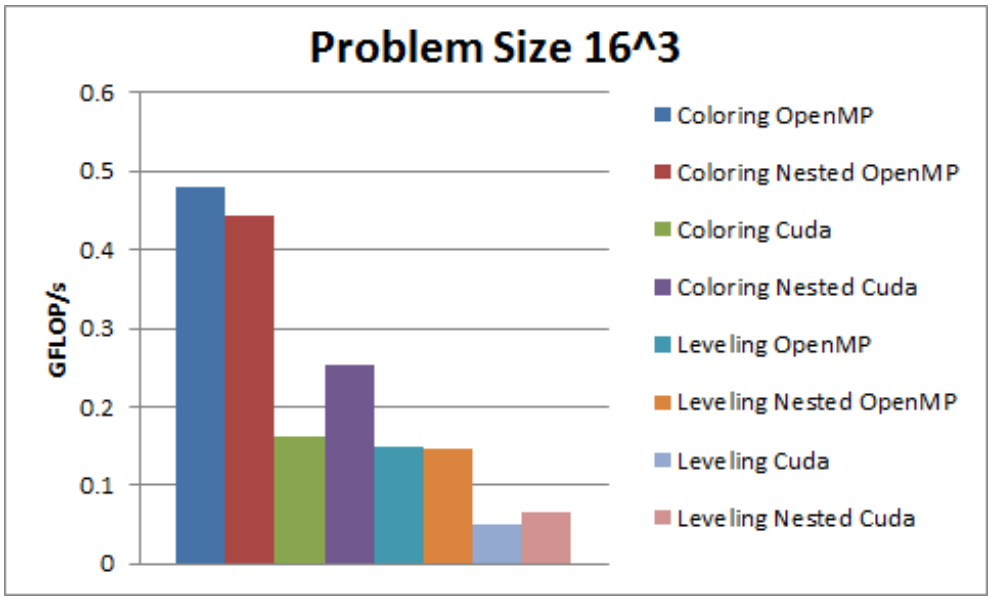


Figure 8: GFLOP/s for all combinations of execution spaces and optimizations of SYMGS in KHPCG on problem size 16^3.

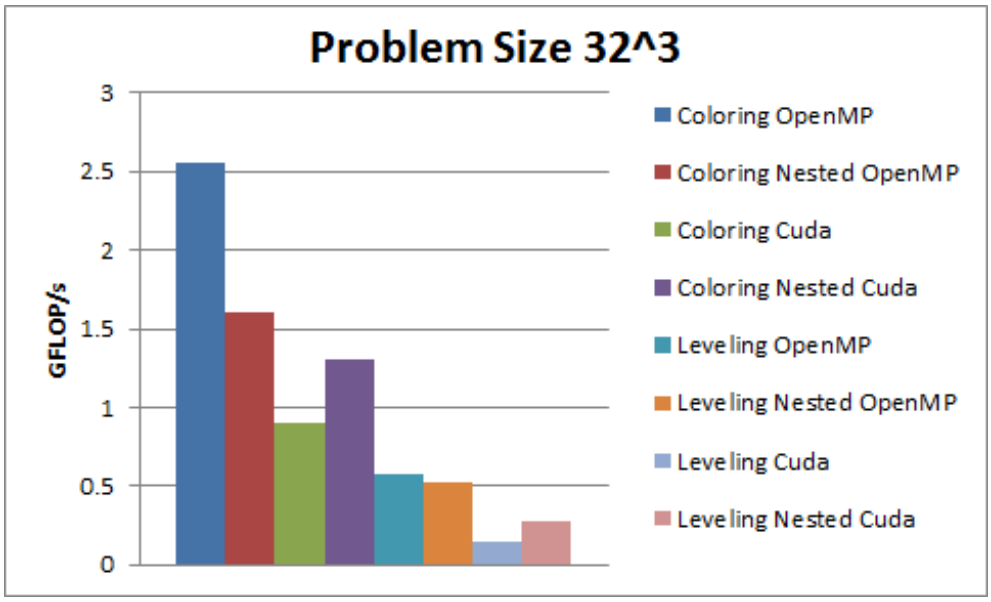


Figure 9: GFLOP/s for all combinations of execution spaces and optimizations of SYMGS in KHPCG on problem size 32^3.

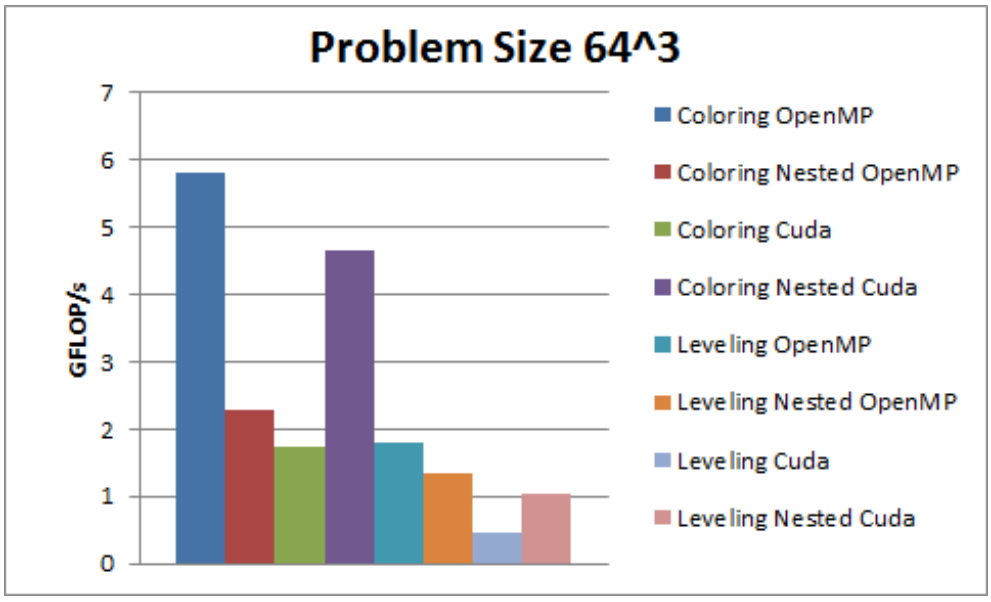


Figure 10: GFLOP/s for all combinations of execution spaces and optimizations of SYMGS in KHPCG on problem size 64^3.

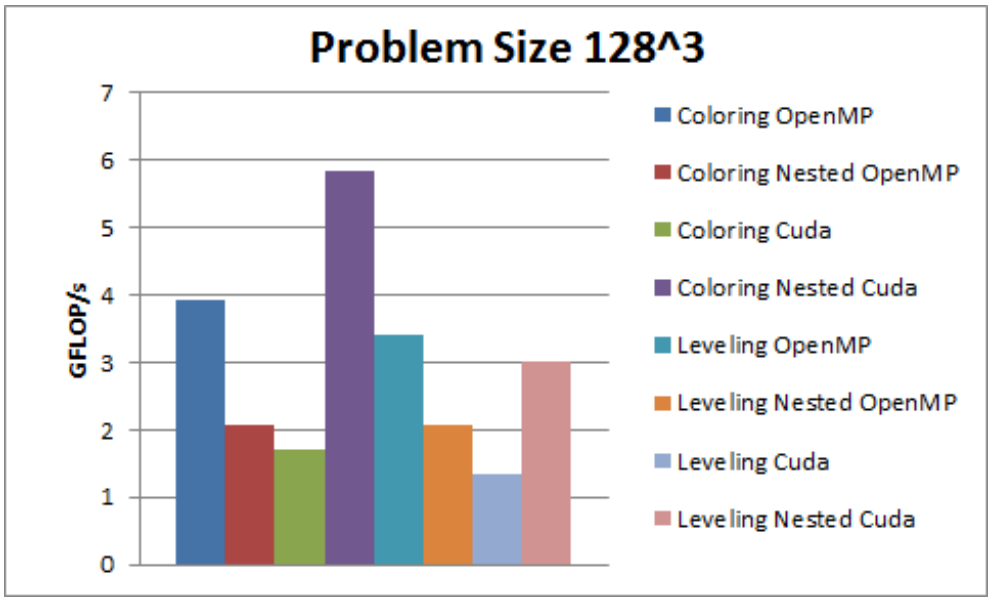


Figure 11: GFLOP/s for all combinations of execution spaces and optimizations of SYMGS in KHPCG on problem size 128^3.

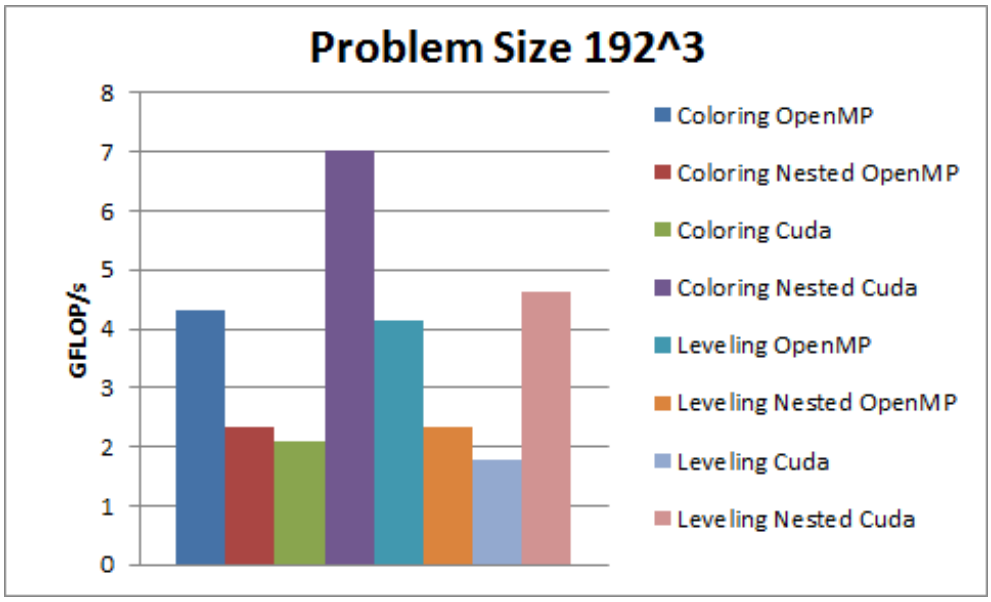


Figure 12: GFLOP/s for all combinations of execution spaces and optimizations of SYMGS in KHPCG on problem size 192^3.

Both optimizations show a similar trend in performance across the various `execution spaces`. But the performance difference is greatly in favor of the coloring algorithm likely due to the fact that level solve is more restrictive in how it forms the row chunks used for parallelism which hinders the amount of parallelism that can be squeezed out of it. Another interesting trend is that hierarchal OpenMP is out performed by single level OpenMP and the opposite is what happens when using Cuda. The exact cause of this is unknown and more testing and tweaking of variables could be done in tangent with using different test beds to explore why. In the final version of KHPCG there is an option to allow for the reordering of the matrix based on the coloring to help reduce the number of cache misses. The option is not included here since a similar optimization was not done for the leveling algorithm as it would require storing two copies of the matrix and we were already experiencing issues with running

out of memory on GPUs. In the future it would be very simple to implement.

7 Conclusion

This work confirmed that while the steepest descent algorithm achieves what it sets out to do, it is often too inefficient to be used in practice. Then the idea of conjugate directions was introduced and although it can address the rate of convergence issue, it runs into its own set of problems. Next the method of conjugate gradients addresses the issues of conjugate directions by computing each search direction at the time its needed, removing the need to store each search directions. Not only did this method act as a better direct method, it converges to a solution much faster than steepest descent so it is used more as an iterative method in practice.

HPCG uses the conjugate gradients method to solve a system of equations and is the new upcoming benchmark to be used in tangent with HPL to measure the performance of the world's largest and fastest supercomputers. Since HPCG is used across a multitude of systems, it is often user optimized to accommodate the architecture it is being run on. Through the use of the Kokkos package developed by Sandia National Laboratories this thesis worked to create a reference version of HPCG that could easily be modified to accommodate a variety of architectures and perform optimally across each. To test performance portability, we tested HPCG and KHPCG across multiple Kokkos `execution spaces` and noted similar performance between the two. Finally we made our own optimizations to KHPCG and noted that with the given solve kernel certain `execution spaces` perform better than others with both hierarchal parallelism and single level parallelism but overall the coloring implementation outperformed the leveling implementation.

7.1 Future Work

The goal of this project was to create a reference version of HPCG using Kokkos but there is still plenty of work to be done. For starters, it may be more in the design of HPCG to use a Kokkos "wrapper" sparse matrix structure than the one found in the Kokkos package allowing the end user greater freedom with their optimizations. On top of this, during the work with HPCG 2.4 we experienced issues with using MPI in tangent with the Kokkos data structures so then it was decided to work on a single node implementation of Kokkos + HPCG. Now that the single node implementation is complete, work needs to be done to re-enable the ability to use MPI with the code.

Since there is now a single node implementation it would be fascinating to compare the optimized results of this code with those of the code provided by large vendors like Intel and Nvidia. This would be a good comparison test to see how the Kokkos programming model stands up to the current models used across each compatible architecture. The code provided here has seen some optimizations but I acknowledge that it may not be the most optimal implementation, and other implementations could be explored to further improve optimized results. On top of these there needs to be better memory management as the current setup did not allow for problem sizes of 256^3 to be run on the GPUs we tested on, as they would run out of memory to store the data.

8 Appendix

8.1 Github Repositories

The repository for HPCG can be found from their webpage `hpcg-benchmark.org`. The Trilinos repository where the Kokkos package is located can be found at `https://github.com/trilinos/Trilinos.git`. There is too much code in

the KHPCG project to be printed in this paper, however, the git repository for this project can be found at `https://github.com/zabookey/KHPCG3.0.git` along with instructions on how to build the project.

8.2 Matlab Code

The Matlab code for each of the iterative methods introduced can be found below.

Steepest Descent Code:

```
function x = SteepDescent(A, x0, b)
x = x0;
r = b - A*x0;
n = sqrt(r'*r);
k = 1;
while n > .000001
    a = (r'*r)/(r'*A*r);
    x = x + a*r;
    r  = b - A*x;
    n = sqrt(r'*r);
    k = k+1;
end
fprintf('Achieved Convergence in %d steps\n', k-1);
end
```

Conjguate Directions Code:

```
function x = ConjDirections(A, x0, b, P)
x = x0;
r = b - A*x0;
n = sqrt(r'*r);
k = 1;
while n > .000001 && k <= length(b)
    p = P(:,k);
    a = (p'*r)/(p'*A*p);
    x = x + a*p;
    r  = b - A*x;
    n = sqrt(r'*r);
    k = k+1;
end
```

```
fprintf('Achieved Convergence in %d steps\n', k-1);
end
```

Conjugate Gradients Code:

```
function x = ConjGradients(A, x0, b, prec)
if nargin < 4
    prec = false;
end
x = x0;
p = x0;
r = b - A*x0;
n = sqrt(r'*r);
k = 1;
while n > .000001
    if prec
        z = zeros(length(r),1);
        z = SYMGS(A, r, z);
    else
        z = r;
    end
    if k == 1
        p = z;
        rz = r'*z;
    else
        rz0 = rz;
        rz = r'*z;
        beta = rz/rz0;
        p = beta*p + z;
    end
    Ap = A*p;
    alpha = rz/(p'*Ap);
    x = x + alpha*p;
    r = r - alpha*Ap;
    n = sqrt(r'*r);
    k = k+1;
end
fprintf('Achieved Convergence in %d steps\n', k-1);
end

function x = SYMGS(A, r, x0)
    x = x0;
    Adim = size(A);
    nrow = Adim(1);
    ncol = Adim(2);
```

```
41      %Forward sweep
42      for i = 1:nrow
43          sum = r(i);
44          for j = 1:ncol
45              sum = sum - A(i,j) * x(j);
46          end
47          sum  = sum + x(i)*A(i,i);
48          x(i) = sum/A(i,i);
49      end
50      %Backward sweep
51      for i = nrow:-1:1
52          sum = r(i);
53          for j = 1:ncol
54              sum = sum - A(i,j) * x(j);
55          end
56          sum = sum + x(i)*A(i,i);
57          x(i) = sum/A(i,i);
58      end
59
60  end
```

References

[1] B. Barney, *Posix threads programming.* `https://computing.llnl.gov/tutorials/pthreads/`.

[2] J. Dongarra and et al., *Top 500 supercomputer sites.* `http://www.top500.org`.

[3] C. H. Edwards, C. Trott, and D. Sunderland, *Kokkos tutorial a trilinos package for manycore performance portability*, 2013.

[4] H. C. Edwards, C. R. Trott, and D. Sunderland, *Kokkos: Enabling manycore performance portability through polymorphic memory access patterns*, Journal of Parallel and Distributed Computing, (2014).

[5] J. D. Faires and R. L. Burden, *NumericalMethods*, Richard Stratton, fourth ed., 2013.

[6] G. H. Golub and C. F. V. Loan, *Matrix Computations*, The John Hopkins University Press, fourth ed., 2013.

[7] A. Greenbaum and T. P. Charter, *Numerical Methods*, Princeton University Press, 2012.

[8] M. A. Heroux, *Solving sparse linear systems.* `http://www.users.csbsju.edu/~mheroux/SC2015HerouxTutorial.pdf`.

[9] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, *An overview of the trilinos project*, ACM Trans. Math. Softw., 31 (2005), pp. 397–423.

[10] M. A. Heroux and J. Dongarra, *Toward a new metric for ranking high performance computing systems*, tech. report, Sandia National Laboratories, 2013.

[11] M. A. Heroux, J. Dongarra, and P. Luszczek, *Hpcg technical specification*, Tech. Report SAND2013-8752, Sandia National Laboratories, 2013.

[12] C. B. Moler, *Numerical Computing with MATLAB*, Society for Industrial and Applied Mathematics, 2004.

[13] NVIDIA, *Cuda programming guide version 3.0*, tech. report, Nvidia Corporation, 2010.

[14] OpenMP Architecture Review Board, *OpenMP Application Program Interface*, 2013.

[15] C. Porzikidis, *Numerical Computation in Science and Engineering*, Oxford University Press, first ed., 1998.

[16] J. R. Shewchuk, *An introduction to the conjugate gradient method without the agonizing pain*, 1994.

[17] C. R. Trott, M. Hoemmen, S. D. Hammond, and H. C. Edwards, *Kokkos programming guide*, 2015.

[18] Wikipedia, *Conjugate gradient method — wikipedia, the free encyclopedia*. `https://en.wikipedia.org/w/index.php?title=Conjugate_gradient_method&oldid=676979261`, 2015. [Online; accessed 1-October-2015].

[19] ——, *Iterative method — wikipedia, the free encyclopedia*. `https://en.wikipedia.org/w/index.php?title=Iterative_method&oldid=695076796`, 2015. [Online; accessed 28-March-2016].