To the Graduate Council:
I am submitting herewith a thesis written by Rosanne Lane West entitled "GPU Implementation of a Novel Approach to Cramer's Algorithm for Solving Large Scale Linear Systems." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Engineering.

Itamar Arel, Major Professor

We have read this thesis and
recommend its acceptance:

Gregory Peterson

Charles Cao

Acceptance for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

GPU Implementation of a Novel Approach to Cramer's Algorithm for Solving Large Scale Linear Systems

A Thesis
Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Rosanne Lane West
May 2010

Copyright © 2010 by Rosanne Lane West
All rights reserved.

Dedication

This thesis is dedicated to my parents. For your love, your support, and for gently pushing me towards my goals, I thank you.

Acknowledgments

There are many people who I would like to acknowledge at this time for their help and support.

First of all, I must thank my advisor, Dr. Itamar Arel, for spending the time and effort to both get me started and guide me down the path towards finishing my studies.

I would also like to thank my manager, Joe Gipson, for being very flexible and understanding of the time needed to complete my academic obligations.

I also express my thanks to all those in my academic career for influencing me and helping me achieve my goals. I thank all of my teachers and professors throughout my studies for their support.

Finally, I must thank my family for their love, support, and patience.

Abstract

Scientific computing often requires solving systems of linear equations. Most software packages for solving large-scale linear systems use Gaussian elimination methods such as LU-decomposition. An alternative method, recently introduced by K. Habgood and I. Arel, involves an application of Cramer's Rule and Chio's condensation to achieve a better performing system for solving linear systems on parallel computing platforms. This thesis describes an implementation of this algorithm on an nVidia graphics processor card using the CUDA language. Increased performance, relative to the serial implementation, is demonstrated, paving the way for future parallel realizations of the scheme.

Contents

1 Introduction **1**

1.1 Motivation and Problem Statement 1

1.2 Thesis Outline 2

2 Literature Review **3**

2.1 LU-Decomposition 3

2.2 Parallel Considerations 4

3 Habgood's algorithm **7**

3.1 Cramer's Rule 7

3.2 Chio's Condensation 7

3.3 Mirroring 9

3.4 Complete Algorithm 10

3.5 Serial Implementation 11

4 Parallel Implementation **12**

4.1 Hardware Used 12

4.2 CUDA Overview 12

4.3 Parallel Implementation 13

4.4 Implementation Results 17

4.5 Alternate Implementations 18

5 Conclusions **23**

5.1 Thesis Summary 23

5.2 Future Work 23

Bibliography **25**

Vita **27**

List of Figures

3.1 Diagram of the the way the original matrix is mirrored and reduced until the resulting matrices are small enough to apply Cramer's algorithm. 10

4.1 Diagram of the memory usage pattern of reducing a 10 x 10 matrix. The dark areas are those containing relevant copies of the matrix. 16

4.2 Diagram of the order in which the matrices created using Habgood's algorithm are reduced. 17

4.3 Chart comparing the performance of the serial and parallel implementations. 19

4.4 Diagram of the flow of data and control between the CPU and GPU. 21

Chapter 1

Introduction

1.1 Motivation and Problem Statement

Systems of linear equations arise in many scientific and engineering areas. The method commonly used by fast linear solvers is LU-decomposition, which is a form of Gaussian elimination[1]. LU-decomposition has a computational complexity of $O(N^3)$[2]. In order to implement LU-decomposition on a parallel platform substantial communication between nodes is required. Therefore, a mesh architecture is commonly used, which poses a challenge in scaling such systems.

An well-known alternative to LU-decomposition for solving systems of linear equations is Cramer's rule. K. Habgood and I. Arel[3] have recently proposed an algorithm that utilizes Chio's condensation to gradually reduce the size of the original matrix until Cramer's rule can be applied efficiently. This algorithm has been implemented on a single-core processor and compared it to LAPACK (Linear Algebra PACKage). K. Habgood found that the Cramer's rule based algorithm exhibited runtimes of approximately 2.5 times that of LAPACK. However, this algorithm does not require as much communication between nodes, so he argued that a parallel implementation of the algorithm could achieve substantial performance gains. This thesis describes an implementation of Habgood's algorithm on an nVidia graphics processor using the CUDA language. The CUDA program resulted in significantly better performance than the original serial version.

1.2 Thesis Outline

Chapter 2 gives a brief review of LU-decomposition techniques and their performance. In Chapter 3, Habgood's algorithm is explained in depth. The main components of this algorithm include Cramer's rule, Chio's condensation, and matrix mirroring. Chapter 4 focuses on the parallel implementation of Habgood's algorithm and discusses the resulting performance. Finally, Chapter 5 offers some conclusions and briefly discusses future work that may follow.

Chapter 2

Literature Review

In this section we briefly review LU-decomposition and discuss its computational and communication complexities. We also review the well-studied parallel implementation of LU-decomposition. This will set the stage for later comparing, in a more accurate manner, Habgood's algorithm to LU-decomposition.

2.1 LU-Decomposition

LU-decomposition is employed in solving systems of linear equations of the general form

$$Ax = b. \tag{2.1}$$

The A matrix is decomposed into the form

$$A = LU, \tag{2.2}$$

where L is a lower triangular matrix and U is an upper triangular matrix. For example, a 3×3 matrix becomes

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}. \tag{2.3}$$

The original equation $Ax = b$ can thus be written as

$$LUx = b. \tag{2.4}$$

Then, the equation

$$Ly = b \tag{2.5}$$

is solved for y when applying forward substitution by use of the following expression:[2]

$$y_1 = \frac{b_1}{l_{11}} y_i = \frac{1}{l_{ii}} \left[b_i - \sum_{j=1}^{i-1} a_{ij} yj \right] i = 2, 3, ..., N \tag{2.6}$$

Lastly,

$$Ux = y \tag{2.7}$$

is solved for x by back substitution:

$$x_N = \frac{y_N}{u_{NN}} x_i = \frac{1}{u_{ii}} \left[y_i - \sum_{j=i+1}^{N} u_{ij} xj \right] i = N-1, N-2, ..., 1. \tag{2.8}$$

The LU-decomposition is also particularly useful for computing the determinant of A, since $det(A) = det(L) * det(U)$ and the determinant of a triangular matrix is the product of its diagonal entries.

LU-decomposition requires $\frac{2N^3}{3}$ floating point operations[2], where N denotes the number of columns/rows of the matrix. Therefore, the computational complexity of the algorithm is $O(N^3)$. It becomes evident that in order for an algorithm to compete with LU-decomposition, it, too, must offer a computational complexity of $O(N^3)$ or less.

2.2 Parallel Considerations

Karniadakis and Kirby[4] discuss several methods of decomposing the LU decomposition computation for parallel machines with distributed memory. They base their discussion on the following sequential pseudocode:

1: **for** $i = 1, N-1$ **do**
2: **for** $j = i+1, N$ **do**
3: $l_{ji} = \frac{a_{ji}}{a_{ii}}$
4: **for** $k = i, N$ **do**
5: $a_{jk} = a_{jk} - l_{ji}a_{ik}$
6: **end for**
7: $b_j = b_j - l_{ji}b_i$
8: **end for**
9: **end for**

10: **for** $i = N, 1$ **do**
11: $x_i = b_i$
12: **for** $j = i+1, N$ **do**
13: $x_i = x_i - a_{ij}x_j$
14: **end for**
15: $x_i = \frac{x_i}{a_{ii}}$
16: **end for**

They first look at the triple for loop in lines 1-9. The common method is to store A in interleaved rows or columns. If every processor is assigned a row of A, in the first iteration of the for loop on line 1, one processor, P_1 sends its row to all of the other processors, $P_2 ... P_N$, which perform lines 3 and 5 in parallel. In the second iteration of the outer for loop P_2 sends its row to processors $P_3 ... P_N$ and then remains idle. In this method, y processors are idle after the y^{th} step. Therefore, storing the matrix by rows does not yield a load-balanced program. If the rows are assigned to processors in an interleaved pattern, after broadcasting its row, every processor will be occupied performing row reductions. This method will remain balanced until the number of rows left is less than the number of processors being used. The best option is to divide A into blocks assigned to processors in an interleaved pattern on a computer with a mesh topology. This decomposition allows BLAS3 routines to be used for greater efficiency. In order to perform the back substitution in the double for loop in lines 10-16 in parallel, it is sufficient to assign the matrix to processors

in blocks of rows.

Chapter 3

Habgood's algorithm

K. Habgood's algorithm combines Cramer's rule, Chio's condensation, and matrix mirroring to achieve a resource-efficient scalable solver for systems of linear equations. Each of these steps and why they are needed are described in this chapter.

3.1 Cramer's Rule

Cramer's rule uses determinant computations as a fundamental building block in solving for x. It states that if the matrix A is square and invertible x can be found by computing

$$x_i = \frac{det(A_i(b))}{det(A)}. \tag{3.1}$$

where $A_i(b)$ represents A with the ith column replaced by b. Cramer's rule is generally considered to be impractical for large matrices because the determinant calculations become very difficult as N increases, and the complexity is $O(N!)$[4]. Therefore, Cramer's rule is generally only used on small matrices.

3.2 Chio's Condensation

In order to reduce the complexity of the determinant calculations using Cramer's rule, Habgood proposed to use Chio's condensation[5] to reduce the size of the matrix before applying Cramer's rule. Chio's condensation reduces a matrix A of size $N \times N$ with $a_{11} \neq 0$

to matrix C of size $(N-1) \times (N-1)$. The elements of C are found by the equation

$$c_{ij} = a_{1,1}a_{i+1,j+1} - a_{1,j+1}a_{i+1,1}. \tag{3.2}$$

It can be proved[5] that

$$det(A) = \frac{(det(C))}{(a_{11}^{n-2})}, \tag{3.3}$$

so the determinant of the reduced matrix is equivalent to the determinant of the original matrix. This is necessary for the application of Cramer's rule.

This process basically replaces each element with the 2×2 determinant of a_{11}, the first value in the element's row, the first value in the element's column, and the element being replaced. The first row and column are discarded, resulting in an $(N-1) \times (N-1)$ matrix. For example, reducing a 3×3 matrix looks like:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 0 \\ 0 & (a_{11}a_{22} - a_{21}a_{12}) & (a_{11}a_{23} - a_{21}a_{13}) \\ 0 & (a_{11}a_{32} - a_{31}a_{12}) & (a_{11}a_{33} - a_{21}a_{13}) \end{bmatrix} \rightarrow \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}. \tag{3.4}$$

Calculating each new matrix element requires two multiplications and one subtraction. However, if the first row is divided by the first element before the reduction occurs, $(N-1)^2$ multiplications can be avoided. Since a_{11} is used to calculate every element in the reduced matrix, if it is equal to 1 it can be ignored in the determinant calculations. Normally, in order to accurately calculate the determinant for Cramer's rule the determinant would have to be divided by the a_{11} value that was removed. However, the a_{11} value can be discarded because when Cramer's rule is applied the a_{11} value would cancel out when the determinants are divided.

This reduction can be repeated until the matrix is small enough to apply Cramer's rule. The number of computations that must be done to reduce an $N \times N$ matrix by one row and column is equal to that required to calculate $(N-1)^2$ 2×2 determinants. Therefore, the complexity required to reduce an $N \times N$ matrix to a single determinant value if Cramer's

rule is applied when the matrix reaches size 2×2 is

$$\sum_{k=1}^{N}(N-k)^2 = \frac{N^3}{3} - \frac{N^2}{2} + \frac{N}{6} \sim O(N^3) \tag{3.5}$$

If this had to be done N times to find all N values of x the complexity would then be $O(N^4)$. However, Habgood utilizes a tree-like structure that involves mirroring the matrix and its derivatives in order to avoid reducing the matrix N times.

3.3 Mirroring

Once columns are removed using Chio's condensation, those x values can no longer be computed. In order to solve for all of the x values, the matrix is mirrored before performing the reduction. To be more specific, a second matrix is created and each column y of matrix A of size $N \times N$ is copied into column $N - y$ of the mirror matrix. For example, mirroring a 3×3 matrix would result in the following two matrices:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \rightarrow \begin{bmatrix} a_{13} & a_{12} & a_{11} \\ a_{23} & a_{22} & a_{21} \\ a_{33} & a_{32} & a_{31} \end{bmatrix} \tag{3.6}$$

Generally, when columns or rows of a matrix are swapped one of them must be negated. However, when the determinant is calculated the negation would cancel out, so for this implementation it is ignored.

In order to prevent solving for the determinant of A and $A_i(b)$ separately the b column is appended as an additional column and undergoes the same operations as the A matrix. The exception is that the b column remains in place when mirroring occurs. Carrying the b column through the calculations as an extra column ensures that it is subjected to the same manipulations that it would have been if it had already been substituted into column i. This enables the substitution for Cramer's rule to be delayed until the matrix has reached an appropriate size and Cramer's rule is applied.

After the matrix is mirrored both the original and mirror are reduced using Chio's

condensation and the x values lost in reducing the original are retained in the mirror and vice versa. This mirroring occurs every time the matrix is reduced by half. If Cramer's rule is not applied until the matrix is size 2×2 the original matrix will be broken into $log_2 N$ matrices. After a matrix is mirrored each of the new matrices is responsible for half of the x values. This process is illustrated in figure 3.1.

3.4 Complete Algorithm

Here is a step-by-step description of how Habgood's algorithm is executed:

1. The algorithm begins by mirroring the matrix.

2. Then, the b column is appended to both copies. The b column is reduced along with the matrix throughout the algorithm.

3. In order to obtain the most accurate solution, the row whose first element has the

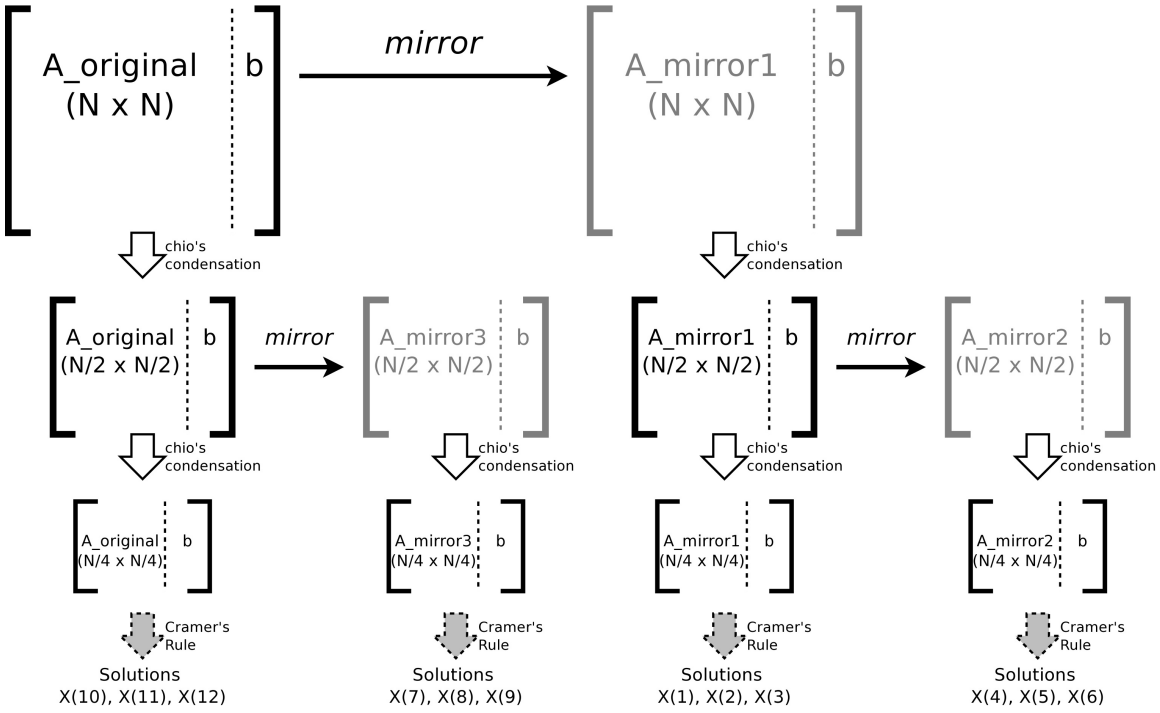


Figure 3.1: Diagram of the the way the original matrix is mirrored and reduced until the resulting matrices are small enough to apply Cramer's algorithm.

largest absolute value is swapped with the first row. This is called partial pivoting and is necessary to maintain numerical stability[4].

4. Then, every element in the first row is divided by the first element in the first row. This division means that the first element in the matrix can be ignored when calculating the determinants for Chio's condensation.

5. Both matrices are reduced using Chio's condensation until they are only half their original size.

6. At this point they are both mirrored, and both matrices are again reduced by half.

This process continues until the matrices reach an appropriate size for Cramer's rule. Then, Cramer's algorithm is applied to each matrix N times, yielding N elements of the x vector.

3.5 Serial Implementation

Habgood implemented this algorithm on a laptop with a 1.5GHz single core Intel Pentium M processor. The laptop contained 1.5 GB of RAM, an L1 cache of 128 KB, and an L2 cache of 1 MB. The bus speed was 400 MHz. The program was written in C and optimizations included cache blocking. He compared his results to LAPACK on the same machine. The algorithm was consistently about 2.5 times slower than LAPACK for matrices from N = 100 to N = 8000, which corresponded to the theoretical analysis. He also experimented with reducing the matrix by more than one row and column at a time during the Chio's condensation step.

Chapter 4

Parallel Implementation

4.1 Hardware Used

For the parallel implementation study an nVidia Tesla C1060 GPU card was employed. This card has 240 cores and 4 GB of memory, and generally represents the current state-of-the-art. The clock rate is 1.3 GHz, and the memory clock rate is 800 MHz. This results in a memory bandwidth of 102 GB/s.

4.2 CUDA Overview

CUDA version 2.3 was used to program the GPU. CUDA stands for Compute Unified Device Architecture and is nVidia's software platform for high performance computing on their GPUs. CUDA is a C-based language that incorporates code that runs on the CPU with kernels that execute on the GPU. It can support 1000s of threads with very little overhead for thread creation. Threads are organized into a grid of thread blocks. Threads in the same block have 16 KB of shared memory and can synchronize, but threads in different blocks cannot communicate. Accesses to shared memory are significantly faster than accesses to global memory because shared memory is on-chip. The number of blocks and threads per block that will execute a kernel is specified when the CPU program calls the kernel. Only one kernel can execute on the GPU at a time. The GPU kernels cannot return any values, information needed by the CPU must be explicitly copied by the CPU from the GPU. The

programmer cannot specify how many or which cores are used. One limitation of CUDA is that it does not support recursion. This was a major drawback for this algorithm, because it is well-suited to recursion.

V. Volkov and J. Demmel[6] documented several important factors to achieving optimal performance when using GPUs. They pointed out that the threads are faster if they remain convergent in an SIMD manner, although the GPU is capable of SIMT (single-instruction, multiple-thread)[7] execution. They also found that shared memory accesses are slower than register file memory accesses, which is contrary to the information found in nVidia's Programming Guide, which states that they are equally fast as long as there are no bank conflicts[7]. Volkov and Demmel also found that the optimal number of threads to have in a block was 64. They also discuss the importance of accessing matrices in row-major order, which doubled their performance for large problems. These techniques influenced the implementation of the systems of linear equations solver.

4.3 Parallel Implementation

The serial implementation of this algorithm on the parallel platform was based on the data parallelism within each operation performed on the matrix. This version used GPU helper functions to perform calculations on the matrix. The general information flow for the system designed involves code running on both the CPU and the GPU, where the latter handled all core linear algebra arithmetic. A CPU-based program begins by reading in from a file or creating the A and b matrices, which consist of integers from -5 to 5 with no zeros. The CPU allocates memory on the GPU for the original matrix plus the b column, its mirror plus the b column, and the x vector. The CPU copies the original matrix to the GPU, then calls a GPU function that mirrors it. The mirror function copies each element from column y to column $N - y$. The mirror function uses a maximum of 240 blocks of 64 threads each. If the matrix is small enough, each thread mirrors one element of the matrix. Otherwise, the elements are mirrored in chunks equal to the total number of threads, which is more efficient when accessing global memory.

The CPU then calls a recursive function, passing it a pointer to the matrix. This

function checks to see if the size of the matrix is 4×4. If so, Cramer's rule is applied four times, four x values are placed in the answers array, and the function returns. Four threads calculate the A_i determinants, and another thread computes the determinant of A. When this last thread completes its calculation, the other threads divide their results by the determinant of A and place their values into the x array. Otherwise, the function enters a while loop until the matrix has been reduced by half the size that was passed in, or it reaches size 4×4. Inside the while loop a GPU function is called to prepare the first row for reduction. This function first searches the first column for the element with the greatest absolute value. If the row with the greatest first value is not the first row, that row is swapped with the first row. Then, the first row is divided by the first element in the first row. Now the first element is 1 and can be ignored when calculating the determinants for Chio's condensation. Since the search for the largest element in the first column must be completed before the other steps can take place, only one block of threads can be used for this function in order for the threads can be synchronized. At the beginning of this function all of the threads are used to place the first column of the matrix into an array in shared memory. Then, one thread searches the array for the largest value and stores it and its index in shared variables. Then, all of the threads swap the rows and divide the first row by the largest value in chunks equal to the total number of threads to coalesce global memory accesses.

Next, a GPU function is called to perform the reduction. In order for the first row and column to not be overwritten before they are used in all the computations for Chios condensation, the reduced matrix is placed in a different memory location. When the memory for the matrix is allocated on the GPU, it is allocated to be twice the size of the matrix. Pointers are kept to the start of the memory and the midpoint. The matrix is copied back and forth between these two halves until it reaches half size. This process is shown in figure 4.1. After each reduction the CPU swaps the pointers to the beginning and midpoint. This ensures that the CPU will know the location of the matrix when it calls the GPU function again because no values are returned from the GPU kernel. To achieve maximum coalescence of global memory accesses, each thread calculates the new values in one column of the matrix. Since these calculations can occur independently, up

to 240 blocks each containing 64 threads are used. The blocks are arranged in two rows, with the threads in the blocks in the first row calculating the new values for the top half of each each column, and those in the blocks in the second row calculating the new value for the bottom half of each column. At the beginning of the reduction kernel all of the threads are used to place the portion of the first column that the block's threads will be referencing into each block's shared memory. Then, each thread stores the element in the first row of its column. The threads then go down their columns calculating the determinants. This allows the global accesses to the matrix and the global stores to be coalesced, and each determinant calculation only requires two global memory accesses. Also, using two rows of blocks allows larger matrices to be solved due to the restriction of having only 16 KB of shared memory per block, as only half of the first column is needed by each block.

If the matrix has an odd number of rows it is reduced to $\frac{N}{2} + 1$, if it is even it is reduced to $\frac{N}{2}$. Once the matrix has been reduced by half a GPU function is called to perform the mirroring. The mirror is placed in the other half of the memory space. The recursive function is then called on both the matrix and its mirror. The resulting matrices are small enough that the memory block can be conceptually divided into fourths with the reduced matrix being copied between fourths in one half during reduction and the mirror being reduced using the two fourths in the other half. When the recursive function returns to main it will have solved for half of the x values. The recursive function is then called on the mirror of the original matrix to solve for the other half of x. This recursion results in a depth-first pattern in reducing the matrices, which is shown in figure 4.2.

When all of the matrices have been reduced the CPU copies the x array from the GPU and compares the answers to those found by LAPACK. The LAPACK answers were stored in a file, along with their respective run times.

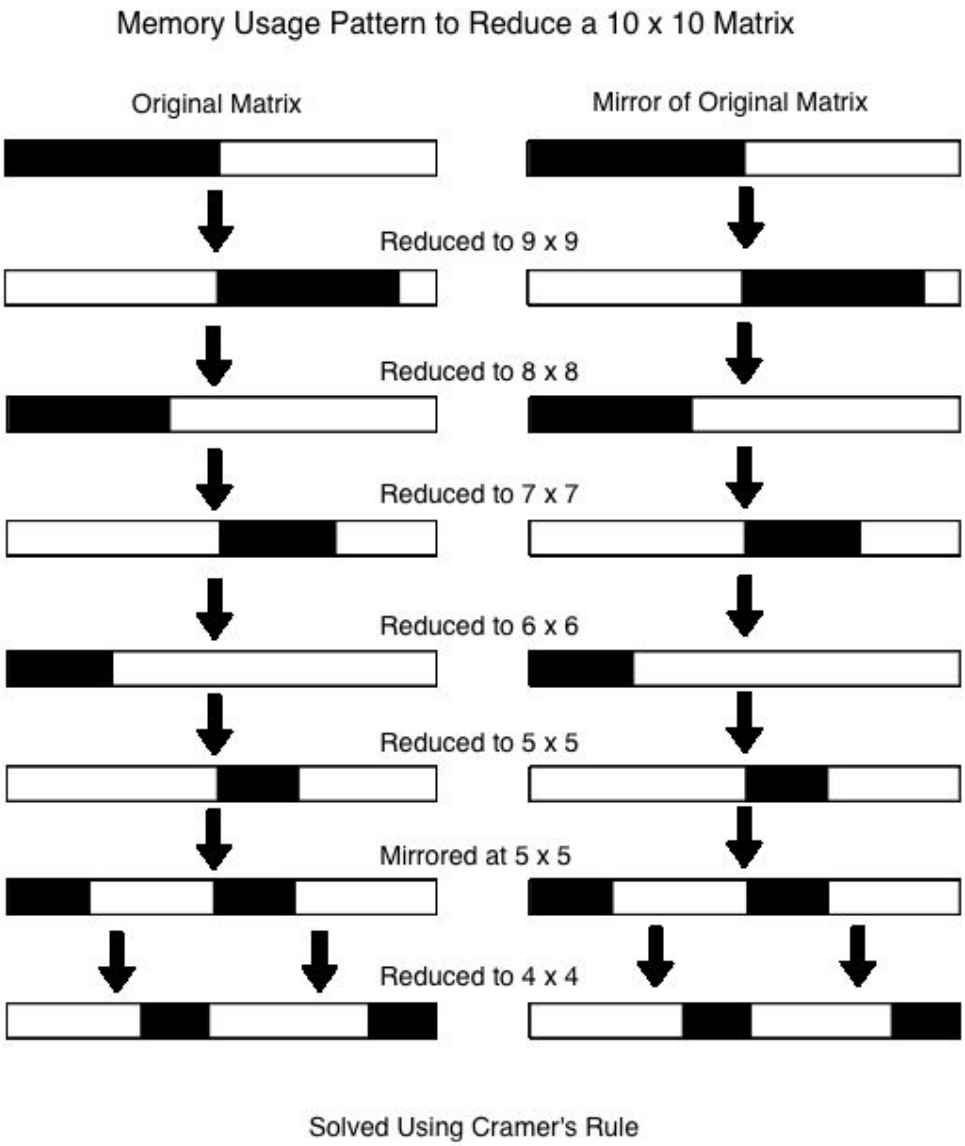


Figure 4.1: Diagram of the memory usage pattern of reducing a 10 x 10 matrix. The dark areas are those containing relevant copies of the matrix.

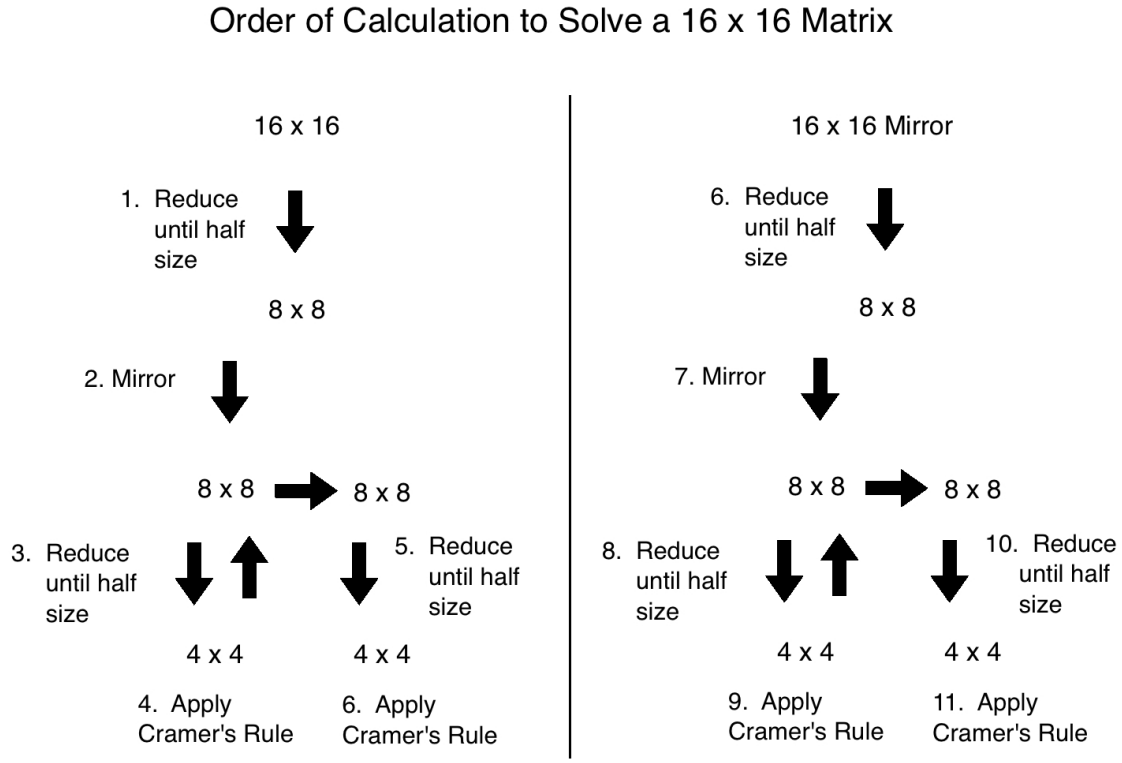


Figure 4.2: Diagram of the order in which the matrices created using Habgood's algorithm are reduced.

4.4 Implementation Results

The performance of the parallel implementation was measured using the gettimeofday function. To compensate for the overhead of gettimeofday, the time was measured for the system of linear equations to be solved twenty times, then the total time was divided by twenty. Twenty trials were sufficient to provide consistent average times for small matrices, and also completed in a reasonable amount of time for larger matrices. This implementation solved systems of linear equations in the times shown in table 4.1.

These results are also graphically illustrated in figure 4.3. The growth of the performance of the parallel version is clearly slower than the serial version, speedups of greater than 14x might be seen for larger values of N. Due to the limit of 16 KB of shared memory per block

N	Serial Implementation (s)	Parallel Implementation (s)	Speedup
100	0.00387	0.023725	.16
300	0.088	0.126814	.69
500	0.461	0.310783	1.48
700	1.955	0.585946	3.34
1000	8.047	1.178758	6.83
1500	36.075	2.991532	12.06
2000	82.652	5.667773	14.58

Table 4.1: Comparison of execution times of serial and parallel implementations of Habgood's algorithm.

the first column of larger matrices does not fit in shared memory for the reduction function. This could be handled by partitioning the matrix so that different blocks work on different segments of the columns so that only part of the first column would need to be stored in memory. However, since these kernels are called with matrices of every size from N to 4 it is very difficult to partition the matrix.

The program was also timed with all of the computations done on the GPU removed. Table 4.2 shows the times without computation and the percentage of total program time that this represents. Since the percentages are very small, this shows that the overhead of creating GPU threads and the bookkeeping that is done on the CPU have little effect on the performance for large matrices. Figure 4.4 shows the communications between the CPU and GPU.

4.5 Alternate Implementations

An alternate implementation of the systems of linear equations solver using Cramer's rule sought to minimize the overhead of transitioning between the CPU and GPU. The most obvious way to do this was to move the entire functionality of the recursive function in the first version to a single GPU function because every time a matrix is mirrored the operations on each of the two resulting matrices are independent. Unfortunately, CUDA does not support recursion, so the CPU recursive function was still needed. However, the while loop that reduced the matrix by half was entirely implemented on the GPU. Since the operations on the matrix must be performed serially so that the correct values are used,

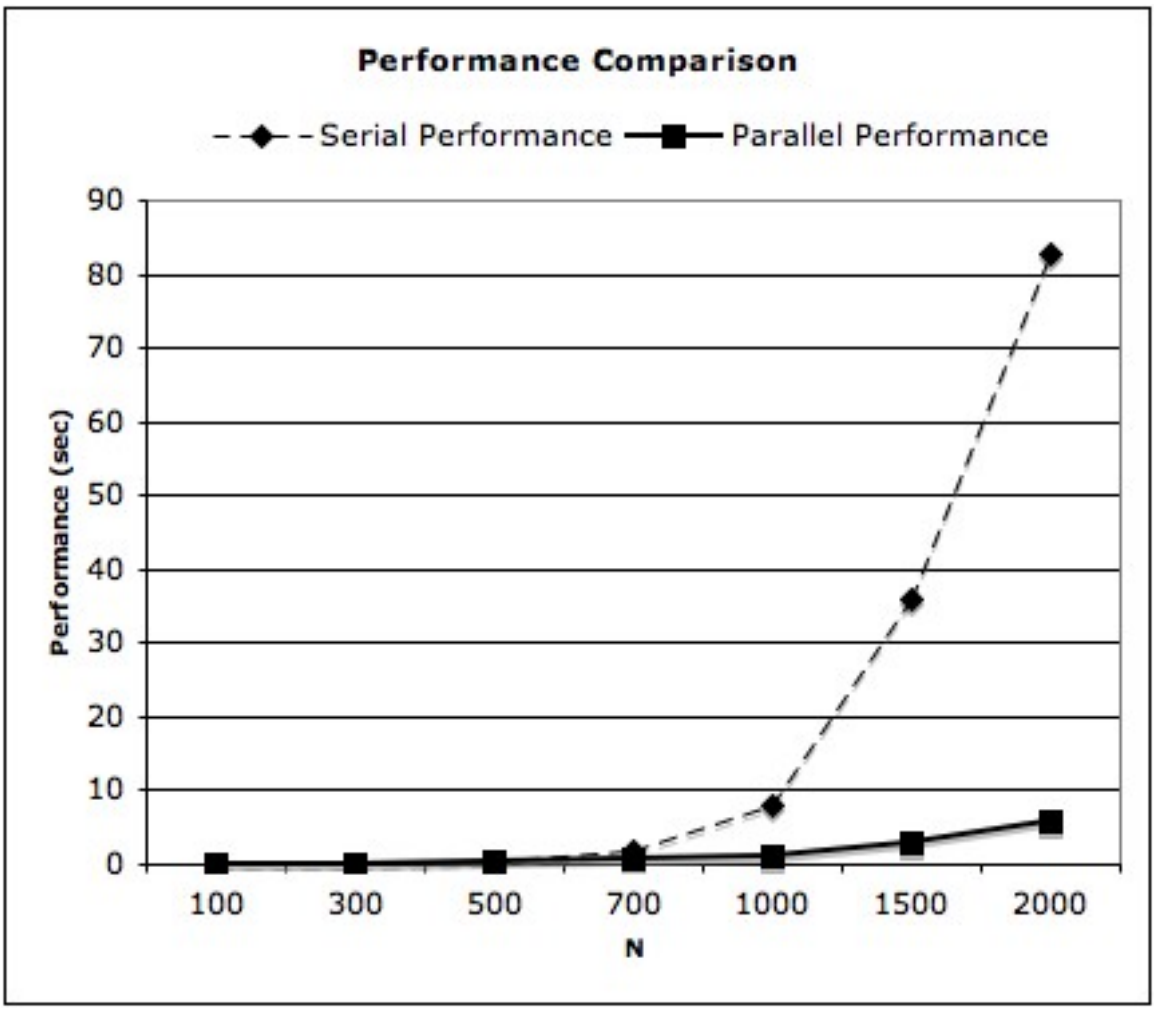


Figure 4.3: Chart comparing the performance of the serial and parallel implementations.

only one block of threads could be used by this function because CUDA does not offer thread synchronization between blocks.

In this version, the A and b matrices are created and mirrored just as in the first version. Next the CPU calls a recursive function that calls a GPU kernel. Thread 0 is designated as the master thread, and it calculates how many reductions it must perform on the matrix. The kernel then enters a loop until the matrix has reached half the size that was passed in. The master thread searches through the first column looking for the largest absolute value. After the largest element in the first column is found, that row is swapped with the first row. Next, the master thread stores the first element of the matrix into a shared variable

N	Time without Computations(s)	Percent of Execution Time
100	0.009184	38.71
300	0.020621	16.26
500	0.032319	10.4
700	0.047289	8.07
1000	0.067853	5.76
1500	0.109910	3.67
2000	0.150901	2.66

Table 4.2: Execution time of the parallel implementation with all GPU computations removed. This is the time used for communications between the CPU and GPU, GPU thread creation, and bookkeeping tasks on the CPU. The percentages shown are the percentage of the execution time of the complete program are taken up by these overheads.

and all of the elements in the first row are divided by the first element. Next, the matrix is reduced by one row and column using Chio's condensation. Then, the loop starts over with finding the largest absolute value in the first row. Before returning control to the CPU, the matrix is mirrored to the second half of the memory block. When the GPU function returns, the CPU recursive function calculates which x values each of the two new matrices will be solving for, and calls itself twice, passing pointers to the respective matrices. Due to the sequential elements of the loop, this kernel could only be run in one block so that the threads could be synchronized. All threads in a block must be on the same core, so this implementation fell far short of good utilization of the 240 cores of the Tesla. If multiple kernels could be executed in parallel this method might be useful to let each core reduce a matrix or mirror and each level of the reduction tree would be calculated at the same time. However, with the restriction of CUDA that only one kernel can be executed at a time this could not be done.

One version was implemented that used many threads to look for the largest value. If each thread in the blocks was responsible for at least 8 elements, each thread found the largest value of a chunk of rows. Each thread placed the largest value it found and the row it was in in arrays at the index corresponding to the index of the thread. When all threads had completed their search, thread 0 searched the array of local largest values and found the absolute largest. The row of the matrix was then retrieved at the same index in the array of local largest rows. This implementation performed slightly worse than simply

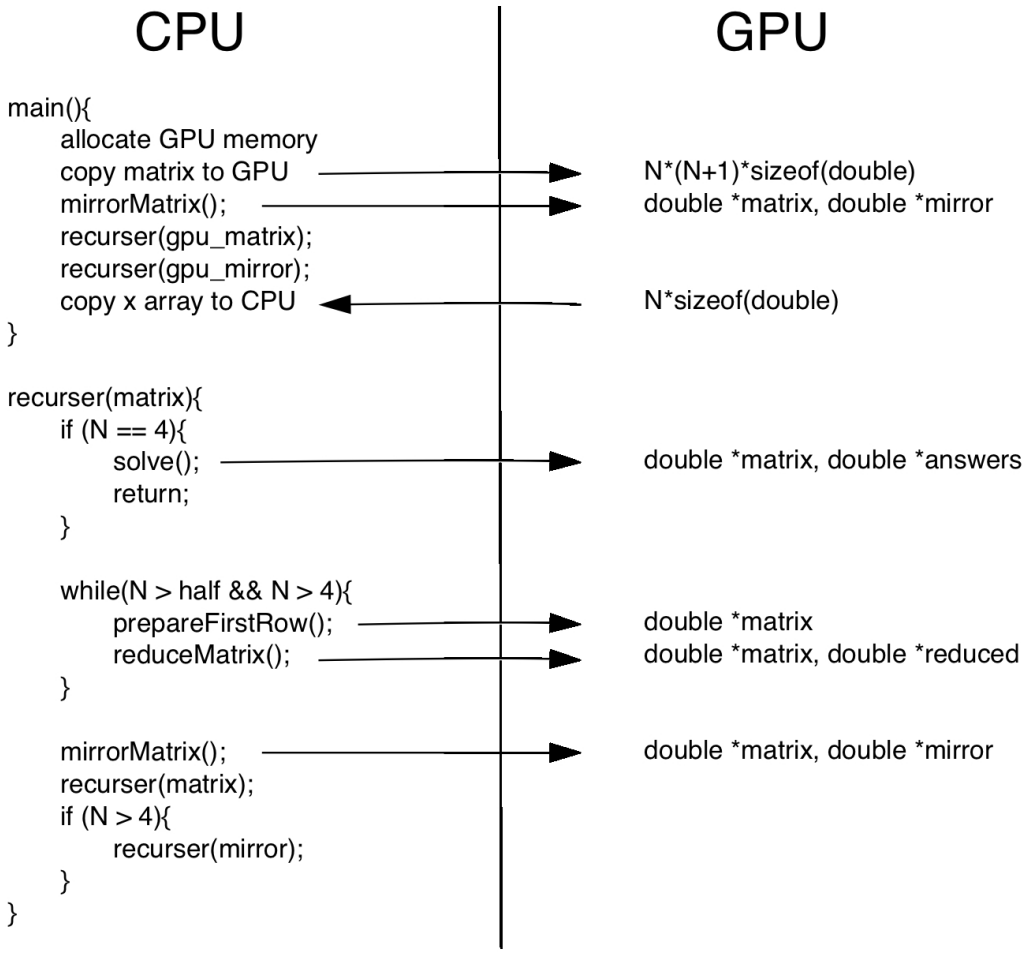


Figure 4.4: Diagram of the flow of data and control between the CPU and GPU.

having one thread examine all of the values in the first column.

Another attempt to improve performance was to store multiple copies of the first row before each reduction. This was done to reduce contention for these values, since they are needed to calculate each new element during reduction. The value in the first column is accessed once by each thread operating on values in that row only once, and each thread stores it to reuse when calculating the rest of its chunk. However, it has to access a new value from the first row each time, and each value in the first row is needed by $N - 1$ threads, although not at the same time because the chunks are not lined up. The extra copies of the first row were copied in the GPU memory by CPU cudaMemcpy calls. Access

to the copies of the first row was interleaved by thread id. The performance was measured with 10, 20, and 30 copies of the first row, and the performance decreased with the number of copies. The overhead of copying the data was greater than the memory access latency due to contention.

Chapter 5

Conclusions

5.1 Thesis Summary

The parallel implementation did achieve a speedup of $14x$ over the serial implementation of Habgood's algorithm. This is less than expected since 240 cores are being used on the Tesla GPU. A conclusion drawn is that the marginal results are due to the restrictions of the CUDA language. The restriction that recursion is not supported in CUDA required control to be transferred between the CPU and GPU once for each mirror operation, twice for each reduction, and once for each application of Cramer's rule to a 4×4 matrix. Another problem was that CUDA does not support synchronization or communication between threads on different cores. This meant that operations requiring any synchronization were limited to one block, and therefore 64 threads. Also, the GPU can only execute one kernel at a time, so this in combination with the previous restriction resulted in very poor utilization of the many cores available.

5.2 Future Work

There are many areas in which this initial attempt to parallelize Habgood's algorithm can be expanded. One area of future work will be to utilize Chio's condensation to reduce matrices by more than one order in each step. Habgood has studied reducing matrices by 4, 8, and 12 rows and columns at a time and found that different values yielded different results for

CPU-based implementation. This suggests that a rigorous treatment of this aspect can be done for GPU systems. The cache offered in upcoming GPU cards paves the way for further memory-management optimization that may be studied in future implementations of this algorithm.

Bibliography

Bibliography

[1] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha, "Lu-gpu: Efficient algorithms for solving dense linear systems on graphics hardware," in *Proceedings of the ACM/IEEE SC 2005 Conference*, Nov. 2005, p. 3.

[2] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *LU Decomposition and Its Applications, Numerical Recipes in FORTRAN: The Art of Scientific Computing (2nd ed.)*. Cambridge University Press, 1992.

[3] K. Habgood and I. Arel, "Revisiting cramer's rule for solving dense linear systems," in *Proceedings of the 2010 Spring Simulation Multiconference*. Society for Computer Simulation International, Apr. 2010.

[4] G. E. Karniadakis and R. M. K. II, *Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and Their Implementation*. Cambridge University Press, 2003.

[5] H. W. Eves, *Elementary Matrix Theory*. Courier Dover Publications, 1980.

[6] V. Volkov and J. W. Demmel, "Benchmarking gpus to tune dense linear algebra," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–11.

[7] *NVIDIA CUDA Compute Unified Device Architecture, Programming Guide, Version 2.0*, 2008.

Vita

Rosanne Lane West was born in Nashville, Tennessee on January 15, 1986. After graduating from Bech High School in 2004, she attended The University of Tennessee in Knoxville. She received her Bachelor of Science degree in Computer Engineering in December of 2008. She remained at The University of Tennessee to continue her studies, where she received her Master of Science degree in Computer Engineering in May of 2010.