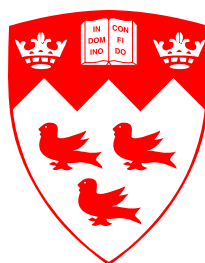


# Krylov Subspace Techniques on Graphic Processing Units

Maryam Mehri Dehnavi



Doctor of Philosophy

Department of Electrical and Computer Engineering

McGill University

Montreal, Quebec, Canada

July 02, 2012

---

A thesis submitted to McGill University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

©Copyright 2012 Maryam Mehri Dehnavi

*To my parents and Farhad*

# ABSTRACT

Computations related to many scientific and engineering problems spend most of their time in solving large, sparse linear systems. Improving the performance of these solvers on modern parallel architecture enables scientists to simulate large accurate models and manipulate massive amounts of data in reasonable time frames. Krylov subspace methods (KSM) are iterative techniques used to solve large sparse systems. The main time consuming kernels in KSMs are sparse matrix vector multiplication (SpMV), vector operations (dot products and vector sums) and preconditioner manipulation. This work presents techniques and algorithms to accelerate some of these kernels on a recent generation of parallel architecture called manycore processors. The performance of the proposed optimizations are tested on graphic processing units (GPUs) and compared to previous work.

The SpMV kernel is accelerated on GPUs and speedups of up to 3.3 times are achieved compared to previous GPU implementations of the algorithm. The conjugate gradient iterative solver is accelerated on NVIDIA graphic cards and a 12.9 fold speedup is achieved compared to optimized implementation of the kernel on multicore CPUs. The sparse approximate inverse preconditioner is accelerated on GPUs and used to enhance the convergence rate of the BiCGStab iterative solver. The preconditioner is generated on NVIDIA GTX480 in the same time as it takes 16 AMD 252 Opteron processors to generate the same preconditioner.

Communicating data between levels of a memory hierarchy and processors is time consuming and costly in KSMs. Communication-avoiding (CA) Krylov solvers take  $k$  steps of a KSM for the same communication cost as one step to reduce the communication overhead in standard KSMs. The matrix powers kernel in communication-avoiding Krylov solvers is accelerated on NVIDIA GPUs and

speedups of up to 5.7 are achieved for the tested problems compared to the standard implementation of  $k$  SpMV kernels.

## ABRÉGÉ

Les calculs liés à de nombreux problèmes scientifiques et techniques demandent qu'on consacre beaucoup de temps à la résolution de grands systèmes linéaires creux. Améliorer la performance de ces résolveurs sur l'architecture parallèle moderne permet aux scientifiques de simuler de grands modèles précis et de manipuler une quantité massive de données dans des délais raisonnables. Les méthodes sous-espaces Krylov (KSM) sont des techniques itératives utilisées pour résoudre de grands systèmes creux. Les noyaux principaux qui demandent beaucoup de temps dans les KSMs sont la multiplication matrice-vecteur creuse (SpMV), les opérations sur les vecteurs (produits scalaires et sommes vectorielles) et la manipulation de préconditionneur. Ce travail présente les techniques et les algorithmes pour accélérer certains de ces noyaux sur une génération récente d'architecture parallèle appelée processeurs multicœurs. La performance des optimisations proposées est testée sur des processeurs graphiques (GPU) et comparée aux travaux antérieurs.

Le noyau SpMV est accéléré sur les processeurs graphiques et des accélérations jusqu'à 3.3 fois plus rapides sont atteintes par rapport aux implémentations de l'algorithme des processeurs graphiques précédents. Le gradient conjugué du résolveur itératif est accéléré sur des cartes graphiques NVIDIA et une accélération 12.9 fois plus rapide est réalisée par rapport à l'implémentation optimisée du noyau sur des processeurs multicœurs. Le préconditionneur approximatif inverse creux est accéléré sur les processeurs graphiques et utilisé pour améliorer le taux de convergence du résolveur itératif BiCGStab. Le préconditionneur est généré sur un NVIDIA GTX480 pour la même durée nécessaire à 16 processeurs AMD Opteron 252 pour générer le même préconditionneur.

La communication de données entre les niveaux d'une hiérarchie de mémoire et des processeurs est longue et coûteuse en KSMs. Les résolveurs sans communication (communication-avoiding ou CA) de Krylov n'utilisent qu'un nombre  $k$  d'étapes d'une méthode de sous-espace de Krylov (KSM) pour un coût de communication équivalent comme une étape qui permet de réduire les frais généraux des communications dans les KSMs standards. Le noyau des pouvoirs de matrice dans les résolveurs de Krylov sans communication est accéléré sur les processeurs graphiques NVIDIA et des accélérations jusqu'à 5.7 plus rapides sont atteintes pour les problèmes testés par rapport à l'implémentation standard de  $k$  des noyaux SpMV.

## ACKNOWLEDGMENTS

I would like to express my greatest gratitude to my supervisor Prof. Dennis Giannacopoulos for his priceless feedback, inspiration and guidance throughout my PhD. His great interest and inspiration in initiating collaboration with other research groups has allowed me to work with world renowned research groups which has opened great windows to my future and work. I would also like to thank Professor Jean-Luc Gaudiot for his guidance and support during my visit to the parallel systems & computer architecture laboratory (PASCAL) in UC-Irvine.

I am very grateful to Professor James Demmel and members of the Berkeley benchmarking and optimization (BeBOP) group which I collaborated with during my six month visit to UC-Berkeley. Finally I would like to greatly thank David Fernandez a very talented colleague of mine whom I have collaborated with throughout my studies.

# TABLE OF CONTENTS

<b>TABLE OF CONTENTS</b> .....	vi
<b>LIST OF TABLES</b> .....	ix
<b>Chapter 1      Introduction</b> .....	1
1.1      Scientific Computing .....	2
1.2      Systems of Linear Equations .....	3
1.2.1      The Solution of Linear Systems of Equations .....	4
1.2.2      Krylov Subspace Techniques .....	5
1.2.3      Kernels in Krylov Methods.....	6
1.2.4      Preconditioning.....	8
1.2.5      Reducing Communication in Krylov Techniques .....	9
1.3      Sparse Matrices.....	10
1.3.1      Types of Sparse Matrices.....	10
1.3.2      Sparse Matrix Storage Formats.....	10
1.4      Parallel Scientific Computation.....	12
1.5      Summary .....	14
1.6      Thesis Outline and Contributions .....	14
<b>Chapter 2      Parallel Computing and Graphic Processing Units</b> .....	17
2.1      Classification of Computer Architectures.....	17
2.2      Parallel Memory Architectures and Programming Models.....	18
2.2.1      Distributed Memory .....	18
2.2.2      Shared Memory .....	18
2.2.3      Hybrid Distributed-Shared Memory Model.....	19
2.3      Graphic Processing Units .....	19
2.4      NVIDIA GPUs .....	22
2.5      CUDA Programming Model.....	23
2.5.1      CUDA Threads and Kernel Execution on GPUs .....	24
2.5.2      Thread Scheduling .....	24
2.6      Performance Optimization in CUDA .....	26
2.6.1      Memory Coalescing.....	26
2.6.2      Avoiding Shared Memory Bank Conflicts .....	27
2.6.3      Increasing Occupancy .....	28
2.6.4      Avoiding Thread Divergence .....	29
2.6.5      Identifying Performance Limiters .....	29
2.6.6      Other optimizations.....	29
2.7      CUDA Libraries .....	31
2.8      Summary .....	32



<b>Chapter 3</b>	<b>Finite Element Sparse Matrix Vector Multiplication on Graphic Processing Units</b>	<b>34</b>
3.1	Introduction.....	34
3.2	GPU Architecture.....	35
3.3	Sparse Matrix Vector Multiplication.....	36
3.4	PCSR (Prefetch-Compressed Row Storage Format) .....	37
3.4.1	Previous Work .....	38
3.4.2	The PCSR Algorithm .....	39
3.4.3	Prefetching.....	42
3.5	Results .....	42
3.6	Conclusion .....	46
<b>Chapter 4</b>	<b>Enhancing the Performance of Conjugate Gradient Solvers on Graphic Processing Units</b>	<b>48</b>
4.1	Introduction.....	48
4.2	GPU Architecture.....	49
4.3	Preconditioned Conjugate Gradient .....	50
4.4	Implementing PCG on GPUs.....	53
4.5	Results .....	56
4.6	Conclusion and Future Work .....	60
<b>Chapter 5</b>	<b>Parallel Sparse Approximate Inverse Preconditioning on Graphic Processing Units</b>	<b>62</b>
5.1	Introduction.....	63
5.2	Sparse Approximate Inverse (SAI) Preconditioning.....	66
5.3	Parallel SAI in NVIDIA GPUs.....	70
5.3.1	GSAI Steps .....	72
5.3.2	Memory Allocation .....	77
5.4	Results .....	80
5.4.1	The GSAI Preconditioning Method.....	81
5.4.2	GSAI vs. ParaSails .....	83
5.5	Conclusion and Future Work .....	84
<b>Chapter 6</b>	<b>Communication-avoiding Krylov Techniques on GPUs</b>	<b>93</b>
6.1	Introduction.....	93
6.1.1	Communication-avoiding Krylov techniques .....	93
6.1.2	NVIDIA GPUS.....	94
6.2	Previous Work.....	95
6.3	Implementation Details .....	97
6.3.1	Matrix Powers on GPU Global Memory .....	97
6.3.2	Matrix Powers on GPU Shared Memory.....	99
6.4	Results .....	101
6.4.1	Matrix Powers on GPU Global Memory .....	102

6.4.2	Matrix Powers on GPU Shared Memory.....	105
6.5	Conclusion and Future Work .....	107
<b>Chapter 7</b>	<b>Conclusion and Future Work .....</b>	<b>109</b>
7.1	Conclusion .....	109
7.2	Future Work.....	111
<b>References</b> .....		<b>112</b>
<b>Appendix I</b>	The BiCGStab Iterative Technique.....	<b>118</b>
<b>Appendix II</b>	NVIDIA GPU Specifications .....	<b>119</b>

# LIST OF TABLES

Table 2.1: CUDA Math Libraries.....	31
Table 2.2: Application-Specific Libraries.....	31
Table 3.1: Non-zeros (nnz) and filling ratio percentage for different padding factors (n) in matrices .....	44
Table 3.2: Speedup of PCSR compared to the row-per-thread and row-per-warp methods on GT8800, the CPU and the Cell architectures. ....	44
Table 4.1: Sparse matrices used for testing.....	58
Table 4.2: Speedup of the optimized PCG compared to PCG row-per-warp (RW) on GPU, vectorized and non-vectorized CPU .....	60
Table 5.1: The number of elements in each of the data structures involved in GSAI and their size based on their data.....	80
Table 5.2: The effect of increasing tolerance ( $\tau$ ) on the number of iterations (GSAI on GTX480).....	86
Table 5.3: Properties of sparse matrices used to test the GSAI preconditioning method.....	86
Table 5.4: The effect of increasing tolerance ( $\tau$ ) on the total execution time involving both the preconditioner construction time and the solve time (GSAI on GTX480). ....	87
Table 5.5: The effect of increasing tolerance ( $\tau$ ) in the GSAI algorithm (on GTX480) on the preconditioner construction time.....	89
Table 5.6: The time spent in computing the stages in Fig. 5.2 for $\tau = 0.9$ on GTX480.....	90
Table 5.7: Preconditioned and unpreconditioned BiCGStab iterative solver on GTX480 and TESLA 2070. ....	90
Table 5.8: ParaSails execution time compared to GPU results. ....	91
Table 6.1: The best speedup of the matrix powers kernel compared to naïve SpMV, fraction of total time spent in communicating data in the naïve SpMV implementation and extra computed flops in the matrix powers kernel performing $k$ . ....	106
Table 6.2: The extra floating point operations performed in the matrix powers kernel for shared memory compared to naïve the SpMV implementation and the total number of thread blocks launched for each $k$ . ....	107

# LIST OF FIGURES

Fig. 1.1: Hierarchy of high performance scientific computing [1].	3
Fig. 1.2: Solvers for linear systems of equations.	5
Fig. 1.3: Compressed sparse row storage format.	11
Fig. 1.4: Compressed format representation of diagonal matrices.	12
Fig. 1.5: The Ellpack-Itpack format of the $A$ matrix from Fig. 1.4.	12
Fig. 2.1: The general memory model of distributed parallel architecture [35].	19
Fig. 2.2: Uniform Memory Access (left figure) and Non-Uniform Memory Access (right figure) shared memory architecture [35].	20
Fig. 2.3: A hybrid CPU memory model (left figure) and a hybrid CPU-GPU memory model (right figure) [35].	20
Fig. 2.4: CPU and GPU floating point operations per second and memory bandwidth (from NVIDIA programming guide [42]).	20
Fig. 2.5: Compute (ALU) control and memory resources in CPU (left figure) and GPUs (right figure).	22
Fig. 2.6: The underlying architecture of NVIDIA Fermi GPUs.	24
Fig. 2.7: Kernel/thread execution model on NVIDIA GPUs (SM represents the streaming multiprocessors on the graphic card, the host and the device are the CPU and GPU respectively).	25
Fig. 2.8: The warp scheduler chooses the next warp ready for execution.	26
Fig. 2.9: The first figure shows threads within a warp accessing data in the 2D array $A$ in strided pattern, when the array is transposed (second figure) data is accesses contiguously allowing for coalesced memory accesses.	27
Fig. 2.10: Row major storage of a $32 \times 32$ matrix in shared memory when each warp accesses one column causes bank conflicts (first figure) which can be resolved by padding the matrix with an extra column (second figure).	28
Fig. 2.11: Padding a vector to be a multiple of 4 and reducing it in parallel.	30
Fig. 3.1: The GT8800 underlying architecture.	36
Fig. 3.2: The CSR SMVM algorithm.	37
Fig. 3.3: PCSR partitioning scheme, (e.g. row 10 is partitioned between blocks 1 and 2 ( $B1$ and $B2$ ); the <i>split vector</i> shows that 3 elements of row 10 are stored in $B1$ and 14 in $B2$ ).	40
Fig. 3.4: The Prefetch-CSR algorithm.	43

Fig. 3.5: Prefetching data in PCSR (a) without prefetching, (b) with prefetching. ....	43
Fig. 3.6: The effect of the padding factor (n) in PCSR. ....	44
Fig. 3.7: Varying the number of prefetches in PCSR.....	45
Fig. 3.8: PCSR performance compared to the row-per-thread and row-per-warp methods on GT8800 as well as the QUAD-Core CPU and Cell architectures. ....	45
Fig. 4.1: NVIDIA GPU architecture. ....	50
Fig. 4.2: Highlighting several bottleneck operations in PCG Shewchuk [7] vs. PCG Chronopoulos [8].....	51
Fig. 4.3: PCG Chronopoulos [8] algorithm implemented on the GPU, optimizing PCSR [53] adds two new kernels to the implementation. ....	54
Fig. 4.4: (a) Percentage of the average execution time of kernels in the PCG Chronopoulos, (b) Fusing kernels in PCG (K1 to K4 represent the kernels in optimized PCG). ....	55
Fig. 4.5: The effect of the optimizations proposed in Section 4.4 in increasing the performance of the PCG algorithm on GT8800.....	58
Fig. 4.6: Performance of the PCG-Row-per-warp [49] method compared to proposed optimized PCG Chronopoulos [8] algorithm on G80 and GT200. ....	59
Fig. 5.1: Steps involved in constructing static sparse approximate inverse preconditioners.....	68
Fig. 5.2: The four stages in implementing SAI preconditioners using GSAI on NVIDIA GPUs ..	70
Fig. 5.3: Constructing local $A$ matrices by first finding $J_{index}$ and $I_{index}$ vector values and then matching the columns referenced in $J_{index}$ to the $I_{index}$ vector. ....	73
Fig. 5.4: The Gram Schmidt QR decomposition with $\langle q, a \rangle = q^T a$ . ....	75
Fig. 5.5: The $M_{pointer}$ vector computed in the <i>compute preconditioner</i> kernel is first modified using the <i>Modify</i> kernel to match the CSC [88] storage format and then the <i>Assemble</i> kernel assembles the $M$ matrix values and stores them in CSC format ( $M_{index}^*$ and $M_{value}^*$ vectors). ....	76
Fig. 5.6: The effect of increasing $\tau$ on the maximum dimension of local $A$ matrices ( $n_{I,max}$ and $n_{2,max}$ ). ....	87
Fig. 5.7: The average fraction of total time (over all matrices) spent in the functions/kernels involved in the first three stages of the GSAI preconditioning algorithm (on GTX480) are shown for an increasing $\tau$ (compute preconditioner consists of all steps in the Compute-GSAI stage)....	88
Fig. 5.8: The average fraction of total time (over all matrices) in generating the SAI preconditioner (the Pre-GSAI, Compute-GSAI and Post-GSAI stages in Fig. 5.2) and solving the problem for an increasing $\tau$ on the GPU using GSAI. ....	88

Fig. 5.9: The speedup achieved from generating the SAI preconditioner on GTX480 and TESLA M2070 using GSAI compared to generating the same SAI preconditioner using ParaSails [59] on 1-32 processors/cores (the generated preconditioner has the same sparsity as A, $\tau = 1$ in GSAI)..	89
Fig. 6.1: The matrix powers implementation on GPU global memory, $x_j^i$ is the $j$ -th component of $x^i = A^i x^{(0)}$ .	98
Fig. 6.2: The steps in the auto-tuner to generate cache blocks for shared memory and find the best performing matrix powers implementation on the GPU.	98
Fig. 6.3: The matrix powers implementation on GPU shared memory, $x_j^i$ is the $j$ -th component of $x^i = A^i x^{(0)}$ .	101
Fig. 6.4: Each matrix is described by its name, description, number of rows, number of non-zeros, average number of non-zeros per row and its non-zero pattern representation.	102
Fig. 6.5: The standard computation of $k$ SpMV's on the GPU, $x_j^i$ is the $j$ -th component of $x^i = A^i x^{(0)}$ .	102
Fig. 6.6: Performance of the matrix powers kernel cache blocking for global memory on NVIDIA GTX480. The "AkX" indicates the best performance obtained for all $k < 40$ . The label "upper bound" shows the performance achievable via scaling the standard $k$ SpMV operations by the change in arithmetic intensity (equation 6.2). The "SpMV" bar shows the performance achieved from the standard $k$ SpMV implementation using CUDA sparse library [89].	104
Fig. 6.7: The speedups achieved for equation 6.1 for the matrix powers kernel on shared memory (test matrix: a pentadiagonal matrix) for different thread blocks per SM (TB/SM) and $k$ .	106
Fig. 6.8: The speedups achieved from equation 6.1 for the matrix powers kernel on shared memory (test matrix: a pentadiagonal matrix) for the best performing number of thread blocks per SM (TB/SM) and different $k$ .	107

## PREFACE TO THE THESIS

### **Format of the Thesis**

This thesis contains four self-contained research papers in Chapters 3, 4, 5 and 6.

The work presented in Chapter 3 entitled “Finite Element Sparse Matrix Vector Multiplication on GPUs” is published in the IEEE Transactions on Magnetics and the short version published in the proceeding of IEEE Conference on Computational Electromagnetics (COMPUMAG 2009). Chapter 4 entitled “Enhancing the Performance of Conjugate Gradient Solvers on Graphic Processing Units” is published in the IEEE Conference on Electromagnetic Field Computation (CEFC 2010) and IEEE Transactions on Magnetics. Chapter 5 presents “Parallel Sparse Approximate Inverse Preconditioning on Graphic Processing Units” accepted for publication in IEEE Transactions on Parallel and Distributed Systems (the short version is published in the proceeding of COMPUMAG 2011). Chapter 6 entitled “Communication-avoiding Krylov Techniques on GPUs” is submitted to CEFC 2012 and is under preparation for journal publication.

### **Contributions of the Authors**

The applicant, Maryam Mehri Dehnavi, is the primary author of all the work presented and the person responsible for all implementations along with major contributions and ideas. Prof. Dennis Giannacopoulos initiated the research, contributed ideas, valuable guidance, supervision, support and manuscript editing throughout the thesis. David Fernandez contributed suggestions and insightful discussions in the first three contributions; multicore results of his work are used for comparison purposes in chapters 3 and 4.

Professor Jean-Luc Gaudiot provided valuable feedback, manuscript editing and guidance for the work presented in Chapter 5. Professor James Demmel initiated the research topic presented in Chapter 6 and contributed ideas, insightful discussions, manuscript editing and valuable supervision for the work presented in the chapter.



## LIST OF ACRONYMS

CPU: central processing units.

GPU: graphic processing units.

CUDA: compute unified device architecture.

SM: streaming multiprocessors.

SP: scalar processors.

API: application programming interface.

TB: thread block.

EM: electromagnetics.

FEM: finite element method.

FDTD: finite difference time domain.

SIMD: single instruction multiple data processing (also short vector processing).

CSR: compressed sparse row, also called compressed row storage (CRS).

CSC: compressed sparse column storage.

SPD: symmetric positive definite matrices.

SMVM/SpMV: sparse matrix vector multiply.

AXPY: alpha  $x$  plus  $y$ ,  $y := \alpha x + y$ .

SAXPY: single precision alpha  $x$  plus  $y$ .

(P)CG: (preconditioned) conjugate gradient algorithm.

SAI/SPAI: sparse approximate inverse preconditioners.

KSM: Krylov subspace method.

DRAM: dynamic random access memory.

BiCGStab: method of biconjugate gradient stabilized.

CA: communication-avoiding.

GSAI: GPU accelerated SAI preconditioning technique.

PCSR/P-CSR: prefetch CSR technique.

---

## Chapter 1 INTRODUCTION

Simulations related to many physics and engineering problems have become larger and more complex in recent years leading to the design of faster and more powerful computing platforms. The new generation of supercomputers—multicore<sup>1</sup>, manycore<sup>2</sup>, petascale<sup>3</sup> and exascale<sup>4</sup> computers— will enable scientists and engineers to solve large accurate models and analyze massive quantities of data from a broad range of natural and engineering systems. On the path to extreme-scale computing, systems with hundreds of thousands of computing cores that can sustain a billion billion calculations per second are being built [1]. The future transition in computer architecture poses numerous scientific and technological challenges. Similar to the migration from vector to parallel computing systems that occurred 15 years ago, the transition to exascale computing will require adaptation, reformulation and redesign of algorithms to effectively exploit future parallel hardware systems. Numerical algorithms involved in simulations related to many complex scientific applications need to rely increasingly on fine grain parallelism and strong scaling.

Krylov subspace methods (KSMs) are a popular class of iterative solvers used to solve systems from many scientific applications and real life problems. The solution of such systems can be a very time consuming process and can take several days or weeks on single-core CPUs. This work accelerates the main computing kernels in KSMs on the most up-to-date manycore architectures namely graphic processing units (GPUs). The reported performance and speedups are compared to the fastest available accelerations on modern multicore, manycore and multiprocessor hardware platforms.

The first chapter is organized as follows:

Section 1.1 describes various stages involved in transferring a physical model from real life problems to a computer program and introduces two of the popular

---

numerical techniques used in scientific simulations. Techniques to solve linear systems of equations specifically Krylov subspace methods are then described along with their main computing kernels in Section 1.2. The aforementioned section also briefly reviews major contributions of this work. Sparse matrices, their types and storage formats are introduced in Section 1.3. The importance of parallel scientific computations on modern architectures is detailed in Section 1.4 and the chapter is summarized afterwards. The outline and major contributions of the thesis are presented in the last section.

## 1.1 Scientific Computing

Many scientific applications can be formulated into mathematical models using a series of differential equations and then transferred into a numerical formula using numerical techniques such as the finite element model (FEM) and the finite difference time domain (FDTD) technique. The numerical representation is then translated into a programming model and executed on the desired hardware platform. The use of modern architectures and parallel processing for running real life applications and problems efficiently, reliably and quickly is called “High Performance Scientific Computing”. Fig. 1.1 shows the steps involved in transferring a physical model to a computer program, to be executed on state-of-the-art hardware platforms.

The finite element method is a widely used numerical technique for the analysis and simulations of electromagnetic problems. Following are the basic steps involved in FEM:

- Discretization of the domain.
- Selection of the interpolation functions and formulation of the system.
- Solution of the assembled system of equations.

One of the most time consuming steps in the finite element analysis is solving the system of equations. The system of equations from many real life electromagnetic problems are very large, thus, enhancing the execution time of FEM solvers is essential. The upcoming sections will describe techniques to solve large linear systems; this work adapts such techniques to better utilize the resources in modern architectures and to reduce the solution time of scientific problems.

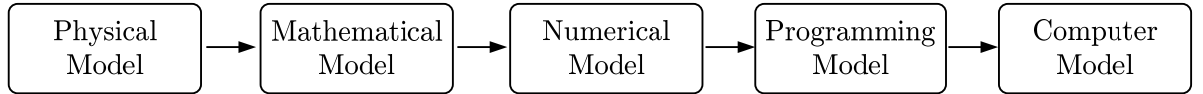


Fig. 1.1: The hierarchy of high performance scientific computing [1].

## 1.2 Systems of Linear Equations

Numerical techniques in many scientific problems result in solving a linear system of equations which can be represented as

$$Ax = b \tag{1.1}$$

where  $x \in \mathbb{R}^N$  is the unknown determined using the coefficient matrix  $A \in \mathbb{R}^{N \times N}$  and  $b \in \mathbb{R}^N$ . The solution of equation 1.1 plays an essential role in simulating scientific applications and real life problems. For practical problems the size of matrix  $A$  is very large which increases memory requirements for storing and solving the system. As the system grows the time to compute the solution will also considerably increase. Thus adapting such algorithms to run on modern architectures in a reasonable time frame with minimum storage requirements is fundamental in high performance scientific computing. Various algorithms to solve equation 1.1 are introduced in this chapter and time consuming computing kernels in these solvers are introduced.

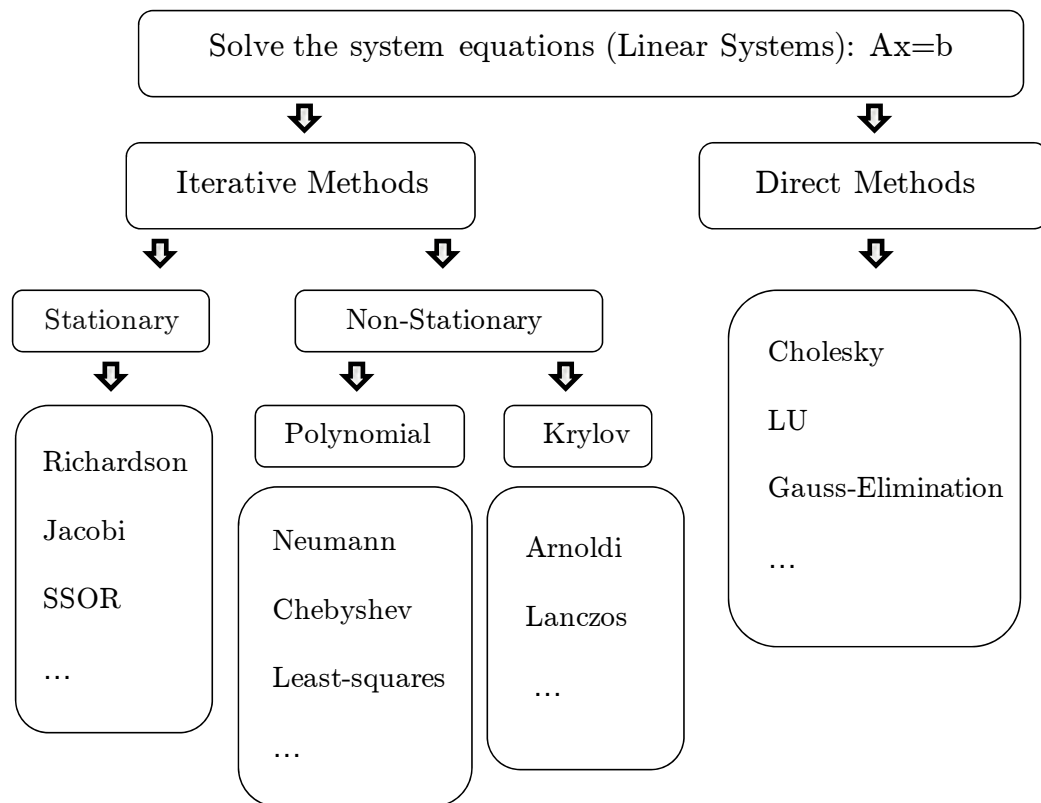


Fig. 1.2: Solvers for linear systems of equations.

### 1.2.1 The Solution of Linear Systems of Equations

A linear system of equations can be solved using *direct* or *iterative* techniques [3]. Direct techniques are not a suitable candidate for vector and parallel machines due to their sequential and recursive nature [4]. A more viable alternative to solving large linear systems is using iterative solvers. Iterative techniques improve the solution of the linear system of equations in a sequence of iterations. Using an initial solution vector, an iterative technique and a termination criterion, the solution of a *convergent* system converges to a desired accuracy. The number of iterations required to reach the termination criteria is determined by the distribution of eigenvalues of the coefficient matrix  $A$ . Fig. 1.2 shows a classification of solvers for linear systems of equations. This work studies Krylov subspace techniques classified as non-stationary iterative solvers.

### 1.2.2 Krylov Subspace Techniques

Krylov subspace methods (KSM's) are a large class of iterative techniques used to solve systems of linear equations from a broad range of applications. A dominant computing kernel in standard KSMs is sparse matrix vector multiplication (SpVM or SMVM). Using one or more SpVM operations in each iteration, KSMs add vector(s) to a basis for one or more "Krylov subspace(s)" and in each iteration the best solution is selected from the expanding subspace,  $(s, A, v) = \text{span}\{v, Av, A^2v, \dots, A^{s-1}v\}$  where  $A$  is an  $n \times n$  square matrix,  $v$  is a length  $n$  vector and  $s$  is a positive integer. Krylov subspace techniques can be categorized based on the choice of subspaces and the way the system is preconditioned [5]. The most commonly used algorithms to compute the basis of these subspaces are Arnoldi, Lanczos and Bi-Lanczos [5]. Lanczos algorithms are known as the symmetric version of Arnoldi methods while Bi-Lanczos techniques are a variant of Lanczos algorithms applicable to non-symmetric problems. A variant of Lanczos and Bi-Lanczos algorithms are used to compute the solution of the tested problems in this work.

The conjugate gradient method (CG) [5], [6] is a Lanczos based Krylov solver used for symmetric positive definite (SPD) matrices. The CG algorithm approximates the solution of the linear systems of equations based on orthogonal residuals and previous search directions. The two variants of the conjugate gradient technique used in this work, namely, Shewchuk [7] and Chronopoulos [8] are detailed in Chapter 4.

The biconjugate gradient stabilized (BiCGStab) iterative solver classified as Bi-Lanczos algorithms is also used in this work (Chapter 5). By generating a CG-like sequence of vectors, one based on the original coefficient matrix  $A$  and the other based on  $A^T$ , the solution of the linear system is solved using bi-orthogonal sequence of vectors and a smooth convergence behaviour [2]. The algorithm for the preconditioned BiCGStab solver used in Chapter 5 can be found in Appendix I. In

the following section compute intensive kernels in the aforementioned Krylov solvers are discussed.

### 1.2.3 Kernels in Krylov Methods

Computing the solution of linear systems using iterative techniques such as KSMs can be very slow for large problems due to communication and computation cost of major computing kernels in KSMs. In this work the term “kernel” is used to represent time-consuming parts of Krylov methods. To accelerate the execution time of Krylov solvers on modern architectures, the most important computing kernels in these algorithms should first be identified. The four main kernels in KSMs are as follows [10]:

- Sparse matrix vector multiplication: The SpMV kernel multiplies a matrix by a vector and stores the result.
- AXPY: This class of kernels are classified as vector-vector operations and are represented in the form  $y := \alpha x + y$ .
- Dot products: Another class of vector-vector operations in the form  $\beta := y^* \cdot x$ .
- Preconditioning: The next section discusses the importance of preconditioners. Generating the preconditioner and applying it to the iterative solver are two separate kernels; the computing cost of the former is considered as part of the iterative solver. Applying the preconditioner  $M$ , to the iterative solver usually involves an SpMV operation which multiplies either  $M$  or  $M^{-1}$  with a vector; the result is then used in the KSM.

This work proposes techniques to accelerate the execution of the aforementioned kernels on modern architectures with many cores. Chapter 3 parallelizes the execution of the SpMV kernel on NVIDIA GPUs; AXPY and dot products are computed in parallel and used in the CG and BiCGStab solvers in chapters 4 and 5.

---

All results in this work are compared to the best available accelerations on both GPUs and CPUs. Previous work on accelerating the above kernels on GPUs (manycore architectures) is surveyed in the related chapters (Chapter 3 and Chapter 4); a survey of previous work on accelerating the aforementioned kernels on CPUs (multicore architectures) is provided in the following. One of the best performing optimizations proposed by Fernandez et al. [11], [12] is used as baseline CPU results for comparisons in Chapter 3 and Chapter 4.

Optimization techniques used in modern architectures such as register and cache blocking, loop transformations, special diagonal storage of matrices and reordering [13], [14], [15] have mostly initiated from the Berkeley benchmarking and optimization (BeBOP) group [16] at UC-Berkeley and used in open source libraries such as OSKI (Optimized Kernel Interface) [17] and POSKI (parallel OSKI) [18]. Such optimizations accelerate the execution of kernels such as SpMV on individual cores of an architecture. Other work such as [5], [19], [20] have also proposed techniques to reduce memory transfers, increase data locality and instruction level parallelism on a single-core. Williams et al. [21], [22] take advantage of the multiple cores on modern CPU architectures and show the importance of exploiting parallelism across multiple cores to enhance the performance of kernels such as SpMV. The conjugate gradient method has also been accelerated on multicore architectures in work such as [23], [24]. Vector units in modern architectures have been efficiently used in [25] by using special matrix formats. Work presented by Fernandez et al. [11], [12] not only exploits parallelism between the multiple cores of modern CPU but also uses vector operations to further enhance the performance of the SpMV and CG algorithms on modern CPUs. GPU accelerations of SpMV and CG kernels in this work have been compared to highly optimized CPU code provided in the aforementioned work.



The “preconditioning” kernel in KSMs is introduced in detail in the next section and the sparse approximate inverse (SAI/SPAI [26]) preconditioner is generated on NVIDIA GPUs in Chapter 5 and then applied to the BiCGStab solver.

#### 1.2.4 Preconditioning

Iterative techniques used to solve large scale problems from practical applications generally have a slow convergence rate. The convergence of these problems depend on the condition number of the coefficient matrix  $A$  which is determined by the spectral property of the matrix [27], [28]. A preconditioner  $M$ , can improve the convergence rate of the linear system  $Ax = b$  by transforming the system to  $M^{-1}Ax = M^{-1}b$  and decreasing the condition number of the preconditioned matrix  $M^{-1}A$ . A system of equations can be preconditioned in three ways:

- Left preconditioning: When the preconditioner is applied to the left hand side of the coefficient matrix resulting in the following system:  $M^{-1}Ax = M^{-1}b$ .
- Right preconditioning: This type of preconditioning does not effect the right hand side of the systems and is applied as follows:  $AM^{-1}u = b$ ,  $x = M^{-1}u$ .
- Split preconditioning: A preconditioned system with split preconditioning is represented as:  $M^{-1}LAM^{-1}Ru = M^{-1}Lb$ ,  $x = M^{-1}Ru$  where  $M = MLMR$ .

Using any of the aforementioned preconditioning techniques, the number of iterations required to reach a desirable tolerance in the linear system reduces at the expense of constructing and storing  $M^{-1}$  and applying it in the iterative solver. Based on how they are constructed, preconditioners are classified as implicit or explicit. Implicit preconditioners compute the approximate of  $A$  while explicit preconditioners are approximates of the inverse of  $A$  [28]. Sparse approximate inverse preconditioners are an important class of explicit preconditioners which are suitable for parallelization. Chapter 5 introduces this class of preconditioners,

proposes techniques to generate them in parallel on NVIDIA GPUs and applies them to the BiCGStab iterative solver in parallel. A very detailed survey of all available work on SPAI preconditioners and work on accelerating this kernel on multicore, manycore and multiprocessors architectures is provided in the aforementioned chapter.

### 1.2.5 Reducing Communication in Krylov Techniques

Communication is defined as data movements in the memory hierarchy of a single processor or between different processors. One of the major bottlenecks in accelerating Krylov techniques on modern architectures is the limited memory bandwidth and data communication overhead. Krylov techniques and most of the kernels in such methods are memory-bound, i.e. communicating data within the memory hierarchy is a major performance limiting factor when accelerating these kernels on modern processors. Techniques proposed to accelerate the performance of Krylov techniques and their kernels in Chapter 3 and Chapter 4 such as memory coalescing, data prefetching, fusing kernels, binding vectors to caches such as the texture memory, etc. reduce the communication overhead of these kernels on GPUs. While benefiting from the aforementioned techniques and optimizations in individual GPU kernel calls, a more aggressive approach in reducing memory communication overhead in Krylov techniques is studied in chapter 6. The aforementioned chapter is based on work titled communication-avoiding (CA) Krylov techniques introduced by the BeBOP research group [18] and extensively studied in Hommen's thesis [10]. CA Krylov techniques reduce communication in Krylov solvers by taking  $k$ -steps of the iterative solvers at the same time; data will be on fast memory while the  $k$  steps of iterative solver are taken at the same time, reducing memory references considerably. A detailed survey of communication-

avoiding Krylov techniques is presented in Chapter 6 and the main computing kernel in these algorithms called the matrix powers kernel is accelerated on GPUs.

### 1.3 Sparse Matrices

Matrices from many partial differential equations (PDEs) based on numerical methods are usually large and have few non-zeros. Whenever the large number of zeros elements and their locations in a matrix can be used to better store and solve the system of equations the matrix is “sparse” [5]. Sparse matrices are defined by Duff [29] as the ratio of the zero to non-zero entries in a matrix and can be represented and operated in compressed formats introduced in Section 1.3.2 [30].

#### 1.3.1 Types of Sparse Matrices

The distribution of non-zero elements in sparse matrices varies based on the properties of the original problem and the mesh generation technique used. If the non-zeros in the matrix form a regular pattern along diagonals, the matrix is referred to as *structured*. An *unstructured* sparse matrix on the other hand, consists of irregularly distributed non-zeros. The structure of a sparse matrix is important in high performance scientific computing. The memory required to store matrices as well as the time to operate on them can be reduced if the non-zeros in a matrix follow a structured pattern.

#### 1.3.2 Sparse Matrix Storage Formats

Sparse matrices can be stored in compressed formats which only require allocating memory to their non-zeros elements. Various schemes exist to store sparse matrices and can be found in books such as [31]. This section briefly introduces some of the more important sparse matrix storage formats. The compressed sparse row (CSR) format stores the matrix using three arrays (Fig. 1.3). The value vector (VAL in Fig. 1.3) stores the non-zero elements of the matrix in consecutive rows and the indices corresponding to each of these values are stored in another array

(INDX in Fig. 1.3). Pointers to the beginning of each row in the aforementioned arrays are stored into a third array (PTR in Fig. 1.3). If the matrix is stored in column order using the above format then the storage schemes is referred to as the compressed sparse column (CSC) storage format.

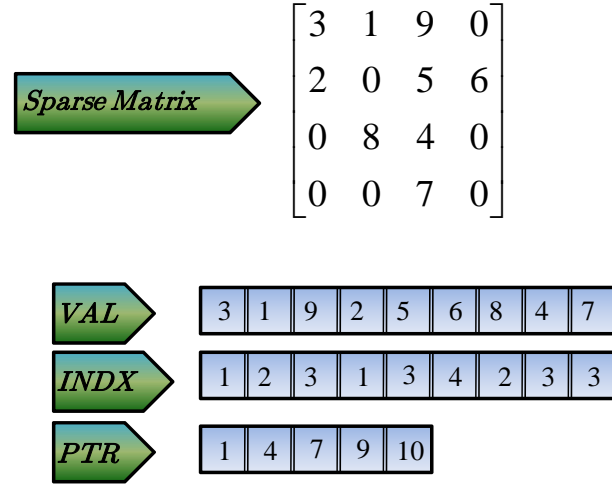


Fig. 1.3: Compressed sparse row storage format.

The non-zeros in diagonally dominant matrices are distributed in a small number of diagonals. These matrices can be stored using their diagonals in a format represented by *Diagonal* (1:n, 1:nd) [32], where nd is the number of diagonals and n is the matrix rank. The diagonal offsets are stored in another array *OFF* (1:nd) (Fig. 1.4). Another compressed storage format for sparse matrices suitable for vector machines is the Ellpack-Itpack format. The matrix is stored using two dense arrays. The non-zero values of the matrix are stored in the first array in row order while padded to match the size of the largest row in the matrix (*C* in Fig. 1.5). The column indices of each elements is stored in the other array (*JC* in Fig. 1.5). The CSR, CSC and diagonal matrix formats are used in this work to store both the coefficient matrices and the preconditioners.

$$A = \begin{bmatrix} 2 & 0 & 1 & 0 & 0 \\ 4 & 5 & 0 & 6 & 0 \\ 0 & 6 & 7 & 0 & 9 \\ 0 & 0 & 12 & 10 & 0 \\ 0 & 0 & 0 & 11 & 10 \end{bmatrix} \quad \begin{matrix} \text{Diagonal} = \begin{bmatrix} * & 2 & 1 \\ 4 & 5 & 6 \\ 6 & 7 & 9 \\ 12 & 10 & * \\ 11 & 10 & * \end{bmatrix} \\ \text{OFF} = \begin{bmatrix} -1 & 0 & 2 \end{bmatrix} \end{matrix}$$

Fig. 1.4: Compressed format representation of diagonal matrices.

$$C = \begin{bmatrix} 2 & 1 & 0 \\ 4 & 5 & 6 \\ 6 & 7 & 9 \\ 12 & 10 & 0 \\ 11 & 10 & 0 \end{bmatrix} \quad JC = \begin{bmatrix} 1 & 3 & 1 \\ 1 & 2 & 4 \\ 2 & 3 & 5 \\ 3 & 4 & 4 \\ 4 & 5 & 5 \end{bmatrix}$$

Fig. 1.5: The Ellpack-Itpack format of the  $A$  matrix from Fig. 1.4.

Proposed algorithms and techniques in this work are tested using matrices from real applications. Two main matrix repositories were used to obtain the tested matrices, the Matrix Market repository [33] and the University of Florida Sparse Matrix collection [34].

#### 1.4 Parallel Scientific Computation

The demand for more precise and complex simulations has increased considerably in the past few decades increasing the execution time of such problems on single-core CPUs. Building faster serial/single-core computers poses many physical and practical challenges such as increased wire delays, miniaturization limitations, manufacturing costs, etc. [35]. During the past 20+ years, Parallel Computing, defined as the simultaneous use of multiple computing resources to compute discrete parts of a problem in parallel, has been increasingly used to simulate large scale scientific problems. Complex problems can potentially be solved in a much shorter time with cheaper resources if executed in parallel. Parallel

---

computing also enables the use of non-local resources over a large network, also referred to as Cloud Computing. For example, over 2.9 million computers in 253 countries are used by SETI@home [36]. Thus, the development of efficient methods to improve the performance of practical scientific problems on parallel processors is almost inevitable in future.

A major challenge in parallel computing is the ability to choose an optimum hardware and parallelize the algorithms to achieve maximum performance and speed on the architecture. Originally introduced in 2001 with the IBM Power4 processor and later integrated to in Sun UltraSPARC IV, AMD dual core Opteron, Cell Broadband engine [37] and Intel Pentium architecture, etc., multicore processors are a more recent and important class of parallel computers. Computations related to large complex problems are divided between threads executing in parallel on the existing cores of such architectures. Multicore processors with several tens and hundreds of cores are also referred to as manycore architectures. NVIDIA graphic cards are currently one of the most popular manycore architectures. Intel will soon be realising their many integrated cores (Intel MIC) [38] processors which have considerably more cores compared to their current multicore CPUs. NVIDIA GPUs are the main computing platforms used throughout this work; most of the optimization and accelerations proposed are applicable to architectures with tens or hundreds of cores and can be adapted to run on future manycore processors.

Modern GPUs are not only powerful graphic engines but also highly parallel programmable manycore processors, allowing very fast manipulation of data. Because graphic cards possess much greater computational parallelism than single or multicore CPU computing platforms, to increase the speed and accuracy of real life problems, compute intensive kernels should be processed on the GPU. Various challenges exist in optimizing algorithms to run on graphic cards. Using the most

---

up-to-date version of GPUs and by achieving a thorough understanding of the compute intensive kernels described in Section 1.2, these kernels are accelerated to run in parallel on the many cores available on modern graphic cards. Whenever possible results achieved from accelerating kernels on graphic cards are compared to their parallel execution on other architectures such as Intel multicore CPUs and the Cell Broadband engine as well as multiple processors.

### 1.5 Summary

The main objective of this work and the importance of high performance computing for scientific applications were discussed in this chapter. Compute intensive kernels in Krylov subspace techniques were introduced and a summary of the compute intensive kernels accelerated in the thesis was provided along with literature review complementary to the previous work survey in each chapter.

### 1.6 Thesis Outline and Contributions

The rest of the thesis is organized as follows:

- Chapter 2: Classifies general computer architectures and parallel computers and gives a detailed introduction to NVIDIA graphic cards, their architecture, programming model and optimization techniques to accelerate the execution of compute intensive kernels on such architectures. Other optimization methods used throughout this work to better utilize the available resources in many core and parallel architectures are also detailed in this chapter.
- Chapter 3: The execution of the sparse matrix vector multiplication is accelerated on NVIDIA GPUs and compared to optimized implementations of the kernel on the CPU and Cell Broadband engine as well as previous work on GPUs.

- Chapter 4: A less common variant of the conjugate gradient method called the Chronopoulos variant is accelerated on the GPU and compared to the more popular Shewchuk variant. The Chronopoulos variant is shown to be better for parallelization on manycore architectures, the performance of the accelerated kernels compared to best available implementations of the CG method on GPU and multicore CPUs.
- Chapter 5: The sparse approximate inverse preconditioner is, to our knowledge, for the first time accelerated on GPUs and then used in the BiCGStab iterative solver which is also implemented on the GPU. The proposed implementations are compared to the best available accelerations of SAI preconditioners on multiprocessor platforms.
- Chapter 6: The communication-avoiding matrix powers kernel is accelerated on graphic cards for the first time on both GPU global and shared memory. The performance of the matrix powers kernel on GPUs is compared to optimized standard implementation of  $k$  step SpMV on NVIDIA GPUs.

Major contributions of the work are as follows:

- A new partitioning scheme and sparse storage format called Prefetch-CSR is proposed to accelerate the execution of the SpMV kernel on manycore architectures which enhanced the performance of this kernel on NVIDIA GPUs considerably compared to previous accelerations. Novel techniques such as padding vectors with zero to enable parallel reductions, hiding memory access delays by prefetching consecutive matrix partitions are also proposed.
- The Chronopoulos variant of the conjugate gradient method is for the first time accelerated on GPUs and shown to be a better alternative to the more common Shewchuk CG variant for manycore architectures. The Prefetch-



---

CSR SpMV kernel is optimized to avoid atomic updates, kernels are fused to reduce kernel launch overheads and increase data locality.

- The sparse approximate inverse preconditioner is accelerated on NVIDIA GPUs for the first time using the proposed GSAI algorithm. Novel techniques to manage GPU memory, compute columns of the preconditioner in parallel via thousands of threads, solve local systems, assemble the matrix in a compressed format on the GPU and transfer it to the iterative solver without going back to the CPU are introduced.
- The communication-avoiding matrix powers kernel used to reduce communication in Krylov subspace techniques is for the first time implemented on NVIDIA GPUs. The matrix is partitioned to fit in the GPU global and shared memory to reduce the communication overhead of KSMs. The best performing matrix powers kernel and cache block size are determined in an auto-tuning stage to efficiently implement the aforementioned kernel on GPUs. The proposed techniques enhance the performance of KSMs on graphic cards and enable the fast execution of KSMs for large problems on manycore architectures.

---

## Chapter 2 PARALLEL COMPUTING AND GRAPHIC PROCESSING UNITS

Graphic cards have recently become very attractive hardware platforms for parallel scientific computations and are the main architecture used in this work. The objective of this chapter is to introduce the architectural and programming details of GPUs specifically NVIDIA graphic cards. A popular classification of general computer architectures and existing parallel processor memory models are introduced in Sections 2.1 and 2.2. Section 2.3 compares CPU and GPU architectures and their ability in accelerating parallel applications. The architectural and programming features of NVIDIA GPUs are introduced in Section 2.4 and Section 2.5, respectively. Techniques to optimize the performance of applications on NVIDIA GPUs are presented in Section 2.6; many of these techniques are used throughout this work to enhance the performance of the computing kernels on graphic cards. Some of the available libraries for accelerating various problems and kernels on GPUs are listed in Section 2.7. Finally, a summary of the chapter is provided in Section 2.8.

### 2.1 Classification of Computer Architectures

A widely used classification of computer architectures was proposed by Flynn [39] in 1966. Based on how instruction and data are processed on the hardware, Flynn classifies computer architectures as:

- Single Instruction, Single Data (SISD): Only one instruction and one data stream are used at each clock cycle, e.g., serial computers.
- Single Instruction, Multiple Data (SIMD): Processors execute the same instruction on different data, e.g., GPUs, vector machines.
- Multiple Instruction, Single Data (MISD): Using a single data stream each processing unit operates on data independently.

- Multiple Instruction, Multiple Data (MIMD): Every processor maybe operating on different data and executing different instruction streams. eg. super computers, networked computing clusters.

SIMD, MIMD, and MISD are all types of parallel processors which are classified based on their memory architecture in the next section.

## **2.2 Parallel Memory Architectures and Programming Models**

Based on the memory configuration, parallel processors are classified into three main categories: shared memory, distributed memory and hybrid models. The programming model used for each class of processors is different and depends on the processor memory model.

### **2.2.1 Distributed Memory**

Inter-processor memories in distributed memory systems are connected using a network. Each processor has a separate memory space (Fig. 2.1) which is not mapped to others. If a processor requires data located in another memory space, the programmer has to explicitly manage how data is transferred. Distributed systems are easy to scale and data local to a processor can be accessed in a short time. Released in 1994, one of the most popular interfaces used for implementations on distributed systems is the message passing interface (MPI) [40].

### **2.2.2 Shared Memory**

The processors in a shared memory system operate independently but share the same memory resources (Fig. 2.2). The shared memory space can either be accessed uniformly by all processors (Uniform Memory Access-UMA) or have a non-uniform access pattern (Non-Uniform Memory Access-NUMA). Programming models such as POSIX threads and OpenMP [41] are used in shared memory architectures such as Intel multicore. Proposed by NVIDIA, compute unified device architecture (CUDA)

is another application programming interface (API) used to program GPUs which are also classified as shared memory architectures.

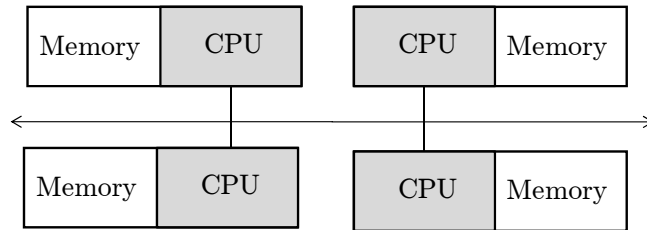


Fig. 2.1: The general memory model of distributed parallel architectures [35].

### 2.2.3 Hybrid Distributed-Shared Memory Model

Hybrid memory models are a combination of shared and distributed memory architectures. By connecting shared memory processors (SMP) or GPUs through a network as shown in Fig. 2.3, a hybrid memory model is constructed. Compute intensive kernels can then be executed on local nodes and communication between different nodes is maintained explicitly using distributed memory programming models. For example, applications executing on a cluster of GPUs, benefit from parallelism on each GPU via programming models such as CUDA while parallelism and data communication between GPUs is maintained using programming interfaces such as MPI.

Most of the hardware platforms used in this work such as NVIDIA GPUs, Intel multicore and the Cell Broadband engine belong to the shared memory model.

## 2.3 Graphic Processing Units

Forced by the fast growing video game industry, a class of shared memory manycore architectures called graphic processing units (GPUs) have recently become a popular architectural resource for scientific computing. As illustrated in Fig. 2.4 [42], the computing power and memory bandwidth of GPUs has grown significantly larger than multicore CPUs in the last few years. For example, GPUs such as NVIDIA GeForce GTX680 can perform up to 3 TFLOPs (tera floating

point operations per second) with bandwidths up to 190 GB/s (gigabytes per second) which is considerably larger than the maximum performance achieved from muticore CPUs.

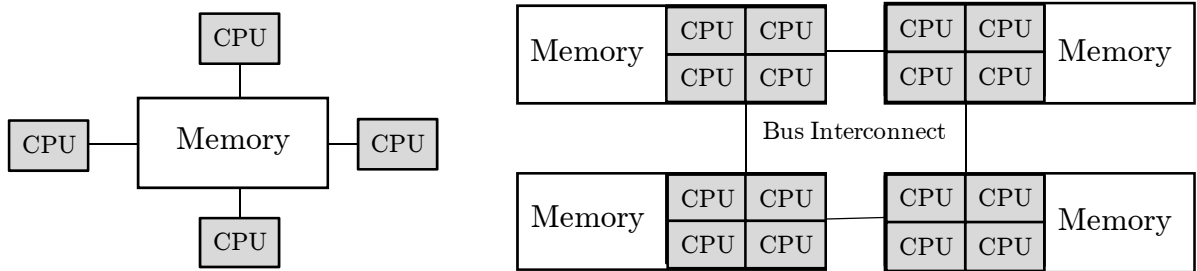


Fig. 2.2: Uniform Memory Access (left figure) and Non-Uniform Memory Access (right figure) shared memory architectures [35].

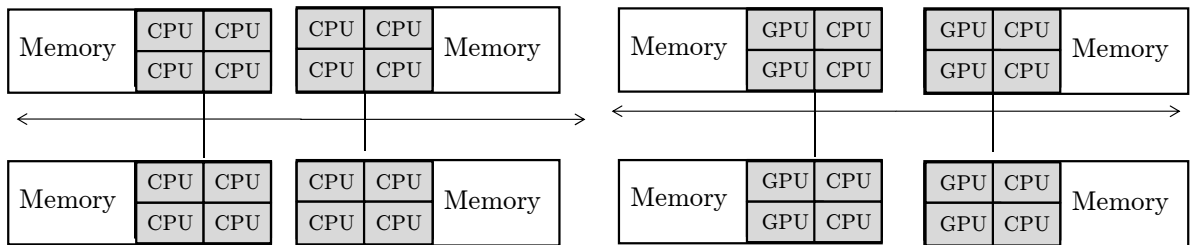


Fig. 2.3: A hybrid CPU memory model (left figure) and a hybrid CPU-GPU memory model (right figure) [35].

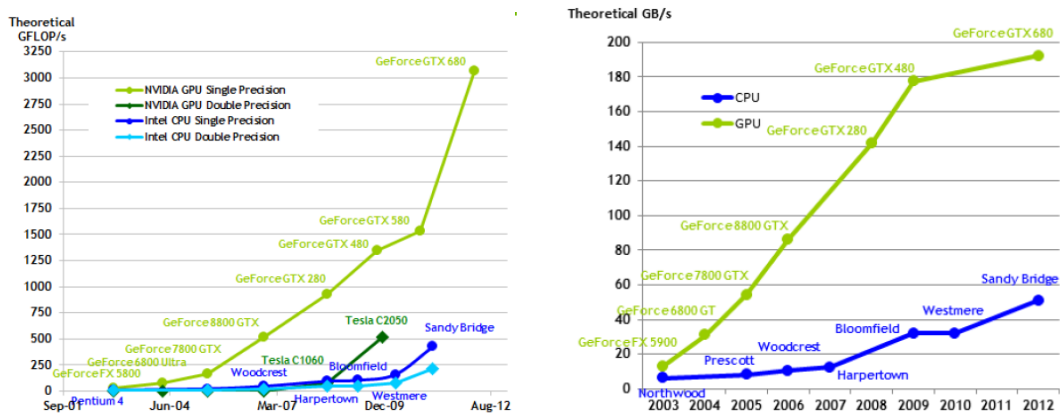


Fig. 2.4: CPU and GPU floating point operations per second and memory bandwidth (from NVIDIA programming guide [42]).

---

The large performance gap between GPUs and multicore CPUs is due to the difference in design philosophies of the two architectures. While cache memories reduce data and instruction access latencies, the large control logic in CPUs (Fig. 2.5) enables the execution of complex sequential code within a thread. GPUs, on the other hand, are designed to compute a large number of floating point operations in a very short time by maximizing chip area and power budget dedicated to arithmetic calculations. Graphic cards are optimized to launch massive numbers of threads, each executing relatively simple tasks which require a small logic unit. The many threads executing in parallel hide memory access latencies and reduce DRAM (dynamic random access memory) accesses via cached memory spaces. To conclude, GPUs are considered as numeric computing engines used to execute compute intensive sections of applications while complex sequential parts of the code are still computed on the CPU.

Four major high-end graphic card vendors are NVIDIA, AMD (formerly ATI), Qualcomm and Intel. AMD Fusion announced in 2006, integrates a CPU and GPU in a mobile stand-alone GPU. The second generation of Fusion is expected to be released in June 2012. Intel released Knights Ferry, a prototype of their many integrated core (MIC) architecture, in 2010 and proposed to release the first commercial version in late 2012. NVIDIA is best known for its gaming cards, but with the introduction of general purpose programming on GPUs researchers and scientists have been using NVIDIA cards for high performance computations in the past few years. NVIDIA GPUs are used throughout this work to measure the performance of proposed optimizations and techniques. The techniques and optimizations presented in this work are not limited to the NVIDIA graphic cards and can be used on other manycore architecture and GPUs with minor adjustments and modifications.

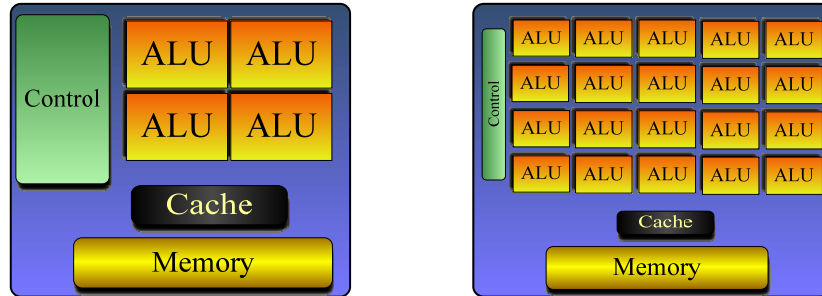


Fig. 2.5: Compute (ALU) control and memory resources in CPU (left figure) and GPUs (right figure).

## 2.4 NVIDIA GPUs

This work accelerates compute intensive kernels in Krylov subspace techniques on NVIDIA graphic cards. The computing resources and architectural specification of these GPUs are described in this section.

Fig. 2.6 shows the architecture of NVIDIA graphic cards (other modern GPUs also have a similar architecture). *Streaming processors* (SPs) are the computing cores of the architecture, which depending on the version of the GPU can have one or multiple computing units. Every 8 SP is grouped into a *streaming multiprocessor* (SM) that is connected to a graphics double data rate (GDDR) DRAM, referred to as *global memory*. Global memory access latencies are high compared to other memory spaces on the GPU (eg. global memory bandwidth in NVIDIA GTX480 is 177 GB/s). The GPU and CPU communicate through the peripheral component interconnect (PCI) express [42] and data is transferred from the CPU to the GPU global memory prior to invoking a GPU kernel (eg. the device/GPU to host/CPU memory bandwidth is 4 GB/s peak for a PCI-express x16). Each SM has access to an on-chip shared memory space and a register file private to each thread. In the new generation of NVIDIA GPUs called Fermi [42], the shared memory space can be configured to 48KB or 16KB with the rest allocated to L1 cache. Accessing data in shared memory and the register files has a low latency. Texture and constant

memory are cached and read-only; however, accesses to texture memory are considerably faster than accesses to constant memory. Fermi GPUs also have a 768 KB unified L2 cache that services all load, store, and texture requests.

NVIDIA GPUs have evolved significantly over the last decade resulting in the development of more than 6 generations through these years. The most up-to-date GPU available at the time was used in the contributions proposed in this work. The GPUs used are from the G80, G200, Fermi and TESLA generations of NVIDIA graphic cards; hardware specifications of each of the cards used are provided in Appendix II.

## **2.5 CUDA Programming Model**

Until 2006, programming GPUs was very difficult and only possible using graphic APIs such as OpenGL and Direct3D. General purpose programming for graphic processing units called GPGPU, was limited by the APIs and only a few people acquired the skills to use GPUs for general applications. The massively parallel architecture of graphic cards motivated GPU manufacturers, to devote silicon area to facilitate general purpose programming on GPUs. Using additional hardware, NVIDIA introduced CUDA (Compute Unified Device Architecture) in 2007 [43], which soon became a fundamental parallel programming language in the scientific computing community and made manycore architectures a popular parallel hardware platform for scientific applications. Released in 2008, OpenCL is an open source framework that executes on heterogeneous platforms of CPU and GPUs. NVIDIA GPUs also support OpenCL; to date speedups achieved from CUDA are higher than OpenCL for most applications [44], therefore CUDA is used for GPU related tests in this work. The proposed optimizations and techniques can be ported to OpenCL and executed on most high-end GPUs (e.g NVIDIA, AMD, Intel MIC).



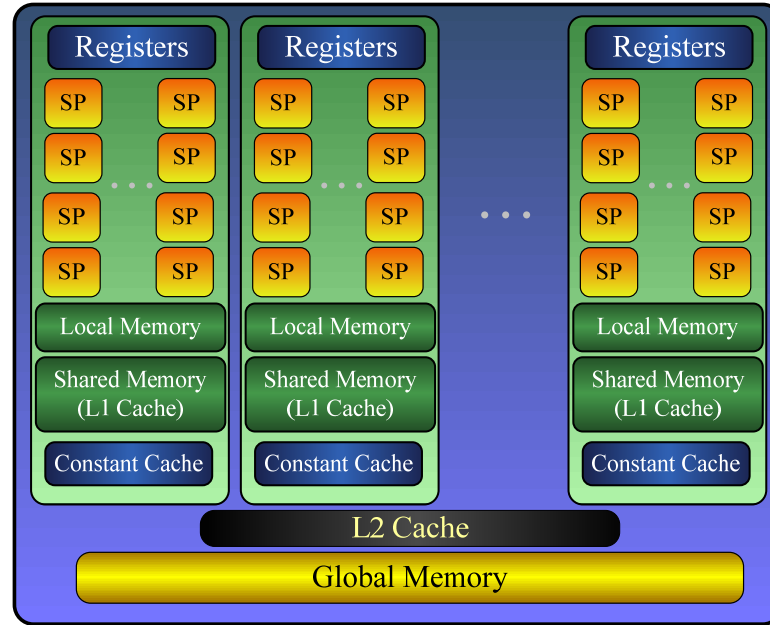


Fig. 2.6: The underlying architecture of NVIDIA Fermi GPUs.

### 2.5.1 CUDA Threads and Kernel Execution on GPUs

Data parallelism is exploited in an application, where many arithmetic operations are simultaneously performed on the data structures. While sequential parts of an application execute on the CPU, data parallel sections of the program are parallelized to run on the graphic card. The threads in a GPU kernel are responsible for performing arithmetic operations on different data in parallel (SIMD). To execute parts of an application on the GPU, data has to be first transferred from the host (CPU) to the device (GPU) global memory, a GPU kernel is then launched to run the application on the GPU, finally results are transferred back to the CPU if required (Fig. 2.7).

### 2.5.2 Thread Scheduling

To execute an application in parallel, the GPU has to launch thousands of threads. The threads inside a GPU kernel are grouped into *blocks* where the threads inside one block share data through GPU shared memory space and their execution can be synchronized with little overhead. Threads belonging to different blocks can

only communicate through GPU global memory and can execute in any order. Up to eight thread blocks are assigned to a streaming multiprocessor simultaneously. The maximum number of threads per SM is also limited and depends on the GPU compute capability. In the Fermi graphic cards, up to 1536 threads can be active simultaneously on one SM.

Active threads in an SM execute in groups of 32 called warps. Warps are scheduled in a scheduler and execute one at a time as shown in Fig. 2.8. To efficiently hide long accesses to global memory, when an instruction executing by the threads in a warp requires data from the device memory, the warp is placed in a waiting list and other warps are scheduled for execution.

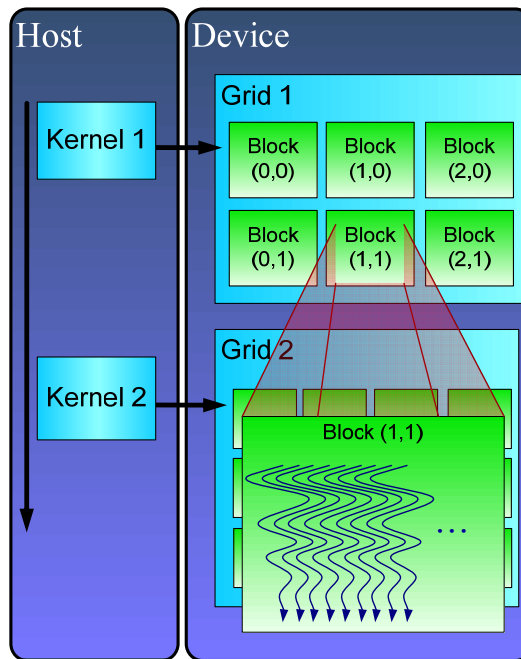


Fig. 2.7: Kernel/thread execution model on NVIDIA GPUs (SM represents the streaming multiprocessors on the graphic card, the host and the device are the CPU and GPU respectively).

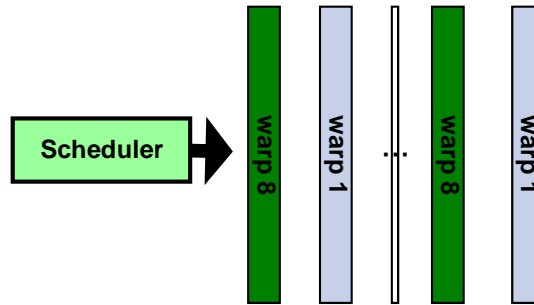


Fig. 2.8: The warp scheduler chooses the next warp ready for execution.

## 2.6 Performance Optimization in CUDA

Various parameters should be considered when developing an application to run in parallel on GPUs. Besides identifying embarrassingly parallel parts of an application and exploiting fine grain parallelism using hundreds and thousands of threads, accesses to various memory spaces on the GPU should be efficiently handled to benefit from the high memory bandwidth on such architectures. Resource occupancy and instruction usage should also be maximized to hide memory access latencies. This section introduces some of the most important techniques used to enhance the performance of GPU kernels on the Fermi architecture (the same methods are used in earlier generations of NVIDIA GPUs with little modification [42]).

### 2.6.1 Memory Coalescing

Many applications are bandwidth bound and all accesses to data begin from global memory. Accesses to GPU global memory are not cached and can take up to 600 cycles; thus optimizing global memory references can enhance the performance of the kernel considerably. Accessing continuous global memory locations by threads in a half warp is called memory coalescing; 32, 64 and 128 bytes by half a warp can be processed in one transaction to reduce global memory accesses. To increase coalesced memory references, data should be stored and accessed contiguously in global memory. Fig. 2.9 shows how accesses to a 2D array can be coalesced if

transposed. Whenever possible accesses to global memory have been coalesced in this work to reduce the execution time of computing kernels on GPUs.

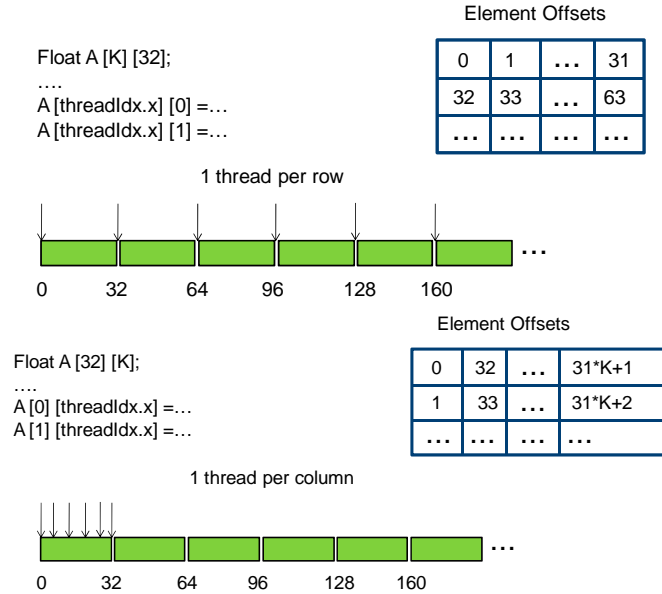


Fig. 2.9: The first figure shows threads within a warp accessing data in the 2D array  $A$  in strided pattern; when the array is transposed (second figure) data is accesses contiguously allowing for coalesced memory accesses.

### 2.6.2 Avoiding Shared Memory Bank Conflicts

Shared memory is on-chip memory space with approximately 20 times lower access latency compared to global memory. The size of shared memory is considerably smaller than GPU global memory and should be used for data that are more frequently accessed.

Data in shared memory is stored in 32 2-byte wide banks where contiguous 4-byte words belong to different banks. Called bank conflicts, if threads within a warp access different 4-byte words of the same bank their access is serialized. To benefit from the shared memory high bandwidth, bank conflicts should be avoided. As shown in Fig. 2.10, if a  $32 \times 32$  matrix is stored in row major (rows are stored consecutively) in shared memory and each warp accesses one column, the memory

accesses will be serialized due to bank conflicts. Adding an extra column to the data array (also called padding), will eliminate bank conflicts since data accessed by the threads in a warp are stored in different banks.

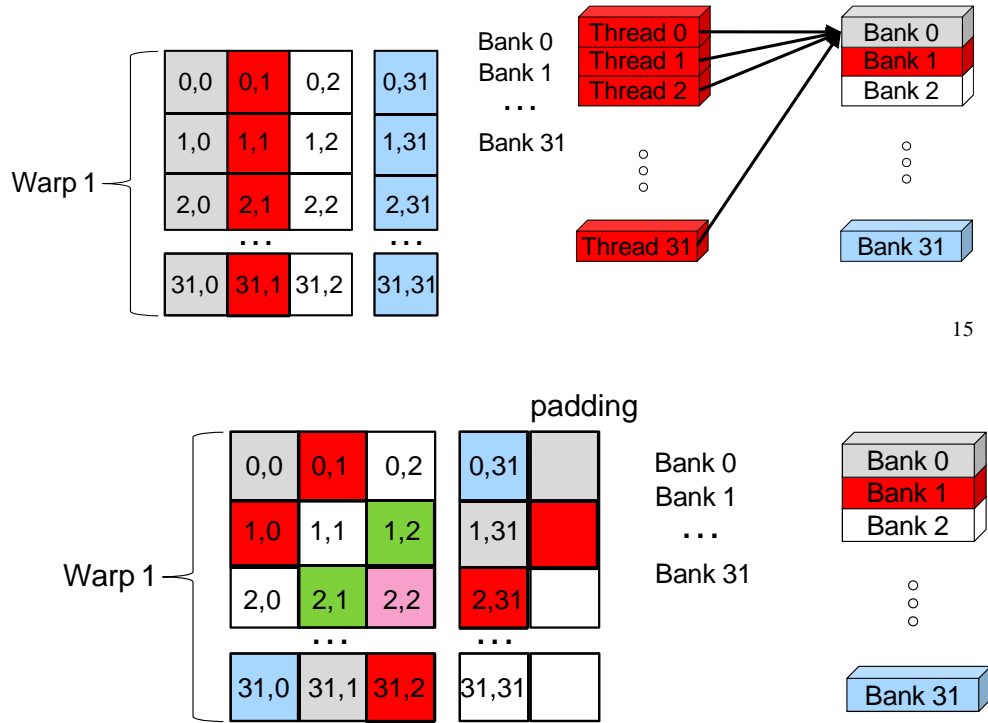


Fig. 2.10: Row major storage of a  $32 \times 32$  matrix in shared memory when each warp accesses one column causes bank conflicts (first figure) which can be resolved by padding the matrix with an extra column (second figure).

### 2.6.3 Increasing Occupancy

The number of active warps divided by the maximum active warps in an SM is used to measure occupancy in a GPU kernel. Higher occupancy improves the performance of a GPU kernel by fully utilizing the available GPU resources; 66% occupancy is usually enough to reach the peak performance. In Fermi graphic cards up to 48 warps and 8 blocks can be active per SM; however, depending on the number of threads per block, available shared memory (and registers) per thread, the number of active warps (and occupancy) can vary. For example, if 32 bytes of shared memory is used by a thread with a 16KB shared memory configuration only

16 warps are active per SM reducing occupancy to 33 percent. The programmer should consider the aforementioned factors while optimizing GPU kernel code in order to enhance resource occupancy. CUDA occupancy calculator [45] can also be used to determine the available resources per thread and occupancy.

#### **2.6.4 Avoiding Thread Divergence**

The threads in a warp execute one instruction at a time, thus parallelism is exploited at warp granularity. If the threads inside one warp go through different execution paths, their execution will be serialized and the threads will diverge. This is called thread divergence and should be avoided since it will decrease the performance of the GPU kernel. Optimizations proposed in this work eliminate or minimize thread divergence resulting in high levels of parallelism in the kernels.

#### **2.6.5 Identifying Performance Limiters**

Accelerating the execution of an application on the GPU can be tedious and desired speedups might not be achieved initially. Performance should be further optimized after running the kernel, detecting performance limiters and addressing them. Major performance limiters in GPU kernels are memory throughput, instruction throughput, latency or a combination of all. Performance can be assessed based on the algorithm's memory and computational requirements, instruction and profiler counters collected using CUDA profiler [42] or using code modified to measure memory and arithmetic execution times independently. The GPU kernels and optimizations proposed in this work are fine-tuned using the aforementioned techniques to obtain high speedups.

#### **2.6.6 Other optimizations**

Other optimizations have also been used throughout this work in order to enhance the performance of computing kernels on GPUs. Some of the more

important optimizations used in future chapters, are presented in detail in this section.

- **Prefetching:** Prefetching is used to hide memory access latencies; while some instructions are waiting for data to be fetched from memory, other instructions perform arithmetic operations. As data in the current partition is being manipulated by some of the threads in the block, other threads access memory and fetch the required data for the next data partition in order to hide global memory access latencies.
- **Padding:** Adding extra elements to a data structure is referred to as padding. We pad some of the data structures with zero in this work to be multiples of a desired number. This will regularize operations and enable a more aggressive manipulation of the data structure in parallel. For example if vectors were padded to be multiples of four as shown in Fig. 2.11, each thread could reduce every 4 elements in parallel enabling efficient parallel reduction of the vector.

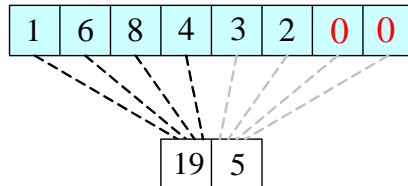


Fig. 2.11: Padding a vector to be a multiple of 4 and reducing it in parallel.

- **Spreading the  $x$  vector:** When solving  $Ax = b$ , the  $x$  *vector* values are accessed in an irregular pattern which can lead to many uncoalesced memory accesses. To regularize these accesses, the corresponding  $x$  vector values are stored/spread in a separate vector in caches in the order they are accessed.

Whenever possible, vector operations such as sort, add, reduce and search are implemented in parallel using the many threads in a thread block and best available algorithms from NVIDIA developers website and libraries [42].

Table 2.1: CUDA Math Libraries

CUFFT	Fast Fourier Transforms Library
CUBLAS	Complex BLAS Library
CUSPARSE	Sparse Matrix Library
CURAND	Random Number Generation Library
THRUST	Performance Primitives for Video Processing
Math.h	C99 Floating Point Library

Table 2.2: Application-Specific Libraries

Molecular Dynamics	OpenMM, HOOMD-blue, ACEMD, ...
Electromagnetic and Acoustic Waves	Acceleware, EM Photonics, ...
Computer Vision	GPU VSIPL, GpuCV, ...
Computational Statistics	R+GPU, ...
Computational Finance	OPLib, ...

## 2.7 CUDA Libraries

Since the introduction of CUDA, many researchers and developers have developed and modified libraries and application-specific software to run their compute intensive kernels on GPUs and harness the power of graphic cards in running embarrassingly parallel problems. As shown in Table 2.1, software used in various application areas has already been modified to run parts of their code on GPUs. Math libraries have also been modified to run on GPUs with only a few listed in Table 2.2. Two of the fastest available libraries in sparse and dense linear algebra released and maintained by NVIDIA called CUBLAS and CUSPARSE are used in this work.



Hundreds of such software and library currently exist (MAGMMA, iCUDA, Barra, decuda, CULA, CUPP, etc.) and with the emerging heterogeneous architectures that uses a combination of GPU and CPU hardware most of the existing sequential software will have to be modified to run on manycore architectures in near future.

## **2.8 Summary**

The architecture and programming model of NVIDIA graphic cards were studied in this chapter along with optimization techniques used to enhance the performance of scientific applications on such platforms. The following chapters will propose various algorithms and methods to accelerate the execution of computation intensive kernels in Krylov techniques on GPUs.

---

## PREFACE TO CHAPTER 3

The following chapter is included as a paper published by the IEEE Transactions on Magnetism (“Finite Element Sparse Matrix Vector Multiplication on GPUs”, IEEE Trans. on Mag., vol. 46, no. 8, pp. 2982-2985, 2010). In this chapter we introduce a new partitioning scheme and sparse storage format (called Prefetch-CSR) to accelerate the execution of SMVM/SpMV kernel on NVIDIA GPUs. Performance results are compared to the SMVM implementation proposed by NVIDIA called *row-per-warp* which is one of the fastest available accelerations of the aforementioned kernel on graphic cards. Results are also compared to optimized implementation of the SMVM kernel on Intel multicore and the Cell Broadband engine.

The proposed acceleration of the SMVM kernel (Prefetch-CSR) is further optimized in the next chapter and used to accelerate the execution of the conjugate gradient method on NVIDIA GPUs.

## Chapter 3 FINITE ELEMENT SPARSE MATRIX VECTOR MULTIPLICATION ON GRAPHIC PROCESSING UNITS

Maryam Mehri Dehnavi, David M. Fernandez, and Dennis Giannacopoulos

**Abstract:** A wide class of finite element electromagnetic applications requires computing very large sparse matrix vector multiplications (SMVM). Due to the sparsity pattern and size of the matrices, solvers can run relatively slowly. The rapid evolution of graphic processing units (GPUs) in performance, architecture and programmability make them very attractive platforms for accelerating computationally intensive kernels such as SMVM. This work presents a new algorithm to accelerate the performance of the SMVM kernel on graphic processing units.

**Index terms:** Computer architecture, Graphic processing units, Parallel processing, Sparse matrix vector multiplication.

### 3.1 Introduction

The performance of finite element (FE) electromagnetic applications can be dominated by the iterative solvers used, such as conjugate gradient (CG) based methods. As problems become larger and more complex, the computation overhead of these kernels dramatically increases the execution time of such solvers on single-core CPUs. Thus, the development of efficient methods to improve the performance of iterative solvers on parallel processors is almost inevitable.

One of the most important kernels in iterative solvers such as the CG method is the sparse matrix vector multiplication. This operation is performed in each iteration and often consumes a majority of the computation time. The main objective of the SMVM kernel is to calculate  $Ax$  where  $A$  is a sparse matrix and  $x$  is a dense vector. Major limitations of SMVM computation involving FE matrices are large memory storage and bandwidth requirements as well as indirect and irregular

memory accesses. Graphic processing units (GPUs) have recently evolved into very attractive commodity data-parallel coprocessors. Easy to learn programming interfaces such as CUDA [43] have allowed massive multithreading and increased utilization of large numbers of cores on the GPU, making them cost efficient highly parallel platforms to solve computationally intensive scientific problems [46].

The main objective of this work is to accelerate the performance of finite element SMVM kernels on the NVIDIA GT8800 graphic cards using a new algorithm, namely PCSR (Prefetch-Compressed Row Storage).

### 3.2 GPU Architecture

Modern GPUs are massively parallel and conform to single instruction multiple data (SIMD) architectures. Several levels of parallelism are offered by GPUs through multiple pipelines and vector processing. GPU architectures such as AMD-ATI X1k series process data in parallel using vector processors while others such as NVIDIA G80 use multiple pipelines to perform parallel operations. With the ability to launch thousands of threads in parallel and processing trillions of operations in seconds, NVIDIA GPUs are among the best for general purpose programming [43], [46]. The NVIDIA GT8800 graphic card (Fig. 3.1) consists of 14 streaming multiprocessors (SMs), each containing eight scalar processors (SPs), or processor cores running at 1.5GHZ. Each of the SMs access a separate 16KB shared memory and a total of 8192 registers. The 14 SMs are connected via 512MB of off-chip device memory.

Using the CUDA programming model, the GPU is viewed as a compute device capable of executing a large number of threads in parallel. While the main core of the code is run on the CPU, parts of the application that exhibit rich data parallelism are implemented as kernel functions on the device (GPU). Data required by the kernel is transferred to the GPU global memory and the parallel portion of

the application is then executed on the device using many different threads. The programmer divides the threads into threads blocks that are distributed amongst the SMs allowing each multiprocessor to run a maximum of eight blocks. Thread blocks allocated to one SM communicate via fast shared memory, but blocks from different SMs can only communicate through global memory with a memory access latency of up to 600 cycles. Every 32 threads in a block execute the same instruction and are called a warp. When threads in the same warp follow different paths of control flow, we say that these threads diverge in their execution. Thread divergence forces the threads in a warp to execute sequentially thus reducing the execution speed of the application and should be avoided [43].

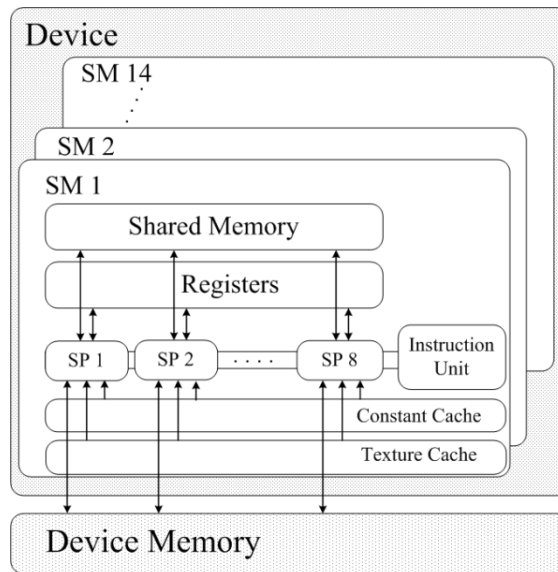


Fig. 3.1: The GT8800 underlying architecture.

### 3.3 Sparse Matrix Vector Multiplication

The SMVM kernel is one of the most popular kernels in solving sparse linear systems for large and complex finite element simulations. A variety of sparse matrix representations exist, each having a distinct form of data storage and access, manipulation of matrix entries and calculation of the matrix vector multiplication

product. The compressed sparse row storage format is one of the most commonly used data structures for SMVM solvers. The non-zero elements of the sparse matrix in this format are stored in a value vector (VAL), while the corresponding index values are held in another vector (INDX). The format also uses a pointer array (PTR), which points to the first entry of each row in VAL and INDX [5]. The sparse vector matrix product in this format is calculated using two nested loop iterations (Fig. 3.2).

for $i = 1$ to number of rows
$Y[i] = 0$
for $j = PTR[i]$ to $PTR[i + 1]$
$Y[i] = Y[i] + VAL[j] * X[INDX[j]]$
end for
end for

Fig. 3.2: The SMVM CSR algorithm.

### 3.4 PCSR (Prefetch-Compressed Row Storage Format)

Many challenges exist in optimizing the performance of scientific applications such as the SMVM kernel on GPU platforms. Some are as follows: global memory access latency, limited shared memory, thread synchronizations, thread divergence, inadequate number of threads and limited global memory bandwidth. The way the programmer addresses these issues differs depending on the application [43].

A new SMVM algorithm, namely PCSR is proposed in this section. By combining CSR with a novel partitioning scheme and computation strategy, the execution time of the SMVM kernel is accelerated on NVIDIA GPUs. To clarify the major advantages of our method, a survey of previous work on SMVM kernel optimization techniques for the GPU is first presented and the details of the new implementation are then described.

### 3.4.1 Previous Work

Since the release of CUDA in 2007, few works have investigated the SMVM kernel optimization on the GPUs. Buatois et al. [47] investigated the performance of blocked-CSR on the G80 series of NVIDIA graphic cards. To increase the performance of their method, the matrix filling ratio is decreased, adding extra non-zeros to the value vector and increasing the number of memory transactions. Sengupta et al. [48] proposed the use of segmented scan for calculating SMVM on GPUs. Wiggers et al. [24] reorders matrix rows to increase parallelism in the SMVM kernel and reduce thread divergence when a row is calculated by a single thread. Sorting matrix rows increases processing overhead considerably increasing the execution time on the host. Comparing the performance of various SMVM representations on the GPU, Bell et al. [49] proposed a new method to optimize the CSR format on the GPU. To decrease thread divergence, instead of calculating each row by a single thread, all threads on a single warp are responsible for computations of one row. Matrices with average non-zeros less than 32 per row do not benefit from their proposed technique and since every element is fetched from the global memory separately and only when their value is required, a majority of memory fetches are uncoalesced when run on the GT8800.

Previous results were implemented on various versions of NVIDIA GPUs each with a different memory bandwidth and processing power. To compare our method with other work we applied the *row-per-thread* and *row-per-warp* methods using the code in [49] on our GPU and present comparison results. Our proposed algorithm introduces new techniques to hide global memory access latency via data perfecting and memory coalescing. The technique also regularizes the data access pattern on the GPU by proper partitioning and padding the matrix with zeros. Detailed description of the method and its major contributions are given in the proceeding sections.

### 3.4.2 The PCSR Algorithm

Details of the partitioning scheme and padding method used in PCSR are proposed in this section. Methods of efficiently accessing the  $x$  vector and the algorithm steps are also presented.

#### A. Partitioning scheme

To obtain a reasonable execution time on the GPU, global memory accesses should be minimized by transferring data on to shared memory. Due to the limited storage of shared memory, vectors require to be partitioned and transferred in small segments. Different row sizes in small matrices complicate the partitioning of the vectors. We propose an efficient partitioning method that benefits from the inherent parallelism on the GPU. To maximize resource usage on an SM, 768 threads should run simultaneously on its architecture. Therefore, if three blocks are active per SM, 256 threads should be executed via one block to maximize performance. The value and index vectors in the CSR representation should also be divided into blocks of 256 elements (vectors are padded with zeros to be divisible to 256). Searching through the row pointer vector, rows split between the blocks are found and their *id* as well as their spreading pattern between two blocks is stored in a new vector called the *split vector* (Fig. 3.3). For matrices with more than 256 average number of non-zeros per row the split vector will store only the id of blocks holding elements of more than one row, to keep the size and transfer time of the split vector to GPU memory negligible compared to the total data transfer time.

Simultaneous loading of data from global memory to shared memory, coalesced memory accesses and reduced memory transfer time are the major benefits of partitioning. Partitioning the vectors and loading them from global memory at the beginning of the kernel, will also reduce the effects of thread divergence. Divergent threads in the computation section of the kernel will fetch their required data from on-chip shared memory, avoiding the serialization of global memory accesses.



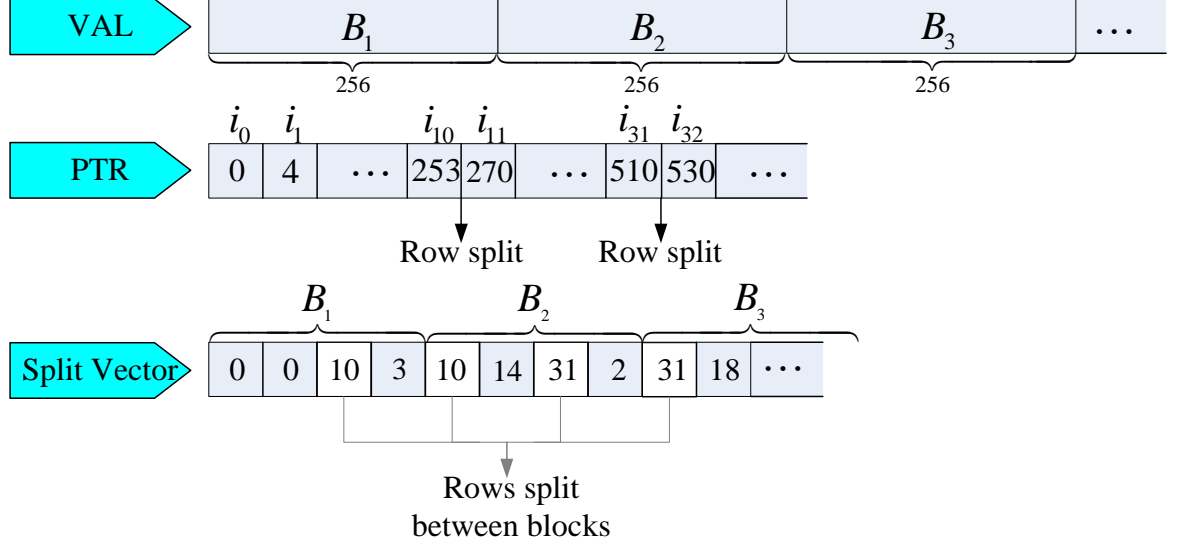


Fig. 3.3: PCSR partitioning scheme, (e.g. row 10 is partitioned between blocks 1 and 2 ( $B_1$  and  $B_2$ ); the *split vector* shows that 3 elements of row 10 are stored in  $B_1$  and 14 in  $B_2$ ).

### B. Zero padding

Minimizing thread divergence on GPUs is essential for achieving good performance. If each thread calculates one row, the diversity in row sizes will cause thread divergence and threads will execute sequentially. Assigning a warp to each row [49] will also cause thread divergence since the number of non-zeros per row are not necessarily multiples of 32. Since the execution is serialized in divergent threads, we reduce the number of operations per thread by padding.

Padding each row to be a multiple of the padding factor ( $n$ ) will allow the kernel to reduce the product vector using parallel reduction. Every  $n$  value in the product vector can be added via parallel reduction and stored in another vector called *sum*. Because of the padding, in the reduction procedure threads will not add values of more than one row. The number of elements corresponding to a row in the sum array is less than the product vector. Thus to calculate the results of each row, a thread will only add the elements in the sum vector corresponding to that row, reducing the number of operations executing sequentially (although increasing the

padding factor is beneficial in reducing thread divergence, larger  $n$  decreases the vector filling ratio and increases the value vector size).

### *C. Texture memory*

The  $x$  vector cannot be divided between blocks due to the irregular indirect access to its elements in the SMVM kernel. Accessing the global memory for every index increases memory latencies. To avoid such accesses, the  $x$  vector is loaded on to texture memory and its elements are spread on the shared memory of each block simultaneously. The texture memory is an on-chip cached memory space, thus a texture fetch costs one memory read from global memory only on a cache miss otherwise it just costs one read from the texture cache. Loading the  $x$  vector to texture memory decreases global memory access latencies and enhances the performance of the SMVM kernel.

In the proposed technique, threads in a block simultaneously load 256 elements of the  $x$  vector corresponding to the index vector values on to shared memory. The technique enables simultaneous spreading of the  $x$  vector on the GPU with minimum memory access latency and also minimizes the effects of thread divergence throughout the kernel.

### *D. Algorithm steps*

Fig. 3.4 shows the seven steps in the PCSR algorithm. Partitions of the index and value vector allocated to each block (256 elements) are first loaded into shared memory simultaneously to coalesce memory accesses and reduce memory transfer time. The  $x$  vector elements are then loaded from texture memory and spread in shared memory. The 256 elements allocated to each block are multiplied with the corresponding values of the  $x$  vector in parallel by the 256 threads in a block. After determining the index and split pattern of the rows in each block using the split vector, required elements of the pointer (PTR) array are loaded into shared memory.

Depending on the padding factor, the product vector is reduced in parallel to generate the sum vector values. Using the sum vector, the final value of each row is calculated by different threads with minimum thread divergence and the results are written into the global memory simultaneously.

### 3.4.3 Prefetching

The time required to load data from global memory is high due to the 300 cycle global memory access latency. Prefetching the required data for the next iteration in each thread block hides much of the global memory access delay. While many threads are waiting on global memory accesses, others process with the necessary calculations for the current data in shared memory. Details of the prefetching methods are shown in Fig. 3.5, the prefetching loop is also unrolled to maximize performance.

## 3.5 Results

We have investigated the performance of our technique on various sparse matrices from [22] with different average non-zeros per row (Table 3.1). The performance of the algorithm is tested on GT8800 NVIDIA graphic cards using CUDA 2.3 and the execution speed of the kernel is represented in GFLOPs (billion floating operations per second). The SMVM kernel is a part of iterative solvers, thus data transfers between host and device memory occur at most twice (at the beginning and the end of iteration) and are neglected over a large number of SMVM operations [49].

In Fig. 3.6 the performance of the proposed technique has been shown. The execution time of the kernel is tested for padding factors of 1, 2, 4 and 8 (the filling ratio of the padded matrices are shown in Table 3.1).

Padding the matrix rows to be multiples of four, increases the performance to 60 percent compared to no padding (padding factor 1). For padding factors larger than

four, the number of zeros added due to padding are increased, decreasing the filling ratio and the SMVM kernel performance. Setting the padding factor to its optimum value (four), Fig. 3.7 shows the effects of prefetching data to hide global memory latency. The results show an average 16 percent increase in performance if each block prefetches and operates on four partitions of 256 value vector elements (Section 3.4.2).

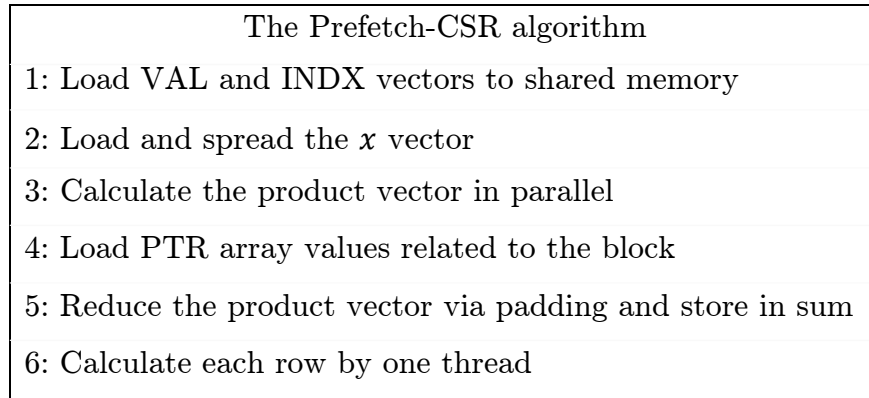


Fig. 3.4: The Prefetch-CSR algorithm.

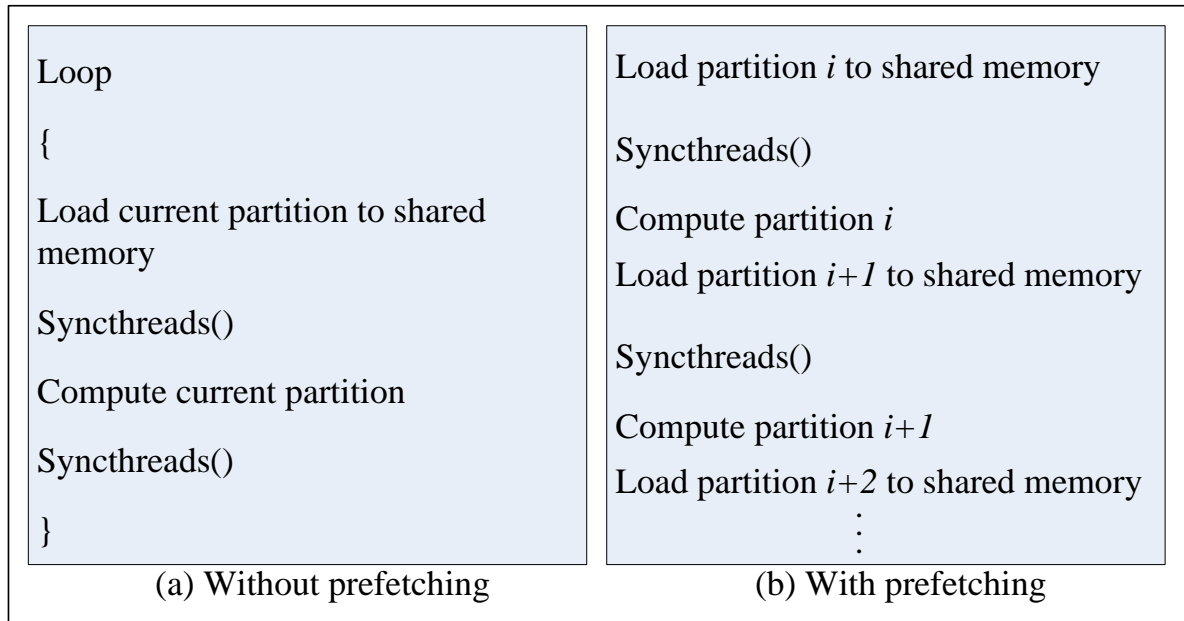


Fig. 3.5: Prefetching data in PCSR (a) without prefetching, (b) with prefetching.

Table 3.1: Non-zeros (nnz) and filling ratio percentage for different padding factors (n) in matrices

Matrix	consph	cant	shipsec	mac-econ	s3dkt3m2
nnz	6010480	4007383	7813404	1273389	3843910
nnz/row	72.1	64.1	55.4	21.24	6.1
n=2	98.8	99.2	98	93.8	97.8
n=4	96.4	98	97.3	80	97.7
n=8	92.2	93.3	96.1	44	68.69

Table 3.2: Speedup of PCSR compared to the row-per-thread and row-per-warp methods on GT8800, the CPU and the Cell architectures.

Matrix	consph	cant	shipsec1	mac-econ	s3dkt3m2	Average
Row thread	3.57	3.71	3.56	2.37	3.64	3.37
Row warp	2.39	2.60	2.26	2.38	2.64	2.45
Cell	5.27	5.04	5.18	5.77	5.41	5.34
CPU	17.03	17.52	18.7	13	18.8	17

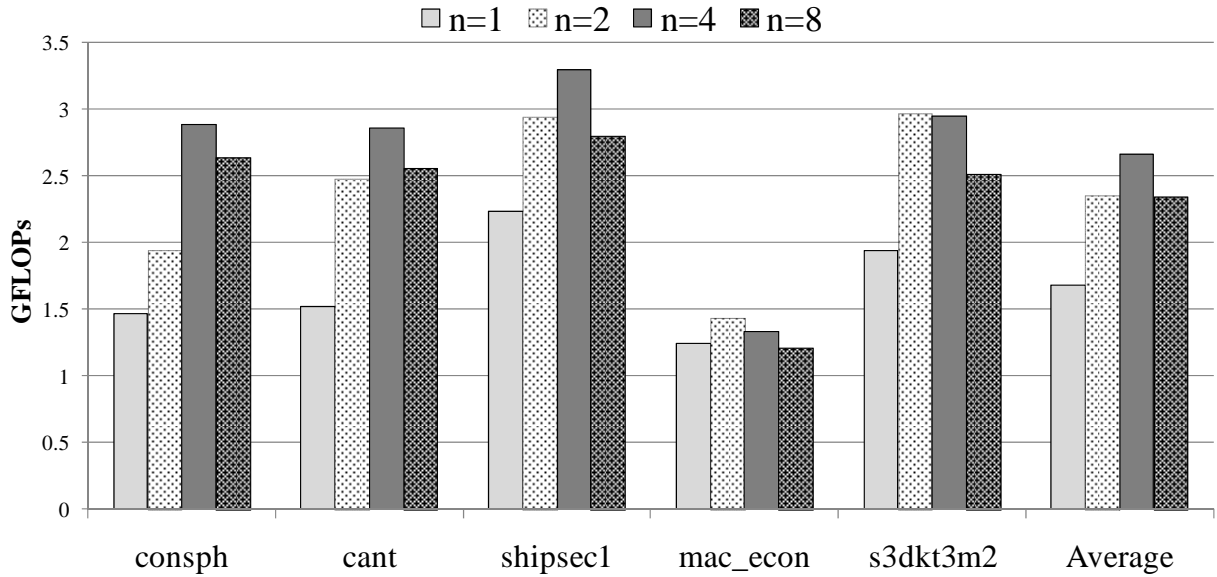


Fig. 3.6: The effect of the padding factor (n) in PCSR.

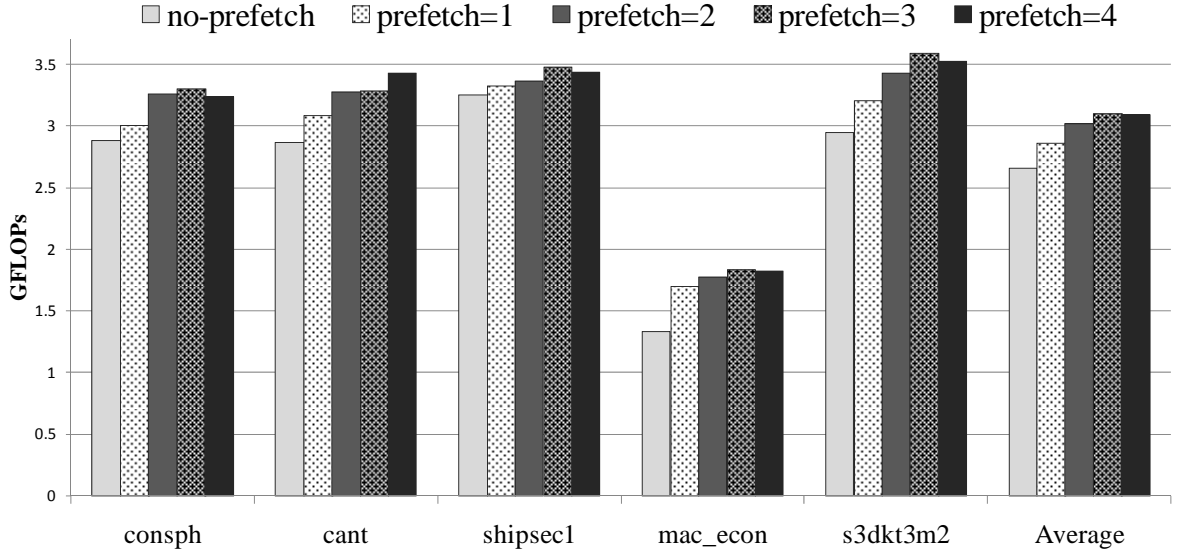


Fig. 3.7: Varying the number of prefetches in PCSR.

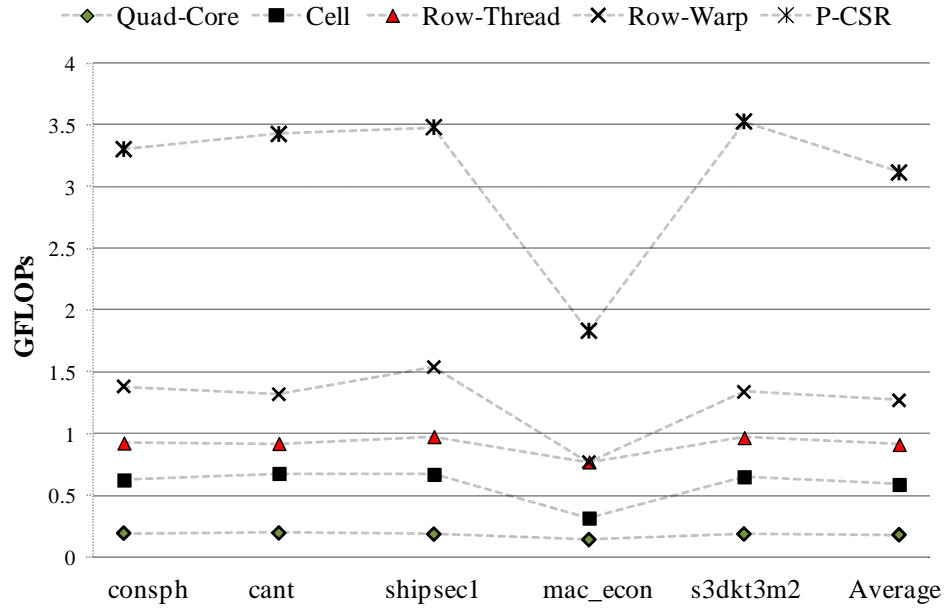


Fig. 3.8: PCSR performance compared to the row-per-thread and row-per-warp methods on GT8800 as well as the QUAD-Core CPU and Cell architectures.

Because of the variety in the memory bandwidth and computation capabilities of different NVIDIA cards, comparisons with other work are done via running their methods on the GT8800. Fig. 3.8 and Table 3.2 provide a comparison of our method to the row-per-warp and row-per-thread methods on GT8800 [49]. The

performance of PCSR is also compared to the execution of the SMVM kernel on a quad-core CPU and the Cell-PPE. The Cell results were obtained using the Cell SDK 3.0 and the PMS method [11]. The CPU platform used was Intel core2 Quad 2.4GHZ architecture with 4 MB of L2 cache per core-pair and 4GB of global DRAM. As shown in Table 3.2, on average our algorithm outperforms the row-per-warp and row-per-thread techniques presented in previous work by 2.45 and 3.37 times respectively. Speedups of up to 18.8 times were achieved compared to the quad core CPU and the execution time was less than what is achieved through optimized SMVM kernel on the Cell.

### 3.6 Conclusion

We have introduced several efficient techniques to accelerate the execution of the sparse matrix vector multiplication on NVIDIA graphic processing units. The proposed methods increased the performance of the SMVM kernel on GT8800 up to 18.8 times compared to the quad core CPU and 3.37 times compared to previous work on accelerating SMVM for GPUs. Reducing the execution time of finite element solvers such as the conjugate gradient method using the proposed optimizations will be investigated in future work.

---

## PREFACE TO CHAPTER 4

The following chapter is included as a paper published by the IEEE Transactions on Magnetics ("Enhancing the Performance of Conjugate Gradient Solvers on Graphic Processing Units", IEEE Trans. on Mag., vol. 47, no. 5, pp.1162-1165, 2011). The Chronopoulos variant of the conjugate gradient method is implemented on NVIDIA GPUs and compared to the Shewchuk variant. Various optimizations such as fusing GPU kernels, binding vectors to caches and SpMV optimizations are used to enhance the performance of the aforementioned kernel on graphic cards. Performances of the proposed optimizations are evaluated on NVIDIA GT8800 and GT200 graphic cards. Performance is also compared to vectorized and non-vectorized parallel implementation of the CG algorithm on Intel multicore architecture.

The convergence rate of iterative solvers such as the conjugate gradient method can be very slow for ill-conditioned matrices. The next chapter accelerates the generation of preconditioners, specifically the sparse approximate inverse preconditioner, used to reduce the number of iterations in iterative solvers. The preconditioner is then used in the BiCGStab iterative solver which is also executed in parallel on the GPU.



## Chapter 4 ENHANCING THE PERFORMANCE OF CONJUGATE GRADIENT SOLVERS ON GRAPHIC PROCESSING UNITS

Maryam Mehri Dehnavi, David M. Fernandez, and Dennis Giannacopoulos

**Abstract:** A study of the fundamental obstacles to accelerate the preconditioned conjugate gradient (PCG) method on modern graphic processing units (GPUs) is presented and several techniques are proposed to enhance its performance over previous work independent of the GPU generation and the matrix sparsity pattern. The proposed enhancements increase the performance of PCG up to 23 times compared to vector optimized PCG results on modern CPUs and up to 3.4 times compared to previous GPU results

**Index terms:** Computer architecture, Graphic processing units, Parallel processing, Conjugate gradient.

### 4.1 Introduction

Real world electromagnetic problems constantly demand more precise and sophisticated simulations in reasonable time frames. To meet such demands in modern finite element method (FEM) applications, programmers must efficiently exploit new technological advancements in modern computing systems. Graphic processing units (GPUs) have evolved very quickly over the last few years and significantly overwhelm CPU specifications in both raw power and memory bandwidth [47]. To benefit from the pervasive computing resources in a GPU, compute intensive data-parallel sections of large problems should be optimized to run on the GPU architecture.

This paper focuses on enhancing the performance of the preconditioned conjugate gradient (PCG) algorithm [7], a popular sparse linear solver in FEM using current GPU processors. Efficient techniques to parallelize PCG on GPUs are presented that overcome the main limitations imposed by both the PCG algorithm

(namely poor data locality and sequential execution), and the programming constraints of modern GPUs (e.g. efficient use of different GPU resources, minimizing data communication, hiding memory access latencies and reducing the number of kernel calls). The effectiveness of these techniques is demonstrated using a range of matrices and speedup results are compared with other state-of-the-art PCG multicore and GPU implementations.

## 4.2 GPU Architecture

Initially driven by the demand for powerful high-definition 3D graphics, modern GPUs have become massively parallel, multithreaded architectures. Easy to learn APIs (Application Programming Interfaces) such as compute unified device architecture (CUDA [43]) has enabled the acceleration of modern scientific applications via massive multithreading. In particular, NVIDIA GPUs offer important computing power for these applications. Fig. 4.1 shows the general architecture of NVIDIA graphic cards. Scalar processors (SPs) are the basic processing units of the architecture and are clustered in groups of eight called streaming multiprocessors (SMs).

Sections of an application that exhibit rich data parallelism are scheduled to run on the GPU. Executing a parallel section on the GPU using CUDA involves: a) transferring required data to GPU global memory; b) launching the device (GPU) kernel; and c) transferring results back to host memory. Threads inside a kernel are grouped into thread blocks, which are executed on SMs. Threads in a block communicate via fast shared memory, but threads in different blocks communicate through long latency global memory. Major challenges in optimizing an application on GPUs are: global memory access latency, different execution paths in each warp (32 consecutive threads in a block) namely thread divergence, communication and synchronizations between threads in different blocks and resource utilization.

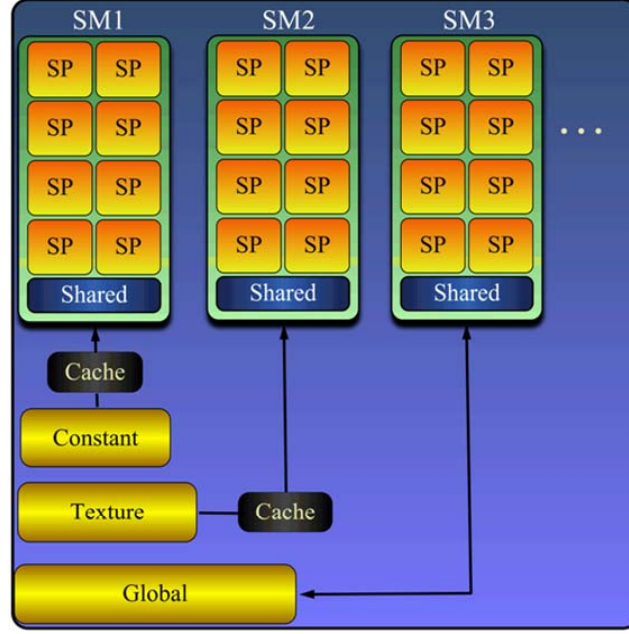


Fig. 4.1: NVIDIA GPU architecture.

### 4.3 Preconditioned Conjugate Gradient

The conjugate gradient (CG) algorithm is one of the most popular iterative linear solvers available today, mainly due to its fast convergence, constant decrease in error-per-iteration and efficient memory usage [7]. Before introducing the parallelization and performance enhancing techniques, one must choose an appropriate PCG version with good parallelization properties as presented in the next section.

#### A. Choosing a PCG algorithm

Many variations of the PCG algorithm exist, depending on their formulation. In this work we implemented a classical PCG algorithm [7] and a variation presented in [8] with better data locality that minimizes the number of kernel calls, the GPU global memory loads, and the communication overhead. Fig. 4.2 presents both algorithms highlighting sections in the main iteration loop where vectors are loaded for the SMVM, SAXPY (vector updates,  $y = \alpha x + y$ ) and dot product operations.

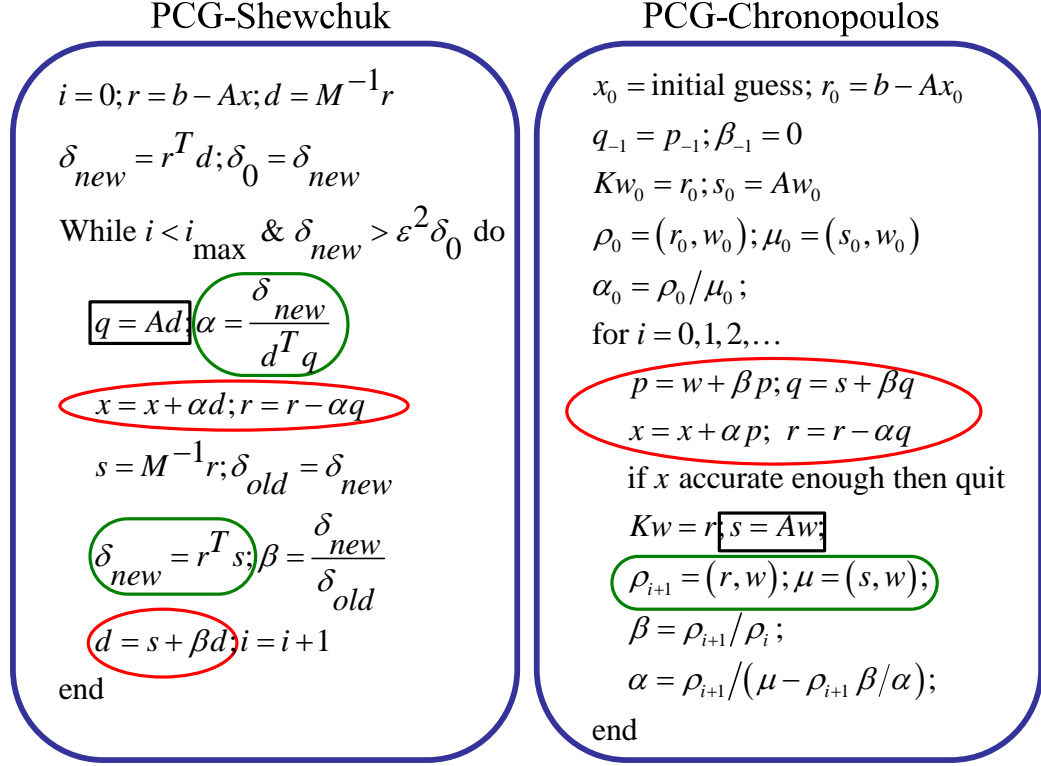


Fig. 4.2: Highlighting several bottleneck operations in PCG Shewchuk [7] vs. PCG Chronopoulos [8].

The main advantages of the Chronopoulos variant of the PCG algorithm compared to the Shewchuk method are as follows:

- In the PCG-Chronopoulos version vectors are loaded in the same place within the main loop as opposed to across the whole loop for the Shewchuk version. This property allows multiple operations to reuse data while on shared memory, reducing long latency memory accesses and exhibiting better data locality.
- Dot products are clustered together in the Chronopoulos variant reducing the number of synchronization steps on both the GPU and the CPU.
- Efficient partitioning of vector and matrix values enables coalesced loading of data and maximum GPU resource utilization during PCG kernel calculations.

### *B. Previous work*

Accelerating the PCG algorithm on massively parallel hardware platforms, especially GPUs, is very challenging due to the sequential nature of the algorithm. Buatois et al. [47] accelerated the CG solver on GPUs using the blocked compressed sparse row storage (BCSR) format. Their algorithm is optimized for a limited set of matrices with specific sparsity patterns. Wiggers et al. [24] reorder matrix rows to decrease the execution time of the SMVM kernel for CG. Sorting rows increases pre-processing and execution time on the CPU. In [50] and [51] a mixed precision iterative refinement algorithm is proposed for the CG. The single precision inner solver in their method is the most time consuming kernel in the overall solution and accelerating its execution is the major focus of our work.

The performance of SMVM using various compressed storage formats on GPUs has been studied in [49]. Using a decision based method, [52] chooses the best performing storage format for SMVM from [49] prior to executing the CG algorithm, at the expense of storing (generating) several copies of the matrices in the various storage formats. Formats such as JDS [9], HYB [52], ELL [49], BCSR [47] require extra pre-processing to benefit from the GPU processors (sorting rows, blocking non-zero values, redundant padding, etc.) that are not negligible compared to the fast execution time of the SMVM and CG algorithms on the GPU. The Prefetch-CSR (PCSR) algorithm proposed in [53] requires very little padding and pre-processing and outperforms the previous SMVM algorithms including one of the best performing algorithms, namely the row-per-warp method from NVIDIA [49]. By using an optimized version of the PCSR algorithm and the row-per-warp method this paper proposes new techniques to overcome major bottlenecks in accelerating PCG on GPUs.

#### 4.4 Implementing PCG on GPUs

We propose four optimizations to the original Chronopoulos PCG in order to decrease its execution time on the GPUs. Without optimization, implementing the Chronopoulos variant of the PCG algorithm leads to eight kernels and some scalar updates on the CPU (Fig. 4.3). Fig. 4.4a shows the percentage of average time spent on each of these kernels in the naive implementation of the PCG algorithm on the GPU. We enhance the performance of the Chronopoulos PCG by optimizing the SMVM kernel, fusing SAXPY operations, using a Jacobi preconditioner and binding vectors to GPU texture memory.

##### *A. Optimizing the SMVM kernel*

As shown in Fig. 4.4a on average 80 percent of the total PCG execution time is spent on the SMVM kernel, thus using the best performing SMVM algorithm is essential in decreasing PCG execution time. In this work we compare the effects of two of the best performing SMVM algorithms proposed in previous work [49], [53] in the PCG algorithm. The first algorithm is the row-per-warp method introduced by Bell et al. [49] and the prefetch compressed row storage (PCSR) [53] is the second SMVM method used. Unlike SMVM algorithms based on other storage formats, the row-per-warp method and PCSR do not require extra pre-processing since they are based on the CSR format.

- *Row-per-warp*

In the row-per-warp [49] method each warp is assigned a row to compute one vector result. The method is efficient if the sparse matrix has a regular sparsity pattern and its bandwidth is approximately equal to a multiple of a warp size.

- *Prefetch-CSR (PCSR)*

The PCSR method [53] partitions the matrix non-zeros to blocks of the same size and distributes them amongst GPU resources. The algorithm pads rows with zeros

to increase data regularity and use of parallel reduction techniques. Prefetching data is also used to hide global memory accesses. To further increase the performance of the algorithm, in this work we have eliminated the atomic updates of the  $Y$  vector by replacing the original SMVM kernel with three sub-kernels, namely, *clear  $Y$  vector*, SMVM and  *$Y$  vector update* (Fig. 4.3). Thus in the optimized version of PCSR, atomic sums of the  $Y$  vector values corresponding to partitioned rows between blocks are removed (the two added kernels, clear  $Y$  vector and  $Y$  vector update are small and have a fast execution time compared to the SMVM kernel).

### B. Jacobi preconditioner

A Jacobi preconditioner was implemented mainly for its ease of parallelization in the PCG method. As an additional benefit, because the solver for this type of preconditioner can be treated as a SAXPY operation, it can be fused with other operations as described in the next section.

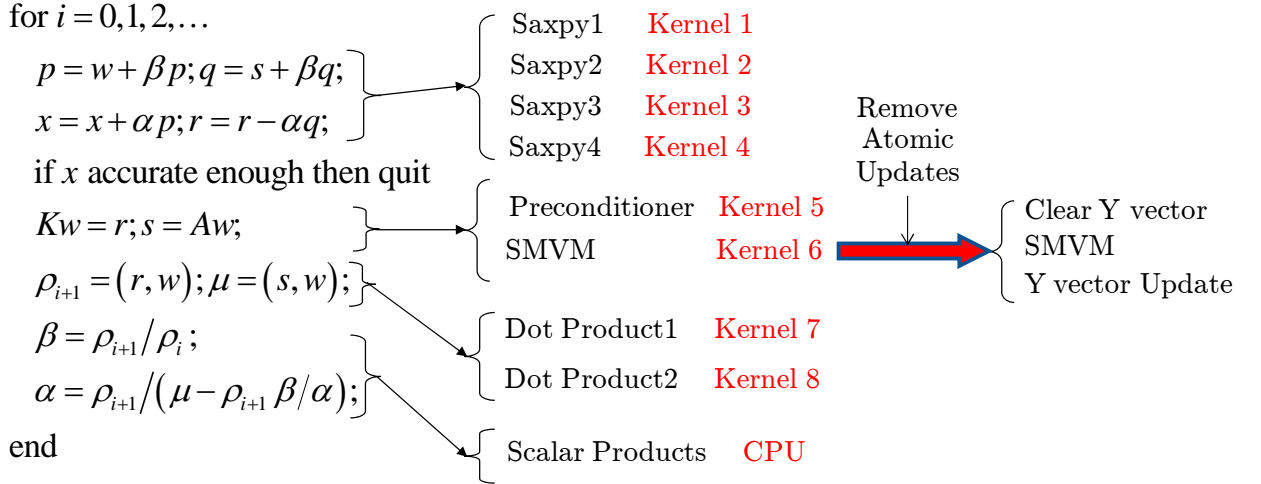


Fig. 4.3: PCG Chronopoulos [8] algorithm implemented on the GPU, optimizing PCSR [53] adds two new kernels to the implementation.

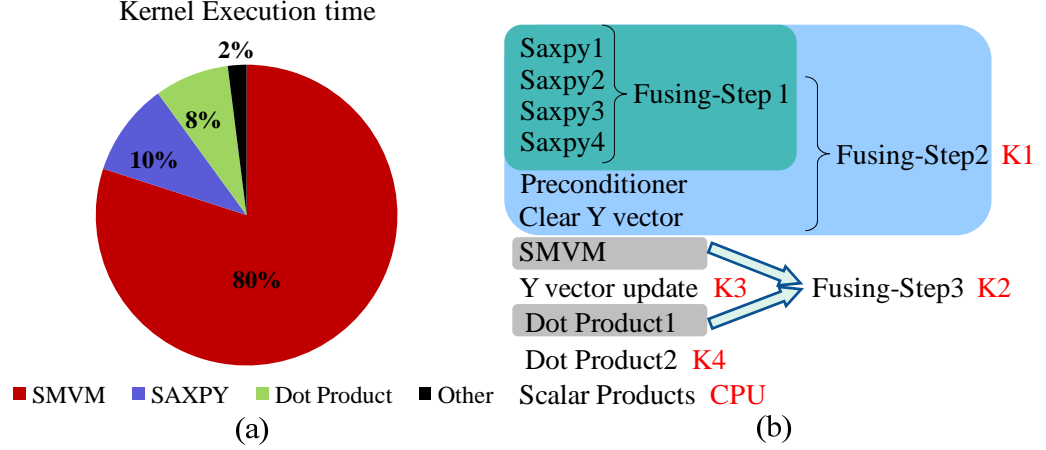


Fig. 4.4: (a) Percentage of the average execution time of kernels in the PCG Chronopoulos, (b) Fusing kernels in PCG (K1 to K4 represent the kernels in optimized PCG).

### C. Fusing kernels

Although the PCG algorithm is mainly implemented on the GPU in previous work, gathering result vector values and performing vector dot products require going back to the CPU, resulting in multiple kernel calls. In each kernel call data is loaded to fast access GPU shared memory in partitions. Upon termination of a kernel, all data is stored back to the GPU global memory, requiring proceeding kernels to reload data to shared memory before their execution. Thus, besides the launching time of each kernel, increased communication is another major drawback of multiple kernel calls.

There are two objectives of fusing individual kernels, the first is to minimize the number of kernels, saving time between kernel calls; and the second is to take advantage of the vectors loaded into shared memory avoiding double loads. The fusions are done in three steps (Fig. 4.4b). In the first step the SAXPY kernels are fused into a single kernel. The second step fuses the preconditioner and clear  $Y$  vector kernels into the SAXPY kernel. The dot product and the SMVM kernels are fused into one kernel in the last step (scalar updates of the dot product are still performed on the CPU).



Fusing reduces the total number of kernels from 8 to 4 in the PCG algorithm. Although optimized implementations of other PCG algorithms might result in small number of kernels, the resulting kernels after fusion in the proposed method have significant implications leading to increased performance:

- Most vectors are only loaded once onto shared memory per iteration.
- Fusing the main operations in the PCG algorithm into one kernel (K1 in Fig. 4.4b) increases coalesced memory fetches reducing global memory accesses.
- Kernels 3 and 4 (K3 and K4 in Fig. 4.4b) are small and do not require large number of memory loads.

#### *D. Texture binding*

The texture memory is a fast on-chip cached memory space. Loading vectors to texture memory decreases the effect of global memory access latencies and enhances the performance of the PCG algorithm kernel. We bind vectors that benefit the most from the cached space to texture memory. By binding vectors to texture memory we increase the execution speed of the PCG algorithm. Since vector values in PCG are updated in each iteration, vectors need to be binded/unbinded to/from texture memory in each iteration.

### **4.5 Results**

The performance of the optimizations proposed is evaluated using 7 sparse matrices from [34] with different sparsity patterns and application areas (Table 4.1). The execution speed of the PCG algorithm is presented in GFLOPs (billion floating point operations per second). For each PCG Chronopoulos iteration, the algorithm computes one SMVM and 7 vector operations, thus  $2 \times nnz + 14 \times n$  flops plus scalar updates are counted ( $nnz$ : number of non-zeros,  $n$ : matrix dimension).

The performance of the optimized algorithm is tested on two different generations of NVIDIA graphic cards the G80 and GT200 series. NVIDIA GT8800

and GTX280 graphic cards are used as representatives of the G80 and GT200 series, respectively. The GTX280 consists of 30 SMs, 16K registers and 1GB of global memory compared to the 14 SMs, 8K register file and 512MB of device memory on the GT8800. Both GPUs have 16KB of shared memory but the GT8800 operates at a higher frequency (1.5GHZ vs. 1.29GHZ). The GT200 generation has higher compute capabilities and handles thread divergence more efficiently while the maximum graphic card power and average cost of the GTX280 is approximately double that of the GT8800 card.

Fig. 4.5 shows the effect of the optimizations proposed in Section 4.4 step by step. Using the row-per-warp algorithm as the SMVM kernel, the PCG Chronopoulos method outperforms the Shewchuk algorithm for all the matrices. By replacing the row-per-warp SMVM with the optimized version of PCSR the average performance of the PCG algorithm increases up to 60 percent as shown in Fig. 4.5. While using PCSR as the SMVM kernel, binding vectors to texture memory increases performance on average 50 percent. Fusing SAXPY operations increases performance on average 6 percent compared to the non-fused version (Fig. 4.4b). The two other fusing steps also contribute to an average 6 percent increase in performance.

Fig. 4.6 shows the performance of the optimized PCG algorithm compared to the row-per-warp method [49] on both G80 (GT8800) and GT200 (GTX280) NVIDIA GPU generations. The proposed algorithm outperforms previous methods on both platforms. Unlike previous methods [49], [52] which are not optimized for matrices with small number of non-zeros per row, the proposed optimizations, independent of the matrix sparsity pattern, are able to increase considerably the performance for such matrices.

Table 4.1: Sparse matrices used for testing

Matrix Name	Matrix Type	Rows	nnz	nnz/row
thermal2	FEM/steady state	1228045	8580313	7
shipsec5	PARASOL ship	179860	10113096	56
g3-circuit	Circuit simulation	1585478	7660826	5
BenElechi1	2D/3D problem	245874	13150496	53
2cubes-sphere	FEM/sphere	101492	1647264	16
s3dkt3m2	FEM/cyl. shell	90449	3753461	41
mt1	Tubular joint	97578	9753570	100

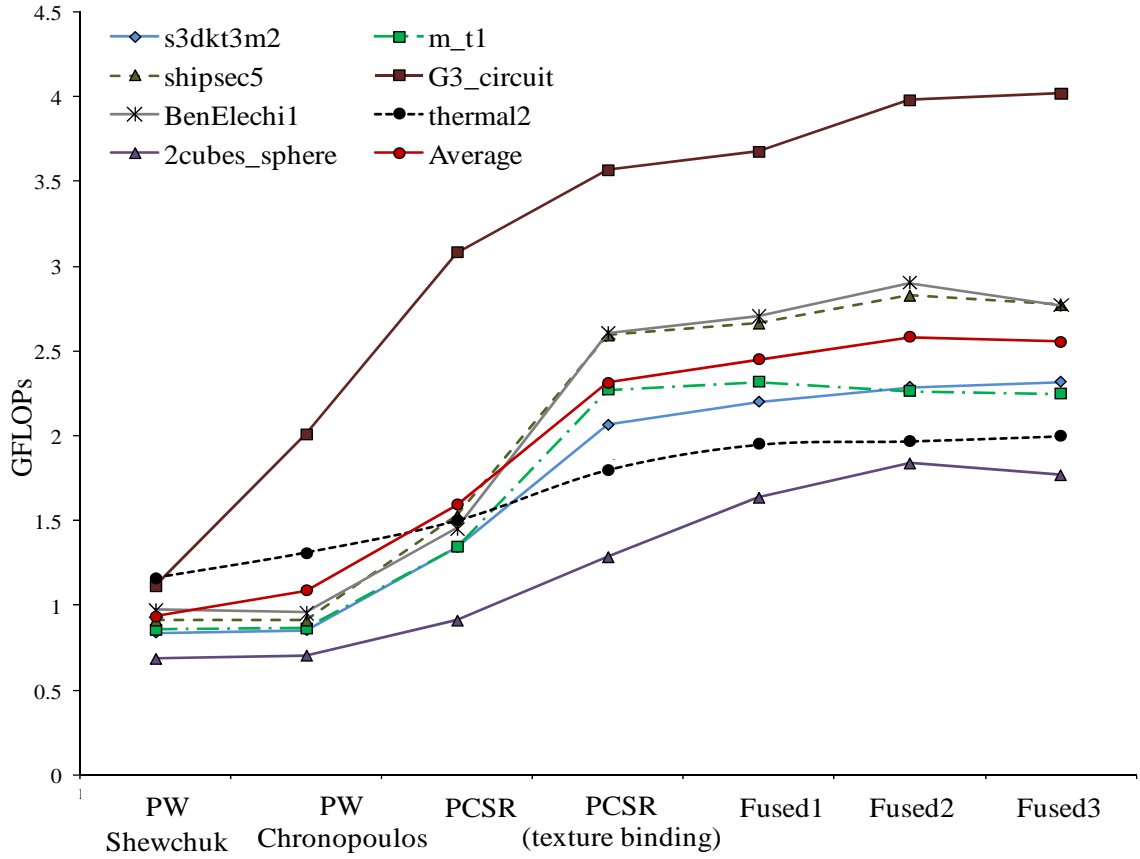


Fig. 4.5: The effect of the optimizations proposed in Section 4.4 in increasing the performance of the PCG algorithm on GT8800.

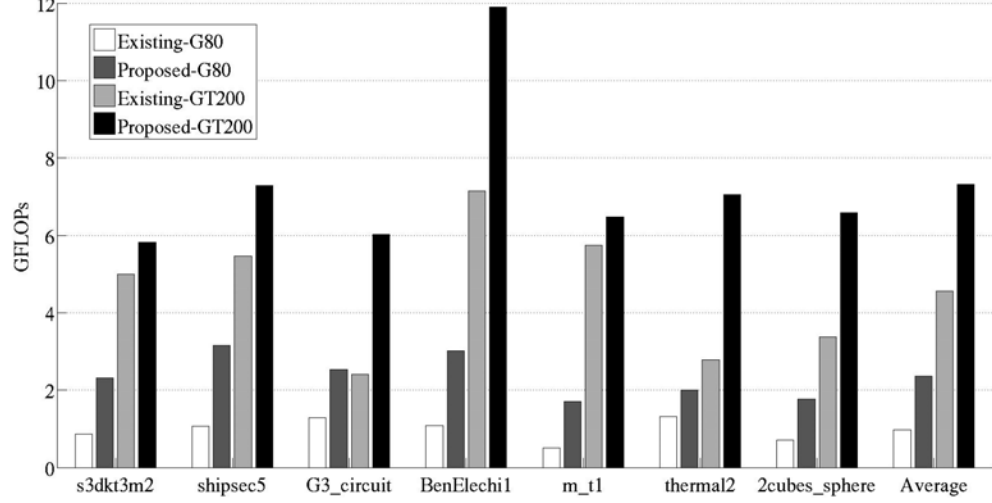


Fig. 4.6: Performance of the PCG row-per-warp [49] method compared to proposed optimized PCG Chronopoulos [8] algorithm on G80 and GT200.

Table 4.2 presents the speedup (SU) of the proposed method compared to the row-per-warp (RW) method implemented on the G80 and G200 architectures, the best vectorized CPU results in [12] as well as a naive CPU implementation. A majority of SMVM algorithms proposed in previous work such as the row-per-warp method introduced in [49] rely on the architecture to address thread divergence, thus PCG algorithms using such methods do not perform well on the G80 generation of NVIDIA GPUs. Since PCSR’s performance is independent of the GPU generation, our PCG implementation outperforms the PCG version of the row-per-warp method on both GPU generations (Table 4.2). Compared to vectorized PCG [12] using 4 threads on an Intel core2 Quad 2.4GHZ architecture (4 MB of L2 cache per core-pair and 4GB of global DRAM) speedups of up to 23 were achieved (Table 4.2). On average 42 times speedup was achieved compared to non-vectorized PCG using a single thread on the same CPU (“CPU Regular” results in Table 4.2).

Compared to single GPU results in [52] (their method uses an SMVM decision algorithm to choose the best performing storage format for each matrix, increasing pre-processing time), for the same matrices we achieve on average 1.5 times speedup

(for similar matrices g3-circuit, thermal2, and BenElechi1 speedups of 1.5, 2 and 1.1 are achieved respectively). Thus the proposed PCG optimizations can, potentially, give average performances of up to 180 GFLOPs on multi-GPU platforms compared to 120 GFLOPs in [52].

#### 4.6 Conclusion and Future Work

The paper introduces several optimizations for the Chronopoulos [8] PCG variant to accelerate the execution of PCG on GPUs. The proposed optimizations increased the performance of PCG on representatives of the G80 and GT200 generations of NVIDIA GPUs up to 3.4 and 2.5 times, respectively, compared to previous methods [49]. In future work we intend to extend our algorithm to multi-GPU platforms and other preconditioners.

Table 4.2: Speedup of the optimized PCG compared to PCG-row-per-warp (RW) on GPU, vectorized and non-vectorized CPU

Overall Speedup	RW G80	RW GT200	Quad-Core	CPU Regular
s3dkt3m2	2.7	1.16	11.11	30.65
shipsec5	2.97	1.33	14.02	72.92
g3-circuit	1.95	2.49	13.25	26.18
BenElechi1	2.79	1.66	23.32	56.65
mt1	3.4	1.12	7.45	34.12
thermal2	1.52	2.53	9.39	41.22
2cubes-sphere	2.5	1.95	11.99	31.35
average	2.55	1.75	12.93	41.87

---

## PREFACE TO CHAPTER 5

The following chapter is included as a paper accepted for publication in IEEE Transactions on Parallel and Distributed Systems (“Parallel Sparse Approximate Inverse Preconditioning on Graphic Processing Units”). The sparse approximate inverse (SAI/SPAI) preconditioner is accelerated on NVIDIA GPUs. The preconditioner is then used to enhance the convergence rate of the BiCGStab iterative solver which is also implemented on the GPU. The performance of the proposed acceleration is compared to ParaSails, a popular implementation of SAI preconditioners on multiprocessors. The work was done in collaboration with Professor Jean-Luc Gaudiot at UC-Irvine. A more aggressive approach in reducing the communication cost of Krylov solvers on NVIDIA GPUs known as the communication-avoiding Krylov techniques is studied in the next chapter.

---

## Chapter 5 PARALLEL SPARSE APPROXIMATE INVERSE PRECONDITIONING ON GRAPHIC PROCESSING UNITS

Maryam Mehri, David M. Fernandez, Jean-Luc Gaudiot and Dennis Giannacopoulos

**Abstract:** Accelerating numerical algorithms for solving sparse linear systems on parallel architectures has attracted the attention of many researchers due to their applicability to many engineering and scientific problems. The solution of sparse systems often dominates the overall execution time of such problems and is mainly solved by iterative methods. Preconditioners are used to accelerate the convergence rate of these solvers and reduce the total execution time.

Sparse approximate inverse (SAI) preconditioners are a popular class of preconditioners designed to improve the condition number of large sparse matrices and accelerate the convergence rate of iterative solvers for sparse linear systems. We propose a GPU accelerated SAI preconditioning technique called GSAI, which parallelizes the computation of this preconditioner on NVIDIA graphic cards. The preconditioner is then used to enhance the convergence rate of the biconjugate gradient stabilized (BiCGStab) iterative solver on the GPU.

The SAI preconditioner is generated on average 28 and 23 times faster on the NVIDIA GTX480 and TESLA M2070 graphic cards respectively compared to ParaSails (a popular implementation of SAI preconditioners on CPU) single processor/core results. The proposed GSAI technique computes the SAI preconditioner in approximately the same time as ParaSails generates the same preconditioner on 16 AMD Opteron 252 processors.

**Index terms:** Numerical algorithms; Parallel algorithms; Graphics processors; Parallel programming; Conditioning.

## 5.1 Introduction

Mathematical physics and engineering problems in a broad range of applications (such as computational electromagnetics, medical and seismic tomography, heat conduction, computational fluid mechanics, etc.) have grown larger and more complex in the past few decades leading to large scale simulations. These simulations generally involve the use of techniques such as the finite element method (FEM) and the finite difference time domain (FDTD) method which are used to discretize, assemble and solve such systems [5], [54]. One of the most time consuming steps in the aforementioned techniques is solving the system of equations proceeding the systems assembly stage. The solution of such systems is often achieved by sparse linear systems and can be obtained by either direct or iterative methods. For larger and sparser systems, direct methods often suffer from high computational complexity and intensive memory requirements. Techniques such as Gaussian elimination, Choleski, LU and QR factorizations [5] are designed to address some of these issues and thus reduce the complexity of computations and required storage in this class of solvers. Direct solvers are notoriously difficult to implement in parallel due to the recursive nature of their computations such as solving large triangular systems [55].

A more viable alternative to solving large linear systems is using iterative solvers. Krylov methods are a popular class of these solvers with techniques such as generalized minimum residual (GMRES), biconjugate gradient (BiCG), biconjugate gradient stabilized (BiCGStab) and conjugate gradient (CG) [5]. Krylov solvers generally involve less computations and memory requirements compared to direct methods. A major limiting factor of iterative solvers is their slow convergence rate especially for ill-conditioned matrices. The convergence rates of most iterative solvers heavily depend on the eigenvalues distribution of the  $A$  matrix when solving the linear system of equations  $Ax = b$  [56]. By clustering the eigenvalues or reducing



the condition number of the matrix, the convergence rate of iterative solvers is improved considerably.

Preconditioners are designed to accelerate the convergence rate of iterative solvers for a majority of applications. Applying the preconditioner  $M$ , to both sides of the linear systems equation  $Ax = b$ , reduces the number of iterations and accelerates the execution time of the solver. Although a considerable number of preconditioning techniques have been developed in previous work [56], e.g., incomplete cholesky (IC), diagonal preconditioners, successive over relaxation (SOR), polynomial preconditioners and sparse approximate inverse (SAI) preconditioners, researchers have not been able to develop an efficient general-purpose preconditioner. A preconditioner is defined as good if it is easy to construct, cheap to store and accelerates the solvers from a broad range of problems. A good preconditioner should also be easy to parallelize and well-suited for modern architectures.

A popular class of preconditioners suitable for parallelization and efficient for a large class of problems is sparse approximate inverse preconditioners. Although computing SAI preconditioners is generally expensive on a single processor, constructing them on parallel architecture is relatively fast. By generating a denser preconditioner, SAI preconditioning can reduce iterations in iterative solvers considerably and be applied to a broad range of applications. Previous work has accelerated the computation of this preconditioner on multiple processors [57], [58], [59], [60], [61], [62], [63], [64], [65] as well as multicore [66], [67] and manycore architecture [68].

Graphic processing units have become an important resource for scientific computing in recent years [69]. With easy to learn APIs (Application Programming Interfaces) such as CUDA [43] (Compute Unified Device Architecture) introduced by NVIDIA, general purpose programming for modern scientific computations on

GPUs gained considerable attention. The GPU consists of streaming multiprocessors (SMs) and each SM contains basic processing units called scalar processors (SPs). To run compute intensive parts of an application on the GPU, initial data has to be transferred from CPU memory to GPU global memory and a GPU kernel is then launched. Using a single data multiple thread paradigm, GPU threads grouped into thread blocks proceed with the computations and transfer the results back to the CPU. The GPU consists of an on-board global memory with long access latency, a fast access shared memory, registers and caches. Threads inside a block communicate via shared memory and their execution can be synchronized. Every 32 threads in a block execute the same instruction and are called a warp.

In this work we present a new GPU accelerated SAI preconditioning technique called GSAI, which parallelizes the computation of sparse approximate inverse preconditioners on NVIDIA GPUs. Major contributions of the proposed GSAI technique are as follows:

- Each GPU warp computes one column of  $M$  and the preconditioner is generated in parallel on the GPU.
- Large data structures are stored in GPU global memory and memory space is reused by dividing the computation of  $M$  between many GPU kernels.
- Memory accesses, vector multiplications and inner products are computed in parallel inside a GPU warp. QR decomposition and triangular solve kernels are also computed in parallel inside each warp via 32 threads.

The preconditioner is assembled in a compressed storage format and then used to solve  $Ax = b$  via the preconditioned BiCGStab iterative solver, which is also accelerated on the GPU.

## 5.2 Sparse Approximate Inverse (SAI) Preconditioning

A sparse approximate inverse preconditioner approximates the inverse of  $A$  using a sparse matrix  $M$  to improve the condition number of the linear system of equations  $Ax = b$ .  $M$  is computed using the least-squares methods and by minimizing the matrix residual norm

$$\|AM - I\|_F^2 \quad 5.1$$

The above equation is then separated into  $n$  independent least square problems

$$\min_{m_k} \|Am_k - e_k\|_2^2, \quad k=1,2,\dots,n \quad 5.2$$

where  $e_k$  is the  $k$ th column of the identity matrix and  $m_k$  represents column  $k$  in matrix  $M$ . The degrees of freedom in solving the above equations are the locations and values of the non-zeros in  $M$ . Based on the degree of freedom used, sparse approximate inverse preconditioner generation is classified as adaptive or static (*a priori*). In adaptive schemes ([61], [70], [71], etc.) the sparsity of  $M$  is initially set to a simple pattern such as diagonal, this pattern is then augmented until a threshold on the residual norm or a maximum on the number of non-zeros in  $M$  is reached. Although adaptive methods have broadened the scope of problems which can be solved using SAI preconditioning, by utilizing additional degrees of freedom in minimizing equation 5.2, the preconditioner generation becomes generally very expensive requiring many reruns to determine the appropriate values of various parameters involved, such as tolerance [72], maximum improvements per step [21], number of non-zeros per step [72], etc. for each problem. On the other hand, static preconditioning ([58], [64], [72], [73], [74]) determines the sparsity of  $M$  in a pre-processing step limiting the degrees of freedom in 5.1 to the non-zero values of  $M$ .

Previous work has introduced various techniques to determine a more accurate approximation of  $M$  prior to computing the preconditioner and have shown that static schemes are more efficient than adaptive techniques in improving the

condition number of the  $A$  matrix if the sparsity of  $M$  is better approximated. Since the focus of this work is not to introduce a better initial guess for the  $M$  preconditioner but to accelerate the computation of  $M$  (equation 5.2), for general static (a priori) SAI preconditioners, we use the most popular approximate of  $M$  which is based on sparsifications [75] of  $A$ .  $M(i,j)$  is considered a non-zero if the condition

$$|A(i,j)| > (1 - \tau) \max_j |A(i,j)|, \quad 0 \leq \tau \leq 1 \quad 5.3$$

is satisfied, where  $\tau$  is a user defined tolerance parameter (the main diagonal is always included). Based on equation 5.3, for smaller  $\tau$  parameters more non-zeros entries in  $A$  are dropped resulting in a sparser preconditioner; for  $\tau$  equal to 1 the sparsity pattern assumed for  $M$  would be the same as the sparsity of  $A$ . If a more accurate approximate of the sparsity of  $M$  is known for a specific application it can be used instead of equation 5.3. Knowing the sparsity of  $M$  before solving equation 5.1, reduces equation 5.2 to

$$\min_{\hat{m}_k} \|\hat{A}\hat{m}_k - \hat{e}_k\|_2^2, \quad k = 1, 2, \dots, n \quad 5.4$$

$\hat{m}_k$  is the reduced vector of unknowns  $m_k(J)$ , where  $J$  is the set of indices  $j$  such that  $m_k(j) \neq 0$ . Considering  $I$  as a set of indices  $i$  such that  $A(i,J)$  is not zero,  $\hat{A}$  is the submatrix  $A(I,J)$  where all zero rows in  $A(.,J)$  are deleted. The dimension of  $\hat{A}$  is equal to  $n_1 \times n_2$  where  $n_1$  and  $n_2$  are the number of indices in  $I$  and  $J$  respectively. Finally  $\hat{e}_k$  represents  $e_k(I)$ . To construct and solve equation 5.4 for each column  $k$  of  $M$ , the steps in Fig. 5.1 should be computed for each  $k$  (more information on the above implementations and the steps in Fig. 5.1 can be found in previous work on SAI preconditioners specifically [58], [59],[61], [63], [72]).

- a)  $J$  is constructed based on (3)
  - b) Columns of  $A$  in  $J$  are selected and matched to construct  $I$
  - c)  $\hat{A}$  is constructed and decomposed using QR Gram-Schmidt [5]
  - d) Values in  $\hat{m}_k$  are computed using  $\hat{m}_k = R^{-1}Q^T \hat{e}_k$  and scattered back to  $M$ .

Fig. 5.1: Steps involved in constructing static sparse approximate inverse preconditioners.

Factorized sparse approximate inverse (FSAI) preconditioners are another class of SAI preconditioning techniques initially introduced by [5], which have been developed in [76], [77], [78], [79], [80], [81], [82]. This class of preconditioners are less popular than the kind based on Frobenius norm minimization (equation 5.1) [56] and can fail due to breakdowns during an incomplete factorization process. FSAI preconditioners are constructed to preserve the symmetric properties of the preconditioned problem and are generally applied to the conjugate gradient iterative solver. A comparative study of various SAI preconditioners is presented in [83].

*Sparsification* is a method used to diminish the pattern of  $A$  when it is relatively full and generate a sparser preconditioner and can be implemented in both adaptive and static SAI preconditioner construction algorithms. Initially introduced by Kolotilina [75] for computing SAI preconditioners for dense matrices, sparsification is also used by [84] to enhance the condition number of anisotropic problems via adaptive SAI preconditioners. Costgrov et al. [85] also propose augmenting the pattern of  $A$  for constructing sparse approximate inverse preconditioners. SAI preconditioner proposed by [86] and *ParaSails* [59] introduced by Chow [58] use a priori sparsity patterns based on powers of sparsified matrices for partial differential equation (PDE) problems. Sparsification is also implemented in SPAI 3.2 [72] by eliminating small values in  $A$  before computing the preconditioner. The equation used in the proposed GSAI technique (equation 5.3) also allows for sparsifying  $A$  using a tolerance parameter  $\tau$ . Applying sparsification to the preconditioner after it has been produced is also studied in [76], [86]. If an effective sparsification is known

for a specific problem it can be added to the Pre-GSAI stage (Fig. 5.2) in the GSAI method proposed.

Most of the work on SAI preconditioners presents techniques to parallelize the computation of the preconditioner on multi-processor architectures [57], [58], [59], [60], [61], [62], [63], [64], [65], by distributing the computation of the columns in  $M$  between multiple processors. Techniques such as grouping communications [63], dictionary based methods [60] and latency-tolerant hybrid SAI preconditioning [62] are proposed in these works, to further enhance the execution time of SAI preconditioners on multiprocessors. ParaSails [59] and SPAI 3.2 [72] are two of the most popular open source implementations of the sparse approximate inverse preconditioner on single and multi-processor platforms and are used for comparison in a majority of previous work [56], [58], [60], [62]. While ParaSails uses a priori approximation of  $M$  to generate the preconditioner, both adaptive and static SAI preconditioners are implemented in SPAI 3.2. Similar to SPAI 3.2 the preconditioned problem in GSAI is solved using the BiCGStab iterative solver (ParaSails implements the GMRES and CG iterative solvers). Chow et al. [58] compare the performance of ParaSails to SPAI 3.2 and show ParaSails generates the SAI preconditioner considerably faster than SPAI 3.2. We compare the preconditioner generation time of the proposed GSAI algorithm on GPUs to ParaSails on single and multi-processor platforms.

Although parallelizing sparse approximate inverse preconditioners on more than one processor has been extensively studied in previous work which succeeded to enhance the execution speed of such preconditioners considerably, few works have studied the possibility of accelerating these preconditioners on multi/many core architectures. Gravvanis et al. [66], [67] attempt to accelerate a sparse approximate inverse preconditioned BiCGStab iterative solver on Intel multicore architecture by allocating the computation of each iteration of the iterative solver to a different

thread; implementation details on how to accelerate the preconditioner computation on a multicore are not presented in this work. Xu et al. [68] accelerate factorized SAI on NVIDIA GPUs. The paper mainly describes how to accelerate the sparse matrix vector multiplication kernel (SpMV) in the iterative solver but details for computing the sparse approximate inverse preconditioner have not been presented (other accelerations of the SpMV kernel are presented in [53], [49] and CUSPARSE [89]).

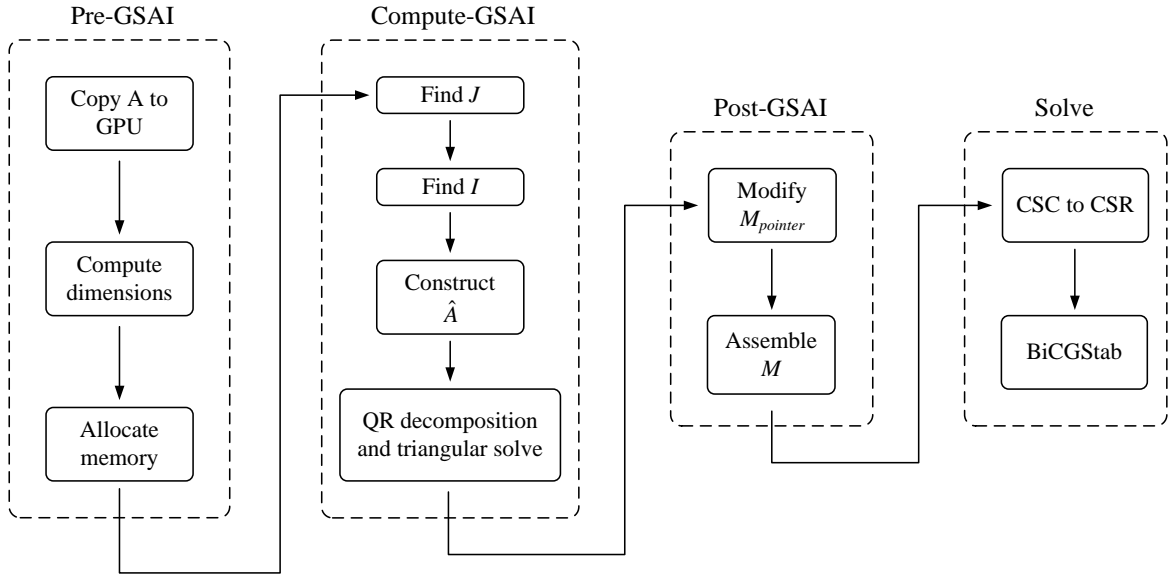


Fig. 5.2: The four stages in implementing SAI preconditioners using GSAI on NVIDIA GPUs .

### 5.3 Parallel SAI in NVIDIA GPUs

The SAI preconditioner is computed in parallel on graphic cards by allocating the computation of each column of  $M$  to one warp. Accelerating the SAI preconditioner involves local (per warp) parallelization of various computing kernels such as QR decomposition, dot products, sorting vector values, finding the maximum value in a vector, etc. One of the major challenges in computing SAI preconditioners on GPUs is the limited size of global and shared memory and the generation of large data structures required and produced by the SAI

preconditioning algorithm. Proposing techniques to free/reuse memory space and minimize the allocated memory to various data structures in the kernel are key factors in producing sparse approximate inverse preconditioners for large problems on GPUs. In the following implementation details to overcome the above constraints and implement in parallel the computing kernels involved in solving  $Ax = b$  using SAI preconditioners are presented.

Computing the SAI preconditioner in parallel on GPUs involves the implementation of steps introduced in Fig. 5.1, which we implemented in a stage called Compute-GSAI (Fig. 5.2). In this stage every 32 threads (one warp) on the GPU computes one column of  $M$  ( $m_k$ ) by executing the steps in Fig. 5.1. Each warp first finds the dimensions of its corresponding  $\hat{A}$  matrix (equation 5.4) and assembles it. The local  $\hat{A}$  matrices, which are very small compared to  $A$ , are then decomposed (local decompositions per warp for each  $\hat{A}$ ) using the Gram Schmidt method [5] and  $m_k$  is computed. SAI preconditioning on GPUs requires two additional steps (Pre-GSAI and Post-GSAI) which handle GPU memory allocation, define required data structures, gather results and determine the required number of kernel calls based on the problem size and available GPU memory. Thus solving the  $Ax = b$  linear systems equations on the GPU using SAI preconditioners consists of four major steps (Fig. 5.2):

- 1) *Pre-GSAI*: involves reading  $A$  in a compressed sparse format [88] and transferring it to GPU, allocating GPU memory space to the preconditioner  $M$  and other data structures and determining the number of kernel calls based on the available global memory space.
- 2) *Compute-GSAI*: computes the sparse approximate inverse preconditioner on the GPU and scatters the produced columns back to  $A$  on GPU global memory.
- 3) *Post-GSAI*: revises the assigned global memory space to  $M$  by releasing extra memory space allocated to  $A$  and assembles  $A$  on the GPU in compressed



column storage (CSC) [88] format.

- 4) *Solver*: converts both  $M$  and  $A$  from CSC to CSR (compressed row storage [88]) on the GPU to accelerate the iterative solver execution time and solves  $Ax = b$  using the computed SAI preconditioner and the BiCGStab iterative solver.

The rest of this section is organized as follows; Section 5.3.1 introduces implementation details of the above steps and the kernel/function calls involved in each stage. Managing global and shared memory, determining the amount of memory required for each data structure and deciding the necessary number of kernel calls are proposed in Section 5.3.2.

### 5.3.1 GSAI Steps

The proposed GSAI preconditioning method computes the SAI preconditioner on NVIDIA GPUs in three major steps namely Pre-GSAI, Compute-GSAI and Post-GSAI, the generated preconditioner is then passed to the Solver stage (Fig. 5.2) to precondition and solve the linear system

#### A. Pre-GSAI Stage

*Copy A to GPU*: Sparse matrices are stored in memory using various compressed sparse storage formats such as CSR, CSC, etc [39]. Such formats reduce the amount of memory used to store the sparse matrix by contiguously storing rows/columns allowing for coalesced memory accesses. To compute the SAI preconditioner the  $A$  matrix is initially stored in CSC format using three vectors called  $A_{value}$ ,  $A_{index}$  and  $A_{pointer}$ . The  $M$  matrix is also produced and stored in columns. A copy of the  $A$  matrix is transferred to GPU global memory.

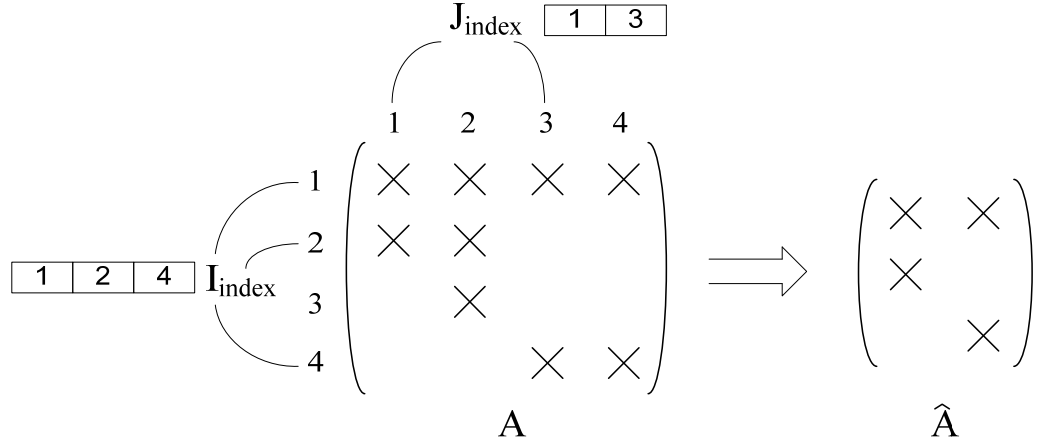


Fig. 5.3: Constructing local  $\hat{A}$  matrices by first finding  $J_{index}$  and  $I_{index}$  vector values and then matching the columns referenced in  $J_{index}$  to the  $I_{index}$  vector.

Compute  $n_1$  and  $n_2$  and allocate memory to  $M$ : The preconditioner  $M$  is stored in global memory, thus memory should be allocated to  $M$  prior to the Compute-GSAI stage. Although the dimensions of  $M$  are the same as  $A$  it has to be stored in compressed format to fit on the GPU global memory. To reduce the amount of computation required to locate data structures used by each warp and regularize global memory accesses, equal memory space is allocated to each column of  $M$  using the compute dimensions kernel (Fig. 5.2). The proposed memory allocation technique, introduces the need for the Post-GSAI step described in the next section, whose execution time is, however, negligible compared to Compute-GSAI as shown in the results section and to the provided benefits. The kernel first finds the dimensions of local  $\hat{A}$  matrices  $(n_1, n_2)$  and stores them on global memory and the maximum  $n_1$  and  $n_2$  values between all columns (called  $n_{1,max}$  and  $n_{2,max}$ ) are then found. Since the number of non-zeros in the largest column of  $M$  is equal to  $n_{2,max}$ , global memory allocated to  $M$  would be equal to the number of columns in  $M$  multiplied by the number of bytes required to store  $n_{2,max}$  floating point values ( $M_{value}$ ). The row indices corresponding to the values of the preconditioner ( $M_{index}$ ) and the number of non-zeros produced for each column of  $M$  ( $M_{pointer}$ ) are stored in

global memory. Besides allocating memory to the preconditioner  $M$ , the Allocate memory step of the Pre-GSAI stage (Fig. 5.2) assigns memory space to other data structures used during the computation of the SAI preconditioner (Compute-GSAI) and determines the number of kernel calls required to compute the SAI preconditioner. Details of these implementations are presented in Section 5.3.2 and Table 5.1.

### *B. Compute-GSAI Stage*

To compute the SAI preconditioner on the GPU, the steps indicated in the Compute-GSAI stage in Fig. 5.2 have to be implemented in parallel on the GPU in a kernel called compute preconditioner. Each column of the preconditioner  $M$  is computed via one warp (32 threads in a block) and every block is assigned 256 threads (eight warps) to compute eight columns in parallel. The number of columns computed in one SM simultaneously will depend on the allocated shared memory per block and available resources per SM.

*Find J:* In this stage the set  $J$  (the first step in Fig. 5.1) is constructed and loaded into a vector called  $J_{index}$ . Each warp in the kernel first loads the column in  $A$  corresponding to its index (the index is assigned to each warp based on the total number of warps launched on the GPU) and finds the largest element in the loaded column. The condition in equation 5.3 is then evaluated for each element of the loaded value vector simultaneously and the column index of elements satisfying the condition is stored in  $J_{index}$ .

*Find I and construct the local  $\hat{A}$ :* To determine  $I$  (Fig. 5.1), the algorithm first loads the row indices of the first column referenced in  $J_{index}$  into a vector called  $I_{index}$ . The row index vector of successive columns referenced by  $J_{index}$  are then loaded in order into shared memory and compared in parallel with values in  $I_{index}$ , new indices are tagged and later added to  $I_{index}$  to construct the set  $I$ . Local  $\hat{A}$  matrices are

constructed on global memory by loading columns indexed in the  $J_{index}$  vector and matching them to the  $I_{index}$  vector in parallel (Fig. 5.3).

*Local QR decomposition and triangular solves:* Local QR decompositions are computed using the Gram Schmidt method (Fig. 5.4) [5], which was easier to parallelize inside a warp compared to other QR decomposition techniques. Each warp decomposes one  $\hat{A}$  matrix, thus many QR decompositions are computed simultaneously via warps executing in parallel. Parallelism is also exploited in a warp by computing the local QR decompositions in parallel using the 32 threads inside a warp, e.g., most of the operations in Fig. 5.4 such as memory loads, multiplications and inner products are computed in parallel.

The orthogonal vectors produced in each step of the QR decomposition algorithm ( $q_i$  in Fig. 5.4) are stored in global memory (Q in Table 5.1) and are used in proceeding steps. At the end of the Compute-GSAI stage  $m_k$  values are computed using  $\hat{m}_k = R^{-1}Q^T\hat{e}_k$  and scattered to global memory space allocated to the  $M$  matrix.

$$\begin{aligned}
 &\text{if } A = [a_1 \quad \dots \quad a_n] \quad \text{and} \quad proj_q a = \frac{\langle q, a \rangle}{\langle q, q \rangle} q \\
 &u_k = a_k - \sum_{j=1}^{k-1} proj_{q_j} a_k \quad q_k = \frac{u_k}{\|u_k\|} \\
 &a_k = \sum_{j=1}^k \langle q_j, a_k \rangle q_j \\
 &Q = [q_1 \quad \dots \quad q_n], \quad R = \begin{pmatrix} \langle q_1, a_1 \rangle & \langle q_1, a_2 \rangle & \langle q_1, a_3 \rangle & \dots \\ 0 & \langle q_2, a_2 \rangle & \langle q_2, a_3 \rangle & \dots \\ 0 & 0 & \langle q_3, a_3 \rangle & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}
 \end{aligned}$$

Fig. 5.4: The Gram Schmidt QR decomposition with  $\langle q, a \rangle = q^T a$ .

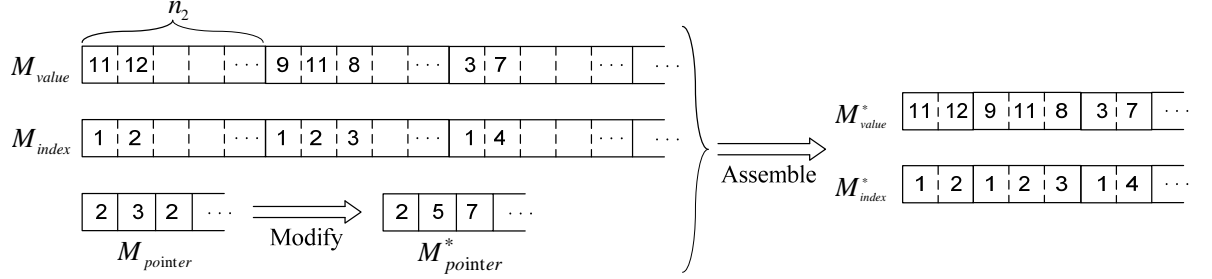


Fig. 5.5: The  $M_{pointer}$  vector computed in the compute preconditioner kernel is first modified using the Modify kernel to match the CSC [88] storage format and then the Assemble kernel assembles the  $M$  matrix values and stores them in CSC format ( $M_{index}^*$  and  $M_{value}^*$  vectors).

### C. Post-GSAI Stage

*Modify and assemble  $M$ :* The values and row indices of the preconditioner generated in the compute preconditioner kernel are stored in  $M_{value}$  and  $M_{index}$  vectors in the format shown in Fig. 5.5. Since the allocated size to each column of  $M$  on global memory is equal to  $n_{2,max}$  (which is not necessarily equal to the number of non-zeros per column), to assemble  $M$  each warp has to store the number of non-zeros of the column it is generating into a vector called  $M_{pointer}$ . In the Post-GSAI stage the  $M_{value}$ ,  $M_{index}$  and  $M_{pointer}$  data structures are modified to match the CSC storage format. The first kernel in the Post-GSAI stage is called Modify which changes  $M_{pointer}$  to match the CSC format ( $M_{pointer}^*$  in Fig. 5.5). Another kernel called *Assemble* then modifies the  $M_{index}$  and  $M_{value}$  vectors on the GPU to match the column storage format ( $M_{index}^*$  and  $M_{value}^*$  in Fig. 5.5). The updated value and index vectors of  $M$  are generated on GPU memory and do not require data to be transferred to the CPU.

### D. The Solver

*Preconditioned BiCGStab solver:* When generating a right preconditioner  $M$  (via minimizing equation 5.2) matrices are stored and generated in column storage format to reduce memory access latencies [5]. On the other hand, to achieve the

best performance and increase coalesced memory accesses on the GPU, the matrices in the sparse matrix vector multiplication kernel should be stored in row storage format [89]. Thus prior to solving  $Ax = b$  the matrices are converted to CSR format (to generate a left preconditioner the CSC to CSR stage in Fig. 5.2 should be removed since all matrices are generated and stored in CSR format). After the conversion step the BiCGStab kernel is called to solve  $Ax = b$  using the produced  $M$ .

The preconditioned BiCGStab iterative solver on GPU is dominated by the multiple sparse matrix multiplies [5]. The CPU is only used for scalar updates in the algorithm and major computing kernels are implemented on the GPU. Since sparse matrix vector multiplication is the most time consuming operation in iterative solvers [26] it has to be accelerated efficiently on the GPU. We used the SMVM implementation from [49], [89] which is one of the fastest implementations of this kernel on GPUs. Other operations in the BiCGStab iterative solver have also been accelerated on the GPU using CUBLAS [90] functions. One of the advantages of using BiCGStab is that  $A$  can be non-symmetric.

### 5.3.2 Memory Allocation

In this section we introduce techniques to overcome GPU memory space limitations and enable the correct implementation of the GSAI stages proposed in Section 5.3.1 for large problems. Since the exact size of data structures (such as  $\hat{A}$  and  $Q$ ) used in the compute preconditioner kernel are only determined during the kernel execution, techniques to allocate memory statically to these data structures in the Pre-GSAI stage (prior to calling the kernel) are also proposed. Based on the allocated memory space to each data structure, the number of compute preconditioner kernel calls required to generate the preconditioner are also determined. The implementations proposed in this section are all a part of the

*Allocate memory* section of the Pre-GSAI stage shown in Fig. 5.2.

Local data structures such as  $\hat{A}$  and  $Q$  are generally large and cannot be stored on GPU shared memory; thus by approximating their size, global memory space is allocated to them in the Pre-GSAI stage prior to calling the compute preconditioner kernel. The maximum number of rows and columns in these matrices is computed in the compute dimensions kernel ( $n_{1,max}$  and  $n_{2,max}$ ) and global memory space equal to the size of an array with  $n_{1,max} \times n_{2,max}$  elements is allocated to them per column (warp). The  $I_{index}$  vector used in the Compute-GSAI kernel also varies in size for each warp and can easily exceed the maximum size of shared memory. This vector is also stored in global memory by allocating memory to arrays of  $n_{1,max}$  elements per column. To compute the preconditioner different columns of  $A$  are required thus the  $A$  matrix should be on global memory at all times. Table 5.1 shows the amount of global memory required to store various vectors and data structures on global memory prior to calling the Compute-GSAI kernel. Because the preconditioner is generated in double precision, data structures such as  $\hat{A}$ ,  $Q$  and  $M_{value}$  are stored in double precision.

For large  $A$  matrices and  $\tau$  parameters that lead to a denser preconditioner, the total size of the data structures in Table 5.1 will exceed the GPU global memory. Since the memory required to store  $\hat{A}$  and  $Q$  for all columns is considerably larger than the size of  $A$  and  $M$ , by calling multiple kernels sequentially and overwriting the memory space allocated to these matrices, computing the SAI preconditioner is made possible on the GPU. Thus after storing  $A$  and  $M$  on global memory depending on the available memory space and the size of other data structures that need to be stored on global memory, the computation of the preconditioner is divided between multiple kernels each producing a few columns of  $M$ . As a result memory allocated to other data structures such as  $\hat{A}$  and  $Q$  can be reused. In the following, steps (implemented on the CPU) to determine the number of required

compute preconditioner kernel calls are presented and the allocated memory space to  $\hat{A}$  which is overwritten in each kernel call is determined (memory assigned to other data structures such as  $Q$  can be computed the same way):

- Memory available to store local data structures ( $\hat{A}$ ,  $Q$ , etc.) in global memory is first determined by subtracting memory allocated to  $A$  and  $M$  matrices from GPU global memory.
- The result is then divided by the size of memory required to store the local data structures for one column, in order to determine the number of columns which can be computed in each kernel call (*columns-per-kernel*).
- The number of compute preconditioner kernel calls is determined via dividing the total number of columns in  $A$  by columns-per-kernel. The memory allocated to storing local  $\hat{A}$  matrices for each kernel will be equal to  $n_{1,max} \times n_{2,max}$  multiplied by columns-per-kernel.

The small size of the GPU shared memory does not limit the size of the problem being solved, because large vectors and data structures in the kernel are stored on GPU global memory. To accelerate computations shared memory is used to store local data structures in the compute preconditioner kernel whenever possible. Before calling the compute preconditioner kernel (in the allocate memory section of the Pre-GSAI stage), the amount of shared memory required for each block to store these data structures is checked and if it reduces the number of active blocks per SM to two all data is read from global memory directly. For larger tolerance parameters which lead to larger data structures most of the data is read directly from global memory. Thus, the number of active blocks per SM is no longer limited to the size of data structures and available shared memory and memory access latencies are reduced via configuring the size of L1 cache to 48KB. To generate SAI preconditioners for very large problems which do not fit on the GPU global memory or to generate very dense preconditioners, a graphic card with larger global memory



could be used or the computation of the SAI preconditioner should be distributed between many GPUs.

Table 5.1: The number of elements in each of the data structures involved in GSAI and their size based on their data.

Data Structure	Number of elements	Type	Size
$A_{value}$	non-zeros	double	elements * 8
$A_{index}$	non-zeros	integer	elements * 4
$A_{pointer}$	Columns	integer	elements * 4
$M_{value}$	Columns $\times n_{2,max}$	double	elements * 8
$M_{index}$	Columns $\times n_{2,max}$	integer	elements * 4
$M_{pointer}$	Columns	integer	elements * 4
$\hat{A}$ ( <i>all columns</i> )	Columns $\times n_{1,max} \times n_{2,max}$	double	elements * 8
$Q$ ( <i>all columns</i> )	Columns $\times n_{1,max} \times n_{2,max}$	double	elements * 8
$I_{index}$ ( <i>all columns</i> )	Columns $\times n_{1,max}$	integer	elements * 4

*elements, columns and non-zeros represent the number of elements computed in the second column of the table, the number of columns in  $A$  and the number of non-zeros in  $A$ , respectively*

## 5.4 Results

The performance of the proposed SAI acceleration on GPUs is evaluated using 7 matrices [34] from various application areas with different sparsity patterns (Table 5.2). Since the SAI preconditioner is not limited to symmetric problems the performance of the preconditioner and the acceleration has also been tested on 4 unsymmetric matrices. These problems are generally difficult to solve and precondition due to their complex geometry and ill-conditioning. GPU results were achieved using NVIDIA GTX480, TESLA M2070 and CUDA-SDK 3.2, CPU programs are executed on a system core Linux cluster from Sharcnet [91] using 1-32 AMD Opteron 252 (2.6GHZ, single-core) processors with a Quadrics Elan4 interconnect. The preconditioned BiCGStab iterations are terminated upon reaching 10,000 iterations or reaching a relative residual of less than  $1e-7$  in under 10,000 iterations using a random Right Hand Side (RHS) for all problems (the same RHS

is used for each matrix in all platforms). Both the preconditioner generation kernel and the iterative solver run in double precision. In the following the performance of the proposed GSAI preconditioner on GTX480 and TESLA M2070 is first presented (Section 5.4.1), the preconditioner computation time on the GPU is then compared to ParaSails (Section 5.4.2) on a single processor/core (a processor/core is an AMD Opteron 252 consisting of one core).

ParaSails computes the preconditioner in parallel on multi-processor platforms by partitioning  $M$  and allocating the computation of its columns to different processors. They propose novel techniques to partition columns/rows amongst processors, hide inter-processor communication latencies, balance load amongst processors, manage one-sided communications, construct  $\hat{A}$  matrices and perform operations such as QR decomposition. Implementation details of how the computation of SAI preconditioners is parallelized in ParaSails can be found in the documentations and publications referenced in [59]. The time to compute the SAI preconditioner using GSAI on GPUs is compared to ParaSails on a cluster of multiple AMD Opteron 252 processors in Section 5.4.2.

#### 5.4.1 The GSAI Preconditioning Method

In this section the effect of increasing the tolerance  $\tau$  in equation 5.3 using GSAI and NVIDIA GTX480 on the preconditioner construction time, iterative solver execution time and the number of iterations are first studied. The total execution time and the number of iterations of the preconditioned iterative solver are then presented for both GTX480 and TESLA M2070.

As shown in Table 5.3 for larger tolerances ( $\tau$ ), the number of iterations considerably decreases for most of the tested problems using GSAI. Because the preconditioner  $M$  is an approximation of  $A^{-1}$  decreasing its sparsity using  $\tau$  does not necessarily guarantee a better preconditioner, for example the number of iterations

in g3-circuit increases when  $\tau$  is increased to 0.6 (Table 5.3). But on average the number of iterations decrease as  $\tau$  increases and the sparsity of  $M$  gets closer to  $A$  [56]. For most of the tested problems the total execution time on GPU also decreases as  $\tau$  increases (Table 5.4). Because more elements of  $A$  satisfy the condition in equation 5.3 the maximum number of rows and columns ( $n_{1,max}$  and  $n_{2,max}$ ) of the local  $\hat{A}$  matrices on the GPU increase with tolerance (Fig. 5.6). As a result the time required by the compute dimensions kernel to determine  $n_{1,max}$  and  $n_{2,max}$  as well as the time required to construct and decompose  $\hat{A}$  in the compute preconditioner kernel also increase with  $\tau$  (Fig. 5.7 and Table 5.5). Fig. 5.7 shows the fraction of total preconditioner execution time spent in all kernels involved in the construction of the SAI preconditioner on GTX480 (kernels in the Pre-GSAI, Compute-GSAI and Post-GSAI stages). Based on Table 5.5 for all tested matrices the preconditioner execution time increases with  $\tau$ . Thus, except for copying  $A$  to the GPU, the execution time of all kernels increases with  $\tau$  due to an increase in the number of non-zeros in preconditioner  $A$  (Fig. 5.7 and Table 5.5).

Fig. 5.8 and Table 5.4 explain why an increase in the SAI computing time for larger tolerances still on average improves the total execution time on GPU. As shown in Fig. 5.8, the total execution time is dominated by the BiCGStab solver. Thus, based on total execution times reported in Table 5.4, by increasing  $\tau$  and generally generating a more accurate preconditioner, the execution time of the iterative solver is decreased (due to an average reduced number of iterations) with a negligible increase in SAI computation time. Since the time spent in generating the preconditioner is considerably less than the time required to solve the problem, the total execution time on average decreases for larger tolerance parameters.

The problem solution time on the GPU decreases when the iterations are reduced on the GPU. This is because the sparse matrix vector multiply kernel involved in the iterative solve uses available GPU resources more efficiently as the

number of non-zeros in  $M$  increase. While the preconditioner becomes denser with larger  $\tau$  parameters, the number of rows in  $M$  is fixed and as a result the number of computing blocks/warps launched on the GPU remain unchanged because of using the sparse matrix vector multiply kernel introduced in [49], [89]. On the other hand the number of non-zeros per row increases, exploiting more parallelism per warp and better utilizing the GPU resources. Thus GPU acceleration of the SAI allows for the generation of more accurate and denser preconditioners and increases the applicability of static preconditioning for sparse approximate inverse preconditioners. Table 5.6 shows the execution time of the steps involved in constructing the sparse approximate inverse preconditioner on GTX480 for  $\tau$  equal to 0.9 (which generated the best preconditioner amongst the tested tolerances) as well as the BiCGStab iterative solver. The time spent in constructing the preconditioner is less than 3 seconds for all matrices (Table 5.6) while the iterative solve can take up to 171 seconds for matrices such as thermal2 on the GPU.

Preconditioners with more than 6 million non-zeros (Table 5.7) are generated in less than 3 seconds (Table 5.6) using the proposed GSAI preconditioner on GTX480. As shown in Table 5.7 without the preconditioner most of the problems would not converge in 10,000 iterations while with the preconditioner the BiCGStab iterative solver would converge to the  $1e-7$  residual error in less than 100 iterations for some matrices (venkat01 and majorbasis). Table 5.7 also shows that although the number of iterations for the preconditioned iterative solver on TESLA M2070 decreases compared to GTX480, the total execution time is still larger for all tested matrices.

#### 5.4.2 GSAI vs. ParaSails

In this section the preconditioner construction time is compared with ParaSails [59] which also uses a priori techniques to determine the sparsity of  $M$  and computes SAI in parallel on multiprocessors. Techniques proposed in ParaSails to

better determine the sparsity of  $M$  prior to its computations for PDE problems can be implemented in the Pre-GSAI stage of GSAI without changing the Compute preconditioner kernel itself (determining the sparsity of  $M$  in a priori SAI preconditioning techniques is negligible compared to the preconditioner computation itself). To compare GSAI with ParaSails, parameters were set so that both ParaSails and GSAI would produce similar preconditioners with the same sparsity as  $A$  ( $\tau = 1$  in GSAI, parameter settings for ParaSails are described in [59]), preconditioners are produced using unfactorized preconditioning in ParaSails.

Table 5.8 shows generating the SAI preconditioner using ParaSails on one processor/core can take up to 100 seconds while the proposed acceleration of sparse approximate inverse preconditioners on GPUs generated the same preconditioner in less than 3 seconds. With GSAI on GTX480, speedups of up to 47 times are achieved compared to ParaSails, decreasing the average generation time of SAI preconditioners 28 times. In Fig. 5.9 the average execution time of ParaSails for all matrices on multiprocessors is compared to average preconditioner generation time of GSAI on NVIDIA GTX480 and TESLA M2070. As shown in Fig. 5.9 constructing the preconditioner on a single GPU using GSAI is equivalent to constructing the same preconditioner on 16 processors/cores using ParaSails.

GSAI computes many columns of  $M$  in parallel, the time spent to construct local  $\hat{A}$  matrices do not accumulate for columns generated simultaneously. This is not the case in ParaSails when run on a single processor, so both the parallel execution of columns on the GPU and the techniques proposed to compute each column of  $M$  are the main reasons behind the reported speedups.

## 5.5 Conclusion and Future Work

The GPU accelerated sparse approximate inverse preconditioning method called GSAI, proposed in this work introduces optimized implementations to parallelize

the computation of sparse approximate inverse preconditioners on NVIDIA GPUs. A sparsified pattern of  $A$  based on tolerance  $\tau$  is used as the sparsity pattern of the preconditioner  $M$ . By allocating the computation of each column of  $M$  to one warp, the GSAI method computes the SAI preconditioner in three stages called Pre-GSAI, Compute-GSAI and Post-GSAI and then solves the linear system of equations in the Solve stage. Techniques to overcome limitations imposed by the small GPU shared and global memory in computing SAI preconditioners on GPUs are proposed as a part of the Pre-GSAI stage. The execution of operations involved in the SAI computation are parallelized per warp in the Compute-GSAI and the generated preconditioner values are assembled and stored in a compressed format in the Post-GSAI step. Finally the preconditioned BiCGStab iterative solver is implemented in parallel (Solver stage) to compute the results of the linear system of equations using the generated preconditioner.

The effects of decreasing the sparsity of the preconditioner using a tolerance parameter  $\tau$  are tested on the GPU using GSAI. The results showed that the number of iterations and total execution time would on average decrease using GSAI for larger tolerances; the preconditioner generation time would remain negligible compared to the problem solution time. The total execution time on the GPU (the time spent on generating the preconditioner and solving the problem) would constantly decrease as  $\tau$  increases making the generation of denser preconditioner more efficient. The generation of the SAI preconditioner was accelerated on average 28 and 23 times on GTX480 and TESLA M2070 respectively using GSAI compared to the time required to create the same preconditioner using ParaSails on a single processor (single-core AMD Opteron 252). The preconditioner generation time on GTX480 and TESLA M2070 (using GSAI) is almost equivalent to creating the SAI preconditioner on 16 processors in parallel using ParaSails. We plan to accelerate the execution time of other variants of SAI preconditioning

techniques such as adaptive methods and also introduce techniques to find better approximations of the preconditioner using GPUs in future work.

Table 5.2: Properties of sparse matrices used to test the GSAI preconditioning method.

Matrix Name	Matrix Type	Rows	non-zeros	Structure
venkat01	CFD problem sequence	62,424	1,717,792	unsymmetric
majorbasis	optimization problem	160,000	1,750,416	unsymmetric
t2em	electromagnetics problem	921,632	4,590,832	unsymmetric
atmosmodd	CFD problem	1,270,432	8,814,880	unsymmetric
thermal2	thermal problem	1,228,045	8,580,313	Symmetric
g3-circuit	circuit simulation problem	1,585,478	7,660,826	Symmetric
apache2	structural problem	715,176	4,817,870	Symmetric

Table 5.3: The effect of increasing tolerance ( $\tau$ ) on the number of iterations (GSAI on GTX480).

Matrix	$\tau = 0.5$	$\tau = 0.6$	$\tau = 0.7$	$\tau = 0.8$	$\tau = 0.9$
venkat01	65	59	50	45	70
majorbasis	49	47	49	43	23
t2em	2390	2390	2390	1264	1264
atmosmodd	268	268	268	145	145
thermal2	6000	5805	5727	3608	2906
g3-circuit	1856	2307	1863	1347	1145
apache2	2922	1674	1674	1143	1226
average	1936	1793	1717	1085	968

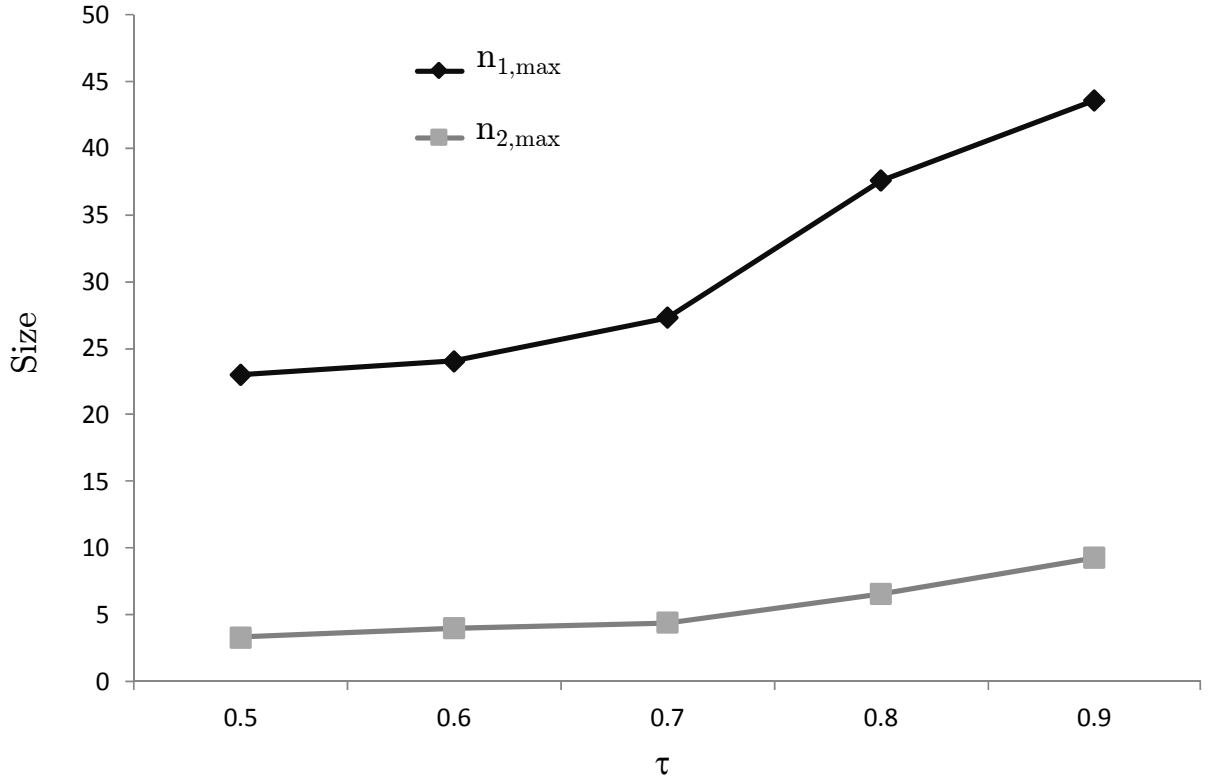


Fig. 5.6: The effect of increasing  $\tau$  on the maximum dimension of local  $\hat{A}$  matrices ( $n_{1,max}$  and  $n_{2,max}$ ).

Table 5.4: The effect of increasing tolerance ( $\tau$ ) on the total execution time involving both the preconditioner construction time and the solve time (GSAI on GTX480).

Matrix	$\tau = 0.5$	$\tau = 0.6$	$\tau = 0.7$	$\tau = 0.8$	$\tau = 0.9$
venkat01	0.43	0.54	0.69	0.83	2.6
majorbasis	0.66	0.64	0.65	0.68	0.6
t2em	108	108	108	59	59
atmosmodd	17	17	17	11	11
thermal2	364	348	331	213	174
g3-circuit	136	170	138	101	87
apache2	110	63	63	44	47
average	105	101	94	61	54



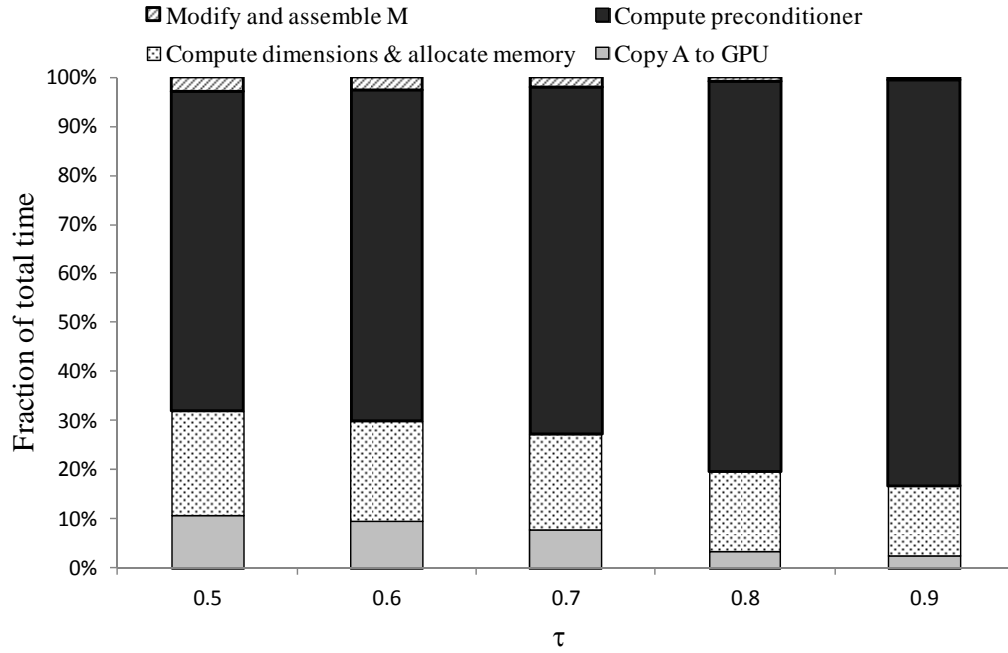


Fig. 5.7: The average fraction of total time (over all matrices) spent in the functions/kernels involved in the first three stages of the GSAI preconditioning algorithm (on GTX480) are shown for an increasing  $\tau$  (compute preconditioner consists of all steps in the Compute-GSAI stage).

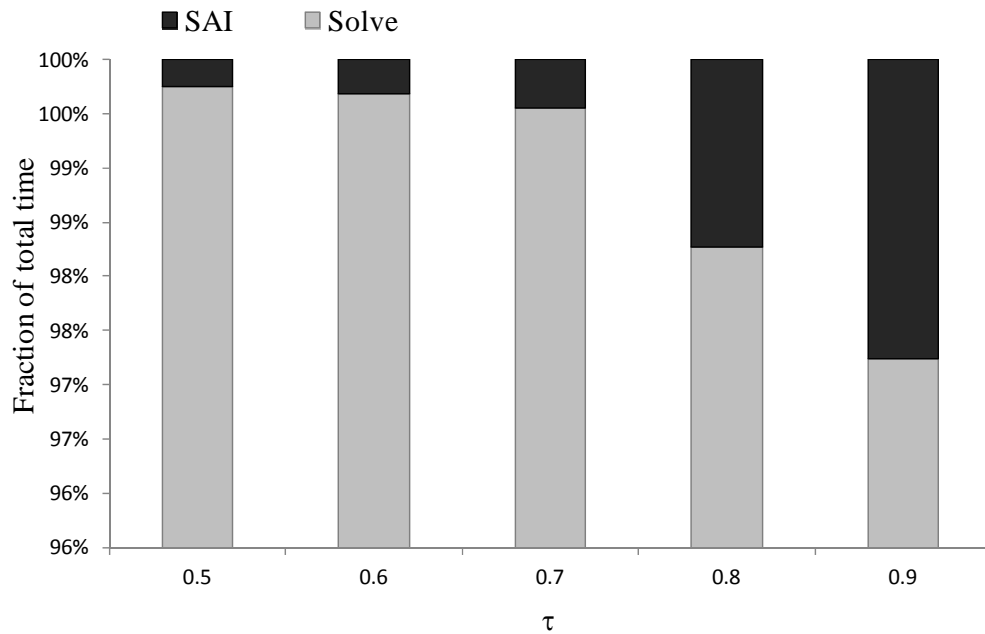


Fig. 5.8: The average fraction of total time (over all matrices) in generating the SAI preconditioner (the Pre-GSAI, Compute-GSAI and Post-GSAI stages in Fig. 5.2) and solving the problem for an increasing  $\tau$  on the GPU using GSAI.

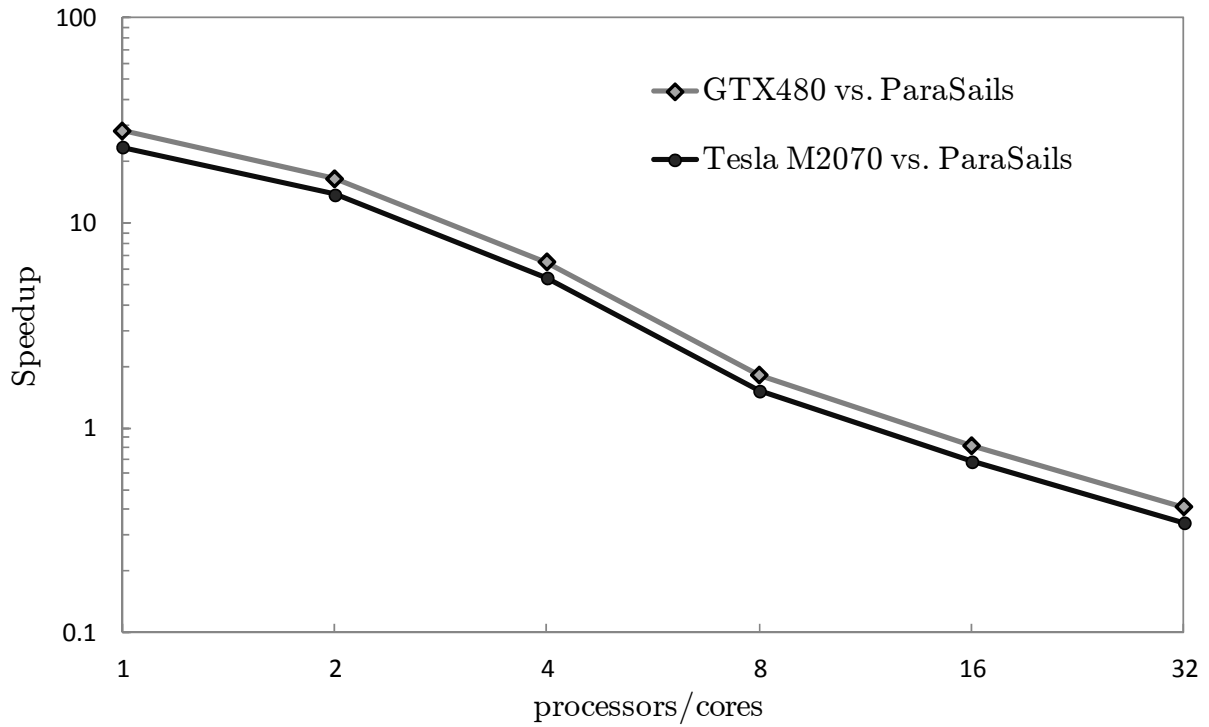


Fig. 5.9: The speedup achieved from generating the SAI preconditioner on GTX480 and TESLA M2070 using GSAI compared to generating the same SAI preconditioner using ParaSails [59] on 1-32 processors/cores (the generated preconditioner has the same sparsity as  $A$ ,  $\tau = 1$  in GSAI).

Table 5.5: The effect of increasing tolerance ( $\tau$ ) in the GSAI algorithm (on GTX480) on the preconditioner construction time.

Matrix	$\tau = 0.5$	$\tau = 0.6$	$\tau = 0.7$	$\tau = 0.8$	$\tau = 0.9$
venkat01	0.11	0.24	0.42	0.58	2.14
majorbasis	0.11	0.11	0.11	0.19	0.3
t2em	0.3	0.3	0.3	1.26	1.26
atmosmodd	0.43	0.43	0.43	1.97	1.97
thermal2	0.42	0.42	0.7	1.65	2.7
g3-circuit	0.65	0.78	0.9	1.51	1.84
apache2	0.31	0.35	0.35	0.78	0.8

Table 5.6: The time spent in computing the stages in Fig. 5.2 for  $\tau = 0.9$  on GTX480.

Matrix	Pre-GSAI $\tau = 0.9$	Compute-GSAI $\tau = 0.9$	Post-GSAI $\tau = 0.9$	Solve
venkat01	0.2	1.94	0.01	0.46
majorbasis	0.06	0.23	0.01	0.3
t2em	0.22	1.03	0.01	57
atmosmodd	0.389	1.57	0.02	9.5
thermal2	0.46	2.23	0.02	171
g3-circuit	0.33	1.49	0.02	85
apache2	0.17	0.62	0.01	47

Table 5.7: Preconditioned and unpreconditioned BiCGStab iterative solver on GTX480 and TESLA 2070.

Matrix	GPU BiCGStab Iterations	Precond. non-zeros	GTX480 Precond. BiCGStab Iterations	GTX480 Total Time	TESLA M2070 Precond. BiCGStab Iterations	Tesla M2070 Total Time
venkat01	>10000	822937	70	2.6	70	2.7
majorbasis	>10000	646524	23	0.6	23	0.72
t2em	>10000	4590832	1264	59	968	63
atmosmodd	>10000	6317824	145	11	140	14
thermal2	6119	6720218	2906	174	2804	195
g3-circuit	>10000	6562707	1145	87	1133	108
apache2	4931	2677127	1226	47	1115	58

The table shows the number of iterations (column one) required to solve the unpreconditioned BiCGStab solver for the tested matrices, the number of non-zeros in the preconditioner produced for  $\tau = 0.9$  and the iterations achieved from the preconditioned BiCGStab solver using this preconditioner on both the GTX480 and TESLA M2070 graphic cards.

Table 5.8: ParaSails execution time compared to GPU results.

Matrix	ParaSails Setup	ParaSails Precond.	ParaSails Total	GTX480 $\tau = 1$	ParaSails Total vs. GTX480 speedup	TESLA M2070 $\tau = 1$	ParaSails Total vs. TESLA M2070 speedup
venkat01	0.1	13.7	13.8	2.22	6.2	2.83	4.8
majorbasis	0.1	14.7	14.8	1.19	12.3	1.43	10.3
t2em	0.4	60	60.4	1.26	47.9	1.55	38.9
atmosmodd	0.7	93.7	94.4	3	31	3.8	24.8
thermal2	0.8	91.7	92.5	3.76	24.5	3.9	23.7
g3-circuit	0.7	99.4	100.1	2.13	46.8	2.64	37
apache2	0.4	52	52.4	1.62	32.3	2	25.8
average speedup	--	--	--	--	28.7	--	23.7

The time to setup (ParaSails-Setup) and compute (ParaSails-Preconditioner) the SAI preconditioner with the same sparsity as  $A$  ( $\tau = 1$  in GSAI) on ParaSails for one processor/core compared to the time required to compute the preconditioner on GTX480 (GPU-SAI) and TESLA M2070 using the GSAI preconditioning algorithm (ParaSails-Total is computed by adding ParaSails-Setup and ParaSails-Preconditioner).

---

## PREFACE TO CHAPTER 6

The following chapter is an extended version of a paper submitted to the IEEE Conference on Electromagnetic Field Computation (CEFC 2012) titled “Communication-avoiding Krylov Techniques on GPUs”. Communication-avoiding Krylov solvers reduce the communication cost of KSMs by computing several vectors of a Krylov subspace “at once”, using a kernel called “matrix powers”. The matrix powers kernel is implemented on NVIDIA GPUs. Speedups of upto 5.7 times are reported for the matrix power kernel compared to regular SpMV implementation. The proposed implementation of matrix powers will be used in communication-avoiding Krylov solvers in future work. This work is done in collaboration with the Berkeley benchmarking and optimization group (BeBOP) and co-supervision of Professor James Demmel at UC-Berkeley.

## Chapter 6 COMMUNICATION-AVOIDING KRYLOV TECHNIQUES ON GPUS

Maryam Mehri Dehnavi, James Demmel and Dennis Giannacopoulos

**Abstract:** Communicating data within the GPU memory system and between the CPU and GPU are major bottlenecks in accelerating iterative solvers on GPUs. The communication-avoiding [92] matrix powers kernel is implemented to reduce data communication between CPU and GPU and within the GPU memory hierarchy in Krylov solvers.

**Index terms:** Numerical algorithms; Parallel algorithms; Graphic processors; Krylov solvers.

### 6.1 Introduction

The sparse matrix vector multiplication (SpMV) kernel is a dominant computing kernel in standard Krylov subspace methods (KSMs). Computing a few arithmetic operations per datum, SpMV operations are classified as communication-bound. The cost of communication (moving data between levels of the memory hierarchy) is considerably higher than the cost of arithmetic computations in modern architectures and this gap is expected to further widen. Thus, in order to enhance the performance of communication bound kernels such as SpMV, new strategies/algorithms should be explored to minimize communication and data movement.

#### 6.1.1 Communication-avoiding Krylov techniques

Communication-avoiding (CA) algorithms [92] communicate less than the state-of-the-art algorithms at the expense of more arithmetic operations. Standard implementations of SpMV in KSMs, require reloading the sparse matrix to caches and fast memory in each iteration when they are too large to fit in fast memory, thus, overwhelming the algorithm with communications and data movement between fast and slow memory. Communication-avoiding Krylov techniques [92],

minimize communication via computing  $k$  steps of the iterative solver at the same time. To take  $k$  steps at the same time, and so potentially reduce memory traffic by a factor of  $k$ , a new sparse matrix kernel is required, called the matrix powers kernel. Where  $p_i$  is a polynomial of degree  $i$ , the matrix powers kernel computes the basis  $[p_1(A)x, p_2(A)x, p_3(A)x, \dots, p_k(A)x]$ . To compute the aforementioned basis for a matrix  $A$  that does not fit into fast memory, the matrix is first divided into partitions (cache-blocks) that fit into the desired memory space. The partitions are then loaded into fast memory to compute the basis. To avoid communication between fast and slow memory and between partitions, non-local rows might also be copied to a partition ("remote/ghost" rows) leading to redundant arithmetic operations [10]. For a well partitioned  $A$  matrix (where  $A$  has a low surface-to-volume ratio), the communication cost of the  $k$ -step matrix powers kernel will be  $O(1)$  compared to  $O(k)$  for  $k$  SpMV operations in a naïve implementation [10].

### 6.1.2 NVIDIA GPUS

Graphic processing units (GPUs) have become an important resource for scientific computing in recent years. With easy to learn APIs such as CUDA [43] introduced by NVIDIA, general purpose programming for modern scientific computations on GPUs have gained considerable attention. The GPU consists of streaming multiprocessors (SMs) and each SM contains basic processing units called scalar processors (SPs). To run compute intensive parts of an application on the GPU initial data has to be transferred from CPU memory to GPU global memory and a GPU kernel is then launched. Using a single data multiple thread paradigm, GPU threads grouped into thread blocks (TBs) proceed with the computations and transfer the results back to CPU. The GPU consists of an on-board global memory with long access latency, a fast access shared memory, registers and caches. Threads inside a block communicate via shared memory and their execution can be

synchronized. Every 32 threads in a block execute the same instruction and are called a warp.

## 6.2 Previous Work

This work implements the matrix powers kernel on NVIDIA GPUs by partitioning (cache-blocking) the matrix to fit into global and shared memory spaces. The kernel will be used in  $k$ -step Krylov solvers in future work. In this section a brief survey of the  $k$ -step Krylov techniques is presented; algorithmic details of these techniques and a complete survey of previous work on  $k$ -step Krylov solvers are presented in [10].

The  $k$ -step Krylov subspace methods were initially introduced by Van Rosendale [93], and later studied in work such as [8], [94]. All this work used a monomial basis and reported convergence for  $k < 5$  in  $k$ -step KSMs. By using a scaled monomial basis [95], a scaled and shifted Chebychev basis [96] and Newton basis [97], the coverage of the  $k$ -step Krylov subspace techniques were further improved at the expense of increased dependency in the algorithm. This problem is resolved by Hoemmen et al. [10] by eliminating the need for scaled basis vectors. A more detailed survey of available work on communication-avoiding KSMs is presented in [10]. The dominant computing kernel in  $k$ -step Krylov solvers is the matrix powers kernel which is implemented on GPUs in this work.

A considerable number of work has been done on accelerating sparse matrix vector multiplication on GPUs [49], [53], [89]. None of the available implementations of SpMV on GPUs consider cache blocking for GPU global or shared memory. If the matrix is larger than GPU global memory, computing  $k$  SpMVs requires reloading the matrix to GPU global memory which is very costly. Most of previous work assumes the matrix is transferred to the GPU once and does not report transfer times between GPU and CPU, which is not applicable to large



matrices that do not fit in global memory. The matrix powers kernel reduces data communication between CPU and GPU global memory and within the GPU memory hierarchy by partitioning the matrix and computing  $k$  SpMV operations at the same time for each partition.

To our knowledge, we are the first to study the performance of the matrix powers kernel on NVIDIA GPUs using global and shared memory as fast memory. Our work is closely related to the work proposed by Mohiyuddin et al. [98], which implements the matrix powers kernel on a 8-core Intel Clovertown. The proposed implementation of the communication-avoiding matrix powers kernel on GPUs will be used in communication-avoiding KSMs in future work. Major contributions of the work are classified in the following:

- Most of the previous work on accelerating SpMV on GPUs [49], [53], [89] does not report the cost of copying data to and from the GPU and assume the  $A$  matrix fully fits on the device memory. With only 1.5GB of global memory in GPUs such as NVIDIA GTX480, matrices from many real problems can not be fully stored on the device. Memory might also be allocated to store preconditioners and other data structures, leaving only a part of GPU global memory for storing  $A$ . As a result, the matrix has to be transferred to the GPU in each iteration, increasing data transfers between GPU and CPU memory in iterative solvers. The matrix powers kernel is implemented on GPUs via global memory cache blocking to reduce data transfers to the GPU global memory in KSMs. The proposed implementation will enable the efficient implementation of communication-avoiding KSMs on NVIDIA GPUs.
- Similar to CPUs, graphic cards also have a memory hierarchy and although references to global memory are efficiently handled by the hardware, reads from this memory space are much more costly than accesses to GPU shared memory and caches. A naive implementation of  $k$  SpMV operations involves reloading

matrix rows from shared memory in each SpMV kernel call, leading to many global memory references. In the second contribution, the GPU shared memory is used as a cache level for the matrix powers kernel to reduce data transfers between GPU global and shared memory.

Implementation details of the proposed contributions are presented in Section 6.3. The performance of the matrix powers kernel for global memory cache blocking is evaluated using several matrices (Section 6.4.1). Initial results for the second contribution are also presented in Section 6.4.2 but will be addressed in more detail in future work.

### 6.3 Implementation Details

Implementation details of the matrix powers kernel on GPU global memory are presented in this section. The auto-tuning stage partitions the matrix to fit into GPU global memory, the partitions are then used in the matrix powers kernel. Details of cache blocking for GPU shared memory are also presented.

#### 6.3.1 Matrix Powers on GPU Global Memory

##### *A. Auto-tuning Stage*

The first stage of the algorithm is the partitioning stage where the matrix is either divided into equal partitions using a naive partitioning strategy or graph and hyper-graph partitioners such as Metis [99] and Zoltan [100]. The results presented in this document are achieved via naive row block partitioning; other partitioning methods will be studied in future work. The matrix is first divided in to equal partitions of row blocks. The partitions are balanced based on the floating point operations required to compute  $k$  steps of the matrix powers for each row block and are recursively reduced to fit into GPU global memory. The size of each partition is equal to the memory required to store local and remote rows in compressed row storage (CSR) format for each partition.

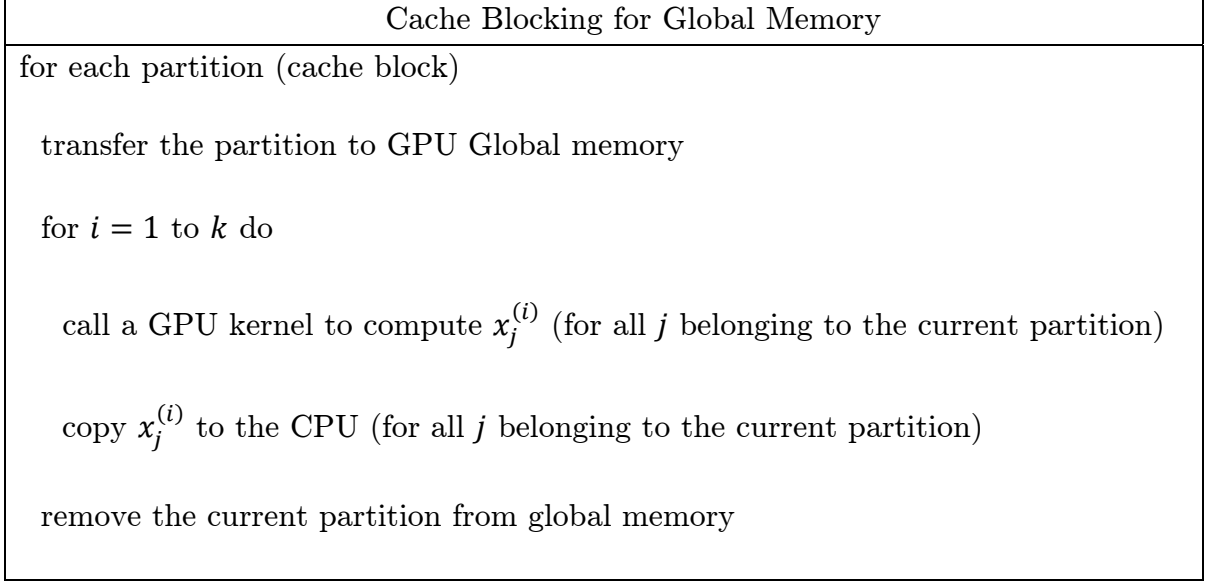


Fig. 6.1: The matrix powers implementation on GPU global memory,  $x_j^i$  is the  $j$ -th component of  $x^i = A^i x^{(0)}$ .

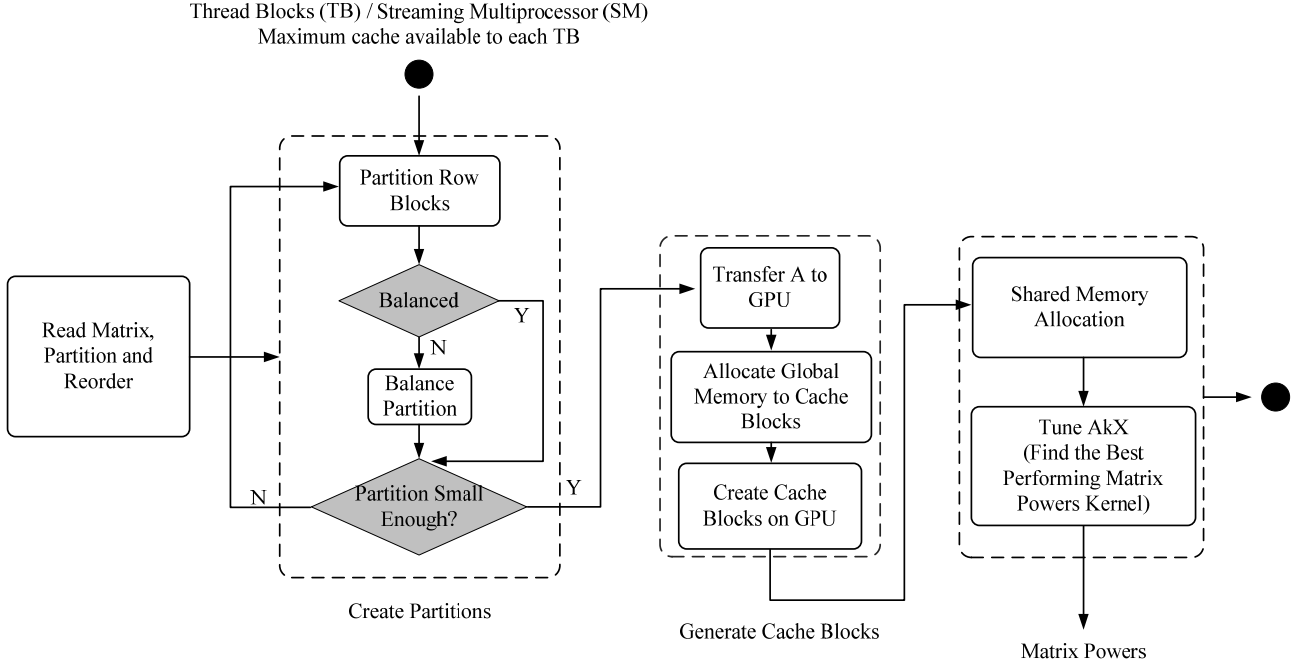


Fig. 6.2: The steps in the auto-tuner to generate cache blocks for shared memory and find the best performing matrix powers implementation on the GPU.

### B. Matrix Powers Kernel

For each partition, the corresponding elements of the source vector along with the matrix partition are then transferred to the global memory to compute  $k$  steps

of the matrix powers via calling the CUSPARSE SpMV kernel [89]  $k$  times. The generated vectors can then be used in the communication-avoiding Krylov solvers. (Fig. 6.1).

### 6.3.2 Matrix Powers on GPU Shared Memory

#### *A. Auto-tuning Stage*

Each streaming multiprocessor on NVIDIA graphic cards has a shared memory which is divided between the active thread blocks in the SM and can be configured to both 16K and 48K bytes. The performance of the matrix powers kernel is studied in this section when partitions (cache blocks) are generated for GPU shared memory. When cache blocking for global memory, each GPU kernel call is responsible for computing a step of the matrix powers kernel for one partition. For shared memory cache blocking on the other hand, the GPU thread blocks are responsible for computing the basis vectors for different partitions of the matrix. Partitioning the matrix to fit into shared memory and be operated on using the active thread blocks per SM can be challenging and may lead to the failure of the auto-tuning/partitioning phase. Some of these challenges are listed in the following:

- Larger cache blocks will lead to fewer extra floating point operations (flops) and smaller ghost zones in the matrix powers kernel. One of the major challenges in cache blocking for GPU shared memory is the small size of this memory space. The shared memory on each streaming multiprocessor is divided between the active thread blocks in that SM. To generate larger cache blocks and reduce arithmetic operations related to ghost rows, the number of thread blocks per SM (TBs/SM) should be reduced. On the other hand, limiting the number of thread blocks per SM, reduces resource occupancy on the GPU which can lead to performance loss.

- The limited number of registers available to each GPU thread block can also reduce GPU resource occupancy. When  $k$  SpMV's are computed per thread block, each active TB will require more registers compared to a naive SpMV implementation (where each thread block only computes one SpMV), which limits the active TBs per SM and can reduce GPU resource occupancy.

The auto-tuner finds the number of active thread blocks per SM which gives the best performance for the matrix powers kernel and leads to successful partitioning (Fig. 6.2). The matrix might be repartitioned and the number of active thread blocks per SM modified in the auto-tuning phase based on the success and performance of the matrix powers kernel for different configurations. To compute  $k$  SpMV's for each partition via one thread block, the auto-tuner should also choose the fastest implementation of the SpMV kernel amongst existing GPU SpMV implementations, specifically the row-per-warp [49] technique (also used in CUSPARSE), the row-per-thread method [49], and Prefetch-CSR [53]. Depending on the matrix sparsity pattern and average number of non-zeros per row, the performance of the aforementioned techniques differ; the auto-tuner chooses the best performing heuristic for each matrix to be used for the matrix powers kernel.

Upon completion of the auto-tuning stage, the best performing implementation of the matrix powers kernel is determined and used in the matrix powers kernel described in the next section.

### *B. Matrix Powers Kernel*

Fig. 6.3 shows the algorithm to compute the matrix powers kernel using GPU shared memory. Each cache block is first loaded to shared memory, using the SpMV algorithm chosen by the auto-tuner, the matrix powers basis vector is then generated for each partition in parallel via GPU thread blocks. We present initial results for the matrix powers kernel shared memory cache blocking for a penta-

diagonal matrix in Section 6.4.2; improved partitioning schemes and more problems will be studied in future work.

## 6.4 Results

Performance results for the matrix powers kernel on NVIDIA GTX480 are presented in this section. The GTX480 graphic card contains 480 CUDA cores and operates at 1.4GHz, the size of global memory is 1.5GB with a bandwidth of 177 GB/s. The shared memory is configured to 48KB. All speedups are calculated using the following formula:

$$\frac{\text{time}(\text{matrix powers kernel for } (Ax, A^2x, \dots, A^kx))}{\text{time}(k \text{ SpMV standard operations})} \quad 6.1$$

Cache Blocking for Shared Memory
<p>for thread block <math>q</math></p> <p>copy partition (cache block) <math>q</math> to shared memory</p> <p>for <math>i = 1</math> to <math>k</math> do</p> <p>compute <math>x_j^{(i)}</math> (for all <math>j</math> belonging to the partition) using the fastest SpMV algorithm for the matrix (exploits parallelism via threads/warps inside a thread block)</p> <p>copy <math>x_j^{(i)}</math> (for all <math>j</math> belonging to the partition) to global memory</p>

Fig. 6.3: The matrix powers implementation on GPU shared memory,  $x_j^i$  is the  $j$ -th component of  $x^i = A^i x^{(0)}$ .

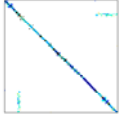



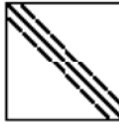

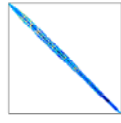
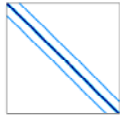
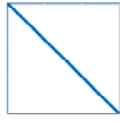

<b>Pwtk</b> <b>Wind Tunnel</b> <b>(218K, 12M, 55)</b> 	<b>Cant</b> <b>FEM cantilever</b> <b>(62K, 4M, 65)</b> 	<b>Cfd2</b> <b>Pressure matrix</b> <b>(123K, 3.1M, 25)</b> 	<b>Gearbox</b> <b>Aircraft flap actuator</b> <b>(153K, 9.1M, 59)</b> 	<b>2d 9-pt</b> <b>9-pt operator on 2Dmesh</b> <b>(1M, 9M, 9)</b> 
<b>mc2depi</b> <b>2D Markov model</b> <b>(525K, 2.1M, 4)</b> 	<b>Shipsec</b> <b>FEM ship section</b> <b>(141K, 7.8M, 55)</b> 	<b>Xenon</b> <b>Complex zelolite csrytals</b> <b>(157K, 3.9M, 25)</b> 	<b>Rajat31</b> <b>Circuit simulation</b> <b>(4.6M, 20.3M, 84)</b> 	<b>Cube_coup3d</b> <b>coupled consolidation</b> <b>(2.1M, 124M, 59)</b> 

Fig. 6.4: Each matrix is described by its name, description, number of rows, number of non-zeros, average number of non-zeros per row and its non-zero pattern representation.

Standard Implementation of $k$ SpMVs
for $i = 1$ to $k$ do  for each partition (cache block)  transfer the partition to GPU Global memory  call a GPU kernel to compute $x_j^{(i)}$ (for all $j$ belonging to the current partition)  transfer $x_j^{(i)}$ to CPU (for all $j$ belonging to the current partition)  remove the current partition from global memory

Fig. 6.5: The standard computation of  $k$  SpMV on the GPU,  $x_j^i$  is the  $j$ -th component of  $x^i = A^i x^{(0)}$ .

#### 6.4.1 Matrix Powers on GPU Global Memory

In this section the performance of the proposed implementation of the matrix powers kernel on GPU global memory is studied using ten matrices (Fig. 6.4) from

the University of Florida matrix repository [35]. All matrices are cache blocked assuming only one fourth of the matrix can be stored in global memory at one time. The  $k$  SpMV standard operations in equation 6.1 are computed using the implementation in Fig. 6.5.

Fig. 6.6 shows the performance of the matrix powers kernel for global memory cache blocking (the best performance obtained for all  $k < 40$ ). Speedups of up to 5.7 and 4.98 are achieved for well structured matrices, cant and 2d-9pt. The naive SpMV performance is lower for matrices with smaller numbers of non-zeros per row such as 2d9pt and mc2depi. The CUSPARSE SpMV implementation performs poorly for such problems due to an increase in thread divergence. The extra flops performed in the matrix powers kernel (for the best  $k$ ) compared to  $k$  steps of the standard SpMV is shown in Table 6.1. For an unstructured matrix such as xenon that achieves the least speedup from the matrix powers kernel, in only 5 steps of the matrix powers kernel up to 23% more flops are computed (Table 6.1). Upperbound in Fig. 6.6 is computed for the best performing  $k$  using:

$$\frac{\text{arithmetic\_intensity}(\text{matrix powers})}{\text{arithmetic\_intensity}(\text{SpMV})} \cdot \text{performance}(\text{SpMV}) \quad 6.2$$

where the arithmetic intensity is the effective flops to bytes transferred ratio. The generated  $x^i$  vectors (where  $x^i = A^i x^{(0)}$ ) are transferred to the CPU for both the naive SpMV and matrix powers kernels at each step. The aforementioned transfers are also included in computing the upperbound. Table 6.1 shows the fraction of total time spent in communicating data between GPU and CPU memory for all the tested problems (for the best performing  $k$ ). The table shows on average 90 percent of the SpMV kernel execution time is spent in transferring data between CPU and GPU global memory which further justifies the importance of avoiding communication using the matrix powers kernel. For matrices such as 2d-9pt and



mc2depi, which have the least number of non-zeros per row, a smaller percentage of total time is spent in communicating data. Also, compared to other matrices, the performance gap between the matrix powers kernel and the upperbound is larger for the aforementioned matrices. This is because the time spent in computing operations such as spreading the initial and source vectors at each step of the matrix powers kernel are no longer negligible for these problems. Increased thread divergence on the GPU for matrices with less number of non-zeros per row also increases the execution time of arithmetic computations for 2d9pt and mc2depi.

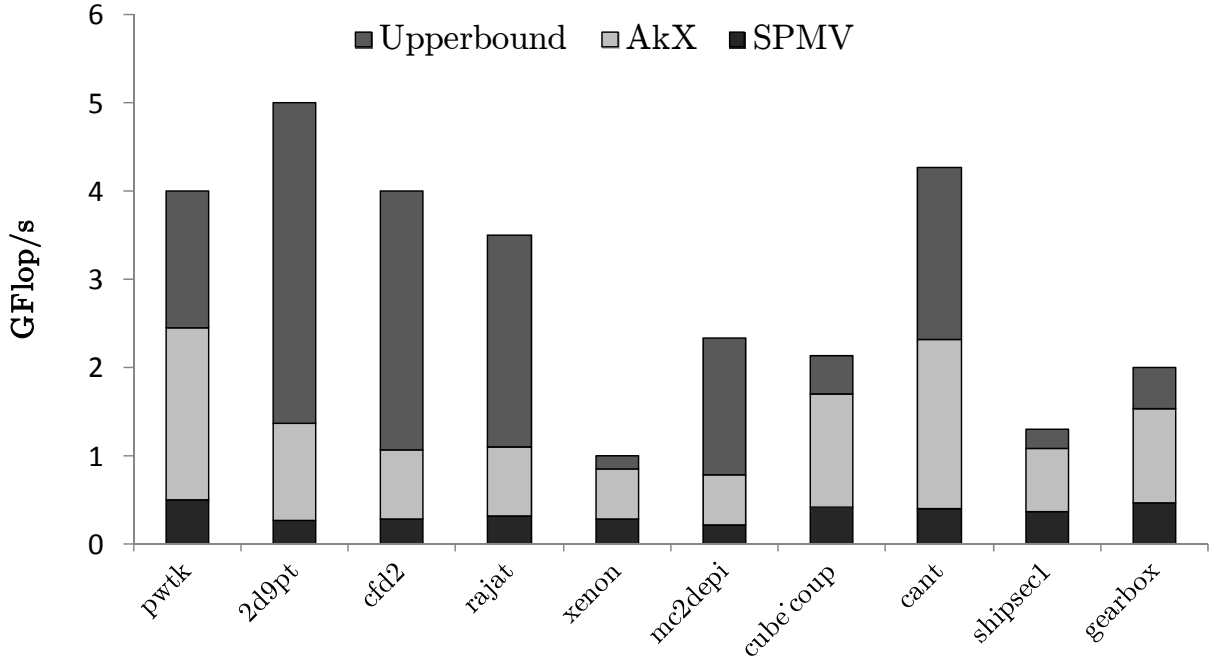


Fig. 6.6: Performance of the matrix powers kernel cache blocking for global memory on NVIDIA GTX480. The “AkX” indicates the best performance obtained for all  $k < 40$ . The label “upper bound” shows the performance achievable via scaling the standard  $k$  SpMV operations by the change in arithmetic intensity (equation 6.2). The “SpMV” bar shows the performance achieved from the standard  $k$  SpMV implementation using CUDA sparse library [89].

### 6.4.2 Matrix Powers on GPU Shared Memory

The performance of the matrix powers kernel on shared memory (cache blocking for shared memory) is tested for a pentadiagonal matrix with 100K rows and 500K non-zeros. Since shared memory is divided between the active thread blocks per GPU streaming multiprocessor, for the tested matrix the partitioning stage was only able to generate partitions for a maximum of four thread blocks per SM. Fig. 6.7 shows the performance of the matrix powers kernel for different  $k$  and possible thread blocks per SM. As shown for a pentadiagonal matrix, maximum performance is achieved when the active thread blocks per SM is set to 3. For larger TBs per SM, the small size of shared memory allocated to each thread block either leads to the failure in the partitioning phase or increases the redundant computations related to ghost rows in the matrix powers kernel. Smaller TBs/SM on the other hand, reduce the GPU resource occupy leading to poor performance.

Fig. 6.8 shows the effects of increasing  $k$  for the best TBs/SM chosen by the auto-tuner, which was 3 for the tested matrix. The matrix powers kernel achieved the best performance for  $k$  equal to 15 leading to 1.4 speedup compared to the standard SpMV implementation. Table 6.2 shows that for the optimum  $k$ , 30 percent extra flops are performed compared to the naive implementation and 1131 thread blocks are launched on the GPU to compute the matrix powers kernel for the tested pentadiagonal matrix.

Comparing Table 6.1 and Table 6.2, we find that the extra flops considerably grow when generating cache blocks for shared memory, decreasing the performance of the matrix powers kernel for shared memory cache blocking. Our experiments show that implementing the matrix powers kernel on architecture such as GPUs that have very small cache sizes, depends mostly on the partitioning technique and matrix structure. The partitioning stage is not able to create very small cache blocks for the tested matrices and the 1d5pt stencil matrix was the only matrix

with solid performance. Also since the hardware manages the execution of threads as well as data movement and prefetching, maintaining a balance between resource occupancy and acceptable partitions highly depends on the auto-tuning phase. We will study other partitioning schemes and enhance our auto-tuner to increase the performance of the matrix powers kernel for shared memory in future work.

Table 6.1: The best speedup of the matrix powers kernel compared to naive SpMV, fraction of total time spent in communicating data in the naïve SpMV implementation and extra computed flops in the matrix powers kernel performing  $k$ .

Matrix	pwtk	2d9pt	cfd2	rajat	xenon	mc2depi	cube coup	cant	shipsec1	gearbox
$k$	15	34	7	15	5	11	8	14	6	7
Speedup	4.92	4.98	3.79	3.49	2.85	3.53	3.98	5.7	2.88	3.21
Communication vs. Total time	91%	84%	90%	87%	87%	78%	88%	93%	93%	96%
AkXflops/ Naiveflops	1.3	1.1	1.2	1.03	1.23	1.02	1.22	1.16	1.24	1.26

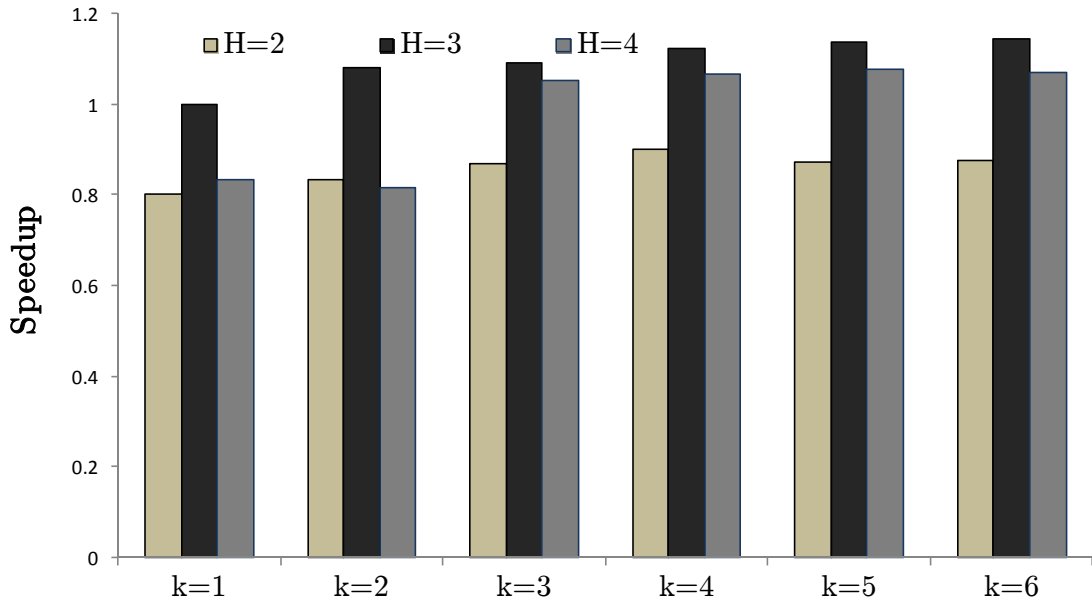


Fig. 6.7: The speedups achieved for equation for the matrix powers kernel on shared memory (test matrix: a pentadiagonal matrix) for different thread blocks per SM ( $H$  in the figure) and  $k$ .

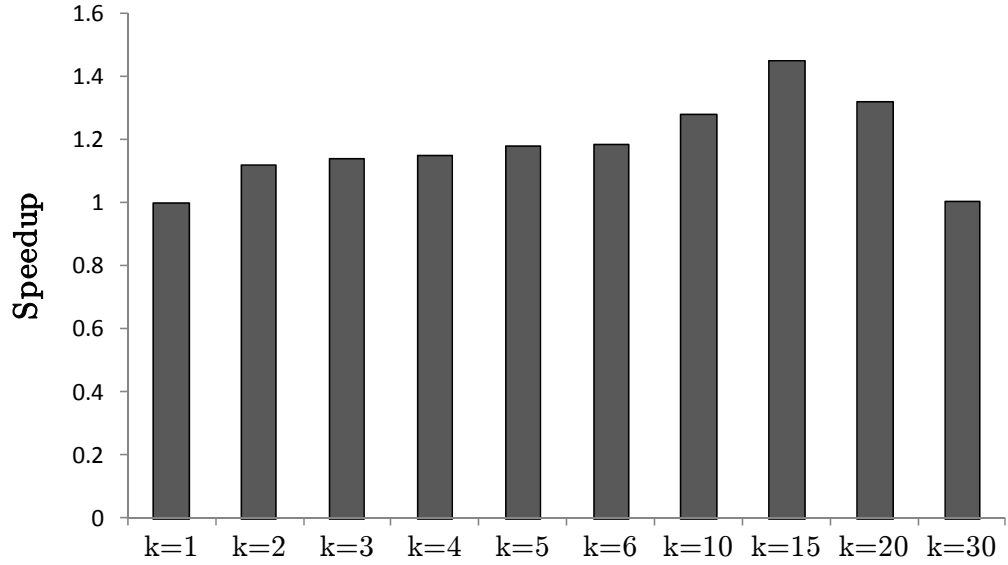


Fig. 6.8: The speedups achieved from equation for the matrix powers kernel on shared memory (test matrix: a pentadiagonal matrix) for the best performing number of thread blocks per SM (TB/SM) and different  $k$ .

Table 6.2: The extra floating point operations performed in the matrix powers kernel for shared memory compared to naive the SpMV implementation and the total number of thread blocks launched for each  $k$ .

$k$	2	3	4	5	6	10	15	20	30
AkXflops/ NaiveFlops	1.02	1.04	1.06	1.08	1.1	1.19	1.3	1.8	2.2
Thread Blocks	1024	1024	1024	1024	1024	1026	1132	2036	2017

## 6.5 Conclusion and Future Work

The matrix powers kernel in communication-avoiding Krylov techniques is accelerated and speedups of upto 5.7 are obtained for global memory cache blocking compared to the standard implementation of  $k$  SpMV operations. The matrix powers kernel shared memory cache blocking is also implemented and tested on a pentadiagonal matrix; in future work we intend to enhance the performance of this kernel by implementing other matrix partitioning schemes and enhancing the auto-tuning phase. The performance of the matrix powers kernel in Krylov subspace

methods will be studied and preconditioners such as the sparse approximate inverse will be used to enhance the convergence of communication-avoiding KSMs.

## Chapter 7 CONCLUSION AND FUTURE WORK

A brief outline of the main content and contributions of this work (Section 7.1) along with possible extensions to future work (Section 7.2) are presented in this chapter.

### 7.1 Conclusion

The initial two chapters of the thesis give an introduction to scientific computing and Krylov subspace techniques (Chapter 1) and architecture specifications and programming challenges of graphic processing units (Chapter 2). The proceeding four chapters introduce major contributions of the work which are outlined in the following:

- The sparse matrix vector multiplication kernel is accelerated on NVIDIA GPUs using a new algorithm called Prefetch-CSR (PCSR). Major contributions in the proposed algorithm are the introduction of a new partitioning scheme, a new sparse storage format suitable for GPUs, parallel reduction of the value vector via zero padding and data prefetching within a GPU thread block. Compared to previous implementations of the SpMV kernel on NVIDIA GT8800 the Prefetch-CSR algorithm was on average 3.37 times faster for the tested matrices.
- The Chronopoulos [8] variant of the conjugate gradient method is for the first time accelerated on GPUs. By fusing the main computing kernels in the aforementioned implementation, memory references and GPU kernel calls are reduced. Frequently used vectors are also loaded in to GPU caches (texture memory) to reduce the data communication overhead. Finally the PCSR SpMV kernel is further optimized and used in the GPU implementation of the Chronopoulos conjugate gradient implementation. The proposed optimizations increased the performance of the preconditioned conjugate

gradient algorithm on NVIDIA G80 and GT200 up to 3.4 and 2.5 respectively compared to previous accelerations of PCG on GPUs.

- The sparse approximate inverse preconditioner is accelerated on GPUs via computing columns of the preconditioner in parallel. Techniques to manage the limited memory space on GPU global memory for large problems, solve local systems within a GPU warp, gather the generated columns and assemble the preconditioner are proposed. Finally, the generated preconditioner is transferred to a BiCGStab iterative solver to enhance its convergence rate. For the tested matrices the SAI preconditioner was generated on average 28 times faster on GTX480 compared to the time required to create the same preconditioner on a single AMD Opteron 252 processor. The preconditioner is generated in approximately the same time on NVIDIA GTX480 and 16 AMD processors.
- The communication-avoiding matrix powers kernel is implemented on NVIDIA GPUs to reduce the communication overhead within the GPU memory hierarchy and between the GPU and CPU memory in Krylov subspace techniques. By dividing the matrix into balanced partitions that fit into the desired memory spaces and choosing the best algorithm in the auto-tuning phase, speedups of up to 5.7 are achieved for the  $k$ -step matrix powers kernel compared to  $k$  steps of the standard SpMV kernel on GTX480.

Optimizations and techniques presented throughout this work are tested on manycore GPUs; however, they are broadly applicable to current and future parallel architectures. The proposed sparse data structures and storage formats, data partitioning schemes, memory allocation strategies, communication-avoiding techniques and many other optimizations, exploit fine grain parallelism in compute intensive kernels in KSMs and can be used to efficiently implement and accelerate these kernels on modern manycore architecture.

## 7.2 Future Work

As the number of computing cores in processors increase, algorithms have to be modified to efficiently exploit fine grain parallelism and fully utilize the available resources on modern architectures. Techniques and algorithms proposed in this work enable the parallel execution of compute intensive kernels in Krylov solvers on modern manycore architectures. The proposed methodologies can be used in future manycore architectures such as Intel MIC processors and heterogeneous computing systems composed of different types of computational units.

Adaptive SAI preconditioning techniques will be implemented on GPUs in future work and used as a smoother in multigrid techniques [101]. SAI preconditioners are also suitable candidates for communication-avoiding Krylov techniques since columns of the preconditioner can be generated independently using only local partitions of the matrix. We intend to use this preconditioner to enhance the convergence rate of communication-avoiding KSMs in future work.

New matrix partitioning schemes will be implemented to enhance the performance of the matrix powers kernel on shared memory and more problems will be tested. Finally the matrix powers kernel on GPUs will be integrated into communication-avoiding Krylov techniques proposed in [10].



## References

- [1] Department of Energy FY 2012 Congressional Budget Req., [www.cfo.doe.gov/budget/12budget/content/volume4.pdf](http://www.cfo.doe.gov/budget/12budget/content/volume4.pdf)
- [2] Y. Liang, "The use of parallel polynomial preconditioners in the solution of systems of linear equations", Thesis, Faculty of informatics, University of Ulster, 2005.
- [3] Z. Bai, et al., "LAPACK users guide", 3rd Edition, SIAM, <http://www.netlib.org/lapack/index>, 1999.
- [4] S. F. Ashby, "Minimax polynomial preconditioning for Hermitian linear systems", SIAM J. Matrix Anal. Appl., 12(4), pp. 766-789, 1991.
- [5] Y. Saad, "Iterative methods for sparse linear systems", SIAM, Philadelphia, pp. 10-349, 2003.
- [6] M. R. Field, "Optimizing a parallel Conjugate Gradient solver", SIAM J. Sci. Comput. 19(1), pp. 27-37, 1998.
- [7] J. R. Shewchuk, "An introduction to the conjugate gradient method without the agonizing pain", Carnegie Mellon University, Pittsburgh, PA, USA, 1994.
- [8] A. Chronopoulos and C. W. Gear, "s-step iterative methods for symmetric linear systems", J. Comput. Appl. Math., vol. 25, no. 2, pp. 153-156, 1989.
- [9] P. Sonneveld, "A fast Lanczos-type solver for nonsymmetric linear systems", SIAM J. Sci. Statist. Comput., vol. 10, no.1, pp. 36-52, 1989.
- [10] M. Hoemmen, "Communication-avoiding Krylov subspace methods". Thesis UC Berkeley, Department of Computer Science, 2010.
- [11] D. Fernandez, D. Giannacopoulos, and W. Gross, "Efficient multicore sparse matrix-vector multiplication for FE electromagnetics", IEEE Trans. on Mag., vol. 45, no. 3, pp. 1392-1395, Mar. 2009.
- [12] D. Fernandez, D. Giannacopoulos, and W. J. Gross, "Multicore acceleration of CG algorithms using blocked-pipeline-matching techniques," IEEE Trans. on Mag., vol. 46, no. 8, pp. 3057-3060, 2010.
- [13] E. J. Im, and K. A. Yelick, "Optimizing sparse matrix vector multiplication on SMPs", in Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing, San Antonio, USA, 1999.
- [14] R. W. Vuduc, "Automatic performance tuning of sparse matrix kernels", PhD Thesis, University of California Berkeley, 2003.
- [15] E. J. Im, K. Yelick, and R. Vuduc, "Sparsity: optimization framework for sparse matrix kernels", International Journal of High Performance Computing Applications, vol. 18, no. 1, pp. 135-158, 2004.
- [16] Berkeley Benchmarking and Optimization Group: <http://bebop.cs.berkeley.edu>
- [17] R. Vuduc, J. W. Demmel, and K. A. Yelick. "OSKI: A library of automatically tuned sparse matrix kernels", Journal of Physics Conference Series, 16, pp. 521-530, 2005.
- [18] <http://bebop.cs.berkeley.edu/poski/index.php>
- [19] S. Toledo, "Improving the memory-system performance of sparse-matrix vector multiplication", IBM Journal of Research and Development, vol. 41, no. 6, pp. 711-725, Nov, 1997.

- 
- [20] R. Nishtala et al., "When cache blocking of sparse matrix vector multiply works and why", *Applicable Algebra in Engineering Communication and Computing*, vol. 18, no. 3, pp. 297-311, May, 2007.
  - [21] S. W. Williams et al., "The potential of the cell processor for scientific computing", in *Proceedings of the 3rd Conference on Computing Frontiers*, pp. 9-20, Italy, 2006.
  - [22] S. Williams et al., "Optimization of sparse matrix vector multiplication on emerging multicore platforms", *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pp. 38:1–38:12, Reno, Nevada, 2007.
  - [23] J. W. Demmel, M. T. Heath, and H. A. Van der Vorst, "Parallel Numerical Linear Algebra", CSD-92-703, UC-Berkeley, EECS Technical Reports, October 6, 1992.
  - [24] W. A. Wiggers et al., "Implementing the conjugate gradient algorithm on multicore systems", 2007 International Symposium on System-on-Chip Proceedings, pp. 11-14, 2007.
  - [25] J. Dongarra et al., "Solving linear systems on vector and shared memory computers", USA: Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1991.
  - [26] M. Mehri Dehnavi et. al., "Parallel Sparse Approximate Inverse Preconditioning on Graphic Processing Units", to appear in *IEEE Transactions on Parallel and Distributed Systems*, 2012.
  - [27] J. N. Shadid and R. S. Tuminaro, "A comparison of preconditioned nonsymmetric Krylov method on a large-scale MIMD machine", *SIAM J. Sci. Comput.* 15(2), pp. 440-459, 1994.
  - [28] A. M. Bruaset, "A Survey of preconditioned iterative methods, longman scientific & technical", Co-published in the United States with John Wiley & Sons, Inc., New York, 1995.
  - [29] G. K. Konstadinidis, "Challenges in microprocessor physical and power management design", in *VLSI Design, Automation and Test*, pp. 9-12, 2009.
  - [30] P. T. Stathis, "Sparse matrix vector processing formats", PhD Thesis, Delft University of Technology, 2004.
  - [31] I. S. Duff, A. M. Erisman, and J. K. Ried, "Direct methods for sparse matrices", Clarendon Press, Oxford, 1986.
  - [32] K. Gallivan, A. Sameh, and Z. Zlatev, "A parallel hybrid sparse linear solver system solver", *Computing Systems in Engineering*, June 1990.
  - [33] R. Boisvert et al., "matrix market", National Institute of Standards and Technology (NIST), <http://math.nist.gov/MatrixMarket/>, Gaithersburg, Maryland, 2011.
  - [34] T. A. Davis, and Y. Hu, "The university of Florida sparse matrix collection", *ACM Transactions on Mathematical Software* (to appear), <http://www.cise.ufl.edu/research/sparse/matrices>, January, 2009.
  - [35] [https://computing.llnl.gov/tutorials/parallel\\_comp/](https://computing.llnl.gov/tutorials/parallel_comp/)
  - [36] [www.setiathome.berkeley.edu](http://www.setiathome.berkeley.edu)
  - [37] "Cell Broadband Engine Programming Handbook", version 1.1, New York: IBM, [www.ibm.com/developerworks/power/cell/documents.html](http://www.ibm.com/developerworks/power/cell/documents.html), 2008.
  - [38] Intel MIC architecture:

- 
- <http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html>
- [39] M. Flynn, "Some computer organizations and their effectiveness", IEEE Trans. Comput. vol. 21, issue 9 pp. 948-960, 1972.
  - [40] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra, MPI: The Complete Reference. MIT Press Cambridge, MA, USA. ISBN 0-262-69215-5, 1995.
  - [41] OpenMP Application Program Interface, version 3.0, 2008, <http://www.openmp.org/mp-documents/spec30.pdf>.
  - [42] NVIDIA downloads and documentation [Online]. Available: <http://developer.nvidia.com/cuda-toolkit-downloads>.
  - [43] NVIDIA CUDA [Online]. Available: <http://developer.nvidia.com/object/cuda.html>.
  - [44] K. Karimi, N.G. Dickson, F. Hamze, "A performance comparison of CUDA and Open CL", Int. J. High Perform. Comput. Appl., 2011.
  - [45] CUDA occupancy calculator:  
[http://developer.download.nvidia.com/compute/cuda/4.0/sdk/docs/CUDA`Occupancy`Calculator.xls](http://developer.download.nvidia.com/compute/cuda/4.0/sdk/docs/CUDA%20Occupancy%20Calculator.xls)  
ble:
  - [46] J. D. Owens et al., "A survey of general-purpose computation on graphics hardware", Comput. Graphics Forum, pp. 80-113, 2007.
  - [47] L. Buatois, G. Caumon, and B. Levy, "Concurrent number cruncher: an efficient sparse linear solver on the GPU", in High Performance Computing and Communications. Berlin, Germany: Springer-Verlag, 2007, vol. 4782, Lecture Notes in Computer Science, pp. 358-371.
  - [48] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for GPU computing", in Proc. Graphics Hardw., 2007, pp. 97-106.
  - [49] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA Tech. Rep., 2008.
  - [50] S. Georgescu and H. Okuda, "GPGPU-Enhanced Conjugate Gradient Solver for Finite Element Matrices", Proc. of The Second international Workshop on Automatic Performance Tuning, 2007.
  - [51] D. Goddeke, R. Strzodka, and S. Turek, "Accelerating double precision FEM simulations with GPUs", ASIM, 2005.
  - [52] A. Cevahir, A. Nukada, and S. Matsuoka, "High performance conjugate gradient solver on multi-GPU clusters using hypergraph partitioning", J. of Research and Development, vol. 5, Issue. 1, pp. 83-91, 2010.
  - [53] M. Mehri Dehnavi, D. Fernandez and D. Giannacopoulos, "Finite element sparse matrix vector multiplication on GPUs", IEEE Trans. on Mag., vol. 46, no. 8, pp. 2982-2985, 2010.
  - [54] J.M. Jin, "The finite element method in electromagnetics", Wiley-IEEE Press, pp. 19-44, 2002.
  - [55] R. Barrett et al., "Templates for the Solution of Linear Systems", SIAM, 1994.
  - [56] J. Zhongxiao and Z. Baochen, "A power sparse approximate inverse preconditioning procedure for large sparse linear systems", Numerical Linear Algebra with Applications, vol. 16, no. 4, pp. 259-299,

2009.

- [57] E. Chow, "Parallel implementation and practical use of sparse approximate inverse preconditioners with a priori sparsity patterns", *Int. J. High Perform. Comput. Appl.*, vol. 15, no. 1, pp. 56-74, 2001.
- [58] E. Chow, "A priori sparsity patterns for parallel sparse approximate inverse preconditioners," *SIAM J. Scientific Computing*, vol. 21, no. 5, pp. 1804-1822, 1999.
- [59] <https://computation.llnl.gov/casc/parasails/parasails.html>.
- [60] T. Huckle, A. Kallischko, A. Roy, M. Sedlacek, and T. Weinzierl, "An efficient parallel implementation of the MSPAI preconditioner," *Parallel Computing*, vol. 36, no. 5-6, pp. 273-284, 2010.
- [61] M.J. Grote and T. Huckle, "Parallel preconditioning with sparse approximate inverses", *SIAM J. Scientific Computing*, vol. 18, no. 3, pp. 838-853, 1997.
- [62] P. Raghavan and K. Teranishi, "Parallel hybrid preconditioning: incomplete factorization with selective sparse approximate inversion," *SIAM J. Scientific Computing*, vol. 32, no. 3, pp. 1323-1345, 2010.
- [63] P. Gonzalez, T. F. Pena, and J. C. Cabaleiro, "Parallel sparse approximate preconditioners applied to the solution of BEM systems," *Engineering Analysis with Boundary Elements*, vol. 28, pp. 1061-1068, 2004.
- [64] M. Benson, J. Krettmann, and M. Wright, "Parallel algorithms for the solution of certain large sparse linear systems", *International J. of Computer Mathematics*, vol. 16, no. 3-4, pp. 245-260, 1984.
- [65] S. T. Barnard, L. M. Bernardo, and H. D. Simon, "An MPI implementation of the SPAI preconditioner on the T3E", *International J. of High Performance Comput. Appl.*, vol. 13, no. 2, pp. 107-123, 2010.
- [66] G. A. Gravavis, P.I. Matskanidis, K.M. Konstantinos, E.A. Lipitakis, "Finite element approximate inverse preconditioning using POSIX threads on multicore systems," *International Multiconference on Computer Science and Information Technology - IMCSIT*, pp. 297-302, 2010.
- [67] G. A. Gravavis, "High performance inverse preconditioning," *Archives of Computational Methods in Engineering*, vol. 16, no. 1, pp. 77-108, 2009.
- [68] K. Xu, D. Z. Ding, Z. H. Fan and R. S. Chen, "FSAI preconditioned CG algorithm combined with GPU technique for the finite element analysis of electromagnetic scattering problems," *Finite Elements in Analysis and Design*, vol. 47, no. 4, pp. 387-393, 2011.
- [69] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. Purcell, "A survey of general-purpose computation on graphics hardware", *Computer Graphics Forum*, vol. 26, no. 1, pp. 80-113, 2007.
- [70] D. F. Cosgrove, J.C. Dias, and A. Griewank, "Approximate inverse preconditioning for sparse linear systems", *International J. of Computer Mathematics*, vol. 44, pp. 91-110, 1992.
- [71] E. Chow and Y. Saad, "Approximate inverse preconditioners via sparse-sparse iterations", *SIAM J. Scientific Computing*, vol. 19, no. 3, pp. 995-1023, 1998.
- [72] <http://www.computational.unibas.ch/software/spai/spaidoc.html>.
- [73] M. J. Grote and H. D. Simon, "Parallel preconditioning and approximate inverses on the Connection

- 
- Machine”, *Parallel Processing for Scientific Computing*, vol. 2, p. 519-523.
- [74] T. Huckle, “Approximate sparsity patterns for the inverse of a matrix and preconditioning”, *Preliminary Proceedings IMACS World Congress on Scientific Computation*, 1997.
  - [75] L. Kolotilina, “Explicit preconditioning of systems of linear algebraic equations with dense matrices”, *SIAM Journal on Matrix Analysis and Applications*, vol. 13, pp. 2566-2573, 1992.
  - [76] L.Y. Kolotilina, and A.Y. Yeregin, “Factorized sparse approximate inverse preconditionings I. theory”, *SIAM Journal on Matrix Analysis and Applications*, vol. 14, no. 1, pp. 45-58, 1993.
  - [77] M. Benzi, C.D. Meyer, and M. Tuma, “A Sparse approximate inverse preconditioner for the Conjugate Gradient method”, *SIAM J. Scientific Computing*, vol. 17, no. 5, pp. 1135-1149, 1998.
  - [78] M. Benzi and M. Tuma, “A sparse approximate inverse preconditioner for nonsymmetric linear systems”, *SIAM J. Scientific Computing*, vol. 19, no. 3, pp. 968-994, 1998.
  - [79] M. Benzi and M. Tuma, “A comparative study of sparse approximate inverse preconditioners”, *Applied Numerical Mathematics*, pp. 305-340, 1999.
  - [80] M. Bollhofer and V. Mehrmann, “Algebraic multilevel methods and sparse approximate inverses”, *SIAM Journal on Matrix Analysis and Applications*, vol. 24, no. 1, pp. 191-218, 2002.
  - [81] M. Bollhofer and Y. Saad, “A factored approximate inverse preconditioner with pivoting”, *SIAM Journal on Matrix Analysis and Applications*, vol. 23, no. 3, pp. 692-705, 2001.
  - [82] S. A. Kharchenko, L. Yu. Kolotilina, A. A. Nikishin, and A. Yu. Yeregin, “A robust AINV-type method for constructing sparse approximate inverse preconditioners in factored form”, *Numerical Linear Algebra with Applications*, vol. 8, no. 3, pp. 165-179, 2001.
  - [83] M. Benzi, M. Tuma, “A comparative study of sparse approximate inverse preconditioners”, *Journal of Applied Numerical Mathematics*, vol. 30, pp. 305-340, 1999.
  - [84] W. Tang, “Towards an effective sparse approximate inverse preconditioner”, *SIAM Journal on Matrix Analysis and Applications*, vol. 20, pp. 970-986, 1999.
  - [85] J. Cosgrove, J. Diaz, and A. Williams, “Structural properties of the graph of augmented sparse approximate inverses”, *Proc. Symposium on Applied Computing*, pp. 131-136, 1990.
  - [86] G. Alljeon G, M. Benzi, and I. Giraud, “Sparse approximate inverse preconditioning for dense linear systems arising in computational electromagnetics”, *Numerical Algorithms*, vol. 16, pp. 1-15, 1997.
  - [87] W. Tang, W. Wan, “Sparse approximate inverse smoother for multigrid”, *SIAM Journal on Matrix Analysis and Applications*, vol. 21, pp. 1236-1252, 2000.
  - [88] T. D. Davis, “Direct methods for sparse linear systems”, *SIAM*, pp. 7-17, 2006.
  - [89] NVIDIA CUSPARSE Library:  
<http://developer.download.nvidia.com/compute/cuda/40rc2/toolkit/docs/CUSPARSELibrary.pdf>.
  - [90] NVIDIA CUBLAS:  
<http://developer.download.nvidia.com/compute/cuda/2.0/docs/CUBLASLibrary2.0.pdf>.
  - [91] <https://www.sharcnet.ca>.

- 
- [92] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick, "Avoiding communication in computing Krylov subspaces", Technical Report UCB/EECS-2007-123, University of California Berkeley EECS, 2007.
  - [93] J.V. Rosendale, "Minimizing inner product data dependencies in conjugate gradient iteration", IEEE Computer Society Press, Silver Spring, 1983.
  - [94] H.F. Walker, "Implementation of the GMRES method using Householder transformations", SIAM Journal on Scientific and Statistical Computing, pp. 9-152, 1988.
  - [95] A.C. Hindmarsh and H.F. Walker, "Note on a Householder implementation of the GMRES method", Technical report, Lawrence Livermore National Lab., USA, 1986.
  - [96] W.D. Joubert and G.F. Carey, "Parallelizable restarted iterative methods for nonsymmetric linear systems", Part I: Theory. International Journal of Computer Mathematics, 44(1), pp. 243-267, 1992.
  - [97] Z. Bai, D. Hu, and L. Reichel, "A Newton basis GMRES implementation", IMA Journal of Numerical Analysis, 14(4), pp. 563-581, 1994.
  - [98] M. Mohiyuddin, M. Hoemmen, J. Demmel and K. Yelick, "Minimizing. communication in sparse matrix solvers", Proceedings of the 2009 ACM/IEEE Conference on Supercomputing, New York, USA, Nov 2009.
  - [99] <http://glaros.dtc.umn.edu/gkhome/metis/metis>.
  - [100] <http://www.cs.sandia.gov/Zoltan/>.
  - [101] W. Tang and W. Wan, "Sprase approxiamte inverse smoother for multigrid", SIAM. J. Matrix Anal. Appl. vo. 21, pp. 1236-1252, 1999.

## Appendix I      The BiCGStab Iterative Technique

$$r_0 = b - Ax_0$$

Choose a vector  $\hat{r}_0$  such that  $(\hat{r}_0, r_0) \neq 0$

$$\rho_0 = \alpha = \omega_0 = 1$$

$$v_0 = p_0 = 0$$

For  $i = 1, 2, 3, \dots$

$$\rho_i = (\hat{r}_0, r_{i-1})$$

$$\beta = (\rho_i / \rho_{i-1}) / (\alpha / \omega_{i-1})$$

$$p_i = r_{i-1} + \beta(p_{i-1} - \omega_{i-1}v_{i-1})$$

$$y = M^{-1}p_i$$

$$v_i = Ay$$

$$\alpha = \rho_i / (\hat{r}_0, v_i)$$

$$s = r_{i-1} - \alpha v_i$$

$$z = M^{-1}s$$

$$t = Az$$

$$\omega_i = (M^{-1}t, M^{-1}s) / (M^{-1}t, M^{-1}t)$$

$$x_i = x_{i-1} + \alpha y + \omega_i z$$

If  $x_i$  is accurate enough then quit

Using a preconditioner  $M$  and the following formulation, the preconditioned BiCGStab iterative technique solves a linear system  $Ax = b$  starting with an initial guess  $x_0$ :

Appendix II      NVIDIA GPU Specifications

NVIDIA GPU generation	CUDA cores	Processor clock	Shared memory per SM	Registers per SM	Off-chip device memory	Memory bandwidth
GT8800	112	1.5GHZ	16KB	8K	512MB	57.6GB/sec
GTX280	240	1.29GHZ	16KB	16K	1GB	141.7GB/sec
GTX480	480	1.4GHZ	48KB	32K	1.5GB	177.4GB/sec
TESLA M2070	448	1.15GHZ	48KB	32K	6GB	150GB/sec