PRECONDITIONED CONJUGATE GRADIENT SOLVER

FOR STRUCTURAL PROBLEMS

A Thesis

presented to

the Faculty of the Graduate School

at the University of Missouri-Columbia

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

---

by

XIANGGE LI

Ye Duan, Thesis Supervisor

MAY 2013

The undersigned, appointed by the dean of the Graduate School, have examined the thesis entitled

"PRECONDITIONED CONJUGATE GRADIENT SOLVER FOR STRUCTRUAL PROBLEMS"

Presented by Xiangge Li,

a candidate for the degree of Master of Science in Computer Science

and hereby certify that, in their opinion, it is worthy of acceptance.

____

Dr. Ye Duan

____

Dr. Jianlin Cheng

____

Dr. Zhihai He

ACKNOWLEDGEMENTS

I would like to express the deepest appreciation to my advisor, Dr. Ye Duan, who gave me a lot of research and financial support during my master program study. He always respects and values my ideas and encourages me to follow my interests. Without his help, I would not have been able to get this far in my master program. I would like to thank Dr. P. Frank Pai for his support in sharing his structural mechanics knowledge and sample applications and Dr. Michela Becchi for teaching me so much concerning GPU architectures and performance tuning. My thanks also goes to Dr. Gordon Springer for his support on GPU cluster and for teaching me a lot about parallel programming. Finally, I would like to thank my committee members, Dr. Jianlin Cheng and Dr. Zhihai He, who valued my research and were generously willing to take the time to evaluate my thesis.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS ii

LIST OF FIGURES vi

LIST OF TABLES viii

ABSTRACT… ix

Chapter 1. Introduction 1

Chapter 2. Background 6

2.1. Matrix Computation 6

2.1.1. Storage Format for Vector and Matrix 6

2.1.1. Matrix/Vector Add/Subtract/Multiplication 7

2.1.1. Matrix Solver – Direct Method 8

2.1.2. Matrix Solver – Multiplication of Inverse Matrix 10

2.2. Sparse Matrix 10

2.2.1. Coordinate Format (COO) 11

2.2.2. Compressed Sparse Row/Column Format (CSR / CSC) 12

2.2.3. Ellpack-Itpack Format (ELL) 12

2.2.4. Diagonal Storage Format (DIA) 13

2.2.5. Hybrid Formats 13

2.3. Iterative Methods 14

2.3.1. Conjugate Gradient Method 15

2.3.2. Preconditioner 15

2.3.2.1. Jacobi, SGS, SOR and SSOR Preconditioners 17

2.3.2.2. Incomplete Cholesky Preconditioner 18
2.3.2.3. Incomplete LU Preconditioners 19
2.3.2 SSOR Approximate Inverse Preconditioner 21
2.4. Triangular Matrix Solver 22
2.5. GPU Computation 23
2.5.1. Bandwidth 24
2.5.2. Architecture Design 24
2.5.3. Program Structure 24
2.5.4. Threads System 25
2.5.5. Multi-Kernels & Multi-Devices Support 25
2.5.6. CUBLAS & CUSPARSE Libraries 25
2.5.7. Floating Point 26
2.6. Related Works 28
Chapter 3. Methodology 32
3.1 Implementation 32
3.2 Convergence Impact by Condition Parameter $\boldsymbol{\omega}$ in SSOR-AI 32
Chapter 4. Results and Discussions 38
4.1. Convergence Reports for SSOR-AI 38
4.2. Wide Range Experiments 44
Chapter 5. Conclusion 47
Appendix A. Reports in Detail 48
Appendix A-1. Description of 125 Tested Matrices and Execution Time 48
Appendix A-2. Number of Iterations of all 125 Matrices in 10 Configurations 52

BIBLIOGRAPHY 57

LIST OF FIGURES

Figure 2-1 Dependency Graph Sample 23

Figure 2-2 Calculation order in floating point context 27

Figure 2-3 FMA extreme case 28

Figure 3-1 Sparse pattern of offshore 33

Figure 3-2 Sparse pattern of af_shell3 33

Figure 3-3 Sparse pattern of Parabolic_fem 34

Figure 3-4 Sparse pattern of Apache2 34

Figure 3-5 Sparse pattern of ecology2 34

Figure 3-6 Sparse pattern of thermal2 34

Figure 3-7 Sparse pattern of G3_circuit 34

Figure 3-8 Sparse pattern of Poisson 34

Figure 3-9 Af_shell3 (1:100,1:100) pattern 35

Figure 3-10 Af_shell3 (1:500,1:500) pattern 35

Figure 3-11 Apache2 (1:100,1:100) pattern 35

Figure 3-12 Apache2 (1:500,1:500) pattern 35

Figure 3-13 Ecology2 (1:100,1:100) pattern 36

Figure 3-14 Ecology2 (1:500,1:500) pattern 36

Figure 3-15 Poisson (1:100,1:100) pattern 36

Figure 3-16 Poisson (1:500,1:500) pattern 36

Figure 4-1 Offshore convergence report 38

Figure 4-2 Af_shell3 convergence report 38

Figure 4-3 Parabolic_fem convergence report... 39

Figure 4-4 Apache2 convergence report... 39

Figure 4-5 Ecology2 convergence report.. 40

Figure 4-6 Thermal2 convergence report 40

Figure 4-7 G3_circuit convergence report 41

Figure 4-8 Poisson convergence report.. 41

Figure 4-9 Percentage of iterations reduction for CG with SSOR-AI (0.01) to CG......... 46

LIST OF TABLES

Table 3-1 Description of Matrices 33

Table 4-1 Execution time comparison of SSOR-AI (1.0) and other preconditioners....... 43

Table 4-2 Convergence comparison for all CG with SSOR-AI preconditioners............. 44

Table A-1 Description of tested matrices and execution time 48

Table A-2 Number of Iterations........ 52

PRECONDITIONED CONJUGATE GRADIENT SOLVER

FOR STRUCTRUAL PROBLEMS

Xiangge Li

Dr. Ye Duan, Thesis Supervisor

ABSTRACT

Matrix solvers play a crucial role in solving real world physics problem. In engineering practice, transition analysis is most often used, which requires a series of similar matrices to be solved. However, any specific solver with/without preconditioner cannot achieve high performance gain for all matrices. This paper recommends Conjugate Gradient iterative solver with SSOR approximate inverse preconditioner for general engineering practice instead of Conjugate Gradient alone. The author uses experiments on 125 symmetric positive definite matrices derived from real structural problems to endorse this recommendation. SSOR approximate inverse preconditioner shows a competitive advantage to provide stable performance improvement (average 12.6x speedup to CG). And, a general setting ($\omega = 0.01$) will effectively prevent the failure of SSOR approximate inverse preconditioner among a wide range of data derived from analysis of structural problems.

CHAPTER 1. INTRODUCTION

The solution to sparse linear systems plays a crucial role in such engineering fields as physics based modeling and simulation, circuit simulation, mechanics of materials, geophysics and many other application fields. For example, the matrix Geo_1438 (Davis and Hu 2011) used in my experiments comes from a real geomechanical problem. The matrix is used to calculate 3D discretized displacement of a deformed region of the earth crust subject to underground force. In this linear system $[K]\{u\} = \{f\}$, the unknown vectors $\{u\}$ represent the 3D discretized displacement of an earth crust region, where right side vectors $\{f\}$ represents the underground force inside and outside of this crust region. The coefficient matrix $[K]$ is the model that couples relationship of the deformation displacements of the crust and underground force.

Two categories of methods are used to solve linear systems: direct methods and iterative methods. Direct methods are based on variants of Gaussian elimination, which is direct and easy to understand. The unknowns will be solved one by one in continuous iterations. However it does not show good scalability. On the contrary, iterative methods are suggested for a sparse linear system especially when matrix size increases to a very large scale.

The iterative methods exhibit better parallelism and then scale well on solving larger problems. Many iterative methods have been proposed and analyzed from a mathematical viewpoint. The non-stationary iterative methods derived from Krylov subspace have been proved to be highly effective (Bai et al. 1987), especially for solving sparse linear systems. This paper targets the Conjugate Gradient iterative method which is one of the families designed for solving symmetric positive definite linear systems. Theoretically,

preconditioners, matrices used to transform a coefficient matrix to show a more favorable spectrum property for converging, also are proved to be an efficient way to improve performance of the iterative method. However, no silver bullet can be found to solve all matrices quickly and easily. Each combination of an iterative method and a configured preconditioner, which from hereon will be referred to as a method configuration in this paper, has its limitations and only works efficiently for particular matrices. Engineering practices expect a method configuration to facilitate solving matrices in a wide range.

In practice, applying a preconditioner into iterative methods introduces extra costs, both for constructing the preconditioner during the initial step and when applying transformation per iteration. A preconditioner with impressive improvement in converging iterations may introduce heavy extra cost during the constructing step and may eventually compromise the whole performance (Naumov 2011a). Too simple of a preconditioner, like the Jacob preconditioner, brings too little impact on the performance of iterative methods (Ament et al. 2010). And, if the coefficient matrix naturally shows good spectral property, which directly fits well within the iterative method, applying a preconditioner is unnecessary. One example of a good fit can be found in the superlinear convergence scenario for the Conjugate Gradient (Concus and Golub 1976). Applying a preconditioner requires solving two triangular matrices per iteration, which is hard to parallel and is the bottleneck to applying preconditioners. To balance the performance improvement by preconditioner and the overhead for applying the preconditioner, approximate inverse preconditioners have been introduced into the world. By taking side effect of accuracy and robustness loss in the approximation, the new approximate inverse preconditioners can be directly applied by an easily paralleled matrix-vector

multiplication operation (Benzi, Cullum, and Tuma 2000; Benzi and Tuma 1999; Benzi and Tuma 1998; Chow and Saad 1998; Cosgrove, Diaz, and Griewank 1992; Gravvanis 2002; Kolotilina and Yeremin 1993; Grote and Huckle 1997). General research suggests that an approximate inverse preconditioner should be used on finite difference discretization of boundary value problems, whose coefficient matrices tends to "more diagonally dominant"(Benzi and Tuma 1999). However, since the approximation impairs the robustness of the SSOR preconditioner, SSOR approximate inverse (SSOR-AI) is more likely to cause convergence failure than SSOR preconditioner. This problem requires an SSOR-AI method configuration that can successfully prevent convergence failure for the majority of the time.

GPU computing, which serves as a cheap massive parallel solution, has been widely used in this field and has proven to be more efficient than CPU, especially for large sparse matrices. Most existing research is focused on tuning the performance for a specific matrix to claim a very high speed up. The Poisson equation has been used for analysis in many papers (Helfenstein and Koko 2011; Gui and Zhang 2012; Michels 2011; Ament et al. 2010), as the non-zero elements in the coefficient matrix of Poisson equations are scattered into larger bands for applying iterative methods. A special preconditioner, Incomplete Poisson (Ament et al. 2010), which evolved from sparse approximate inverse algorithms, has been developed for solving Poisson equations only. Previous research analyzed various preconditioners on GPU, such as Parallel Conjugate Gradient algorithm combined with Jacobi (Georgescu and Okuda 2007; Ament et al. 2010), Incomplete Cholesky (Gui and Zhang 2012), Incomplete LU (Gui and Zhang 2012), ILU (0)/SSOR (Ament et al. 2010; Yu et al. 2012), and SSOR Approximate Inverse (Helfenstein and

Koko 2011) preconditioners. Although the studies mentioned above all claimed very high speed up (up to 15x) for performance of GPU compared to CPU in solving linear systems, papers from Intel researches (Lee et al. 2010) and (Gregg and Hazelwood 2011) challenged previous reports of 10x to 1000x speed up reports on GPU stating that these results were obtained only when comparing GPU with an insufficient CPU tuning code, and the reasonable GPU performance speed up narrows to an average of 2.5x that of CPU. Gui and Zhang (2012) claim more than 7x speed up in solving Poisson equations, but their study only compares the GPU results with a single thread CPU implementation without exploiting the intuitive performance improvement methods: multi-threading technology or BLAS/LAPACK libraries. The paper from NVIDIA researchers (Naumov 2011b) also supports the low speed up results. The NVIDIA paper compares Conjugate Gradient with multiple preconditioners among multiple matrices set to report only an average speed up range from 2.07x to 2.69x. The reports are based on compatible experimental environments with leading commercial linear algebra libraries, which are NVIDIA CUBLAS & CUSPARSE libraries for GPU and the Intel MKL library for CPU. Li and Saad (2013) also compared different preconditioners for Conjugate Gradient method among a wide range of data. The overall performance reported in this study of GPU to CPU was also less than 4x.

A speed up cannot be guaranteed when applying GPU solver with the same configuration among different matrices (Li and Saad 2013; Naumov 2011b). When conducting continuous computation/simulation on several time sensitive matrices (Bolz et al. 2003), preconditioner tuning is limited in its ability to adapt to continuous data sets. Some previous research (Bolz et al. 2003) focusing on simulation of specific problem

provided approximately a 2x speed up on GPU, which is still a very impressive speedup compared to CPU. Therefore solving matrices on GPU has proven to be efficient in engineering practice.

On the other hand, method configuration with a higher speed up does not mean the calculation time for this configuration is the best among all configurations (Yu et al. 2012). Preconditioners that are easier to be paralleled and show higher speed up on GPU are not guaranteed to provide the best convergence improvement in calculation.

In this study, I chose SSOR approximate inverse preconditioner for Conjugate Gradient solver to provide stable performance improvement to CG on GPU. SSOR preconditioner has proven effective on improving convergence and is easy to construct. SSOR-AI preconditioner keeps effective convergence improvement feature of SSOR and is also easy to construct. Furthermore, the application of the SSOR-AI only needs an extra matrix-vector multiplication in each iteration, so it is easy to be paralleled on GPU with relatively low overhead. The robustness loss, which may deteriorate convergence, in the approximation was the major problem encountered in this research. The following research questions are addressed in this paper:

1. When applying SSOR approximate inverse preconditioner, will the convergence rate suddenly deteriorate with a specific or range of ω value?
2. Are any patterns visible in the relationship of convergence rate to ω values?
3. Is there any method configuration which can be used to make sure applying SSOR approximate inverse preconditioner gains universal advantages to Conjugate Gradient solver without a preconditioner?

CHAPTER 2. BACKGROUND

2.1. Matrix Computation

2.1.1. Storage Format for Vector and Matrix

Two categories of storage formats exist for vector and matrix: dense format and sparse format. Dense format is a storage policy that linearly stores all elements of matrix/vector in memory. On the contrary, sparse format is a storage policy that only stores non-zero elements in memory along with position information of those elements. The common used position indices are zero-based and one-based, whose index base respectively are zero and one.

The following matrix is an M*N size zero-based indexing matrix:

$$\begin{bmatrix} A_{0,0} & A_{0,1} & \cdots & A_{0,N-1} \\ A_{1,0} & A_{1,1} & \dots & A_{1,N-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{M-1,0} & A_{M-1,1} & \dots & A_{M-1,N-1} \end{bmatrix}$$

The following matrix is an M*N size one-based indexing matrix:

$$\begin{bmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,N} \\ A_{2,1} & A_{2,2} & \dots & A_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ A_{M,1} & A_{M,2} & \dots & A_{M,N} \end{bmatrix}$$

In mathematics world, row vector and column vector are different. However, their storage in memory are the same with extra explicit (some tag in structure) or implicit (semantics in algorithm) information to identify them. Dense format vector is a single data array linearly stored in memory. Sparse format vector is composed by a data array stored all non-zero elements in the vector and an integer index array that stored positions of those non-zero elements in equivalent dense format vector.

In solving linear systems, two dimensional matrices are used. Memory is linear structured, so two dimensional matrixes have to be serialized into one dimensional memory representation. Two major formats are used to represent two dimensional matrices: row-major format and column-major format. In row-major format matrix, all elements in a row are sequentially (column index from low to high) placed in memory before storing any element in rows with higher row index. In column-major format matrix, all elements in a column are sequentially (row index from low to high) placed in memory before storing any element in columns with higher column index.

For example, a one-based M * N dense matrix A

$\begin{bmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,N} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ A_{M,1} & A_{M,2} & \cdots & A_{M,N} \end{bmatrix}$ will be serialized in memory as

$[\, A_{1,1}\ \ A_{1,2}\ \cdots A_{1,N}\ A_{2,1}\ \ A_{2,2} \cdots A_{2,N}\ \cdots A_{M,1}\ \ A_{M,2} \cdots A_{M,N}\,]$ (In Row-major format) or

$[\, A_{1,1}\ \ A_{2,1}\ \cdots A_{M,1}\ A_{1,2}\ \ A_{2,2} \cdots A_{M,2}\ \cdots A_{1,N}\ \ A_{2,N} \cdots A_{M,N}\,]$ (In Column-major format).

Sparse matrix format will be discussed in Chapter 2.2.

2.1.1. Matrix/Vector Add/Subtract/Multiplication

It is easy to use block division or row/column division to distribute whole matrix computation into multiple independent smaller matrix computation. This key feature of these operations' domain division is independent subdomain resolution. Any subdomain problem can be solved independent without any interference of other subdomain problems. So only one synchronization operation will be needed to make sure all subdomain results are solved. That is the perfect fit for Single Instruction Multiple Data (SIMD) parallel architecture: multiple data blocks will be processed homogeneously.

Element-wise operations for Matrix/Vector Addition and Subtraction are $C[i,j] = A[i,j] \pm B[i,j]$. All operations to each element are independent, so it is easy to be parallel.

Element-wise operations for matrix-vector multiplication is $C[i] = \sum_{k=1}^{n}(A[i,k] * B[k])$. The operations for different $C[i]$ is also independent, and it also easy to be parallel.

2.1.1. Matrix Solver – Direct Method

Direct method is typically derived from Gaussian elimination. Direct method Solver use two steps to solve $[A]\{x\} = \{b\}$:

1. Compute the factorization of coefficient matrix $[A]$
2. Use the factorization to solve $[A]\{x\} = \{b\}$

The first step is the most intensive part. After the first step, the second step can be easily applied and cost only trivial effort comparing to first step. Many methods have been developed to reuse the factorization result in step1 to solve many right side vectors$\{b\}$. Factorization step, or step 1, only need to be done once or fewer times than step 2, so computation time would be saved.

In Gaussian elimination, an N*(N+1) augment matrix would be used to solve

$[A]\{x\} = \{b\}$. Initially the augment matrix will be initiated as $[A|b]$, then multiple steps of elimination will be applied to augment matrix. The final result in augment matrix would be $[I|x]$, where matrix $[I]$ is unit matrix.

For example, the augment matrix of equations

$$\begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \\ A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{Bmatrix}$$

Would be

$$\left[\begin{array}{cccc|c} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} & b_1 \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} & b_2 \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} & b_3 \\ A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} & b_4 \end{array}\right] \qquad (2.1.1)$$

Gaussian elimination will first eliminate all elements in lower triangular part of the matrix to zeros.

$$\left[\begin{array}{cccc|c} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} & x_1' \\ 0 & A_{2,2} & A_{2,3} & A_{2,4} & x_2' \\ 0 & 0 & A_{3,3} & A_{3,4} & x_3' \\ 0 & 0 & 0 & A_{4,4} & x_4' \end{array}\right]$$

Upper triangular part of the matrix will be eliminated to zeros in next step.

$$\left[\begin{array}{cccc|c} A_{1,1} & 0 & 0 & 0 & x_1'' \\ 0 & A_{2,2} & 0 & 0 & x_2'' \\ 0 & 0 & A_{3,3} & 0 & x_3'' \\ 0 & 0 & 0 & A_{4,4} & x_4'' \end{array}\right]$$

And finally remaining matrix will be normalized with diagonal elements.

$$\left[\begin{array}{cccc|c} 1 & 0 & 0 & 0 & x_1 \\ 0 & 1 & 0 & 0 & x_2 \\ 0 & 0 & 1 & 0 & x_3 \\ 0 & 0 & 0 & 1 & x_4 \end{array}\right]$$

Gauss-Jordon elimination is alternative algorithm to Gaussian elimination. Gauss-Jordon elimination will eliminate all elements except the diagonal element in a column to zeros and normalize the diagonal element to one in each step.

$$(2.2.1) = \left[\begin{array}{cccc|c} 1 & A_{1,2} & A_{1,3} & A_{1,4} & x_1' \\ 0 & A_{2,2} & A_{2,3} & A_{2,4} & x_2' \\ 0 & A_{3,2} & A_{3,3} & A_{3,4} & x_3' \\ 0 & A_{4,2} & A_{4,3} & A_{4,4} & x_4' \end{array}\right] = \left[\begin{array}{cccc|c} 1 & 0 & A_{1,3} & A_{1,4} & x_1'' \\ 0 & 1 & A_{2,3} & A_{2,4} & x_2'' \\ 0 & 0 & A_{3,3} & A_{3,4} & x_3'' \\ 0 & 0 & A_{4,3} & A_{4,4} & x_4'' \end{array}\right]$$

$$= \left[\begin{array}{cccc|c} 1 & 0 & 0 & A_{1,4} & x_1''' \\ 0 & 1 & 0 & A_{2,4} & x_2''' \\ 0 & 0 & 1 & A_{3,4} & x_3''' \\ 0 & 0 & 0 & A_{4,4} & x_4''' \end{array}\right] = \left[\begin{array}{cccc|c} 1 & 0 & 0 & 0 & x_1 \\ 0 & 1 & 0 & 0 & x_2 \\ 0 & 0 & 1 & 0 & x_3 \\ 0 & 0 & 0 & 1 & x_4 \end{array}\right]$$

These two algorithms require the same amount of calculation in total. However, Gauss-Jordon Elimination processes the same amount of calculation in half of inter-dependent iterations which are needed for Gaussian Elimination. By reducing number of iterations to half, Gauss-Jordon Elimination requires less synchronization between iterations and exposes more parallelism per iteration. By performing row elimination on all rows per iteration, Gauss-Jordon Elimination redistributes homogenous loads to all iterations that shows balanced job load among all iterations.

2.1.2. Matrix Solver – Multiplication of Inverse Matrix

Another intuitive method to solve $[A]\{x\} = \{b\}$ is $\{x\} = [A]^{-1}\{b\}$. By first calculating inverse matrix $[A]^{-1}$of coefficient matrix $[A]$, then right side vectors $\{b\}$ is left multiplied by inverse matrix $[A]^{-1}$ to get vector result $\{x\}$. This method can also be used to solve multiple right side vectors $\{b\}$ to save computation time. However, the memory requirements for solving $[A]^{-1}$ are much larger than other methods.

The inverse matrix $[A]^{-1}$ can be solved with following equation:

$$[A][A]^{-1} = [I]$$

The solving process also can use Gaussian/Gauss-Jordon elimination with an N*(2N) size augment matrix.

2.2. Sparse Matrix

In real world practice, the non-zero elements of the coefficient matrix are relative fewer to the zero elements of the matrix. Calculations that involved zero elements in the matrix can be ignored in most cases. This observation motivates the idea of sparse matrix format that only stores non-zero elements of the matrix with extra information to locate elements.

As the calculation of majority elements of matrix, zero elements, can be ignored, the complexity of algorithm is also dramatically changed. For example, Gauss-Jordon elimination in sparse matrix is typically worse than Gaussian elimination because the new non-zero elements along with new calculation based on them are introduced in upper triangular matrix in the column elimination process. The algorithm for dense matrix needs to be reanalyzed in sparse matrix context.

There are varies of sparse matrix formats. Different formats facilitate different sparse matrix patterns and different algorithms.

2.2.1. Coordinate Format (COO)

In COO format representation, only non-zero elements in the matrix are recorded. COO format is composed by three equal elements arrays, respectively storing values of all non-zero elements, their row indices and column indices along with the number of non-zero elements in the matrix. By default, row-major format would be used to serialize 2D matrix into value array. For example, the matrix below

$$\begin{bmatrix} 20 & 0 & 0 & 9 \\ 0 & 15 & 0 & 0 \\ 11 & 0 & 8 & 0 \\ 7 & 0 & 0 & 25 \end{bmatrix} \tag{2.2.1}$$

is presented in zero-based COO format as three arrays:

Row Indices: $[0 \quad 0 \quad 1 \quad 2 \quad 2 \quad 3 \quad 3]$

Column Indices: $[0 \quad 3 \quad 1 \quad 0 \quad 2 \quad 0 \quad 3]$

Non-zero Values: $[20 \quad 9 \quad 15 \quad 11 \quad 8 \quad 7 \quad 25]$

2.2.2. Compressed Sparse Row/Column Format (CSR / CSC)

In many scenarios, we need to iterate all elements in a row or multiple rows. When using row-major COO format, an O (log (number of non-zero elements)) search process has to be done to locate the index range of a row's data. If keeping track of the index range of each row's data in serialized storage, only an O (1) complexity would be enough to locate the index range of each row's data. That introduces row-major CSR format, which stores index range of each row's data instead of storing row indices of all non-zero elements. Also for the same reason, we get the column-major CSC format.

The zero-based CSR format of matrix (2.2.1) is

Row Pointers: $[0 \quad 2 \quad 3 \quad 5 \quad 7]$

Column Indices: $[0 \quad 3 \quad 1 \quad 0 \quad 2 \quad 0 \quad 3]$

Non-zero Values: $[20 \quad 9 \quad 15 \quad 11 \quad 8 \quad 7 \quad 25]$

2.2.3. Ellpack-Itpack Format (ELL)

Ellpack-Itpack sparse matrix format (ELL) uses an N*K data matrix and an N*K indices matrix, which contains corresponding column indices of data matrix elements, to represent an N*N dense matrix. The K is at least the maximum number of non-zero elements per row in the original dense matrix. If the number of non-zero elements in a row is less than K, the padding elements of the row will be filled with 0 in data matrix. And corresponding column indices of the row will be filled with -1.

For example, the zero-based ELL format of matrix (2.2.1) is

$$\text{Data} = \begin{bmatrix} 20 & 9 \\ 15 & 0 \\ 11 & 8 \\ 7 & 25 \end{bmatrix} \qquad \text{Indices} = \begin{bmatrix} 0 & 3 \\ 1 & -1 \\ 0 & 2 \\ 3 & -1 \end{bmatrix}$$

Comparing to COO or CSR format, the ELL format can be calibrated padded to meet specific requirements of machine word alignment, cache line alignment, or memory page boundary alignment. With the alignment, penalty of misaligned memory access can be bypassed. However the memory spatial locality may be compromised due to the padding operations. So the K value has to be chosen carefully.

2.2.4. Diagonal Storage Format (DIA)

Diagonal Storage Format (DIA) uses a data matrix and a distance vector to store all non-zero elements that reduces the information needed for locating the non-zero elements. A K band N*N size dense matrix will be represented as an N*K data matrix and a K element distance vector. The elements from same diagonal band will be stored in same column of data matrix and the elements from same row will also be stored in same row of data matrix. Distance vector stores the distance of each diagonal to main diagonal. The negative and positive distance value respectively represents diagonal in lower and upper triangular part of matrix. It is particularly usefully for diagonal dominant sparse matrices, which normally generated from finite element or finite difference discretization.

For example, the zero-based DIA format of matrix (2.2.1) is

$$\text{Data} = \begin{bmatrix} 0 & 0 & 20 & 9 \\ 0 & 0 & 15 & 0 \\ 0 & 11 & 8 & 0 \\ 7 & 0 & 25 & 0 \end{bmatrix}$$

$$\text{Distance} = \begin{bmatrix} -3 & -2 & 0 & 3 \end{bmatrix}$$

2.2.5. Hybrid Formats

Many hybrid formats have been developed to use two sparse matrix format respectively represented part in regular pattern and irregular part of non-zero elements of matrix.

Among all formats mentioned above, DIA and ELL are effective for sparse matrix-vector multiplication, while COO and CSR are more flexible and easy to operate. So combination formats of these two categories can be useful. The most widely used hybrid format is HYB, a combination of ELL and COO format.

2.3. Iterative Methods

Unlike the direct methods, iterative methods use an iterative representation to converge to the solution with more accurate result iteratively. The solution from iterative methods is an approximation of exact solution. All unknowns are solved at the same iteration when specified tolerance of residuals is reached. It is impossible to predict the amount of iteration needed for convergence of the iterative methods except Jacobi method. However, if converged, other iterative methods are converged faster than Jacobi method.

There are two types of Iterative methods: stationary methods and non-stationary methods. Stationary methods are those iterative methods that can be expressed in iterative form $\{x\}^k = C_1\{x\}^{k-1} + C_0$, where C_0 and C_1 are independent to iterative count k. If converged with proper parameter configuration, solution error per iteration will gradually downgrade. That is what the "stationary" refers to. Stationary methods are older and simpler to implement and usually not as effective as non-stationary methods. Most non-stationary methods are based on the idea of sequences of orthogonal vectors and can be highly effective. The convergence rate of iterative methods substantially depends on the spectrum of the coefficient matrix. So a transformation matrix (called preconditioner) that transforms the coefficient matrix into one with more favorable spectrum would be crucial to the iterative methods. Common preconditioners can be expressed in multiplication of

two triangular matrices (Barrett et al. 1987) and those two triangular matrices will be solved in sequence.

Most of the non-stationary iterative methods are based on Krylov subspace. Conjugate Gradient (CG) method is the one for solving symmetric positive definite matrices. Positive definite is said to an N*N size symmetric matrix $[A]$ if for any N elements non-zero vector$\{z\}$, $\{z\}^T[A]\{z\}$ is positive.

2.3.1. Conjugate Gradient Method

The Conjugate Gradient (CG) method is the oldest and effective non-stationary method for solving symmetric positive definite systems. The conjugate gradient method consists of three tightly coupled iterative vectors in any i^{th} iteration: iterates $x^{(i)}$, residuals $r^{(i)}$, and search directions/paths $p^{(i)}$.

The Conjugate Gradient algorithm is:

Initial step: $p^{(0)} = r^{(0)} = b - Ax^{(0)}$

Iterative steps: start from iteration 0, loop step 1 to step 4 until stop criteria meets.

Step 1: $\alpha_i = \frac{|r^{(i)}|^2}{dot(p^{(i)}, Ap^{(i)})} = \frac{{r^{(i)}}^T r^{(i)}}{{p^{(i)}}^T Ap^{(i)}}$

Step 2: $x^{(i+1)} = x^{(i)} + \alpha_i p^{(i)} \;\; and \;\; r^{(i+1)} = r^{(i)} - \alpha_i Ap^{(i)}$

Step 3: $\beta_{i-1} = \frac{|r^{(i+1)}|^2}{|r^{(i)}|^2} = \frac{{r^{(i+1)}}^T r^{(i+1)}}{{r^{(i)}}^T r^{(i)}}$

Step 4: $p^{(i+1)} = r^{(i+1)} + \beta_i p^{(i)}$

2.3.2. Preconditioner

The spectral properties of the coefficient matrix determines the convergence rate of iterative methods (Hestenes and Stiefel 1952; van der Sluis and van der Vorst 1986). So

if we can transform the linear system into equivalent one that has the same solution but with more favorable spectral properties, we may boost up the cost of solving process. The matrix that served in the transformation is called preconditioner. In other word, preconditioner matrix $[M]$ approximates the coefficient matrix $[A]$, and spectral properties of $[M]^{-1}[A]$ is more favorable than [A]. Then transformed system

$$[M]^{-1}[A]\{x\} = [M]^{-1}\{b\}$$

has the same solution as the original system $[A]\{x\} = \{b\}$ and iterative methods applied in new transformed system will converge in less iterations than in original system (van der Sluis and van der Vorst 1986).

For symmetric positive definite matrices, the spectral condition number κ is defined as

$$\boldsymbol{\kappa} = \frac{\boldsymbol{\lambda_{max}}}{\boldsymbol{\lambda_{min}}}$$

with the maximum and minimum eigenvalues λ_{max} and λ_{min}. Note that the identity matrix has a value of κ = 1. The objective of preconditioning is to transform the original system into an equivalent system with the same solution, but a lower condition number. So preconditioner M^{-1} should approximate A^{-1} to make new κ approximate to 1.

The above theoretical transformation will not be directly used in real world scenarios. When original coefficient matrix A is symmetric, it is crucial to preserve the symmetry for success of some iterative methods. However, transformed coefficient matrix $[M]^{-1}[A]$ is not guaranteed to remain symmetric nor definite even if [A] and [M] are. That is why in practice, the preconditioner M would be represented in the factored form:

$$[M] = [M_L][M_R],$$

Where $[M_L]$ and $[M_R]$ are two triangular matrices. The transformed system will be

$$[M_L]^{-1}[A][M_R]^{-1}([M_R]\{x\}) = [M_L]^{-1}\{b\}$$

and the preconditioned process will be split into two phase:

Phase 1: solving $\{y\}$ from $[M_L]^{-1}[A][M_R]^{-1}\{y\} = [M_L]^{-1}\{b\}$

Phase 2: compute $\{x\}=[M_R]^{-1}\{y\}$.

If coefficient matrix A is symmetric and positive definite, transformed coefficient matrix $[M_L]^{-1}[A][M_R]^{-1}$ can preserve symmetric and positive definite properties with choosing $[M_L] = [M_R]^T$.

2.3.2.1. Jacobi, SGS, SOR and SSOR Preconditioners

One of the simplest ways to generate a preconditioner is performing incomplete factorization of coefficient matrix A. There are multiple preconditioners.

First we decomposed coefficient matrix A as A = L + D + U, where D, L, U represents its diagonal, strictly lower and strictly upper triangular part respectively.

Jacobi preconditioner (or diagonal scaling in Georgescu and Okuda (2007)) can be expressed by

$$M_{\text{Jacobi}} = [D]$$

Symmetric Gauss Seidel preconditioner can be expressed by

$$M_{SGS} = ([D]+[L]) * [D]^{-1}([D]+ [U])$$

SOR preconditioner can be expressed by

$$M_{SOR} = ([D]+\omega[L])/\omega * ([D]+ \omega[U])/ \omega$$

SSOR preconditioner can be expressed by

$$M_{SSOR}(w) = \frac{1}{2-w}\left(\frac{1}{w}D + L\right)(\frac{1}{w}D)^{-1}(\frac{1}{w}D + U)$$

SSOR preconditioner is a very useful preconditioner. It can be derived from the coefficient matrix with minor work and the number of iterations can be reduced to a lower order with the optimal ω value. However in practice, the spectral information required for calculating the optimal ω is prohibitively expensive to compute.

optimal ω in SSOR preconditioner is attainable according to Axelsson and Barker (1987). However in practice, the effort for calculating matrix spectrum information that is required for finding the optimal ω is even harder than solving the matrix itself. The number of iterations to converge could be reduced to a lower order (Barrett et al. 1987).

For symmetric positive definite coefficient matrix, SSOR preconditioner with parameter range in $0 < \omega < 2$ can make sure to converge (from Householder-John theorem).

2.3.2.2. Incomplete Cholesky Preconditioner

Incomplete Cholesky preconditioner cannot present in simple matrix form, the algorithm for generating it is:

For k=0 to N-1 do

$A_{k,k} = \sqrt{A_{k,k}}$;

For i=k+1 to N-1 do

$A_{i,k} = \frac{A_{i,k}}{A_{k,k}}$;

end

For j=k+1 to N-1 do

For i=j to N-1 do

If ($A_{i,j}$==0) then contineu

$A_{i,j} = A_{i,j} - A_{i,k}A_{j,k}$;

end

end

end

2.3.2.3. Incomplete LU Preconditioners

Incomplete LU (ILU) preconditioner is constructed using Incomplete LU (ILU) factorization process. The ILU factorization process computes two triangular matrices, sparse lower triangular matrix $[L]$ and sparse upper triangular matrix $[U]$, from coefficient matrix $[A]$ so that the residual matrix $[R] = [L][U] - [A]$ satisfies certain constrains(Saad 2003).

ILU (0), the Incomplete LU factorization with no fill-in, presents same zero patterns with coefficient matrix A. The algorithm for building it is:

For i=2 to N do

For k=1 to i-1 do

If $A_{i,k}$ == 0 then continue;

$A_{i,k} = \frac{A_{i,k}}{A_{k,k}}$;

For j=k+1 to N do

If $A_{i,j}$ == 0 then continue;

$A_{i,j} = A_{i,j} - A_{i,k}A_{k,j}$;

end

end

end

ILU (p), the Incomplete LU factorization with p level fill-in, keeps all fill-in elements whose level of fill does not exceed p. The algorithm for building it is:

Assume NZ is set of all nonzero elements in N*N size coefficient matrix A.

For each $A_{i,j} \in NZ,\ where\ i \in [1..N], j \in [1..N]$

$Level_{i,j} = 0$

end

For i=2 to N do

For k=1 to i-1 do

If $\text{Level}_{i,j} > p$ then continue;

$A_{i,k} = \frac{A_{i,k}}{A_{k,k}}$;

For j=1 to N do $A_{i,j} = A_{i,j} - A_{i,k}A_{k,j}$;

$\text{Level}_{i,j} = min\{\text{Level}_{i,j},\ \text{Level}_{i,k} + \text{Level}_{k,j} + 1\}$

End

For each $A_{i,j}$, where $j \in [1..N]$

If $\text{Level}_{i,j} > p$ then

$A_{i,j} = 0$

end

end

end

There are many ILU variants with different dropping strategies applied. The details can be referred to (Saad 2003).

2.3.2 SSOR Approximate Inverse Preconditioner

Normal preconditioner requires a matrix solver process (normally overhead too heavy to apply except for very simple preconditioner) or two consecutive triangular solver processes per iteration in iterative method. The solver process is intensive calculation and hard to be parallel. Another group of preconditioner called approximate inverse preconditioner has been developed motivated by multiplication of inverse matrix idea that described in Chapter 2.1.2. By approximating the inverse matrix of normal preconditioner first, the approximated inverse matrix only need an extra matrix-vector multiplication per iteration to apply. The performance improvement by this transformation is dramatic, however along with accuracy or robustness loss in the approximation process, which compromise the effect of improvement of convergence rate.

SSOR approximate inverse preconditioner is just a first order approximate inverse of SSOR preconditioner:

$$M = \overline{K}^T\overline{K}, \text{ where } \overline{K} = \sqrt{\omega(2-\omega)}D^{-1/2}I - \omega\sqrt{\omega(2-\omega)}D^{-1/2}LD^{-1}$$

Strictly speaking, this preconditioner is not approximate inverse method; it can be regards as a hybrid of ILU and polynomial preconditioning techniques. And it is also called as truncated Neumann SSOR preconditioner(Benzi and Tuma 1999).

We don't have evidence to prove it still remaining convergence even for symmetric positive definite coefficient matrix. And due to the information loss in the approximation, condition parameter ω in SSOR-AI should show less impact on convergence rate to SSOR.

2.4. Triangular Matrix Solver

Since most of the time preconditioner will be constructed as multiplication of two triangular matrices, a lower triangular matrix and an upper triangular matrix, triangular matrix solver plays a very important role for applying preconditioner.

For triangular matrix solver, the solving process will be easier to analyze. Level scheduling (Saad 2003), derived from topological sorting, has been developed to reveal parallelism of forward/backward substitution. The idea is very simple, by grouping unknowns x_i into different levels so that all unknowns in same level can be solved simultaneously. If the solving of one unknown x_i depends on results of set of other unknowns X_i, the level of x_i will be one plus maximum level of any $x_j \in X_i$.

To facilitate solving multiple right sides vectors, a two phases solver has been proposed and implemented in two phases in NVIDIA CUSPARSE library as standard Level-3 functions (Naumov 2011a). Author builds a dependency graph based on coefficient matrix's sparsity pattern and groups independent rows to levels. This dependency graph is a directed acyclic graph (DAG) that can be generated by a modified breadth-first-search (BFS) algorithm.

For example, a lower triangular matrix T involved in the $[T]\{x\} = \{b\}$

$$\begin{bmatrix} T_{1,1} & & & & & & & & \\ & T_{2,2} & & & & & & & \\ & & T_{3,3} & & & & & & \\ T_{4,1} & & & T_{4,4} & & & & & \\ T_{5,1} & & & & T_{5,5} & & & & \\ & T_{6,2} & & & & T_{6,6} & & & \\ & & T_{7,3} & & & & T_{7,7} & & \\ & & & T_{8,4} & T_{8,5} & & & T_{8,8} & \\ & & & & T_{9,5} & T_{9,6} & & & T_{9,9} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \\ b_8 \\ b_9 \end{Bmatrix}$$

Has dependency graph in Figure 2-1.

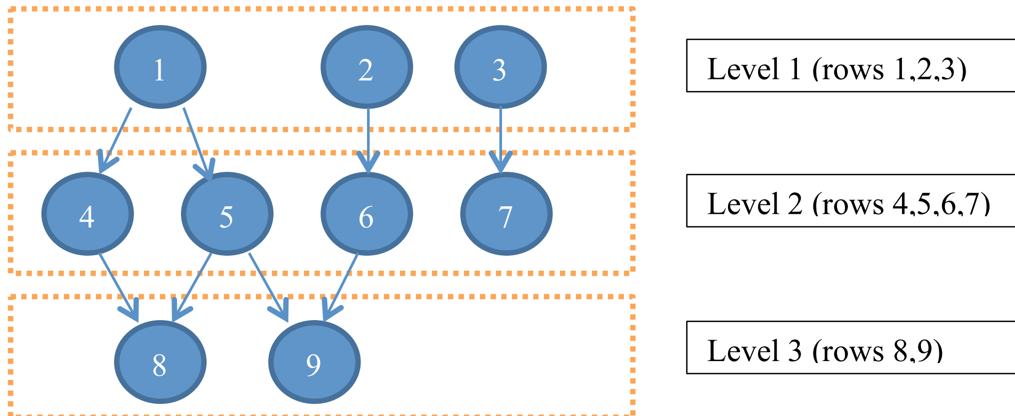

Figure 2-1 Dependency Graph Sample

From the dependency graph, initially $x_1, x_2, and\ x_3$ have no any dependent on other variants, and they can be solved immediately in parallel. And the variants that depend on the result of $x_1, x_2, and\ x_3$, that are $x_4, x_5, x_6, and\ x_7$ can be solved in parallel in next step. Finally after all dependent variants of $x_8\ and\ x_9$ are solved, $x_8\ and\ x_9$ can be solved in parallel. In other words, variants/rows are solved from lowest level to highest level; no variants/rows will be solved until all their dependent variants/row are solved; all variants/rows in same level can be solved in parallel.

2.5. GPU Computation

GPU computing is Single Instruction Multiple Data (SIMD) architecture, which executes identical operation/instruction on multiple data load simultaneously. SIMD is a light weight parallel architecture which has heavy penalties on synchronization and heterogeneous treatment on data (branch operations). I target on NVIDIA Compute Unified Device Architecture (CUDA™) for experiments.

2.5.1. Bandwidth

For supporting massive threads running at the same times with very high floating point calculation capability, graphic chips requires enormous memory bandwidth to feed data into the processors. NVIDIA Tesla M2070 provide 384-bit memory I/O interface and up to 150Gbytes per second on-board memory bandwidth. The on-board memory bandwidth capability of GPU is significant larger than contemporary CPU, PCI Express (interface from Graphic card to main board) and fabric link in cluster (InfiliBand enhanced data rate is 25.78125 Giga bits/s).

2.5.2. Architecture Design

CPU depends on sophisticated cores to achieve high calculation capability. Many advanced features such as out-of-order execution, branch prediction, help CPU successfully handle control logic and data dependency. GPU depends on massive lightweight cores and threads to accumulate high calculation capability. Without too many fancy features, fast thread switch help GPU bypass the effect of memory access stall. From specification, GPU can provide ten to thousands times floating point calculation capability to CPU. However, GPU high performance naturally can’t survive under complicated control logics and/or frequently/complicated synchronizations.

2.5.3. Program Structure

The parallel massive threads capability and bandwidth requirements dominate the programming structure design for GPU. A host(traditional CPU), and one or more devices(Graphic cards) will cooperate to achieve the GPU-enabled computation: the required data will first be moved from main memory to GPU on-board memory, and GPU starts the computation with access only to its onboard memory, then the result will

be transferred back to main memory after calculation. GPUs only take responsible for code logic that shows rich amount of data parallelism, CPUs handle the rest. Due to intuitively parallel semantics of GPU threads, any function executed in GPU, called kernels, are within specific lightweight thread context to process data slices only with synchronization barrier supported.

2.5.4. Threads System

GPU provides a two layers hierarchy thread system: Grids and Blocks. Block is composed of set of 3D indexed threads; Grid is composed of set of 3D indexed blocks. Preset variables threadIdx and blockIdx are respectively used to identify thread of a block and block of a grid; Preset variables blockDim and gridDim are respectively used to store number of threads in a block and number of blocks in a grid. Each of those four preset variables has three dimensional fields: field x, field y, and field z. All three fields in threadIdx and blockIdx are zero-based index values.

2.5.5. Multi-Kernels & Multi-Devices Support

Due to algorithm limitation, not all the kernels can consume all resource provided by GPU. CUDA does support asynchronous execution mechanism on the same GPU device, which is called stream. Kernels resided in different streams can be executed parallel in a GPU device. CUDA also provides programming supports for multiple GPU devices. However, no direct access support from one device to another.

2.5.6. CUBLAS & CUSPARSE Libraries

CUBLAS library implements BLAS (Basic Linear Algebra Subroutines) on NVIDIA CUDA runtime. CUSPARSE library implements a set of BLAS functions for sparse

matrices. Both of these two libraries are provided by NVIDA to facilitate calculation of matrices and vectors. Both of them cannot automatically parallelize across multiple GPU devices/cards.

CUBLAS library is composed of three levels of functions and some helper functions. Level-1 functions perform scalar and vector based operations; Level-2 functions perform matrix-vector operations; Level-3 functions perform matrix-matrix operations.

CUSPARSE library is also composed of three levels of functions and some helper functions. Level-1 functions perform operations between a sparse format vector and a dense format vector; Level-2 functions perform operations between a sparse format matrix and a dense format vector; Level-3 functions perform operations between a sparse format matrix and a set of dense format vectors.

Each function in three levels from two libraries support four matrices element data types: single precision, double precision, single precision complex, and double precision complex.

2.5.7. Floating Point

In engineering practice, implementation of floating point in different hardware varies a lot. And rounding approximation exists in every call for floating point calculation. Different calculation order of a same equation may produce slightly different results. Those errors may accumulate to finally compromise the entire calculation result. For example, direct method especially suffered from rounding error that is error made in one step spreads in all following steps. That is also one reason why direct method does not suit for calculation in large scale matrix.

(Whitehead and Fit-Florea 2011) shows many examples of impact of calculation in different order and in different round options/orders. For example, the following figure from Whitehead and Fit-Florea (2011) shows the (A+B)+C does equal A+(B+C) in floating point calculation.

$$
\begin{aligned}
A &= 2^1 \times 1.00000000000000000000001 \\
B &= 2^0 \times 1.00000000000000000000001 \\
C &= 2^3 \times 1.00000000000000000000001 \\
(A+B)+C &= 2^3 \times 1.01100000000000000000001011 \\
A+(B+C) &= 2^3 \times 1.01100000000000000000001011
\end{aligned}
$$

Figure 2-2 Calculation order in floating point context

A common standard IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754-1985) is widely adopted by mainstream computing systems to provide compatible behaviors among each other. (Whitehead and Fit-Florea 2011) describes the NVIDIA CUDA compliance to IEEE 754 standard.

Single precision:	1 bit [sign]	8 bits [exponent]	23 bits [fraction]
Double precision:	1 bit [sign]	11 bits [exponent]	52 bits [fraction]

The difference of single precision and double precision affect not only the accuracy but also the memory bandwidth requirements and cache locality. The application choice between double precision and single precision may somehow manipulate the performance and result. (Georgescu and Okuda 2007) shows “single precision GPUs make very good accelerators for the CG method if and only if the matrix is well behaved, meaning that its condition number, computed after preconditioning, should be below 10^5. For ill-conditioned matrices, in the current solver setting, double precision is necessary.”

IEEE 754 defines four rounding modes: round towards negative, round towards positive, round towards zero and round-to-nearest. CUDA explicitly supports all four modes in its basic floating point functions. However IEEE 754 standard does not require all common mathematical functions to be correctly rounded. For example, denormal numbers, small numbers close to zero, will be flushed to zero in single precision floating point from CUDA compute capability 1.2 and below.

Fused multiply-add operation (FMA) was included in 2008 of the IEEE 754 standards to increase the accuracy for (X * Y + Z). The two round steps normally will be applied in this operation: result rounding of multiplication and result rounding of addition. Only one round operation will be applied to this FMA operation. In some extreme case, computing one multiply followed by one addition may lose all bits of precision. The following example from Whitehead and Fit-Florea (2011) shows the case in single precision:

$$
\begin{aligned}
A &= \quad 2^{0} \times 1.00000000000000000000001 \\
B &= -2^{0} \times 1.00000000000000000000010 \\
\mathrm{rn}(A \times A + B) &= 2^{-46} \times 1.00000000000000000000000 \\
\mathrm{rn}(\mathrm{rn}(A \times A) + B) &= 0
\end{aligned}
$$

Figure 2-3 FMA extreme case

2.6. Related Works

The University of Florida sparse matrix collection (Davis and Hu 2011) is widely used as matrices source for analysis (Georgescu and Okuda 2007; Naumov 2011b; Li and Saad 2013). There are 231 symmetric positive definite matrices in the collection, 125 of them come from structural problems.

The CG method was introduced by Hestenes and Stiefel (1952). Superlinear phenomenon was first observed by Concus and Golub (1976). Van der Sluis and van der Vorst (1986)

proved the condition when the superlinear phenomenon will show up. Many preconditioners have been analyzed in previous research. Preconditioned CG method has been proven effective by Knyazev and Lashuk (2007). The Jacobi preconditioner for CG method has been studied by Georgescu and Okuda (2007). The research proved that the Jacobi preconditioner could not work for most of matrices from the University of Florida sparse matrix collection (Davis and Hu 2011). Gui and Zhang (2012) solved the Incomplete Cholesky preconditioner for CG with parallelized Jacobi iterative methods and applied a new sparse storage format. However, the Preconditioned CG method outperformed the CG method for only the smallest matrices in the eight matrices from Poisson equations; in fact, even the iterations for convergence of PCG are less than ¼th iterations of CG. The overhead cost of applying Incomplete Cholesky preconditioner is still too high. Li and Saad (2013) compared multiple preconditioners and sparse storage format. Their recent Journal of Supercomputing article stated that the sparse triangular solver in GPU can only attain a very low speed up to CPU, and may be even lower than serial implementation in CPU. The overall performance speed up for Incomplete Cholesky and ILU preconditioner in GPU can only outperform CPU implementation for up to 3 and 4 times respectively.

Applying approximate inverse of a normal preconditioner is a tradeoff between performance gain and cost overhead of preconditioner. Benzi and Tuma (1999) summarized all approximate inverse techniques stating that an SSOR approximate inverse preconditioner cannot be better than SSOR preconditioner and none-diagonal dominant coefficient matrices will suffer from degradation of rate of convergence due to robustness loss in the first or second order of approximation. Ament et al. (2010) derived

a SSOR based heuristic approximate inverse preconditioner from regular grid (finite difference method) called Incomplete Poisson Preconditioner $M^{-1} = (I - LD^{-1})(I - D^{-1}L^T)$. This preconditioner is only designed for solving Poisson equations. Helfenstein and Koko (2011) determined that an SSOR approximate inverse preconditioner is derived by using first order approximation of SSOR and is for more general usage.

Not much effort was put on general preconditioners for a wide range of data. Domain-specific preconditioners showed stable relative improvements in the following research.

Georgescu and Okuda (2007) tested the CG with Jacobi preconditioner on all symmetric positive definite matrices with more than 10,000 rows from the University of Florida sparse matrix collection (Davis and Hu 2011). And though an average 3-5x speed up has been achieved, the solver does not work for majority of the test matrices.

Yu et al. (2012) tested the GMRES GPU solver with block ILU (0) preconditioner in nonlinear simulation for 100 days. From the comparison in this study, solver/preconditioner with a higher speed up definitely does not mean the calculation time would also be better. The 10x speed up is just comparison to serial implementation of CPU. And it should be highlighted that the relative tolerance for the experiments (2 million square matrices) is too low, only 1e-3. Comparing to mostly 1e-6 of other research, the low relative tolerance cannot provide convincing results.

Chou et al. (2011) developed a domain-specific preconditioner, SEVA, for power grid simulation and achieved a 43% iterations reduction and 23% speed up over CG without preconditioner and CG methods with universal preconditioners Jacobi and ILU respectively.

Approximate inverse preconditioners almost have the least application overhead as they only need an extra matrix-vector multiplication instead of solving two triangular matrices. SSOR has proven very effective on convergence improvement. A combination of SSOR preconditioner and approximate inverse technology, that is SSOR approximate inverse preconditioner, can be expected to operate efficiently and effectively. The only problem is the robustness loss introduced in the approximation process.

CHAPTER 3. METHODOLOGY

3.1 Implementation

I implemented Conjugate Gradient iterative solver with SSOR approximate inverse preconditioner using CUBLAS and CUSPARSE. Based on commercial linear algebra libraries on GPU, I can show the results with only general tuning for no preset knowledge of input matrices.

I conducted the experiments in a GPU cluster node. The Linux cluster node is built with Shared Memory Processor (SMP) architecture equipped with twelve Intel Xeon X5650 @ 2.67 GHz CPUs, 48Gbytes memory and two NVIDIA Tesla M2070 448 Cores 1.15GHz GPU cards with 6Gbytes GDDR5 onboard memory.

3.2 Convergence Impact by Condition Parameter ω in SSOR-AI

To answer the first and second research questions in chapter 1, I used eight symmetric positive definite matrices. The first seven of them are from the University of Florida sparse matrix collection (Davis and Hu 2011), which were chosen in the analysis of paper by Naumov (2011b); the eighth matrix comes from Helfenstein and Koko (2011). Those eight matrices cover matrices encountered in real world practice from many application fields and show very unique sparse patterns.

The eight matrices in Table 3-1 come from real world problems in different application fields, which represent matrices' requirements of different fields. And the non-zero element distribution patterns that come from the same application fields are dramatically different. Figures from Figure 3-1 to Figure 3-8, respectively show their non-zero element distribution patterns (Davis and Hu 2011).

Table 3-1 Description of Matrices

#	Matrix	Rows/Cols	NNZ	Application Field
1	offshore	259,789	4,242,673	electromagnetics problem
2	af_shell3	504,855	17,562,051	subsequent structural problem
3	parabolic_fem	525,825	3,674,625	computational fluid dynamics problem
4	apache2	715,176	4,817,870	structural problem
5	ecology2	999,999	4,995,991	landscape ecology problem
6	thermal2	1,228,045	8,580,313	steady state thermal problem
7	G3_circuit	1,585,478	7,660,826	circuit simulation
8	Poisson	130,009	908,357	computational fluid dynamics problem

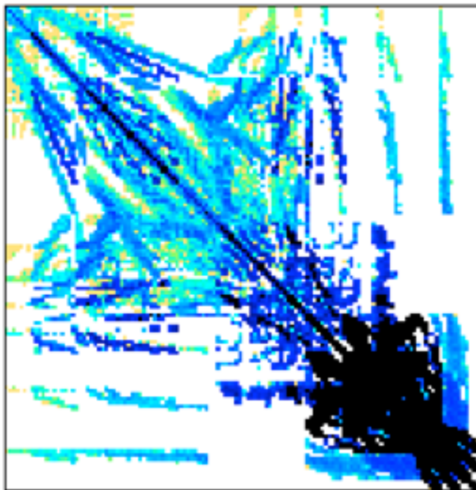

Figure 3-1 Sparse pattern of offshore

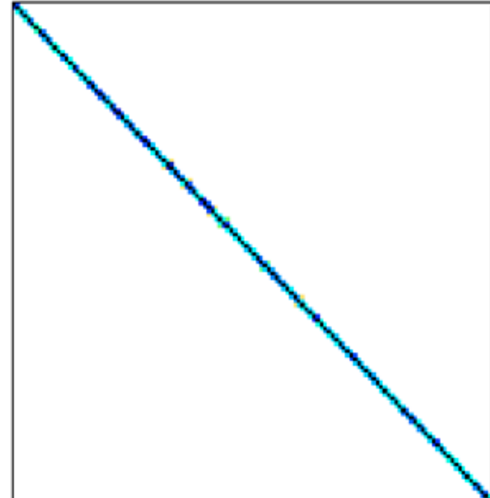

Figure 3-2 Sparse pattern of af_shell3

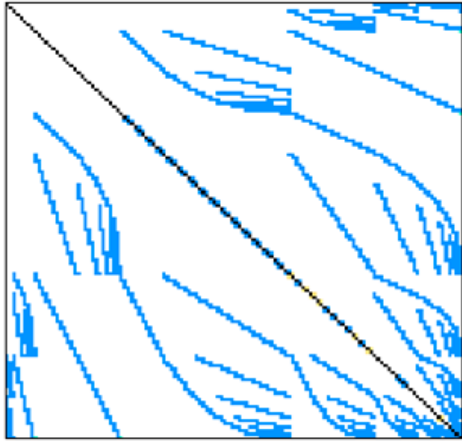

Figure 3-3 Sparse pattern of Parabolic_fem

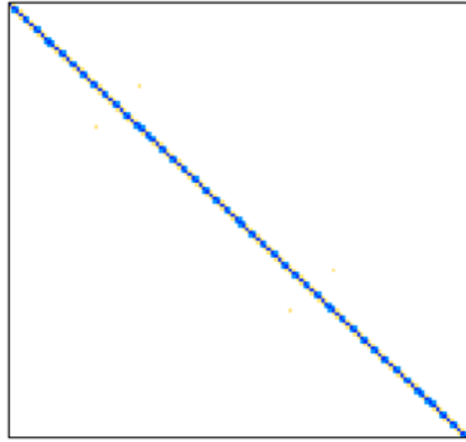

Figure 3-4 Sparse pattern of Apache2

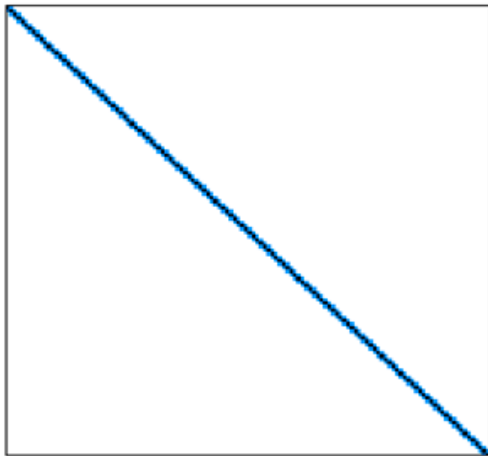

Figure 3-5 Sparse pattern of ecology2

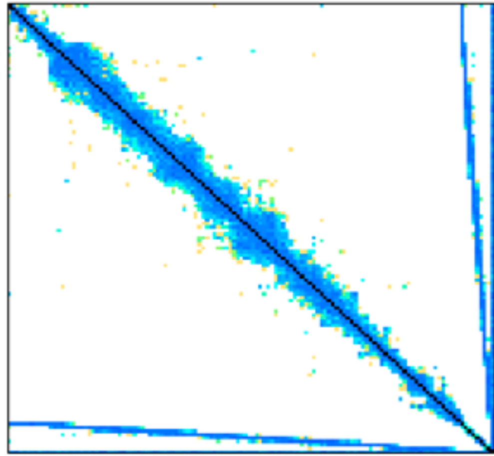

Figure 3-6 Sparse pattern of thermal2

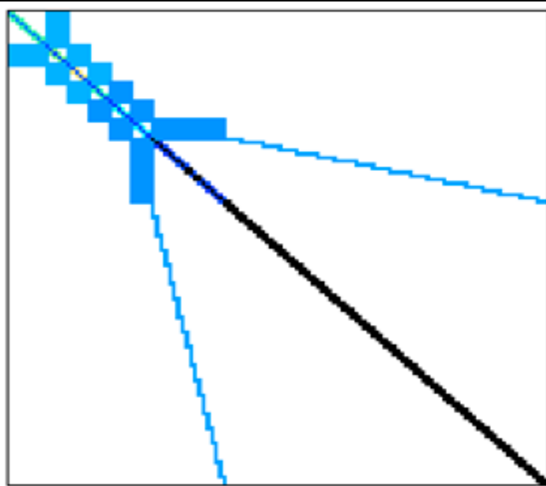

Figure 3-7 Sparse pattern of G3_circuit

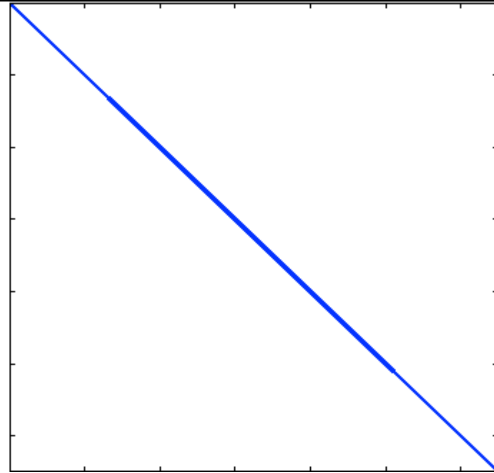

Figure 3-8 Sparse pattern of Poisson

The sparse pattern in [2][4][5][8] in coarse scale looks similar, however in fine scale it still shows different patterns. The following figures show sparse patterns of top left 100*100 and 500*500 contents of those matrices.

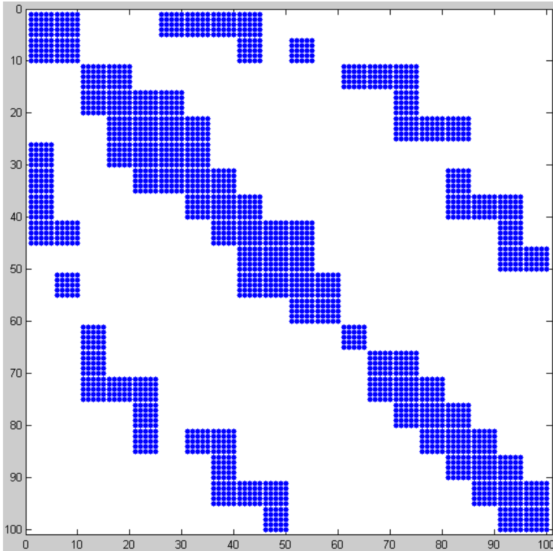

Figure 3-9 Af_shell3 (1:100,1:100) pattern

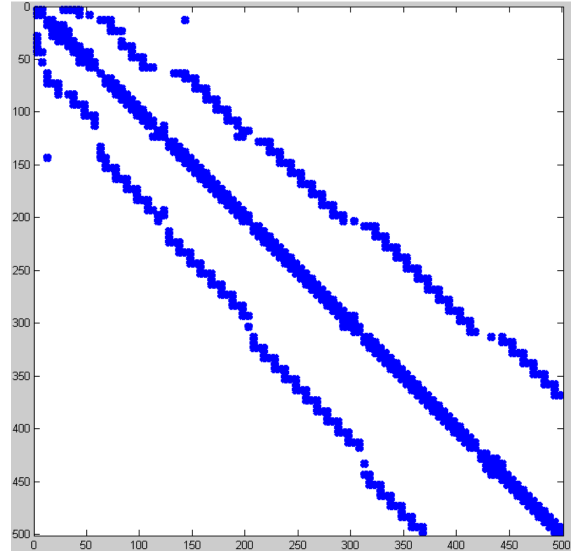

Figure 3-10 Af_shell3 (1:500,1:500) pattern

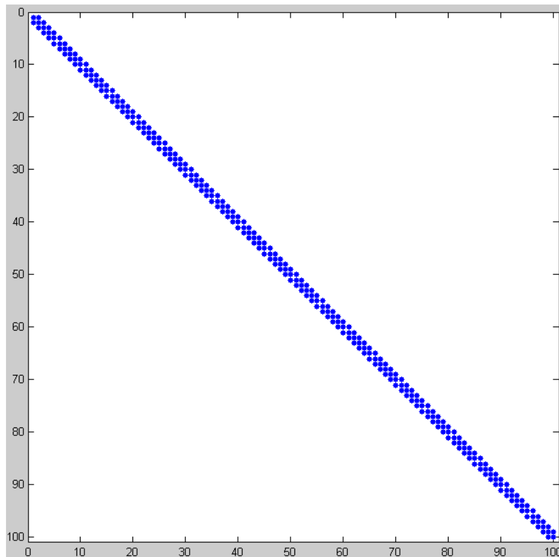

Figure 3-11 Apache2 (1:100,1:100) pattern

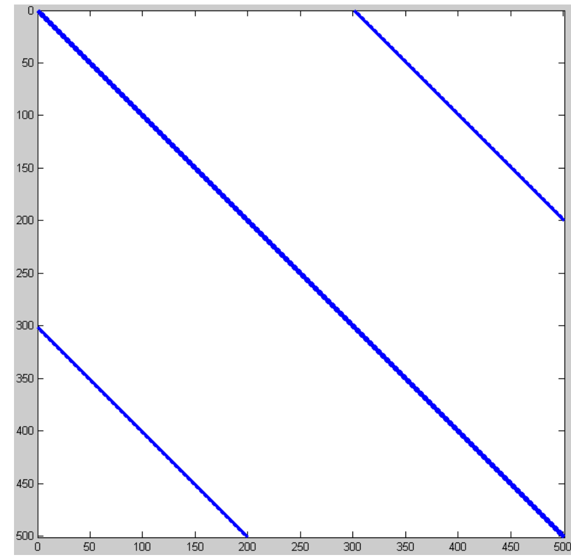

Figure 3-12 Apache2 (1:500,1:500) pattern

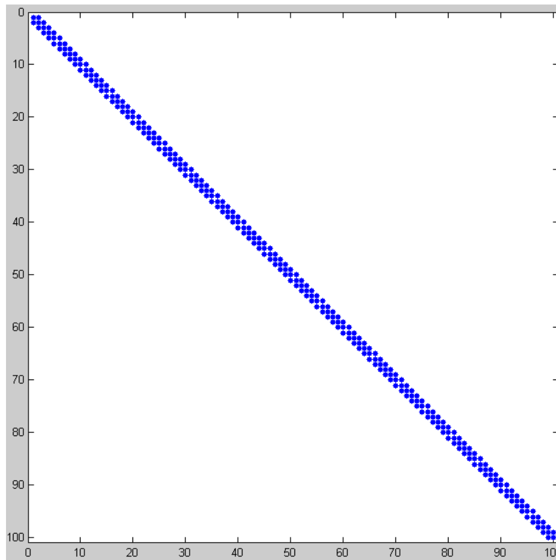

Figure 3-13 Ecology2 (1:100,1:100) pattern

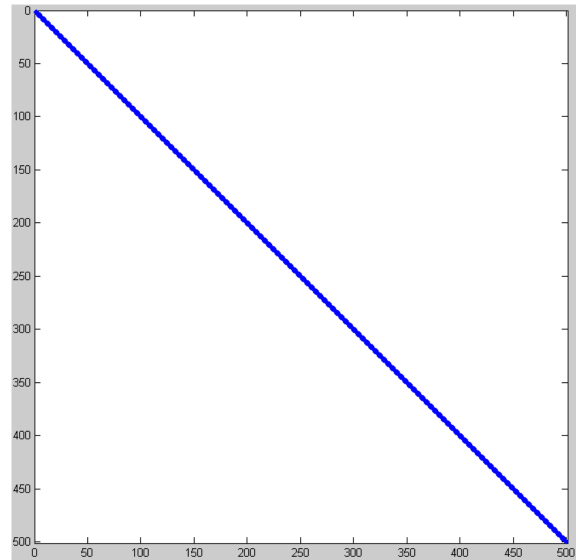

Figure 3-14 Ecology2 (1:500,1:500) pattern

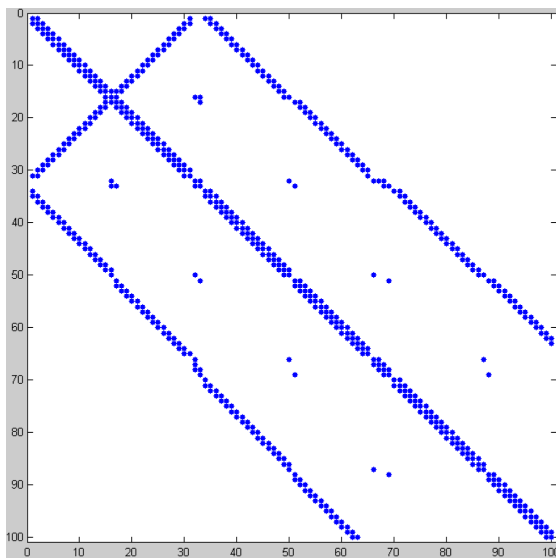

Figure 3-15 Poisson (1:100,1:100) pattern

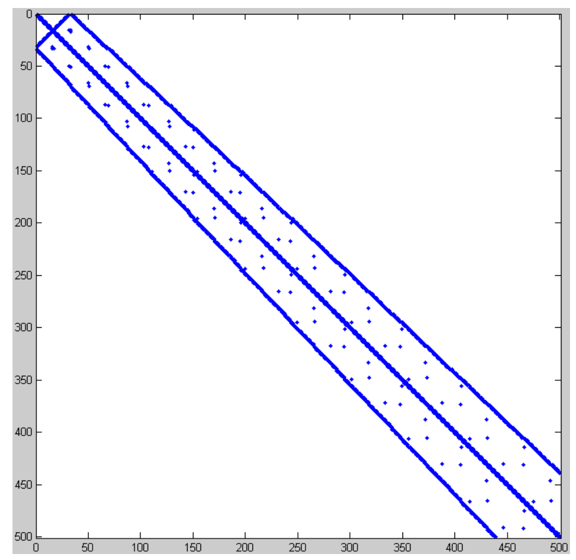

Figure 3-16 Poisson (1:500,1:500) pattern

I ran Conjugate Gradient iterative method with SSOR-AI preconditioner in GPU for all the choice of ω values in range $[0.01,1.99]$ with a 0.01 interval, which included 199 samples per matrix. The stop criteria were either error tolerance reached to 1e-7 or maximum of 20,000 iterations reached. And for each sample, I collected three data:

number of iterations for convergence, execution time for iterative method, and time for data transfer from CPU to GPU.

CHAPTER 4. RESULTS AND DISCUSSIONS

4.1. Convergence Reports for SSOR-AI

Figures from Figure 4-1to Figure 4-8 are the results that reveal the relationship of number of iterations to converge and ω values in SSOR approximate inverse preconditioner.

Figure 4-1 Offshore convergence report

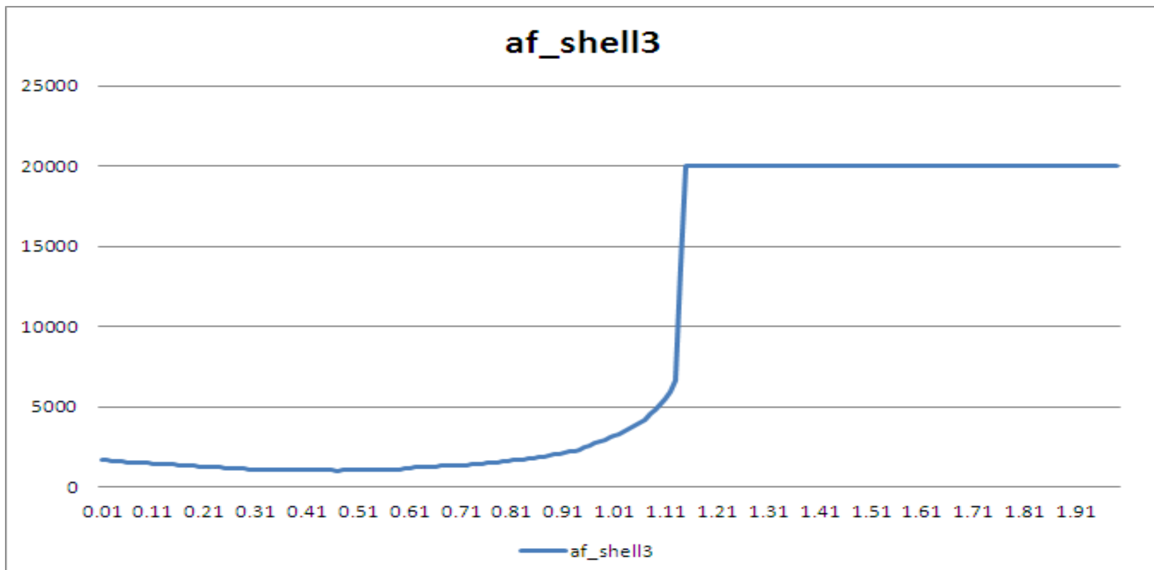

Figure 4-2 Af_shell3 convergence report

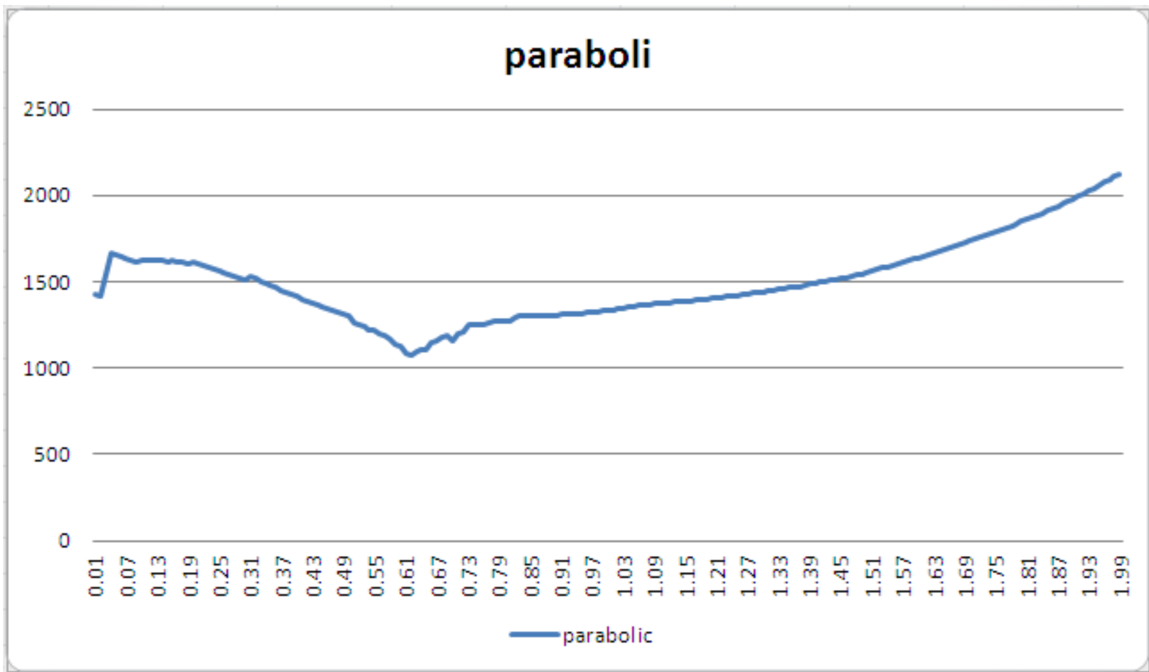

Figure 4-3 Parabolic_fem convergence report

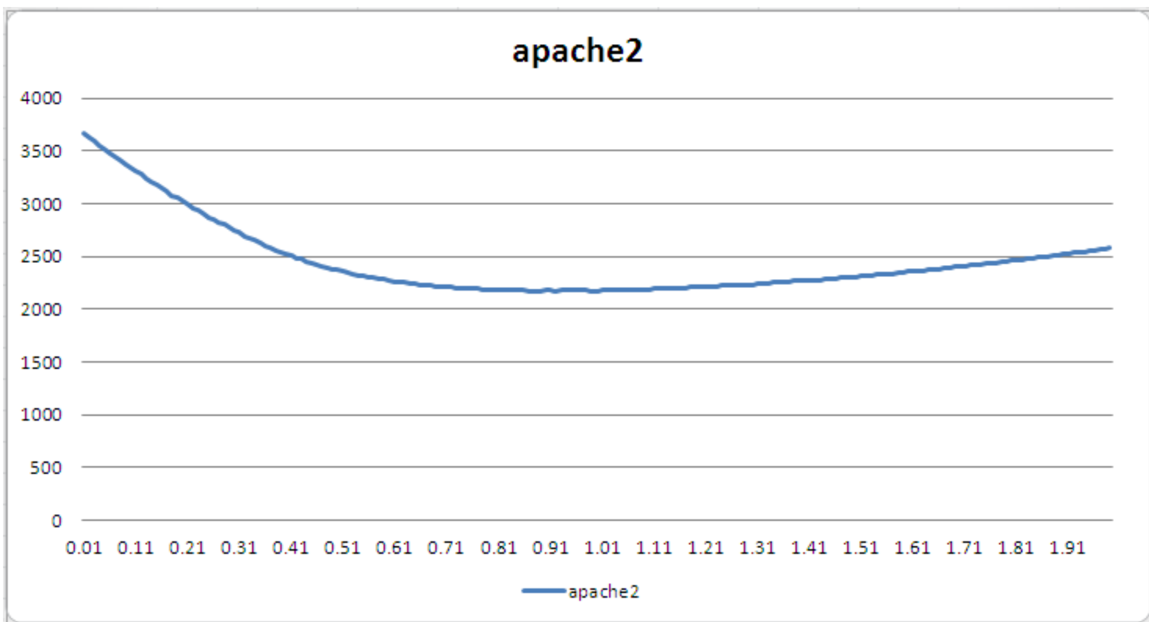

Figure 4-4 Apache2 convergence report

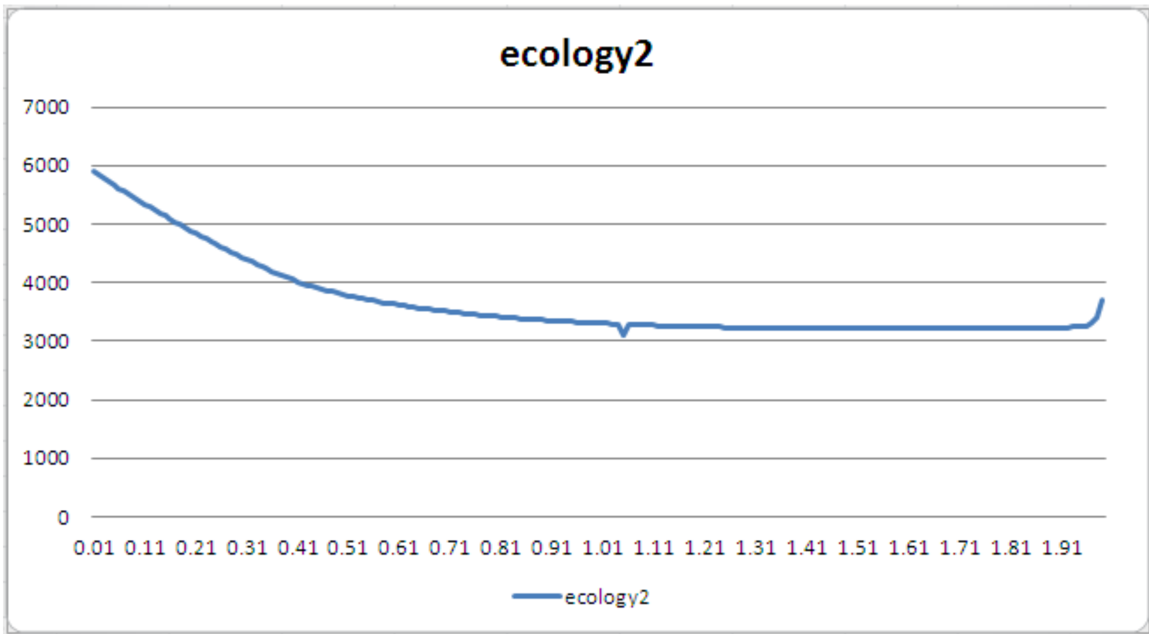

Figure 4-5 Ecology2 convergence report

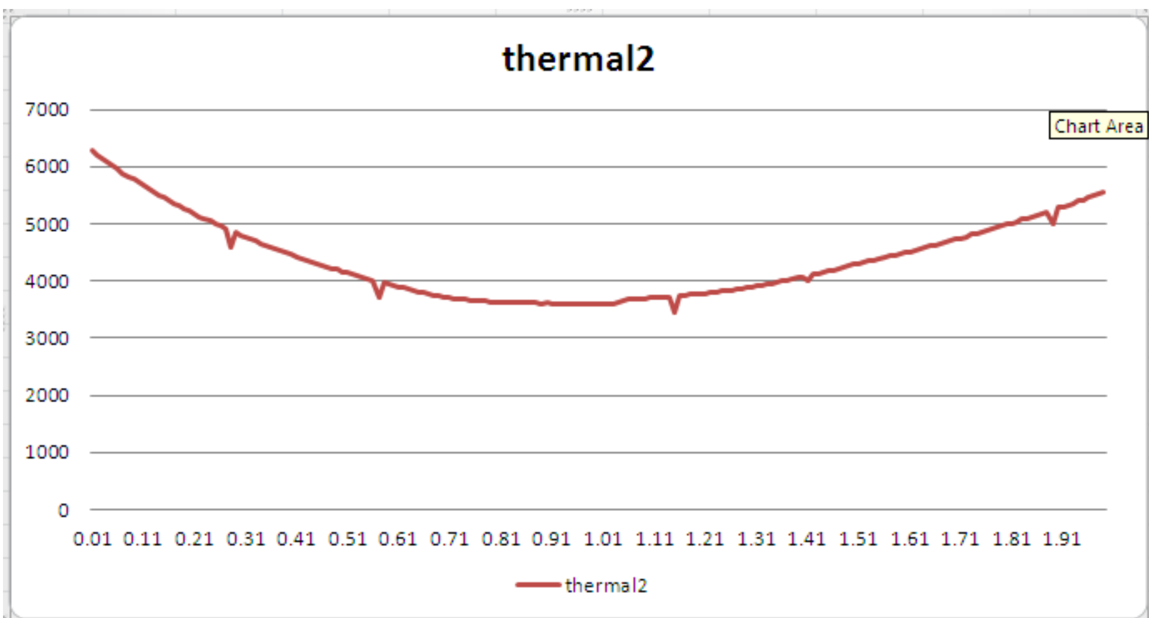

Figure 4-6 Thermal2 convergence report

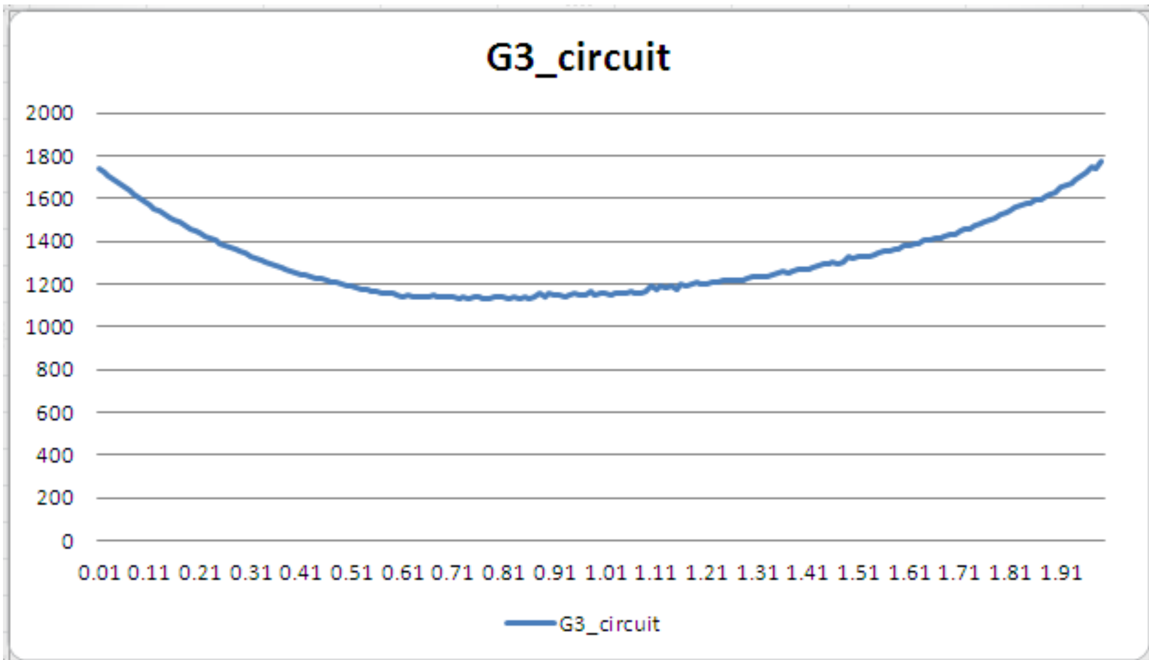

Figure 4-7 G3_circuit convergence report

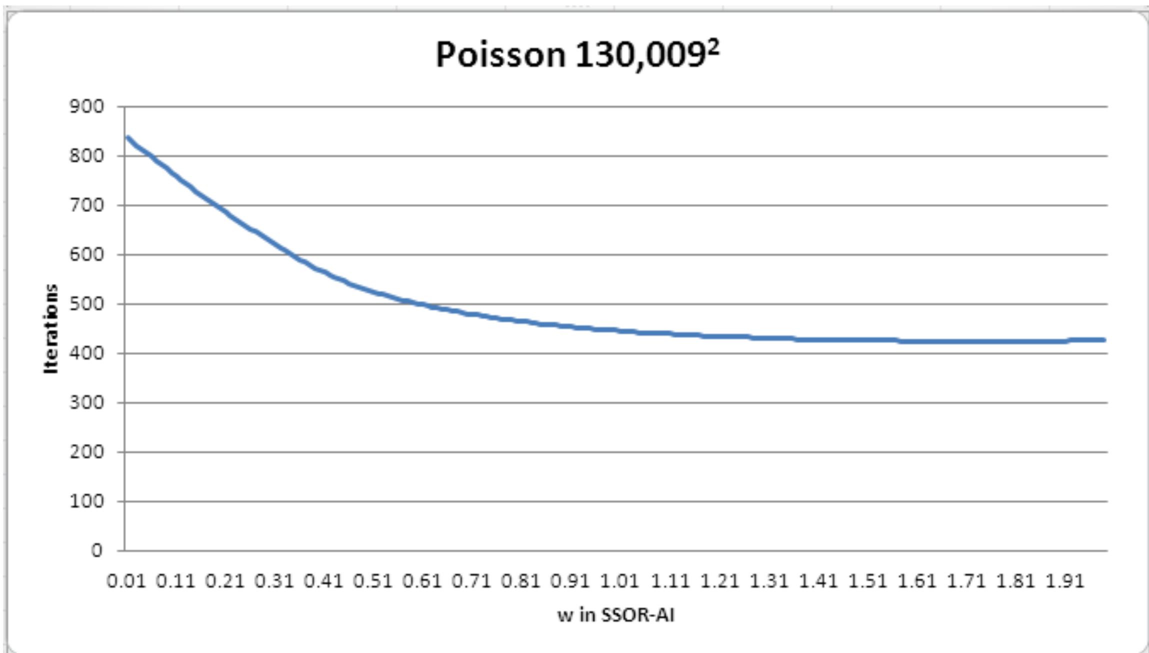

Figure 4-8 Poisson convergence report

From the figures above, I found out that bad convergence behavior shows for some ω values. In other words, the convergence rate will deteriorate for some ω values. And

those ω values are grouped in an area instead of shown as isolated points on the axle. Offshore and af_shell3 both show these behaviors.

Also from the data, the convergence rate (or number of iterations needed for convergence) does show very clear trends to the change of ω values. The change rate of convergence rate, or slope of convergence rate to ω, can be considered as continuous most of the time. Only thermal2 show some spike data.

The number of iterations' difference for optimal ω and worst ω are huge enough, that at least 50% iterations for worst cases can be saved for optimal cases.

The ω value range with minor convergence difference to optimal convergence is grouped around the optimal ω value and the area is large enough to be easy identified. Hitting the suboptimal range is relatively easy in 0.01 unit.

I cannot reproduce the experiments in exact same environment settings with Naumov (2011b). Those GPU settings in my experiments are almost equivalent with GPU settings in Naumov (2011b). And I use CPU construction time for ILU (0) preconditioner in Naumov (2011b) as approximation of CPU construction time for SSOR-AI preconditioner.

Table 4-1 shows execution time of SSOR-AI ($\omega = 1.0$) compared to the minimum execution time of all preconditioners tested in the study by Naumov (2011b). The execution time for GPU implementation includes construction time for preconditioner on CPU, copy time for preconditioner from CPU to GPU, and solving time on GPU. The execution time for CPU implementation includes construction time for preconditioner on CPU and solving time on CPU.

Compared to Naumov (2011b), my implementation of CG with SSOR-AI preconditioner can almost defeat any preconditioner settings on GPU with the simple set $\omega = 1.0$. And my implementation of SSOR-AI preconditioner can defeat all preconditioner settings on CPU. The average speed up to best execution time of CPU is 2.05x. The data proved the SSOR-AI preconditioner can provide very stable impressive performance improvement with low overhead for applying it.

Table 4-1 Execution time comparison of SSOR-AI (1.0) and other preconditioners

Name	CPU Execution Time Minimum (seconds)	GPU Execution Time Minimum (seconds)	GPU Execution Time SSOR-AI (ω =1.0)
offshore	1.1	1.92	0.846
af_shell3	40.12	34.14	26.417
parabolic_fem	12.91	7.05	4.902
apache2	23.82	12.93	8.706
ecology2	29.55	55.4	16.238
thermal2	50.59	54.82	18.863
G3_circuit	14.79	8.79	9.048

However, the default choice $\omega = 1.0$ for SSOR approximate inverse preconditioner may not be a good idea for general usage. The af_shell3 and offshore reveals a potential risk for convergence a breakdown with ω less than 1.0.

Two patterns exists for relationship of convergence and ω. One pattern is a U style pattern where the optimal ω is in the middle of range (0, 2) and the iterations for convergence in approximate optimal ω value are almost half the value of the worst ω choice iterations. The other pattern is an ascendant style pattern where the optimal ω is located near the leftmost extreme of range (0, 2). In this pattern, the iterations for

convergence in approximate optimal ω value are only 1/10 to 1/1000 of the worst ω choice iterations.

4.2. Wide Range Experiments

I ran all positive definite matrices for all structural problems in the University of Florida sparse matrix collection (Davis and Hu 2011). A total 125 matrices were tested. CG method and CG with SSOR-AI preconditioner with ω = [0.01, 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 1.99] were tested. The stop criteria were either a maximum 500,000 iterations reached or relative residual reached to 1e-7. Matrices “x104” and “thread” from same group “DNVS” will not converge using CG without a preconditioner or CG with SSOR-AI preconditioners, so these two matrices were excluded from the report. Table 4-2 is the convergence report for these 10 method configurations. It is easy to see that when the SSOR approximate inverse preconditioner’s condition parameter ω equals 0.01, which is the best setting among all 123 tests. Instead of the commonly used condition parameter ω=1.0, an extreme minimum ω value closer to the lower bound of range (0, 2) showed very good performance among a wide range of data sets and effectively prevented most of the convergence failures.

Table 4-2 Convergence comparison for all CG with SSOR-AI preconditioners

Method Configuration	# of failing to converge	# of reach min iterations
CG	8	7
CG+SSOR-AI(0.01)	1	75
CG+SSOR-AI(0.25)	16	35
CG+SSOR-AI(0.50)	23	29

CG+SSOR-AI(0.75)	34	31
CG+SSOR-AI(1.00)	43	24
CG+SSOR-AI(1.25)	50	17
CG+SSOR-AI(1.50)	51	16
CG+SSOR-AI(1.75)	59	16
CG+SSOR-AI(1.99)	62	16

Figure 4-9 presents the iterations relationship of CG and CG with SSOR-AI (0.01) preconditioner. The difference of number of iterations for each sample is normalized by larger number of iterations in the comparison of CG and CG with SSOR-AI preconditioner, so that the percentage of iteration reduction maps to (-100%, 100%). The detailed reports are put in Appendix A. For all tests, 90.9% tested matrices had better convergence with SSOR approximate inverse (ω =0.01) preconditioner when applied and an average 57.6% iterations reduction was achieved from all test matrices. For those test matrices with positive iteration reduction value, average 67.9% iteration reduce was achieved. An average 12.6x speed up can be achieved for CG with SSOR-AI (ω =0.01) preconditioner over CG performance without preconditioner among all test matrices.

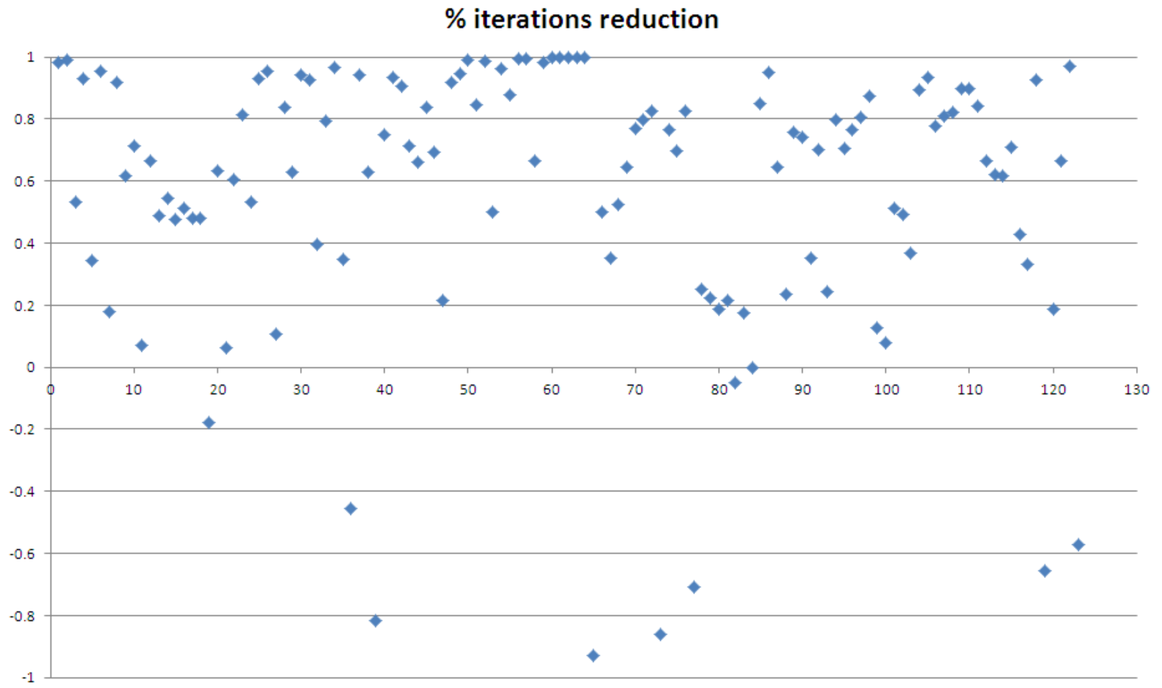

Figure 4-9 Percentage of iterations reduction for CG with SSOR-AI (0.01) to CG

CHAPTER 5. CONCLUSION

It was amazing to observe that SSOR approximate inverse preconditioner works well for general usage in solving structural problems on an extreme minimal ω value, which is close to a lower bound of range (0, 2). The risk of convergence failure increased with increment of ω value. The experiments on 125 matrices from structural problems showed that applying SSOR approximate inverse preconditioner with $\omega = 0.01$ reduced the convergence failure and obtained the best convergence behaviors. The report also showed where 90.9% of the test matrices had better convergence with SSOR approximate inverse ($\omega = 0.01$) preconditioner applied and an average 67.9% iterations reduction was achieved from those positive cases. An average 12.6x speed up was achieved for CG with SSOR-AI ($\omega = 0.01$) preconditioner when compared to CG without a preconditioner.

With the extreme minimal ω value, applying SSOR approximate inverse preconditioner showed very stable performance improvement and prevented most of the robust loss issues introduced by approximation of inverse preconditioner.

APPENDIX A. REPORTS IN DETAIL

Appendix A-1. Description of 125 Tested Matrices and Execution Time

Table A-1 Description of tested matrices and execution time

Name	Group	rows/cols	nonzeros	CG (ms)	SSOR-AI (ω =0.01) (ms)
Emilia_939	Janna	923,136	40,373,538	932,624.00	154,408.00
Fault_639		638,802	27,245,944	649,916.00	63,224.20
Flan_1565		1,564,794	114,165,372	212,482.00	197,786.00
Geo_1438		1,437,960	60,236,322	235,213.00	33,213.90
Hook_1498		1,498,023	59,374,451	102,279.00	132,079.00
Serena		1,391,349	64,131,971	245,220.00	21,877.40
apache2	GHS_psdef	715,176	4,817,870	10,766.40	14,380.60
audikw_1		943,695	77,651,847	1,100,240.00	179,709.00
inline_1		503,712	36,816,170	607,391.00	460,681.00
ldoor		952,203	42,493,817	146,561.00	82,885.20
Kuu	MathWorks	7,102	340,200	222.03	292.21
Muu		7,102	170,134	28.79	18.39
af_0_k101	Schenk_AFE	503,625	17,550,675	86,021.60	84,727.80
af_1_k101		503,625	17,550,675	93,619.30	81,683.40
af_2_k101		503,625	17,550,675	69,447.80	69,938.40
af_3_k101		503,625	17,550,675	39,255.40	36,887.20
af_4_k101		503,625	17,550,675	70,608.00	70,378.10
af_5_k101		503,625	17,550,675	67,370.90	67,364.20
apache1	GHS_psdef	80,800	542,184	822.18	1,476.38
bcsstk01	HB	48	400	46.22	23.11
bcsstk02		66	4,356	19.65	23.04
bcsstk03		112	640	106.78	53.38
bcsstk04		132	3,648	130.99	36.12
bcsstk05		153	2,423	100.36	61.23
bcsstk06		420	7,860	903.92	84.59
bcsstk08		1,074	12,960	841.35	57.45
bcsstk09		1,083	18,437	70.77	76.98
bcsstk10		1,086	22,070	808.63	163.07
bcsstk11		1,473	34,241	944.53	449.81
bcsstk14		1,806	63,454	1,479.11	116.62

bcsstk15		3,948	117,816	2,257.90	220.28
bcsstk16		4,884	290,378	115.51	99.27
bcsstk17		10,974	428,650	4,850.44	1,471.53
bcsstk18		11,948	149,090	6,750.03	312.94
bcsstk19		817	6,853	510.80	393.18
bcsstk20		485	3,135	86.52	188.36
bcsstk21		3,600	26,600	1,652.66	126.41
bcsstk22		138	696	113.71	54.90
bcsstk23		3,134	45,178	654.35	4,537.62
bcsstk24		3,562	159,910	2,273.14	757.14
bcsstk25		15,439	252,241	5,876.04	547.27
bcsstk26		1,922	30,336	3,720.88	442.80
bcsstk27		1,224	56,126	252.06	98.61
bcsstk28		4,410	219,024	5,156.26	2,366.56
bcsstk34	Boeing	588	21,418	209.76	48.43
bcsstk36		23,052	1,143,140	291,432.00	143,855.00
bcsstk38		8,032	355,460	1,981.24	2,329.15
bcsstm02	HB	66	66	5.69	2.26
bcsstm05		153	153	12.21	6.82
bcsstm06		420	420	44.10	9.97
bcsstm07		420	7,252	88.09	22.38
bcsstm08		1,074	1,074	31.00	9.84
bcsstm09		1,083	1,083	9.53	9.76
bcsstm11		1,473	1,473	18.23	9.76
bcsstm12		1,473	19,659	1,087.48	169.59
bcsstm19		817	817	99.63	5.67
bcsstm20		485	485	79.67	9.79
bcsstm21		3,600	3,600	7.43	6.99
bcsstm22		138	138	23.95	6.80
bcsstm23		3,134	3,134	1,308.53	6.66
bcsstm24		3,562	3,562	3,842.09	6.95
bcsstm25		15,439	15,439	2,114.48	7.70
bcsstm26		1,922	1,922	837.74	9.75
bcsstm39	Boeing	46,772	46,772	133.32	12.39
bmw7st_1	GHS_psdef	141,347	7,318,399	1,379.86	34,953.20
bmwcra_1		148,770	10,641,602	24,893.20	24,314.00
bodyy4	Pothen	17,546	121,550	57.85	51.12
bodyy5		18,589	128,853	135.75	88.85
bodyy6		19,366	134,208	305.96	149.61
cbuckle	TKK	13,681	676,515	1,422.49	547.02
crankseg_1	GHS_psdef	52,804	10,614,210	4,857.00	1,964.24

crankseg_2		63,838	14,148,858	8,121.08	2,844.22
ct20stif	Boeing	52,329	2,600,295	5,740.07	73,150.30
hood	GHS_psdef	220,542	9,895,422	26,354.00	11,870.70
lund_a	HB	147	2,449	110.48	43.07
lund_b		147	2,441	126.48	29.82
m_t1	DNVS	97,578	9,753,570	149,628.00	997,938.00
mesh1e1	Pothen	48	306	9.48	8.72
mesh1em1		48	306	13.33	13.26
mesh1em6		48	306	9.07	9.25
mesh2e1		306	2,018	31.15	29.83
mesh2em5		306	2,018	30.59	31.17
mesh3e1		289	1,377	14.20	13.82
mesh3em5		289	1,377	13.18	12.87
msc00726	Boeing	726	34,518	275.77	57.35
msc01050		1,050	26,198	7,376.73	484.92
msc01440		1,440	44,998	417.70	188.70
msc04515		4,515	97,707	1,487.28	1,388.02
msc10848		10,848	1,229,776	4,450.60	1,850.33
msc23052		23,052	1,142,686	299,498.00	124,719.00
msdoor	INPRO	415,863	19,173,163	191,588.00	237,997.00
nasa1824	nasa	1,824	39,208	1,525.69	598.28
nasa2146		2,146	72,250	103.11	98.55
nasa2910		2,910	174,296	2,486.12	712.06
nasa4704		4,704	104,756	5,675.62	2,205.69
nasasrb		54,870	2,677,324	23,045.90	9,470.15
nos1	HB	237	1,017	686.47	170.06
nos2		957	4,137	9,702.24	1,452.95
nos3		960	15,844	91.66	94.28
nos4		100	594	30.58	34.27
nos5		468	5,172	145.41	83.14
oilpan	GHS_psdef	73,752	2,148,558	28,315.50	24,384.80
olafu	Simon	16,146	1,015,156	17,846.90	17,632.20
plbuckle	TKK	1,282	30,644	645.89	86.00
pwtk	Boeing	217,918	11,524,432	1,549,860.00	186,571.00
raefsky4	Simon	19,779	1,316,789	563.86	216.64
s1rmq4m1	Cylshell	5,489	262,411	1,379.77	353.54
s1rmt3m1		5,489	217,651	1,617.91	388.86
s2rmq4m1		5,489	263,351	6,169.27	867.26
s2rmt3m1		5,489	217,681	8,508.76	1,156.27
s3dkq4m2	GHS_psdef	90,449	4,427,725	668,155.00	188,462.00
s3dkt3m2		90,449	3,686,223	558,995.00	331,401.00

s3rmq4m1	Cylshell	5,489	262,943	24,709.30	11,712.40
s3rmt3m1		5,489	217,669	33,483.40	16,997.90
s3rmt3m3		5,357	207,123	52,018.90	19,536.40
ship_001	DNVS	34,920	3,896,496	36,991.00	39,336.60
ship_003		121,728	3,777,036	635,500.00	752,947.00
shipsec1		140,874	3,568,176	219,107.00	27,991.60
shipsec5		179,860	4,598,604	259,165.00	1,341,540.00
shipsec8		114,919	3,303,553	158,141.00	227,646.00
smt	TKK	25,710	3,749,582	9,807.78	5,910.60
sts4098	Cannizzo	4,098	72,356	4,198.82	183.63
vanbody	GHS_psdef	47,072	2,329,056	8,713.63	35,287.80

Appendix A-2. Number of Iterations of all 125 Matrices in 10 Configurations

Table A-2 Number of Iterations

Name	CG	ω(0.01)	ω(0.25)	ω(0.5)	ω(0.75)	ω(1)	ω(1.25)	ω(1.5)	ω(1.75)	ω(1.99)
Emilia_939	∞	8,501	5,492	4,764	5,241	∞	∞	∞	∞	∞
Fault_639	∞	5,074	3,260	3,410	11,580	∞	∞	∞	∞	∞
Flan_1565	8,960	4,196	3,121	3,246	∞	∞	∞	∞	∞	∞
Geo_1438	16,730	1,194	854	702	669	721	1,017	18,161	∞	∞
Hook_1498	6,958	4,553	3,233	2,674	2,910	3,494	4,127	5,177	7,380	30,778
Serena	16,335	724	529	651	1,148	2,430	6,328	13,286	∞	∞
apache2	4,453	3,662	2,874	2,370	2,198	2,176	2,223	2,309	2,432	2,579
audikw_1	65,164	5,350	18,410	136,720	∞	∞	∞	∞	∞	∞
inline_1	70,735	27,179	20,267	∞	∞	∞	∞	∞	∞	∞
ldoor	14,260	4,095	3,488	∞	∞	∞	∞	∞	∞	∞
Kuu	531	493	394	339	353	538	4,141	106,178	∞	∞
Muu	57	19	15	14	17	27	45	90	566	5,248
af_0_k101	20,626	10,570	8,334	6,653	6,388	6,545	7,250	8,205	9,431	11,513
af_1_k101	22,466	10,190	8,049	6,567	6,270	6,631	7,448	8,213	9,307	11,407
af_2_k101	16,650	8,719	5,962	4,594	4,351	4,584	5,540	6,122	6,833	9,441
af_3_k101	9,391	4,590	3,517	2,861	2,686	2,699	2,890	3,314	5,507	7,685
af_4_k101	16,928	8,775	6,877	5,346	4,942	5,254	5,600	6,128	6,535	5,214
af_5_k101	16,147	8,397	6,506	5,276	4,855	5,042	5,392	5,771	6,691	5,718
apache1	1,553	1,889	1,789	1,521	1,428	1,412	1,475	1,521	2,006	2,179
bcsstk01	125	46	71	90	110	137	163	186	208	232
bcsstk02	47	44	60	53	54	61	72	84	92	100
bcsstk03	301	119	322	580	897	1,242	1,446	1,866	2,172	2,423

bcsstk04	362	67	71	99	158	216	290	367	468	559
bcsstk05	273	128	97	102	125	167	228	307	375	431
bcsstk06	2,614	183	1,027	2,030	2,965	4,471	7,129	15,671	25,498	40,839
bcsstk08	2,431	116	537	898	1,266	1,808	2,455	3,228	4,383	5,719
bcsstk09	193	172	184	197	255	358	510	660	1,027	1,572
bcsstk10	2,337	376	257	271	432	1,106	2,638	15,035	212,167	∞
bcsstk11	2,712	1,008	1,378	4,631	13,219	40,740	240,913	∞	∞	∞
bcsstk14	4,305	243	2,147	7,788	17,420	37,819	91,922	262,680	∞	∞
bcsstk15	6,337	456	2,593	7,289	16,260	32,545	71,428	250,915	∞	∞
bcsstk16	276	167	118	94	95	109	274	1,057	6,635	183,503
bcsstk17	11,424	2,359	2,532	8,084	91,023	∞	∞	∞	∞	∞
bcsstk18	17,974	607	24,491	115,184	339,025	∞	∞	∞	∞	∞
bcsstk19	1,428	931	38,959	79,667	144,059	218,940	304,379	423,082	∞	∞
bcsstk20	243	446	33,885	64,150	90,612	148,623	213,657	312,902	434,665	∞
bcsstk21	4,769	285	239	228	260	295	361	436	519	704
bcsstk22	311	116	100	113	130	165	200	247	300	350
bcsstk23	1,871	10,169	∞	∞	∞	∞	∞	∞	∞	∞
bcsstk24	6,276	1,568	159,671	∞	∞	∞	∞	∞	∞	∞
bcsstk25	14,405	964	162,506	∞	∞	∞	∞	∞	∞	∞
bcsstk26	10,748	1,010	12,876	52,273	134,597	456,027	∞	∞	∞	∞
bcsstk27	713	203	429	2,580	17,155	86,830	∞	∞	∞	∞
bcsstk28	13,547	4,565	2,970	3,969	18,586	∞	∞	∞	∞	∞
bcsstk34	592	97	67	69	99	286	807	2,539	6,352	16,131
bcsstk36	∞	153,547	∞	∞	∞	∞	∞	∞	∞	∞
bcsstk38	4,401	3,447	78,272	366,004	∞	∞	∞	∞	∞	∞
bcsstm02	12	1	1	1	1	1	1	1	1	1
bcsstm05	18	1	1	1	1	1	1	1	1	1

bcsstm06	102	1	1	1	1	1	1	1	1	1
bcsstm07	235	36	347	1,223	2,237	3,578	5,133	6,859	9,676	12,988
bcsstm08	65	1	1	1	1	1	1	1	1	1
bcsstm09	2	1	1	1	1	1	1	1	1	1
bcsstm11	26	1	1	1	1	1	1	1	1	1
bcsstm12	3,159	385	222	698	5,570	41,689	∞	∞	∞	∞
bcsstm19	275	1	1	1	1	1	1	1	1	1
bcsstm20	206	1	1	1	1	1	1	1	1	1
bcsstm21	3	1	1	1	1	1	1	1	1	1
bcsstm22	53	1	1	1	1	1	1	1	1	1
bcsstm23	3,770	1	1	1	1	1	1	1	1	1
bcsstm24	11,110	1	1	1	1	1	1	1	1	1
bcsstm25	5,783	1	1	1	1	1	1	1	1	1
bcsstm26	2,397	1	1	1	1	1	1	1	1	1
bcsstm39	319	1	1	1	1	1	1	1	1	1
bmw7st_1	668	9,502	∞	∞	∞	∞	∞	∞	∞	∞
bmwcra_1	10,027	5,014	2,923	4,108	106,348	∞	∞	∞	∞	∞
bodyy4	134	87	72	63	62	67	75	84	107	373
bodyy5	337	160	137	125	130	141	158	185	233	1,419
bodyy6	801	285	261	247	262	298	329	369	568	4,499
cbuckle	2,918	669	2,226	4,925	14,475	300,286	∞	∞	∞	∞
crankseg_1	2,086	423	1,510	58,794	∞	∞	∞	∞	∞	∞
crankseg_2	2,716	473	2,683	125,089	∞	∞	∞	∞	∞	∞
ct20stif	6,251	45,160	∞	∞	∞	∞	∞	∞	∞	∞
hood	10,161	2,370	1,520	15,792	∞	∞	∞	∞	∞	∞
lund_a	277	84	111	212	300	384	492	582	748	985
lund_b	301	53	57	102	173	313	481	680	871	1,064

m_t1	68,035	232,229	∞	∞	∞	∞	∞	∞	∞	∞
mesh1e1	16	12	10	9	8	9	11	14	17	21
mesh1em1	27	21	17	15	14	15	17	20	24	29
mesh1em6	16	13	11	9	9	9	9	11	13	16
mesh2e1	70	55	45	39	41	50	66	91	141	197
mesh2em5	56	59	46	40	39	44	56	75	96	127
mesh3e1	17	14	11	10	10	10	11	15	21	29
mesh3em5	12	12	11	9	9	9	10	11	14	16
msc00726	759	115	80	75	109	289	668	1,749	4,139	8,943
msc01050	20,863	1,015	947	1,924	3,790	7,807	17,580	37,819	66,239	118,431
msc01440	1,138	405	946	3,542	8,435	15,831	27,799	39,595	62,475	88,644
msc04515	4,086	3,118	2,962	5,495	11,016	173,760	∞	∞	∞	∞
msc10848	7,530	1,814	21,489	92,396	∞	∞	∞	∞	∞	∞
msc23052	∞	129,259	∞	∞	∞	∞	∞	∞	∞	∞
msdoor	40,058	25,881	17,836	∞	∞	∞	∞	∞	∞	∞
nasa1824	4,403	1,308	2,227	5,200	13,812	34,225	95,589	356,728	∞	∞
nasa2146	264	200	155	201	703	5,389	∞	∞	∞	∞
nasa2910	6,589	1,333	1,296	2,301	4,985	14,065	42,805	195,105	∞	∞
nasa4704	15,753	4,670	49,977	190,616	∞	∞	∞	∞	∞	∞
nasasrb	25,529	5,978	21,850	82,679	∞	∞	∞	∞	∞	∞
nos1	1,934	379	652	1,339	2,012	2,695	3,514	4,263	4,909	5,114
nos2	27,362	3,458	52,454	103,177	132,011	161,595	187,044	211,297	243,428	300,669
nos3	245	214	159	127	117	117	165	344	1,026	3,476
nos4	78	72	59	49	45	44	48	55	69	90
nos5	368	180	145	122	115	114	119	146	167	187
oilpan	32,080	16,253	7,197	185,774	∞	∞	∞	∞	∞	∞
olafu	32,141	20,369	69,267	452,639	∞	∞	∞	∞	∞	∞

plbuckle	1,775	188	166	184	213	338	529	1,328	3,582	9,621
pwtk	∞	32,872	∞	∞	∞	∞	∞	∞	∞	∞
raefsky4	920	203	705	1,099	4,174	132,824	60,931	182,062	∞	∞
s1rmq4m1	3,470	666	7,570	23,643	89,030	∞	∞	∞	∞	∞
s1rmt3m1	4,252	755	3,831	9,582	17,779	253,661	∞	∞	∞	∞
s2rmq4m1	16,167	1,668	92,631	308,464	∞	∞	∞	∞	∞	∞
s2rmt3m1	22,614	2,312	50,580	169,851	∞	∞	∞	∞	∞	∞
s3dkq4m2	∞	79,292	∞	∞	∞	∞	∞	∞	∞	∞
s3dkt3m2	∞	167,138	∞	∞	∞	∞	∞	∞	∞	∞
s3rmq4m1	56,464	21,364	∞	∞	∞	∞	∞	∞	∞	∞
s3rmt3m1	81,987	31,297	∞	∞	∞	∞	∞	∞	∞	∞
s3rmt3m3	127,235	36,912	∞	∞	∞	∞	∞	∞	∞	∞
ship_001	32,581	18,652	430,440	∞	∞	∞	∞	∞	∞	∞
ship_003	∞	333,512	∞	∞	∞	∞	∞	∞	∞	∞
shipsec1	175,059	12,713	∞	∞	∞	∞	∞	∞	∞	∞
shipsec5	171,899	∞	∞	∞	∞	∞	∞	∞	∞	∞
shipsec8	138,202	112,060	∞	∞	∞	∞	∞	∞	∞	∞
smt	9,429	3,141	2,014	11,130	146,509	∞	∞	∞	∞	∞
sts4098	10,392	296	616	1,291	2,339	5,214	14,081	45,126	146,108	∞
vanbody	10,457	24,415	∞	∞	∞	∞	∞	∞	∞	∞

BIBLIOGRAPHY

1. Ament, Marco, Gunter Knittel, Daniel Weiskopf, and Wolfgang Strasser. 2010. “A Parallel Preconditioned Conjugate Gradient Solver for the Poisson Problem on a Multi-gpu Platform.” In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference On*, 583–592.

2. Axelsson, Owe, and Vincent Allan Barker. 1987. *Finite Element Solution of Boundary Value Problems: Theory and Computation*. Vol. 35. Society for Industrial and Applied Mathematics.

3. Bai, Zhaojun, James Demmel, Jack Dongarra, Axel Ruhe, and Henk Van Der Vorst. 1987. *Templates for the Solution of Algebraic Eigenvalue Problems: a Practical Guide*. Vol. 11. Society for Industrial and Applied Mathematics.

4. Barrett, Richard, Michael Berry, Tony F. Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. 1987. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. 43. Society for Industrial and Applied Mathematics.

5. Benzi, Michele, Jane K. Cullum, and Miroslav Tuma. 2000. “Robust Approximate Inverse Preconditioning for the Conjugate Gradient Method.” *SIAM Journal on Scientific Computing* 22 (4): 1318–1332.

6. Benzi, Michele, and Miroslav Tuma. 1998. “A Sparse Approximate Inverse Preconditioner for Nonsymmetric Linear Systems.” *SIAM Journal on Scientific Computing* 19 (3): 968–994.

7. ———. 1999. “A Comparative Study of Sparse Approximate Inverse Preconditioners.” *Applied Numerical Mathematics* 30 (2): 305–340.

8. Bolz, Jeff, Ian Farmer, Eitan Grinspun, and Peter Schröoder. 2003. “Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid.” In *ACM Transactions on Graphics (TOG)*, 22:917–924.

9. Chou, Chung-Han, Nien-Yu Tsai, Hao Yu, Che-Rung Lee, Yiyu Shi, and Shih-Chieh Chang. 2011. “On the Preconditioner of Conjugate Gradient method—A Power Grid Simulation Perspective.” In *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference On*, 494–497.

10. Chow, Edmond, and Yousef Saad. 1998. “Approximate Inverse Preconditioners via Sparse-sparse Iterations.” *SIAM Journal on Scientific Computing* 19 (3): 995–1023.

11. Concus, Paul, and Gene H. Golub. 1976. *A Generalized Conjugate Gradient Method for Nonsymmetric Systems of Linear Equations*. Springer.

12. Cosgrove, J. D. F., J. C. Diaz, and A. Griewank. 1992. “Approximate Inverse Preconditionings for Sparse Linear Systems.” *International Journal of Computer Mathematics* 44 (1-4): 91–110.

13. Davis, Timothy A., and Yifan Hu. 2011. “The University of Florida Sparse Matrix Collection.” *ACM Transactions on Mathematical Software (TOMS)* 38 (1): 1.

14. Georgescu, Serban, and Hiroshi Okuda. 2007. “Conjugate Gradients on Graphic Hardware: Performance & Feasibility.”

15. Gravvanis, G. A. 2002. “Explicit Approximate Inverse Preconditioning Techniques.” *Archives of Computational Methods in Engineering* 9 (4): 371–402.

16. Gregg, Chris, and Kim Hazelwood. 2011. “Where Is the Data? Why You Cannot Debate CPU Vs. GPU Performance Without the Answer.” In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium On*, 134–144.

17. Grote, Marcus J., and Thomas Huckle. 1997. “Parallel Preconditioning with Sparse Approximate Inverses.” *SIAM Journal on Scientific Computing* 18 (3): 838–853.

18. Gui, Yechen, and Guijuan Zhang. 2012. “An Improved Implementation of Preconditioned Conjugate Gradient Method on GPU.” *Journal of Software* 7 (12): 2695–2702.

19. Helfenstein, Rudi, and Jonas Koko. 2011. “Parallel Preconditioned Conjugate Gradient Algorithm on GPU.” *Journal of Computational and Applied Mathematics*.

20. Hestenes, Magnus Rudolph, and Eduard Stiefel. 1952. *Methods of Conjugate Gradients for Solving Linear Systems*. NBS.

21. Knyazev, Andrew V., and Ilya Lashuk. 2007. “Steepest Descent and Conjugate Gradient Methods with Variable Preconditioning.” *SIAM Journal on Matrix Analysis and Applications* 29 (4): 1267–1280.

22. Kolotilina, L. Yu, and A. Yu Yeremin. 1993. “Factorized Sparse Approximate Inverse Preconditionings I. Theory.” *SIAM Journal on Matrix Analysis and Applications* 14 (1): 45–58.

23. Lee, Victor W., Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, and Per Hammarlund. 2010. “Debunking the 100X GPU Vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU.” In *ACM SIGARCH Computer Architecture News*, 38:451–460.

24. Li, Ruipeng, and Yousef Saad. 2013. “GPU-accelerated Preconditioned Iterative Linear Solvers.” *The Journal of Supercomputing* 63 (2): 443–466.

25. Michels, Dominik. 2011. “Sparse-matrix-CG-solver in CUDA.” In *Proceedings of the 15th Central European Seminar on Computer Graphics*.

26. Naumov, Maxim. 2011a. “Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU”. NVIDIA Technical Report, NVR-2011-001.

27. ———. 2011b. “Incomplete-LU and Cholesky Preconditioned Iterative Methods Using CUSPARSE and CUBLAS.” *Nvidia White Paper*.

28. Saad, Yousef. 2003. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics.

29. Van der Sluis, Abraham, and Henk A. van der Vorst. 1986. “The Rate of Convergence of Conjugate Gradients.” *Numerische Mathematik* 48 (5): 543–560.

30. Whitehead, Nathan, and Alex Fit-Florea. 2011. “Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs.” *Rn (A+ B)* 21: 1–1874919424.

31. Yu, Song, Hui Liu, Zhangxin John Chen, Ben Hsieh, and Lei Shao. 2012. “GPU-based Parallel Reservoir Simulation for Large-scale Simulation Problems.” In *SPE Europec/EAGE Annual Conference*.