Optimizing Data Locality for Iterative Matrix Solvers on CUDA

Raymond Flagg, Jason Monk, Yifeng Zhu PhD., Bruce Segee PhD.
Department of Electrical and Computer Engineering, University of Maine, Orono, ME, USA

Abstract— *Solving systems of linear equations is an important problem that spans almost all fields of science and mathematics. When these systems grow in size, iterative methods are used to solve these problems. This paper looks at optimizing these methods for CUDA Architectures. It discusses a multi-threaded CPU implementation, a GPU implementation, and a data optimized GPU implementation. The optimized version uses an extra kernel to rearrange the problem data so that there are a minimal number of memory access and minimum thread divergence. The normal GPU implementation achieved a total speedup of 1.60X over the CPU version whereas the optimized version was able to achieve a total speedup of 1.78X. This paper demonstrates the importance of pre-organizing the data in iterative methods and its impact.*

Keywords: Block Jacobi, CUDA, Multi-GPU, Iterative Methods, GPGPU

1. Introduction

Systems of linear equations are ubiquitous in computer engineering, as well as throughout all of science. They are used to solve problems from linear circuits all the way to discretization of more complex problems such as Finite Element Methods. The fact that they are deeply imbedded in so much of the work done in these areas means that there has always been a scientific advantage in being able to solve these systems faster.

Generally these systems are solved through algorithms that are referred to as direct approaches. They are called direct approaches because they involve a series of steps that can be followed and, when completed, the matrix has been solved. These direct approaches generally have two drawbacks; they are inherently sequential and are often extremely slow for large numbers of variables. This is where indirect or iterative solvers take the stage. Iterative methods involve performing a step or series of steps over and over again. There is no direct way of knowing how many steps are required; instead they converge towards a solution. There are a variety of ways of telling if the solution is close to convergence, and therefore close to correct.

This paper focuses on the implementation of the Jacobi Algorithm, which is used mostly for simplicity. This modification to the algorithm should be easily adaptable to use a variety of iterative methods. This algorithm has been implemented for a single and multiple CPUs using C++ and the pthreads library. This implementation will serve as a baseline for the tests performed on a single GPU. The addition of GPU support will be a CUDA-based solver.

2. Related Work

The natural parallel nature of iterative linear solvers are generally very attractive to the GPGPU environment [1]. Iterative linear solvers have been implemented on GPUs for a variety of problems in several areas of science. Both CUDA and OpenGL implementations were tested by Amorim et al in 2009 [2]. CUDA based systems were successfully tested by Wang et al 2009 [3], Zhang et al 2009 [4], and Amador and Gomes 2009 [5] to mention a few examples. In most cases there was a noticeable benefit for iterative solvers running on the GPU.

Amorim et al 2009 performed a comparison of OpenGL and CUDA implementations with a single threaded SSE-enabled CPU baseline. The OpenGL code was implemented by writing a shader that would read the current iteration from one texture and write the next iteration into another. The CUDA implementation showed to be noticeably better than OpenGL, with a maximum speedup of 31x vs only 17x achieved through OpenGL. This was a good comparison of the two approaches however it was only tested on fairly small problems ($n < 10$). [2]

Wang et al 2009 performed some testing up to much larger sizes ($n < 4000$). This implementation was, however, a much less efficient implementation that was only able to reach speedups between 1.5x and 3x. It did show that to utilize the full potential of the GTX280 being tested (240 Processors) that the n must be greater than or equal to 512. The performance showed the most computational power at $n = 512$, decreasing slightly for $n > 512$. [3]

Zhang et al 2009 created an implementation that performed well and was tested on matrix sizes up to 10,000. This implementation had support for both single and double precision, which both performed significantly better than the CPU. The single precision had a peak speedup of 59x, while the double had a peak speedup of 19x. [4]

Amador and Gomes 2009 did a comparison of three different iterative solvers. They were the Jacobi Method, Gauss-Seidel (A Derivative of Jacobi), and Conjugate Gradient. Their results showed significantly greater speedup of Jacobi and Gauss-Seidel implementations. This suggested that Jacobi-based methods are significantly more parallelizable, and a better candidate for multi-GPU applications. [5]

Iterative methods have been implemented on a GPU cluster at least once before by Ali Cevahir 2010. [6] It was an

implementation of a conjugate gradient method. It had 15x more processing power by using 2 GPUs/node rather than 2 CPUs/node. This method required a significant amount of preconditioning to the matrix.

3. Iterative Solvers

There are many types of iterative methods that are used commonly in solving large linear systems. Many of them are arguably better at producing solutions than the Jacobi Method. What the Jacobi Method lacks in convergence rate it makes up for in how parallelizable it is as well as the simplicity in implementing it.

3.1 Jacobi

For a brief explanation of the Jacobi Method let us examine the matrix equation below. Where A is a matrix of size n by n, b is a column vector of size n, and x is a column vector of unknowns of size n.

$$A * x = b \tag{1}$$

The simplest explanation of the Jacobi Method can be described by splitting the system into a series of rows. Each row can represent a single linear equation. The following would be a single equation representing row i, in a system of size n.

$$A_{i,0} * x_0 + A_{i,1} * x_1 + ... + A_{i,n} * x_n = b_i \tag{2}$$

The basis of the Jacobi Method is to solve the i^{th} equation for the i^{th} unknown for all i.

$$x_i = \frac{b_i - \sum_{j!=i} A_{i,j} * x_j}{A_{i,i}} \tag{3}$$

By inserting the current values of x into the right hand side of the above equation, the Jacobi method produces the next iteration for x. The equation below shows how x moves from one iteration to the next.

$$x_i^{k+1} = \frac{b_i - \sum_{j!=i} A_{i,j} * x_j^k}{A_{i,i}} \tag{4}$$

Now the benefit of the Jacobi Method is clear because each of the unknowns can be solved for completely independently within each iteration, making this a very parallelizable algorithm.

3.2 Block-based Jacobi

The Block Jacobi Algorithm takes this already parallel method and lends itself even more so to the CUDA architecture. The modification of the algorithm is a small one. It divides the matrix into blocks of unknowns and each block is iterated separately; the results are only communicated after several iterations have passed.

If the block algorithm was being applied with two groupings then each of the blocks B1 and B2 could be [x_0, $x_{n/2}$] and [$x_{n/2+1}$,x_n]. Both B1 and B2 are iterated separately without updating any values outside their respective blocks; this is referred to as an inner iteration. Several inner iterations are completed before values outside a block are updated. When all of the values are updated this is referred to as an outer iteration.

The division of the unknowns cuts down on the communication bandwidth that is needed. It also allows for some computational optimization that is discussed in the Reorganization section.

4. Parallel Block Jacobi

This section describes how the version of the Block Jacobi Algorithm was implemented. First it describes the CPU version and how the block algorithm applies to multithreading. Then this section discusses the CUDA structuring and applications related to the architecture.

4.1 CPU Structure

To focus on the calculation of the solution the process is divided into three different stages. The program reads from input, then calculates the solution in parallel, and finally writes the output. Figure 1 shows this flow for three processing threads.

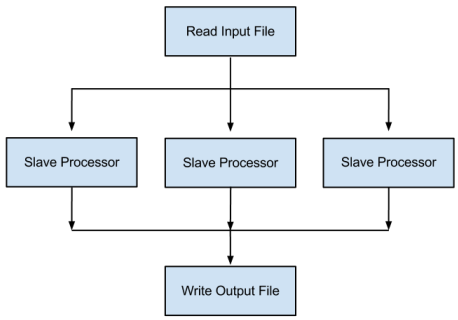

Fig. 1: Program Flow for 3 Threads

While in the processing stage there are a number of processing threads that handle inner iterations and a master thread that tracks convergence of the group. When an outer iteration occurs, each of the slave processors transmits updates to each of the other slave processors and sends information about its convergence state to the master thread. Figure 2 shows the flow between threads during an outer iteration.

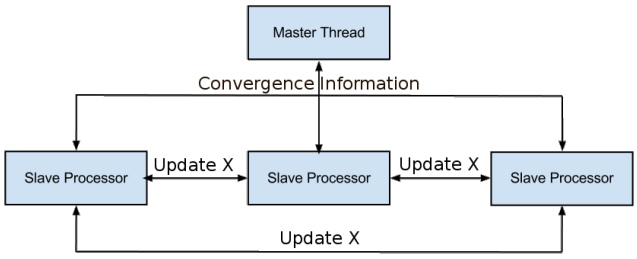

Fig. 2: Outer Iteration Communication

Given that on the CPU each of the threads controls a block, the size of the blocks being used in the algorithm can be controlled by the number of threads being used.

4.2 GPU Structure

The Block GPU Algorithm lends itself extremely well to the CUDA kernel structures. A CUDA Block can share memory and fits well to hold a single block of the Block Jacobi algorithm. This allows each CUDA thread to update the values of a single variable, unlike the CPU where each thread controls an entire Jacobi Block.

Each group of inner iterations is contained within a single kernel, and between each kernel an outer iteration is performed by doing a Device-to-Device memory copy.

4.3 Multi-GPU Structure

This structure lends itself well when porting to a multiple GPU system using each processing thread to control a GPU. Figure 3 below shows the modified data flow for a multi-GPU system.

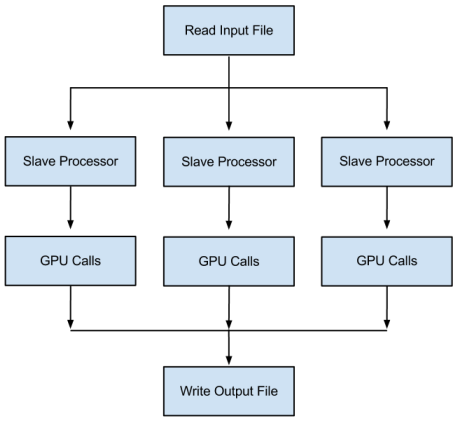

Fig. 3: Program Flow for 3 GPUs

The kernel requires little modification because inner iterations are isolated to occur on a single GPU. Outer iterations require that the updated X values are shared between each of the GPUs using a similar structure to the CPU Implementation.

4.4 Sparse Storage

A **sparse matrix** is a matrix that is composed mostly of zeros. These kinds of matrices are very common in some types of modeling (e.g., FEM). Figure 4 shows an example of a matrix generated by a 2-D FEM problem, where black pixels represent non-zero elements.

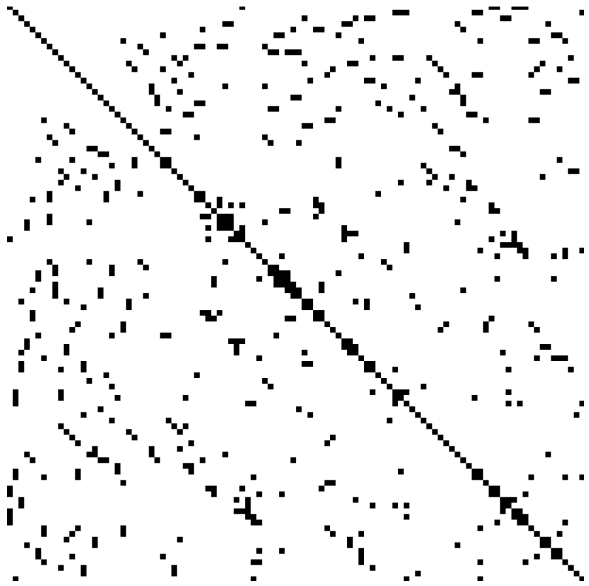

Fig. 4: Non-zero Elements in Sparse Matrix

When a matrix is composed mostly of zeros it becomes more efficient to only store the non-zero elements rather than the entire array in memory. The linear equations solver implements a simple form of sparse storage. Each row of the matrix is stored as a list of indices and a list of coefficients. A pair consisting of an index and a coefficient represent one value in the row and the index into the row where the value should reside.

4.5 Reorganization

To handle large systems the data passed to the GPU is sparsely stored to save space. Each CUDA thread loops across the summation shown in Equation 4. This is not a good structure for CUDA because for each time through the summation there are two categories it can fall into. A variable can be within the Jacobi block so the value must be pulled from an updated list in shared memory, or it can be outside of the Jacobi block meaning a static value should be pulled from global memory. Since CUDA has 32-thread groups operating in an SIMD nature, having divergent threads in each loop is bad for performance.

To make the code more optimized for CUDA, a second kernel was added. This second kernel was run once at the beginning of operation and would reorganize the data to remove conditional situation within the main loop. To do this the kernel takes the lists describing the coefficients in a specific equation and divides them into the groups for static coefficients (coefficients multiplied by values outside the Jacobi block) and dynamic coefficients (coefficients multiplied by values within the jacobi block). Figure 5 shows an example of this division, where static coefficients are indicated by an S. Global indices are represented by I and local indices (within a Jacobi block) are represented by L.

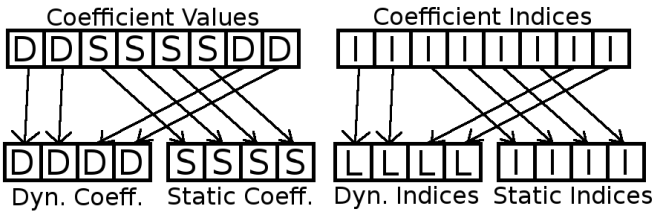

Fig. 5: Reorganization of Sparse Matrix Row

Figure 5 also shows that when storing the indices for the dynamic list the problem converts the indices from global indices to local indices. These lists are created for each of the threads. When stored in memory the array is transposed to ensure that each thread read happens within a single cache line to minimize memory accesses.

5. Performance Analysis

The code was tested on both a small and large linear system for analysis. The inputs were generated using Finite Element Models. The small test was generated with a simple 1-D heat flow finite element model with 100 nodes (unknowns). The larger test case is one generated using the University of Maine Ice Sheet Model, a 2-D problem with 168,861 nodes (unknowns).

5.1 Hardware Configuration

Having insufficient supporting hardware can be detrimental to performance of the GPU. The configuration used was built for GPU computing at the University of Maine. Table 1 shows an overview of the hardware for the machine.

Table 1: GPGPU Machine Configuration

CPU:	Intel Core i7 990X
GPU:	2 x NVIDIA GTX 580 1 x NVIDIA GTX 680
Memory:	24 GB DDR 1600
Motherboard:	ASUS Rampage III Extreme
Storage:	50 GB SSD 500 GB HDD

5.2 CPU Tests

The CPU implementation was tested on each of the inputs described above. The number of processing threads was varied from one to eleven. Figure 6 shows the runtime averaged over three runs for each of the numbers of threads for the smaller test data.

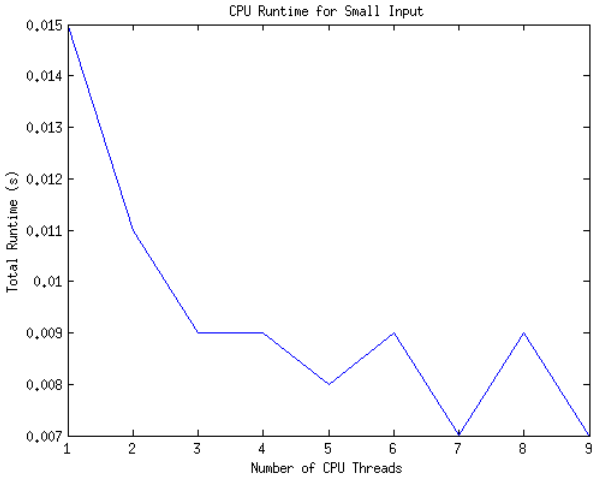

Fig. 6: CPU Runtime for Small Test Data

The results clearly show the total runtime starting to level off around six threads, the number of physical cores the processor being used for testing. Figure 7 below is the results for the same tests with the larger input set.

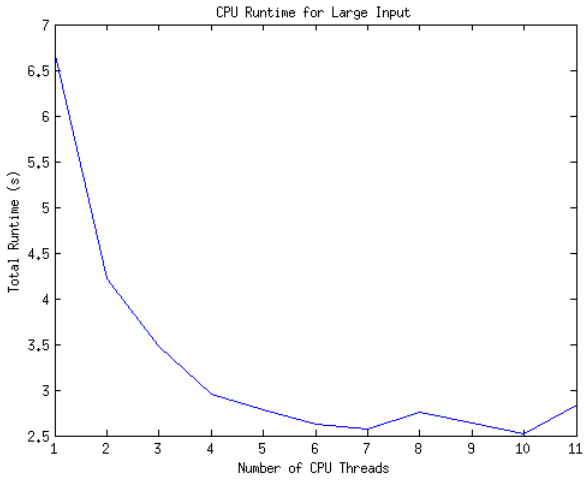

Fig. 7: CPU Runtime for Large Test Data

5.3 GPU Tests

The same data was tested on 1-3 GPUs. All single GPU tests use the NVIDIA GTX 680. All multi-GPU tests use one NVIDIA GTX 680 and all additional GPUs are NVIDIA

GTX 580s. Figure 8 shows the runtimes for these tests. It shows that the runtime increased as the GPUs were added. This was a fairly small test set (only 100 unknowns), so the lack of performance is expected.

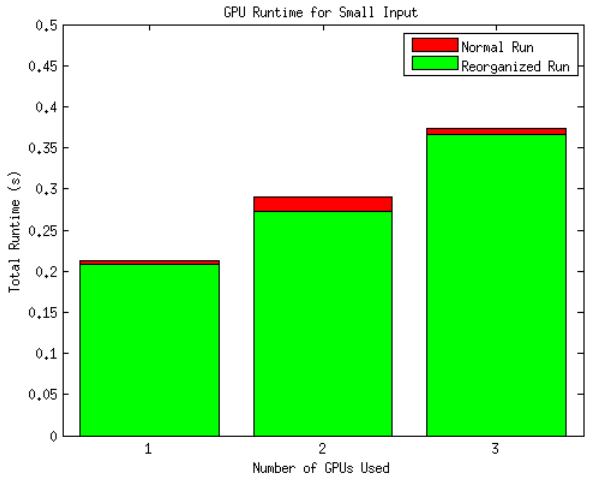

Fig. 8: GPU Runtime for Small Test Data

Figure 9 shows the GPU performance on the larger dataset. With the larger dataset there was a very slight decrease in runtime from one to two GPUs being used (not existent in the reorganized version).

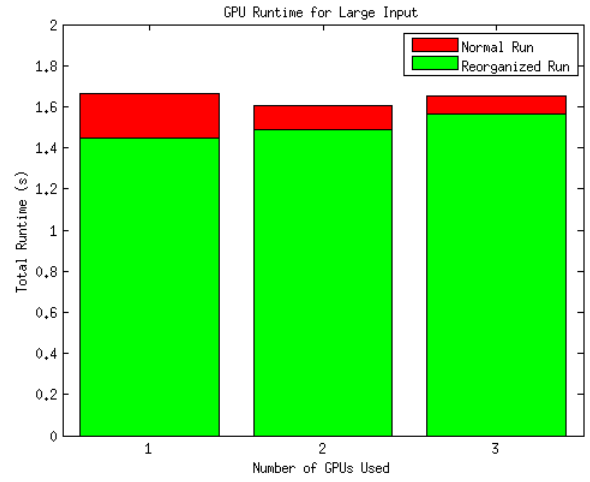

Fig. 9: GPU Runtime for Large Test Data

Table 2 shows the total speedup for the large dataset with one, two, and three GPUs for both normal and reorganized runs. For this dataset, two GPUs offered the best speedup, 1.60X.

Table 2: Table of Large Dataset Total Speedup

Number of GPUs	Normal Speedup	Reorganized Speedup
1	1.55X	1.78X
2	1.60X	1.73X
3	1.56X	1.65X

Given the lack of significant increase in performance from the addition of GPUs, the processing time (excluding setup and shutdown of program) was examined. Figures 10 and 11 show the processing time for each of the datasets.

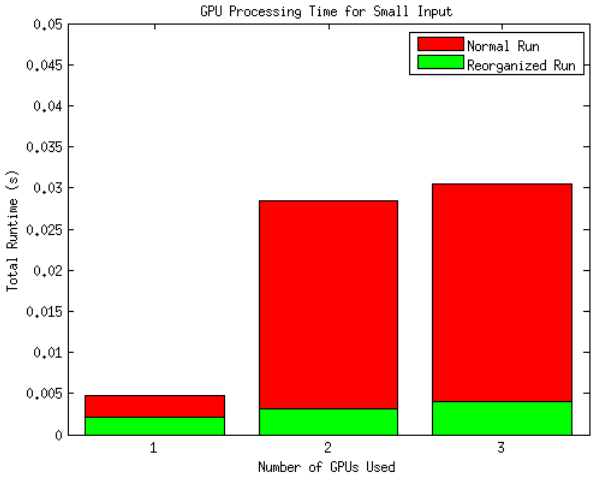

Fig. 10: GPU Processing Time for Small Test Data

The small dataset shows a noticable increase in processing time taken from adding GPUs. This indicates that the communication overhead of the GPUs is not worth the extra computational power given.

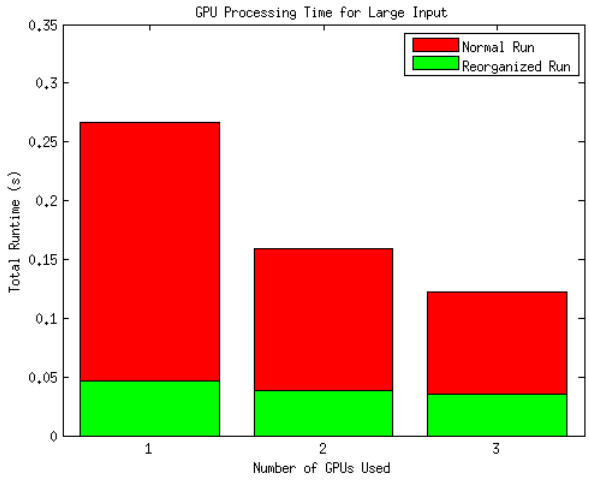

Fig. 11: GPU Processing Time for Large Test Data

Table 3 shows the processing speedup for the large dataset with one, two, and three GPUs for both normal and reorganized runs. For this dataset, one GPU offered the best speedup for the reorganized data, 6.72X, while two GPUs offered the best speedup for normal data, 3.86X.

Table 3: Table of Large Dataset Processing Speedup

Number of GPUs	Normal Speedup	Reorganized Speedup
1	3.30X	6.72X
2	3.86X	5.77X
3	3.82X	4.78X

The larger dataset shows much better results for adding GPUs. The processing time decreased for GPUs being added for both the normal and reorganized versions of the code. Analysis of the rest of the code showed that there was an increase in the setup time for the GPUs as more were added. This setup time was used to transfer the problem data to all of the GPUs. Figures 12 and 13 show the setup times for each of the tests.

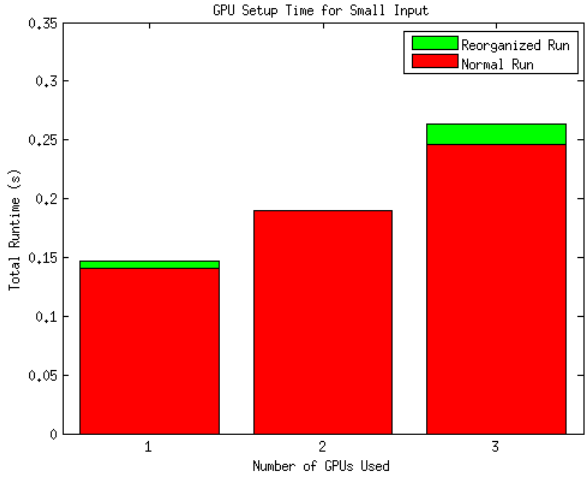

Fig. 12: GPU Setup Time for Small Test Data

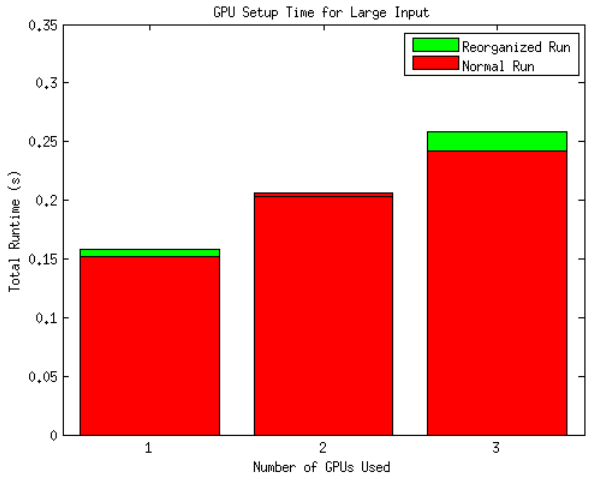

Fig. 13: GPU Setup Time for Large Test Data

The increase in setup time is nearly linear for both cases. This is easily explained by the limited bandwidth of the PCI-E bus. As the number of GPUs increases the amount of data that must be transferred increases almost linearly as each of the GPUs requires the full initial state of most of the values.

6. Conclusion

The Linear Equations Solver described in this paper had good results when ported to the GPU. Including the complete runtime, the GPU had a maximum speedup of 1.78X over the CPU. Looking at only the processing time, the GPU had a speedup of 6.72X over the CPU. The processing went 6.72X faster on a data set significantly larger than the related work described in Section 2. The reorganization of the data for CUDA had a significant impact on the runtime in all tests.

7. Acknowledgements

This work was partially funded by NFS grants EAR 1027809, OIA 0619430, DRL 0737583, CCF 0754951, EPS 0918018, EPS 0904155, and NIH grant R01 HL092926.

References

[1] J. Monk, "Using GPU Processing to Solve Large-Scale Scientific Problems," Master's thesis, University of Maine, May 2013.

[2] R. Amorim, G. Haase, M. Liebmann, and R. Weber dos Santos, "Comparing CUDA and OpenGL implementations for a Jacobi iteration," in *High Performance Computing Simulation, 2009. HPCS '09. International Conference on*, june 2009, pp. 22–32.

[3] T. Wang, Y. Yao, L. Han, D. Zhang, and Y. Zhang, "Implementation of Jacobi iterative method on graphics processor unit," in *Intelligent Computing and Intelligent Systems, 2009. ICIS 2009. IEEE International Conference on*, vol. 3, nov. 2009, pp. 324–327.

[4] Z. Zhang, Q. Miao, and Y. Wang, "CUDA-Based Jacobi's Iterative Method," in *Computer Science-Technology and Applications, 2009. IFCSTA '09. International Forum on*, vol. 1, dec. 2009, pp. 259–262.

[5] G. Amador and A. Gomes, "CUDA-Based Linear Solvers for Stable Fluids," in *Information Science and Applications (ICISA), 2010 International Conference on*, april 2010, pp. 1–8.

[6] A. Cevahir, "Scalable Implementation Techniques for Sparse Iterative Solvers on GPU Clusters," Computational Science & Engineering Laboratory.