ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Προσομοίωση κυκλωμάτων μεγάλης κλίμακας με προρυθμιστές Steiner κόμβων σε παράλληλες αρχιτεκτονικές

Simulation of large-scale circuits with Steiner node preconditioners on parallel architectures

Διπλωματική Εργασία

Δημήτριος Κ. Γαρυφάλλου

Επιβλέποντες Καθηγητές : Νέστωρας Ευμορφόπουλος
Επίκουρος Καθηγητής

Χρήστος Δ. Αντωνόπουλος
Επίκουρος Καθηγητής

Βόλος, Σεπτέμβριος 2014

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Προσομοίωση κυκλωμάτων πολύ μεγάλης κλίμακας με χρήση steiner κόμβων σε παράλληλες αρχιτεκτονικές

Διπλωματική Εργασία

Δημήτριος Κ. Γαρυφάλλου

Επιβλέποντες : Νέστωρας Ευμορφόπουλος
Επίκουρος Καθηγητής

Χρήστος Δ. Αντωνόπουλος
Επίκουρος Καθηγητής

Εγκρίθηκε από την διμελή εξεταστική επιτροπή την $26^{η}$ Σεπτεμβρίου 2014

.....	.....
Ν. Ευμορφόπουλος	Χ. Δ. Αντωνόπουλος
Επίκουρος Καθηγητής	Επίκουρος Καθηγητής

Διπλωματική Εργασία για την απόκτηση του Διπλώματος του Ηλεκτρολόγου Μηχανικού και Μηχανικού Υπολογιστών, του Πανεπιστημίου Θεσσαλίας, στα πλαίσια του Προγράμματος Προπτυχιακών Σπουδών του Τμήματος Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Πανεπιστημίου Θεσσαλίας.

.....

Δημήτριος Γαρυφάλλου

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών
Πανεπιστημίου Θεσσαλίας

To my family and friends

Στην οικογένειά μου και στους φίλους μου

Ευχαριστίες

Με την περάτωση της παρούσας εργασίας, θα ήθελα να ευχαριστήσω θερμά τους επιβλέποντες της διπλωματικής εργασίας κ. Νέστορα Ευμορφόπουλο και κ. Χρήστο Δ. Αντωνόπουλο για την εμπιστοσύνη που επέδειξαν στο πρόσωπό μου με την ανάθεση του συγκεκριμένου θέματος, την άριστη συνεργασία και τη συνεχή καθοδήγηση, η οποία διευκόλυνε την εκπόνηση της διπλωματικής εργασίας.

Επίσης, θα ήθελα να ευχαριστήσω τους φίλους και συνεργάτες του εργαστηρίου Ε5 για την υποστήριξη και την δημιουργία ενός ευχάριστου και δημιουργικού κλίματος και ιδιαίτερα τον διδακτορικό φοιτητή Κωνσταντή Νταλούκα για τις εύστοχες υποδείξεις του και την συνεχή στήριξή του.

Τέλος, οφείλω ένα μεγάλο ευχαριστώ στην οικογένειά μου και στους φίλους μου για την αμέριστη υποστήριξη και την ανεκτίμητη βοήθεια που μου παρείχαν τόσο κατά την διάρκεια των σπουδών μου όσο και κατά την εκπόνηση της διπλωματικής μου εργασίας.

Δημήτριος Γαρυφάλλου

Βόλος, 2014

Contents

List of Tables

List of Figures

List of Algorithms

List of Acronyms

AMG	Algebraic Multigrid
BiCG	BiConjugate Gradient
CCS	Compressed Column Storage
CG	Conjugate Gradient
CMG	Combinatorial Multigrid
COO	Coordinate
CPU	Central Processing Unit
CSR	Compressed Row Storage
CUDA	Compute Unified Device Architecture
DC	Direct Current
DIA	Diagonal Format
DRAM	Dynamic Random-Access Memory
ECC	Error Checking & Correction
EDA	Electronic Design Automation
ELL	ELLPACK
FLOPs	Floating-Point Operations Per second
GCC	GNU Compiler Collection
GMG	Geometric Multigrid
GMRES	Generalized Minimal Residual
GPGPU	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
HPC	High Performance Computing
HYB	Hybrid
IBM	International Business Machines Corporation
IC	Integrated Circuit
ICC	Intel C++ Compiler
MG	Multigrid
MNA	Modified Nodal Analysis
NVCC	Nvidia CUDA Compiler
OP	Operating System
PCG	Preconditioned Conjugate Gradient
PKT	Packet
SDD	Symmetric Diagonally Dominant
segSpMV	Segmented Sparse Matrix Vector Multiplication
SIMD	Single Instruction, Multiple Data
SIMT	Single-Instruction Multiple-Thread

SM	Streaming Multiprocessor
SOR	Successive Overrelaxation
SPD	Symmetric Positive Definite
SpMV	Sparse Matrix Vector Multiplication
SSOR	Symmetric Successive Overrelaxation

Περίληψη

Η επίλυση γραμμικών συστημάτων της μορφής $Ax = b$, για συμμετρικούς πίνακες με κυρίαρχη διαγώνιο αποτελεί πρόβλημα θεμελιώδους θεωρητικής σημασίας καθώς επίσης χρησιμοποιείται σε αμέτρητες εφαρμογές στην αριθμητική ανάλυση, τη μηχανική και τις επιστήμες. Ο πρώτος αξιόπιστα αποδοτικός επιλυτής τέτοιων συστημάτων για γενικές και αυθαίρετα σταθμισμένες τοπολογίες, προτάθηκε μόλις πριν λίγα χρόνια. Ο επιλυτής αυτός στηρίζεται στις αρχές της θεωρίας γράφων και επιτυγχάνει εξαιρετικά αποτελέσματα ενώ παράλληλα παρέχει ισχυρές εγγυήσεις για την ταχύτητα σύγκλισης.

Σκοπός αυτής της διπλωματικής εργασίας είναι η επιτάχυνση της απόδοσης του συγκεκριμένου επιλυτή για συστήματα τα οποία εμφανίζονται στην προσομοίωση κυκλωμάτων πολύ μεγάλης κλίμακας. Οι πίνακες που εμφανίζονται σε αυτά τα μεγάλα συστήματα έχουν αραιή δομή με αποτέλεσμα οι μέθοδοι για τον αποδοτικό χειρισμό τους να είναι συχνά κρίσιμοι για την επίδοση πολλών εφαρμογών συμπεριλαμβανομένης και της προσομοίωσης κυκλωμάτων. Έχει αποδειχτεί πως οι πράξεις πολλαπλασιασμού αραιού πίνακα με διάνυσμα (SpMV) έχουν ιδιαίτερη σημασίας στην υπολογιστική επιστήμη. Αποτελούν το κυρίαρχο κόστος σε πολλές επαναληπτικές μεθόδους που χρησιμοποιούνται στην επίλυση μεγάλων γραμμικών συστημάτων και η επιτάχυνσή τους παραμένει πρόκληση για την επιστημονική κοινότητα.

Το εύρος ζώνης της μνήμης αποτελεί μείζον περιοριστικό παράγοντα για την απόδοση των επαναληπτικών μεθόδων που βασίζονται στις πράξεις πολλαπλασιασμού SpMV. Προκειμένου να ξεπεράσουμε αυτό το εμπόδιο, προσπαθήσαμε να ενσωματώσουμε έναν καινούριο γρήγορο παράλληλο αλγόριθμο υλοποιημένο σε μια κάρτα γραφικών (GPU), η οποία προσφέρει τεράστιες επιδόσεις σε πολλές υψηλής-απόδοσης εφαρμογές. Ο καινούριος αλγόριθμος, που αποκαλείται segSpMV, βασίζεται στην μορφή αποθήκευσης αραιών πινάκων συμπιεσμένων γραμμών (CSR) και μπορεί να εφαρμοστεί σε ένα εύρος αριθμητικών εφαρμογών τόσο δομημένων όσο και μη δομημένων πινάκων. Στην εργασία μας μελετάμε τις επιπτώσεις της χρήσης του αλγορίθμου αυτού στον γραφοθεωρητικό μας επιλυτή, ο οποίος αποκαλείται CMG.

Λέξεις Κλειδιά:
Γραμμικά συστήματα, Μέθοδοι επίλυσης, Προρυθμιστές, Πολλαπλασιασμός Αραιού Πίνακα με Διάνυσμα, Κάρτα Γραφικών, Προγραμματισμός Υψηλών Επιδόσεων

Abstract

The solution of linear systems in the form $Ax = b$, on symmetric diagonal dominant matrices (SDDs) is a problem of fundamental theoretical importance but also one with a myriad of applications in numerical mathematics, engineering and science. The first reliably efficient SDD solver for general and arbitrary weighted topologies was first proposed in recent years. The solver is based on support theory principles and it achieves state of the art empirical results while providing robust guarantees on the speed of convergence.

In this thesis, we try to accelerate the performance of this solver for systems that occur in very large scale circuit simulation. Matrices that arise in those very large systems are sparse matrices, and as a result, methods for efficiently manipulating them are often crucial to the performance of many applications including circuit simulation. Sparse matrix-vector multiplication (SpMV) operations have proven to be of particular importance in computational science. They represent the dominant cost in many iterative methods for solving large-scale linear systems and it remains a challenge for the research community to accelerate them.

Memory bandwidth is a major limiting factor in the performance of iterative algorithms that rely on SpMV. To overcome that limit, we try to apply a new fast parallel algorithm on GPU platforms, which offers a tremendous performance in many high-performance computing applications [1]. The new algorithm, called *segSpMV*, is based on the Compressed Sparse Row (CSR) format and can be applied to wide computational applications with both structured and unstructured matrices. We study the implications of using that SpMV kernel in the CMG solver.

Keywords:
Linear Systems, Solution Methods, Preconditioners, Sparse Matrix-Vector Multiplication, Graphics Processing Unit, High Performance Computing

Chapter 1

Introduction

1.1 Problem Description

Circuit simulation is a technique where a computer software is used to simulate the behavior of an electronic circuit or system, using mathematical models. New designs can be tested, evaluated and diagnosed without actually constructing the circuit or device. It is used across a wide spectrum of applications, ranging from integrated circuits(IC) and microelectronics to electrical power distribution networks and power electronics. Simulating a circuit's behavior before actually building it can greatly improve design efficiency by making faulty designs known as such, and providing insight into the behavior of electronics circuit designs. In particular, for integrated circuits, the tooling is expensive, breadboards are impractical, and probing the behavior of internal signals is extremely difficult. Therefore almost all integrated circuits design relies heavily on simulation.

1.2 Thesis Contribution

The core of circuit simulation is based on the solution of linear systems in the form $Ax = b$. Those systems arise after the Modified Nodal Analysis. In Electrical Engineering Modified Nodal Analysis or MNA is an extension of nodal analysis which not only determines the circuit's node voltages (as in classical nodal analysis), but also some branch currents [2]. Several algorithms are based on solving such sort of linear systems. The contribution of this thesis is the acceleration of the CMG solver for SDD systems that arise in circuit simulation.

Starting from a C implementation [11] of the algorithm we ported the most time consuming part of the solver to a GPU with a view to improve the performance of the solve phase of the CMG solver. The results of our evaluation showed that if we do not include the time spent transferring data between the CPU and GPU, we achieve time speedups up to 7.9x over the sequential version.

For the implementation we used the Compute Unified Device Architecture (CUDA) [3], which is an open-source programming and interfacing tool provided by NVIDIA. The GPU device we used for the benchmarking is the NVIDIA® TESLA™ C2075 with 448 CUDA cores.

1.3 Formation of the thesis

In chapter 2 we give background material on the existing solution methods of linear systems. We begin with a review of what sparsity means and we describe the most useful sparse matrix storage. Then, we mainly refer to stationary, nonstationary and multigrid methods.

In chapter 3 we review some basic notions of preconditioner matrices. We discuss about the importance of preconditioning, how it is used and how it helps to the convergence of the methods.

In chapter 4 we give some background material on support theory for graphs and we describe the Steiner preconditioner. In the second section we give some background material on solvers and we present CMG.

In chapter 5 we present the GPU architecture and the CUDA programming model. Also, we present the basic characteristics of the NVIDIA® TESLA™ C2075.

Finally, in chapter 6 we present our attempt to improve the performance of CMG. Firstly, we review both the segSpMV method and the existing methods that are relevant with segSpMV.

Then we describe our implementation and we present the results from the experiments conducted. In the last section we make a conclusion and we give some tips for further future improvement of our implementation.

Chapter 2

Solution Methods of the $Ax = b$

2.1 Introduction

There are two broad categories of methods for solving linear equations in the form $Ax = b$ when A is large and sparse: ***direct*** and ***iterative***. While for some techniques such as direct solvers, we only provide brief descriptions, for iterative solvers, we go into more depth to describe the algorithms, since they are of interest to us here.

A direct method for solving the system of equations $Ax = b$ is any method that produces the solution x after a finite number of operations. An example of a direct method is using Gaussian elimination to factor A into matrices L and U where L is lower triangular and U is upper triangular and then solving the triangular systems by forward and back substitution. Direct methods are typically preferred for dense linear systems. The problem with direct methods for sparse systems is that the amount of computational effort and storage required can be prohibitive [4].

An alternative to direct methods of solution are iterative methods, which involve the construction of a sequence $\{x^{(i)}\}$ of approximations to the solution x, for which $x^{(i)} \rightarrow x$. Iterative methods for solving general, large sparse linear systems have been gaining popularity in many areas of scientific computing. Until recently, direct solution methods were often preferred to iterative methods in real applications because of their robustness and predictable behavior. However, a number of efficient iterative solvers were discovered and the increased need for solving very large linear systems triggered a noticeable and rapid shift toward iterative techniques in many applications [5].

In this thesis we are interested only in iterative methods on sparse matrices. But before we analyze some of the most well-known, let's see what the term ***sparse*** refers to.

2.1.1 Sparsity Overview

Consider the solution of linear systems of the form

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

where A is a nxn matrix, and both x and b are $nx1$ vectors. Of special interest is the case where A is large and sparse. The term ***sparse*** above refers to the relative number of non-zeros in the matrix A. A nxn matrix A is considered to be ***sparse*** if A has only $O(n)$ non-zero entries. In this case, the majority of the entries in the matrix are zeros, which do not have to be explicitly stored. An *nxn* dense matrix has $\Omega(n^2)$ non-zeros. There are many ways of storing a sparse matrix. Whichever method is chosen, some form of compact data is required that avoids storing the numerically zero entries in the matrix. It needs to be simple and flexible so that it can be used in a wide range of matrix operations. This need is met by the primary data structure in CSparse[1], a compressed-column matrix [6]. Some basic operations that operate on this data structure are matrix-vector multiplication, matrix-matrix multiplication, matrix addition, and transpose.

The simplest sparse matrix data structure is a list of the nonzero entries in arbitrary order. The list consists of two integer arrays i and j and one real array x of length equal to the number of entries in the matrix

[1] CSparse is a C library which implements a number of direct methods for sparse linear systems.

For example, the matrix [7]

$$A = \begin{bmatrix} 4.5 & 0 & 3.2 & 0 \\ 3.1 & 2.9 & 0 & 0.9 \\ 0 & 1.7 & 3.0 & 0 \\ 3.5 & 0.4 & 0 & 1.0 \end{bmatrix} \tag{2.2}$$

is presented in ***zero-based triplet*** form below. A zero-based data structure for a mxn matrix contains row and column indices in the range 0 to m-1 and n-1, respectively.

$$\begin{aligned} i &= \{2,\ 1,\ 3,\ 0,\ 1,\ 3,\ 3,\ 1,\ 0,\ 2\} \\ j &= \{2,\ 0,\ 3,\ 2,\ 1,\ 0,\ 1,\ 3,\ 0,\ 2\} \\ x &= \{2,\ 1,\ 3,\ 0,\ 1,\ 3,\ 3,\ 1,\ 0,\ 2\} \end{aligned}$$

The triplet form is simple to create but difficult to use in most sparse matrix algorithms. The ***compressed-column* storage** (CCS) is more useful and is used in almost all functions in CSparse. An m-by-n sparse matrix that can contain up to $nzmax$ entries is represented with an integer array p of length $n + 1$, an integer array i of length $nzmax$, and a real array x of length $nzmax$. Row indices of entries in column *j* are stored in $i[p[j]]$ through $i[p[j\ + 1] - 1]$, and the corresponding numerical values are stored in the same locations in x. The first entry $p[0]$ is always zero, and $p[n]\ \leq\ nzmax$ is the number of actual entries in the matrix. The example matrix (2.2) is represented as

$$\begin{aligned} p &= \{\ 0,\ \ 3,\ \ 6,\ 8,\ \ 10\} \\ i &= \{\ 0,\ \ 1,\ \ 3,\ 1,\ \ \ 2,\ \ 3,\ \ 0,\ \ 2,\ \ \ 1,\ 3\} \\ x &= \{4.5,\ 3.1,\ 3.5,\ 2.9,\ 1.7,\ 0.4,\ 3.2,\ 3.0,\ 0.9,\ 1.0\} \end{aligned}$$

One of the goals of dealing with sparse matrices is to make efficient use of the sparsity in order to minimize storage throughout the computations, as well as to minimize the required number of operations. Sparse linear systems are often solved using different computational techniques than those employed to solve dense systems.

2.2 Overview of the Methods

Below are short descriptions of each of the methods to be discussed, along with brief notes on the classification of the methods in terms of the class of matrices for which they are most appropriate. In later sections of this chapter more detailed descriptions of these methods are given [8].

- **Stationary Methods**

  – Jacobi.
  The Jacobi method is based on solving for every variable locally with respect to the other variables; one iteration of the method corresponds to solving for every variable once. The resulting method is easy to understand and implement, but convergence is slow.

  – Gauss-Seidel
  The Gauss-Seidel method is like the Jacobi method, except that it uses updated values as soon as they are available. In general, if the Jacobi method converges, the Gauss-Seidel method will converge faster than the Jacobi method, though still relatively slowly.

– SOR
Successive Overrelaxation (SOR) can be derived from the Gauss-Seidel method by introducing an extrapolation parameter ω. For the optimal choice of ω, SOR may converge faster than Gauss-Seidel by an order of magnitude.

• **Nonstationary Methods**

– Conjugate Gradient (CG).
The conjugate gradient method derives its name from the fact that it generates a sequence of conjugate (or orthogonal) vectors. These vectors are the residuals of the iterations. They are also the gradients of a quadratic functional, the minimization of which is equivalent to solving the linear system. Conjugate gradient (CG) is an extremely effective method when the coefficient matrix is symmetric positive definite (SPD), since storage for only a limited number of vectors is required.

– Generalized Minimal Residual (GMRES).
The Generalized Minimal Residual method computes a sequence of orthogonal vectors, and combines these through a least-squares solve and update. However, it requires storing the whole sequence, so that a large amount of storage is needed. For this reason, restarted versions of this method are used. In restarted versions, computation and storage costs are limited by specifying a fixed number of vectors to be generated. This method is useful for general nonsymmetric matrices.

– BiConjugate Gradient (BiCG).
The biconjugate gradient (BiCG) method generates two CG-like sequences of vectors, one based on a system with the original coefficient matrix A, and one on A^T. Instead of orthogonalizing each sequence, they are made mutually orthogonal, or "bi-orthogonal". This method, like CG, uses limited storage. It is useful when the matrix is nonsymmetric and nonsingular; however, convergence may be irregular, and there is a possibility that the method will break down. BiCG requires a multiplication with the coefficient matrix and with its transpose at each iteration.

2.3 Stationary Methods

Iterative methods that can be expressed in the simple form

$$x^{(k)} = Bx^{(k-1)} + c, \tag{2.3}$$

(where neither B nor c depend upon the iteration count k) are called $stationary$ iterative methods. In this section, we present the three main stationary iterative methods: the ***Jacobi method***, the ***Gauss-Seidel method*** and the ***Successive Overrelaxation (SOR) method.***

2.3.1 The Jacobi Method

The Jacobi method is easily derived by examining each of the *n* equations in the linear system $Ax = b$ in isolation. If in the i^{th} equation

$$\sum_{j=1}^{n} a_{i,j}\, x_j = b_i,$$

we solve for the value of x_i while assuming the other entries of x remain fixed, we obtain

$$x_i = (b_i - \sum_{j \neq i} a_{i,j}\, x_j)/a_{i,i}\,. \tag{2.4}$$

This suggests an iterative method defined by

$$x_i^{(k)} = \left(b_i - \sum_{j \neq i} a_{i,j}\, x_j^{(k-1)}\right)/a_{i,i} \tag{2.5}$$

which is the Jacobi method. Note that the order in which the equations are examined is irrelevant, since the Jacobi method treats them independently. For this reason, the Jacobi method is also known as the method of ***simultaneous displacements***, since the updates could in principle be done simultaneously.

In matrix terms, the definition of the Jacobi method in (2.3) can be expressed as

$$x^{(k)} = D^{-1}(L + U)x^{(k-1)} + D^{-1}b, \tag{2.6}$$

where the matrices D, $-L$ and $-U$ represent the diagonal, the strictly lower-triangular, and the strictly upper-triangular parts of A, respectively.

The pseudocode for the Jacobi method is given below in Figure 2.1. Note that an auxiliary storage vector, x is used in the algorithm. It is not possible to update the vector x in place, since values from $x^{(k-1)}$ are needed throughout the computation of $x^{(k)}$

```
Choose an initial guess x^(0) to the solution x.
for k = 1, 2, ...
    for i = 1, 2, ..., n
        x̄_i = 0
        for j = 1, 2, ..., i − 1, i + 1, ..., n
            x̄_i = x̄_i + a_{i,j} x_j^(k−1)
        end
        x̄_i = (b_i − x̄_i)/a_{i,i}
    end
    x^(k) = x̄
    check convergence; continue if necessary
end
```

Figure 2.1: The Jacobi Method

2.3.2 The Gauss-Seidel Method

Consider again the linear equations (2.2). If we proceed as with the Jacobi Method, but now assume that the equations are examined one at a time in sequence, and the previously computed results are used as they are available, we obtain the Gauss-Seidel method pseudocode in Figure 2.2.

Choose an initial guess $x^{(0)}$ to the solution x.
for $k = 1, 2, \ldots$
  for $i = 1, 2, \ldots, n$
    $\sigma = 0$
    for $j = 1, 2, \ldots, i-1$
      $\sigma = \sigma + a_{i,j} x_j^{(k)}$
    end
    for $j = i+1, \ldots, n$
      $\sigma = \sigma + a_{i,j} x_j^{(k-1)}$
    end
    $x_i^{(k)} = (b_i - \sigma)/a_{i,i}$
  end
  check convergence; continue if necessary
end

Figure 2.2: The Gauss-Seidel Method

$$x_i^{(k)} = \left(b_i - \sum_{i>j} a_{i,j}\, x_j^{(k)} - \sum_{j>i} a_{i,j}\, x_j^{(k-1)}\right)/a_{i,i} \tag{2.7}$$

Two important facts about the Gauss-Seidel method should be noted. First, the computations in (2.5) appear to be serial. Since each component of the new iterate depends upon all previously computed components, the updates cannot be done simultaneously as in the Jacobi method. Second, the new iterate $x^{(k)}$ depends upon the order in which the equations are examined. The Gauss-Seidel method is sometimes called the method of successive displacements to indicate the dependence of the iterates on the ordering. If this ordering is changed, the components of the new iterate (and not just their order) will also change.

These two points are important because if A is sparse, the dependency of each component of the new iterate on previous components is not absolute. The presence of zeros in the matrix may remove the influence of some of the previous components. Using a judicious ordering of the equations, it may be possible to reduce such dependence, thus restoring the ability to make updates to groups of components in parallel. However, reordering the equations can affect the rate at which the Gauss-Seidel method converges. A poor choice of ordering can degrade the rate of convergence; a good choice can enhance the rate of convergence.

In matrix terms, the definition of the Gauss-Seidel method in (2.5) can be expressed as

$$x^{(k)} = (D - L)^{-1}(Ux^{(k-1)} + b) \tag{2.8}$$

As before D, $-L$ and $-U$ represent the diagonal, lower-triangular, and upper-triangular parts of A, respectively.

2.3.3 The Successive Overrelaxation Method (SOR)

The Successive Overrelaxation Method, or SOR, is devised by applying extrapolation to the Gauss-Seidel method. This extrapolation takes the form of a weighted average between the previous iterate and the computed Gauss-Seidel iterate successively for each component:

$$x_i^{(k)} = \omega \overline{x_i}^{(k)} + (1 - \omega) x_i^{(k-1)}.$$

(where $\overline{x_i}$ denotes a Gauss-Seidel iterate, and ω is the extrapolation factor). The idea is to choose a value for ω that will accelerate the rate of convergence of the iterates to the solution.

In matrix terms, the successive overrelaxation (SOR) algorithm can be written as follows:

$$x^{(k)} = (D - \omega L)^{-1}(\omega U + (1 - \omega)D)x^{(k-1)} + \omega(D - \omega L)^{-1}b\,. \qquad (2.9)$$

The pseudocode for the SOR algorithm is given above in Figure 2.3.

Choose an initial guess $x^{(0)}$ to the solution x.
for $k = 1, 2, \ldots$
  for $i = 1, 2, \ldots, n$
    $\sigma = 0$
    for $j = 1, 2, \ldots, i-1$
      $\sigma = \sigma + a_{i,j}x_j^{(k)}$
    end
    for $j = i+1, \ldots, n$
      $\sigma = \sigma + a_{i,j}x_j^{(k-1)}$
    end
    $\sigma = (b_i - \sigma)/a_{i,i}$
    $x_i^{(k)} = x_i^{(k-1)} + \omega(\sigma - x_i^{(k-1)})$
  end
  check convergence; continue if necessary
end

Figure 2.3: The SOR Method

2.4 Nonstationary Methods

Nonstationary methods differ from stationary methods in that the computations involve information that changes at each iteration. Typically, constants are computed by taking inner products of residuals or other vectors arising from the iterative method.

2.4.1 Generalized Minimal Residual (GMRES)

The GMRES method generates a sequence of orthogonal vectors, but in the absence of symmetry this can no longer be done with short recurrences; instead, all previously computed vectors in the orthogonal sequence have to be retained. For this reason are used restarted versions of the method. The GMRES algorithm has the property that residual norm $\| b - Axi \|$ can be computed without the iterate having been formed. Thus, the expensive action of forming the iterate can be postponed until the residual norm is deemed small enough. The GMRES iterates are constructed as:

$$x^i = x^0 + y_1u^1 + \ldots + y_iu^i, \qquad (2.10)$$

The GMRES method retains orthogonality of the residuals by using long recurrences, at the cost of a larger storage demand.

The pseudocode for the restarted GMRES algorithm with preconditioner M is given in Figure 2.4.

$x^{(0)}$ is an initial guess
for $j = 1, 2, ...$
  Solve r from $Mr = b - Ax^{(0)}$
  $v^{(1)} = r/\|r\|_2$
  $s := \|r\|_2 e_1$
  for $i = 1, 2, ..., m$
    Solve w from $Mw = Av^{(i)}$
    for $k = 1, ..., i$
      $h_{k,i} = (w, v^{(k)})$
      $w = w - h_{k,i}v^{(k)}$
    end
    $h_{i+1,i} = \|w\|_2$
    $v^{(i+1)} = w/h_{i+1,i}$
    apply $J_1, ..., J_{i-1}$ on $(h_{1,i}, ..., h_{i+1,i})$
    construct J_i, acting on ith and $(i+1)$st component
    of $h_{.,i}$, such that $(i+1)$st component of $J_i h_{.,i}$ is 0
    $s := J_i s$
    if $s(i+1)$ is small enough then (UPDATE$(\tilde{x}, i)$ and quit)
  end
  UPDATE$(\tilde{x}, m)$
end

In this scheme UPDATE$(\tilde{x}, i)$
replaces the following computations:

Compute y as the solution of $Hy = \tilde{s}$, in which
the upper $i \times i$ triangular part of H has $h_{i,j}$ as
its elements (in least squares sense if H is singular),
$\tilde{s}$ represents the first i components of s
$\tilde{x} = x^{(0)} + y_1 v^{(1)} + y_2 v^{(2)} + ... + y_i v^{(i)}$
$s^{(i+1)} = \|b - A\tilde{x}\|_2$
if $\tilde{x}$ is an accurate enough approximation then quit
else $x^{(0)} = \tilde{x}$

Figure 2.4: The Preconditioned GMRES(m) Method

2.4.2 Conjugate Gradient (CG)

The Conjugate Gradient method is an effective method for symmetric positive definite systems. It is the oldest and best known of the nonstationary methods discussed here. The method proceeds by generating vector sequences of iterates (***i.e.***, successive approximations to the solution), residuals corresponding to the iterates, and search directions used in updating the iterates and residuals. Although the length of these sequences can become large, only a small number of vectors needs to be kept in memory. In every iteration of the method, two inner products are performed in order to compute update scalars that are defined to make the sequences satisfy certain orthogonality conditions. On a symmetric positive definite linear system these conditions imply that the distance to the true solution is minimized in some norm.

Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
for $i = 1, 2, \ldots$
  solve $Mz^{(i-1)} = r^{(i-1)}$
  $\rho_{i-1} = r^{(i-1)^T} z^{(i-1)}$
  if $i = 1$
    $p^{(1)} = z^{(0)}$
  else
    $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
    $p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$
  endif
  $q^{(i)} = Ap^{(i)}$
  $\alpha_i = \rho_{i-1}/p^{(i)^T} q^{(i)}$
  $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$
  $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$
  check convergence; continue if necessary
end

Figure 2.5: The Preconditioned Conjugate Gradient Method

The pseudocode for the Preconditioned Conjugate Gradient (PCG) Method is given above in Figure 2.5. It uses a preconditioner *M*; for *M* = *I* one obtains the unpreconditioned version of the Conjugate Gradient Algorithm.

2.4.3 BiConjugate Gradient (BiCG)

The Conjugate Gradient method is not suitable for nonsymmetric systems because the residual vectors cannot be made orthogonal with short recurrences. The GMRES method retains orthogonality of the residuals by using long recurrences, at the cost of a larger storage demand. The BiConjugate Gradient method takes another approach, replacing the orthogonal sequence of residuals by two mutually orthogonal sequences, at the price of no longer providing a minimization.

The update relations for residuals in the Conjugate Gradient method are augmented in the BiConjugate Gradient method by relations that are similar but based on A^T instead of A. The pseudocode for the Preconditioned BiConjugate Gradient Method with preconditioner *M* is given in the top of the next page in Figure 2.6.

Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$.
Choose $\tilde{r}^{(0)}$ (for example, $\tilde{r}^{(0)} = r^{(0)}$).
for $i = 1, 2, \ldots$
  solve $Mz^{(i-1)} = r^{(i-1)}$
  solve $M^T\tilde{z}^{(i-1)} = \tilde{r}^{(i-1)}$
  $\rho_{i-1} = z^{(i-1)^T}\tilde{r}^{(i-1)}$
  if $\rho_{i-1} = 0$, **method fails**
  if $i = 1$
    $p^{(i)} = z^{(i-1)}$
    $\tilde{p}^{(i)} = \tilde{z}^{(i-1)}$
  else
    $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
    $p^{(i)} = z^{(i-1)} + \beta_{i-1}p^{(i-1)}$
    $\tilde{p}^{(i)} = \tilde{z}^{(i-1)} + \beta_{i-1}\tilde{p}^{(i-1)}$
  endif
  $q^{(i)} = Ap^{(i)}$
  $\tilde{q}^{(i)} = A^T\tilde{p}^{(i)}$
  $\alpha_i = \rho_{i-1}/\tilde{p}^{(i)^T}q^{(i)}$
  $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$
  $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$
  $\tilde{r}^{(i)} = \tilde{r}^{(i-1)} - \alpha_i\tilde{q}^{(i)}$
  check convergence; continue if necessary
end

Figure 2.6: The Preconditioned BiConjugate Gradient Method

2.5 Computational Aspects of the Methods

Efficient solution of a linear system includes the selection of the proper choice of iterative method. However, to obtain good performance, consideration must also be given to the computational kernels of the method and how efficient they can be executed on the target architecture. The performance of direct methods, is largely that of the factorization of the matrix. However, this lower efficiency of execution does not imply anything about the total solution time for a given system. Furthermore, iterative methods are usually simpler to implement than direct methods, and since no full factorization has to be stored, they can handle much larger systems than direct methods. Table 2.1 lists the type of operations performed per iteration and the storage required for each method (without preconditioning).

Method	Inner Product	SAXPY	Matrix-Vector Product	Precond Solve	Storage Reqmnts
JACOBI			1^a		Matrix + 3n
Gauss Seidel		1	1^a		
SOR		1	1^a		Matrix + 2n
GMRES	i+1	i+1	1	1	Matrix + (i+5)n
CG	2	3	1	1	Matrix + 6n
BiCG	5	5	1/1	1/1	Matrix + 10n

Table 2.1: Summary of Operations for Iteration i:"a/b" means "a" multiplications with the matrix and "b" with its transpose. Storage requirements for the methods in iteration i: n denotes the order of the matrix.

2.6 Multigrid Methods

Before closing this chapter we would like to discuss about the multigrid (MG) methods. MG methods in numerical analysis is defined as a group of algorithms for solving differential equations using a hierarchy of discretizations. They are an example of a class of techniques called multiresolution methods, very useful in problems exhibiting multiple scales of behavior. For example, many basic relaxation methods exhibit different rates of convergence for short- and long-wavelength components, suggesting these different scales be treated differently, as in a Fourier analysis approach to multigrid. MG methods can be used as solvers as well as preconditioners.

The main idea of MG is to accelerate the convergence of a basic iterative method by global correction from time to time, accomplished by solving a coarse problem[2]. This principle is similar to interpolation between coarser and finer grids. The typical application for multigrid is in the numerical solution of elliptic partial differential equations in two or more dimensions.

Multigrid can be applied in combination with any of the common discretization techniques.MG methods are among the fastest solution techniques known today. In contrast to other methods, multigrid methods are general in that they can treat arbitrary regions and boundary conditions. They do not depend on the separability of the equations or other special properties of the equation.

[2] Coarse problem is an auxiliary system of equations used in an iterative method for the solution of a given larger system of equations. It is basically a version of the same problem at a lower resolution, retaining its essential characteristics, but with fewer variables.

Chapter 3

Introduction to Preconditioners

3.1 Introduction

In chapter 2 we discussed about many iterative methods. The convergence rate of iterative methods depends on spectral properties of the coefficient matrix. Hence one may attempt to transform the linear system into one that is equivalent in the sense that it has the same solution, but that has more favorable spectral properties. A ***preconditioner*** is a matrix that effects such a transformation. For SPD systems, the rate of convergence of the conjugate gradient method depends on the distribution of the eigenvalues of A. The purpose of preconditioning is that the transformed matrix in question will have a smaller spectral condition number, and/or eigenvalues clustered around 1. For nonsymmetric problems the situation is more complicated, and the eigen-values may not describe the convergence of nonsymmetric matrix iterations like GMRES. On parallel machines there is a further tradeoff between the efficacy of a preconditioner in the classical sense, and its parallel efficiency. Many of the traditional preconditioners have a large sequential component.

If M is a nonsingular matrix that approximates A, then the linear system (3.1) has the same solution as (2.1) but must be significantly easier to solve.

$$M^{-1}Ax = M^{-1}b\ , \tag{3.1}$$

$$AM^{-1}y = b,\ x = M^{-1}y \tag{3.2}$$

$$M_1^{-1}AM_2^{-1}y = M_1^{-1}b,\ x = M_2^{-1}y \tag{3.3}$$

The system (3.1) is preconditioned from the left, (3.2) is preconditioned from the right. At (3.3) is performed split preconditioning where the preconditioner is $M = M1M2$.

Iterative algorithms such as the Conjugate Gradient method, converge to a solution using only matrix-vector products with A. It is well known that iterative algorithms suffer from slow convergence properties when the condition number of A, $\kappa(A)$, which is defined as the ration of the largest over the minimum eigenvalue of A, is large. What preconditioned iterative methods attempt to do is to remedy the problem by changing the linear system to $M^{-1}Ax = M^{-1}b$. In this case, the algorithms use matrix-vector products with A, and solve linear systems of the form $My = z$. So now the speed of convergence depends on the ***condition number*** $\kappa(A, M)$.

The condition number is defined as:

$$\kappa(A, M) = max_x \frac{x^T Ax}{x^T Mx} \cdot max_x \frac{x^T Mx}{x^T Ax}\ . \tag{3.4}$$

where x is taken to be outside the null space of A. There are two contradictory goals one has to deal in constructing a preconditioner M: (i) The linear systems in M must be easier than those in A to solve, (ii) The condition number must be small so it will minimize the number of iterations.

Historically, preconditioners were natural parts of the matrix A. We analyze some of the most well-known preconditioners below.

3.2 Jacobi Preconditioner

The simplest preconditioner consists of just the diagonal of the matrix

$$m_{i,j} = \begin{cases} a_{i,i}, & if\ i = j \\ 0, & otherwise \end{cases}$$

This is known as the (point) Jacobi preconditioner.

For the model problem, $\kappa(B^{-1}A) = O(n) = \kappa(A)$, so the asymptotic rate of convergence is not improved with diagonal scaling. *B* in this case does not need to be factored. The storage required for the preconditioner is $O(n)$ since it is a sparse matrix. And, the preconditioner system is very easy to solve, since it simply requires dividing each vector entry by the corresponding diagonal entry of *B*.

Even through the asymptotic rate of convergence is not improved, diagonal scaling can sometimes make the difference between convergence and non-convergence for an ill-conditioned matrix A. Moreover, diagonal scaling generally achieves some reduction in the number of iterations, and is so cheap to apply that it might as well be done.

3.3 SSOR Preconditioner

Another example of a preconditioner is the SSOR preconditioner which like the Jacobi preconditioner, can be easily derived from the coefficient matrix without any work.

Assume we have a symmetric matrix *A*. If this matrix is decomposed as

$$A = D + L + L^T$$

in its diagonal, lower, and upper triangular part, the SSOR matrix is defined as

$$M = (D + L)D^{-1}(D + L)^T$$

or, parametrized by ω

$$M(\omega) = \tfrac{1}{2-\omega}\left(\tfrac{1}{\omega}D + L\right)\left(\tfrac{1}{\omega}D\right)^{-1}\left(\tfrac{1}{\omega}D + L\right)^T.$$

The SSOR matrix is given in factored form, so this preconditioner shares many properties of other factorization-based methods. For example, its suitability for vector processors or parallel architectures depends strongly on the ordering of the variables.

3.4 Incomplete Factorization Preconditioners

A broad class of preconditioners is based on incomplete factorizations of the coefficient matrix. We call a factorization incomplete if during the factorization process certain fill elements, nonzero elements in the factorization in positions where the original matrix had a zero, have been ignored. Such a preconditioner is then given in factored form $M + LU$ with L lower and U upper triangular. The efficacy of the preconditioner depends on how well M^{-1} approximates A^{-1}.

When a sparse matrix is factored by Gaussian elimination, fill-in usually takes place. In that case, sparsity-preserving pivoting techniques can be used to reduce it. The triangular factors L and U of the coefficient matrix A are considerably less sparse than A.

Sparse direct methods are not considered viable for solving large linear systems due to time and space limitations, however, by discarding part of the fill-in in the course of the factorization process, simple but powerful preconditioners can be obtained in the form $M = LU$, where L and U are the incomplete (approximate) LU factors.

Summarizing, it can be said that existing solutions to the problem for incomplete factorization preconditioners for general SPD matrices follow one of two cases: simple inexpensive fixes that result in low quality preconditioners in terms of convergence rating, or sophisticated, expensive strategies that produce high quality preconditioners.

Chapter 4

A Multigrid-Like SDD solver

In this chapter we give some background material on support theory of preconditioning and we describe CMG, the solver that we studied and tried to optimize. The CMG was proposed by I. Koutis and Gary Miller and is characterized by the form of the preconditioner [9] [10] . The first implementation was in MATLAB [11] and later transformed into C code [12] . The basis of our implementation is the C code of the CMG solver.

4.1 Support Theory for Preconditioning

The main goal of the support theory is to provide techniques to bound the generalized eigenvalues and condition number for a matrix pencil (A, B) where B is a preconditioner for A. In this section we review fragments of support theory that are relevant to the design of the CMG. We refer the reader to [13] for an extensive explosion of support theory.

4.1.1 Electric Networks as Graphs – Support Basics

The cornerstone of combinatorial preconditioners is the following intuitive yet paradigm-shifting idea explicitly proposed by Vaidya [14]: **A preconditioner for the Laplacian of a graph A should be the Laplacian of a simpler graph B, derived in a principled fashion from A**.

There is a fairly well known analogy between graph Laplacians and resistive networks [15]. If G is seen as an electrical network with the resistance between nodes i and j being $1/w_{i,j}$, then in the equation $Av = i$, if v is the vector of voltages at the node, i is the vector of currents. Also, the quadratic form $v^T Av = \sum_{i,j} w_{i,j} \left(v_i - v_j\right)^2$ expresses the **power dissipation** on G, given the node voltages v. In view of this, the construction of a good preconditioner B amounts to the construction of a simpler resistive network (for example by deleting some resistances) with an energy profile close to that of A.

The **support** of *A* by *B*, defined as $\sigma(A/B) = max_v \frac{v^T Av}{v^T Bv}$ is the number of copies of B that are needed to support the power dissipation in A, for all settings of voltages. The principal reason behind the introduction of the notion of support, is to express its local nature, captured by the Splitting Lemma.

Lemma 4.1 (Splitting Lemma) *If* $A = \sum_{i=1}^{m} A_i$ *and* $B = \sum_{i=1}^{m} B_i$ where A_i, B_i are Laplacians, then $\sigma(A, B) \leq max_i \sigma(A_i, B_i)$

The Splitting Lemma allows us to bound the support of A by B, by splitting the power dissipation in A into small local pieces, and "supporting" them by also local pieces in B.

For example, in his work Vaidya proposed to take B as the maximal weight spanning tree of A. Then, it is easy to show that $\sigma(B, A) \leq 1$, intuitively because more resistances always dissipate more power. In order to bound $\sigma(A, B)$, the basic idea to let the A_i be edges on A (the ones not existing in B), and let B_i be the unique path in the tree that connects the two end-points of A_i . Then one can bound separately each $\sigma(A_i, B_i)$. In fact, it can be shown that any edge in A that doesn't exist in *B*, can be supported only by the path B_i

As an example, consider the example in Figure 4.1 of the two (dashed) edges $A1$, $A2$ and their two paths in the spanning tree (solid) that share one edge "e".

In this example, the **dilation** of the mapping is equal to 3, i.e. the length of the longest of two paths. Also, as "e" is uses two times, we say that the **congestion** of the mapping is equal to 2. A core Lemma in Support Theory [16] [13] is that the support can be upper bounded by the product **congestion** $*$ **dilation**.

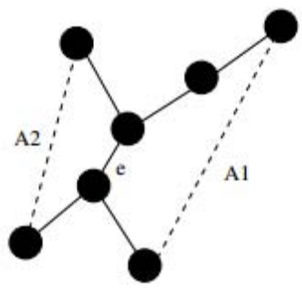

Figure 4.1: A graph and its spanning tree - obtained by deleting the dashed edges

4.1.2 Steiner Preconditioners

Steiner preconditioners, introduced in [17] and extended in [18] introduce external nodes into preconditioners. The proposed preconditioner is based on a partitioning of the n vertices in V into m vertex disjoint clusters V_i For each V_i, the preconditioner contains a star graph $S_i$with leaves corresponding to the vertices in V_i rooted at a vertex r_i. The roots r_i are connected and form the **quotient** graph Q. This general setting is illustrated in Figure 4.2.

Let D' be the total degree of the leaves in the Steiner preconditioner S. Let the **restriction** R be an n × m matrix, where $R(i,j)$ = 1 if vertex i is in cluster j and 0 otherwise. Then, the Laplacian of *S* has $n + m$ vertices, and the algebraic form

$$S = \begin{pmatrix} D' & -D'R \\ -R^T D' & Q + R^T D' R \end{pmatrix}. \tag{4.1}$$

A troublesome feature of the Steiner preconditioner S is the extra number of dimensions/vertices. Gremban and Miller [17] proposed that every time a system of the form $Bz = y$ is solved in an usual preconditioned method, the system

$$S\begin{pmatrix} z \\ z' \end{pmatrix} = \begin{pmatrix} y \\ 0 \end{pmatrix}$$

should be solved instead, for a set of **don't care** variables z'. They also showed that the operation is equivalent to preconditioning with the dense matrix

$$B = D' - V\left(Q + D_Q\right)^{-1} V^T \tag{4.2}$$

where $V = D'R$ and $D_Q = R^T D' R$. The matrix *B* is called the Schur complement of S with respect to the elimination of the roots r_i. It is a well-known fact that B is also a Laplacian.

The analysis of the support $\sigma(A/S)$, is identical to that for the case of subgraph preconditioners. For example, going back to Figure 4.2 the edge $(v1, v4)$ can only be supported by the path $(v1, r1, v4)$, and the edge $(v4, v7)$ only by the path $(v4, r1, r2, v7)$. Similarly we can see the mappings from edges in A to paths in S for every edge in A. In the example, the **dilation** of the mapping is 3, and it can be seen that to minimize the **congestion** on every edge of S (i.e. make it equal to 1), we need to take $D' = D$, where D are the total degrees of the nodes in A, and $w(r1, r2) = w(v3, v5) + w(v4, v7)$. More generally, for two roots r_i,r_j we should have:

$$w(r_i, r_j) = \sum_{i' \in V_i, j' \in V_j} w_{i,j}.$$

Under this construction, the algebraic form of the quotient Q can be seen to be $Q = R^T AR$.

In [18] it was shown that the support $\sigma(S/A)$ reduces to bounding the support $\sigma(Si, A[V_i])$, for all i, where $A[V_i]$ denotes the graph induced in A by the vertices V_i. The key behind bounding $\sigma(Si, A[Vi])$ is called **conductance**. Let us give the definition of conductance.

Definition 4.1 The conductance $\phi(A)$ of a graph $A = (V, E, w)$ is defined as

$$\Phi(A) = min_{S \subseteq V} \frac{w(S, V - S)}{\min(w(S), w(V - S))}$$

where $w(S, V - S)$ denotes the total weight connecting the sets S and $V - S$, and where $w(S)$ denotes the total weight incident to the vertices in S.

The main result of [18] is captured by the following Theorem.

Theorem 4.1 The support σ(S/A) is bounded by a constant c independent from n, if and only if for all i the conductance of the graph $A^0[V_i]$ induced by the nodes in V_i augmented by the edges leaving V_i is bounded by a constant c'.

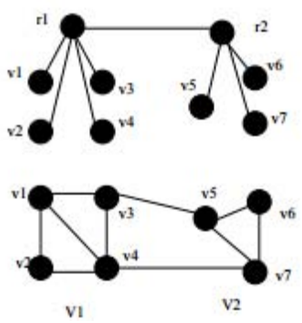

Figure 4.2: A graph and its Steiner preconditioner.

4.1.3 Predicting the performance of solvers

Theorem 4.1 doesn't give a way to pick clusters, but it does provide a way to $avoid$ bad clustering. In recent work [19], Grady proposed a multigrid method where the construction of the "coarse" grid follows exactly the construction of the **quotient** graph in the previous section. Specifically, Grady's algorithm proposes a clustering such that every cluster contains exactly one pre-specified 'coarse' nodes. It then defines the restriction matrix *R* and he lets the coarse grid be $Q = R^T AR$, identically to the construction of the previous Section. The algorithm is iterated to construct a hierarchy of grids. The question then is whether the proposed clustering provides the guarantees that by Theorem 4.1 are necessary for the construction of a good Steiner preconditioner. The following figure, is the Figure 2 of [19], with a choice of weights that force the depicted clustering.

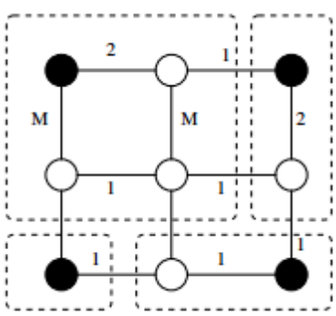

Figure 4.3: A bad clustering.

Every cluster in Figure 4.3 contains exactly one black/coarse node. The problem with the clustering is that the top left cluster, has a very low conductance when $M \gg 1$. In general, in order to satisfy the requirement of the previous Theorem, there are cases where the clustering has to contain clusters with no coarse nodes in them. As we will discuss in later the behavior of the multigrid algorithm proposed in [19] is closely related to the quality of the Steiner preconditioner induced by the clustering. This implies that the multigrid of [19] can suffer bad convergence.

The canonical clustering in Grady's algorithm is very suitable for GPU implementations, when other solvers may be less suitable. This gives to it an advantage on this type of hardware. Even in the presence of a number of relatively bad clusters, it can be faster relative to a solver that uses better clusters. However the advantage is lost when the computed clusters cross a negative threshold in quality, a threshold that depends on several hardware-dependent factors. The value of Support Theory is evident in this case. Grady's algorithm can be instrumented with a very fast routine that measures the quality of the formed clusters and predicts its performance, and reverts to another solver when needed. One can also imagine hybrid clustering algorithms where the majority clusters are formed using the algorithm [19] and the "sensitive" parts of the system are treated separately.

4.2 The Combinatorial Multigrid Solver

The present chapter describes the Combinatorial Multigrid Solver (CMG).At the beginning, we give a short review of multigrid solvers and then we describe the basic components of CMG.

4.2.1 Related work on SDD solvers

Multigrid was originally conceived as a method to solve linear systems that are generated by the discretization of the Laplace (Poisson) equation over relatively nice domains [20]. The underlying geometry of the domain leads to a hierarchy of grids $A = A_0, \ldots, A_d$ that look similar at different levels of detail; the picture that the word multigrid often invokes to mind is that of a tower of 2D grids, with sizes $2^{d-i}x2^{d-i}$ for $i = 0, \ldots, d$. Its provably asymptotically optimal behavior for certain classes of problems soon lead to an effort, known as Algebraic Multigrid (AMG), to generalize its principles to arbitrary matrices. In contrast to classical Geometric Multigrid (GMG) where the hierarchy of grids is generated by the discretization process, AMG constructs the hierarchy of "coarse" grids/matrices based only on the algebraic information contained in the matrix. Various flavors of AMG, based on different heuristic coarsening strategies, have been proposed in the literature. AMG has been proven successful in solving more problems than GMG, though some times at the expense of robustness, a by-product of the limited theoretical understanding.

A solver with provable properties for arbitrary SDD matrices, perhaps the "holy grail" of the multigrid community, was discovered only recently. The path to it was Support Theory [13], a set of mathematical tools developed for the study of combinatorial subgraph preconditioners, originally introduced by Vaidya [14] [21] .It has been at the heart of the seminal work of Spielman and Teng [22] who proved that SDD systems can be solved in nearly-linear time. Koutis and Miller [23] proved that SDD matrices with planar connection topologies (e.g. 4-connectivity in the image plane) can be solved asymptotically optimally, in $O(n)$ time for n-dimensional matrices. The complexity of the Spielman and Teng solver was recently significantly improved by Koutis, Miller and Peng [24] [25], who described an $O(m \log n)$ algorithm for the solution of general SDD systems with m non-zero entries.

It is fair to say that these theoretically described solvers are still impractical due to the large hidden constants, and the complicated nature of the underlying algorithms. Combinatorial Multigrid (CMG) [9] is a variant of multigrid that reconciles theory with practice. Similarly to AMG, CMG builds a hierarchy of matrices/graphs. The essential difference from AMG is that

the hierarchy is constructed by viewing the matrix as a graph, and using the discrete geometry of the graph, for example notions like graph separators and expansion. It is, in a way, a hybrid of GMG and AMG, or a discrete-geometric MG. The re-introduction of geometry into the problem allows us to prove sufficient and necessary conditions for the construction of a good hierarchy and claim strong convergence guarantees for symmetric diagonally dominant (SDD) matrices based on recent progress in Steiner preconditioning [17] [26] [18].

4.2.2 SDD linear systems as graphs

In this subsection we discuss how SDD linear systems can be viewed entirely as graphs. Combinatorial preconditioning advocates a principled approach to the solution of linear systems. The core of CMG and all other solvers designed in the context of combinatorial preconditioning is in fact a solver for a special class of matrices, graph Laplacians. The **Laplacian** A of a graph $G = (V, E, w)$ with positive weights, is defined by:

$$A_{i,j} = A_{j,i} = -w_{i,j} \; and \; A_{i,i} = -\sum_{i \neq j} A_{i,j}$$

More general systems are solved via light-weight transformations to Laplacians. Consider for example the case where the matrix A has a number of positive off-diagonal entries, and the property $A_{i,i} = \sum_{i \neq j} |A_{i,j}|$. Positive off-diagonal entries have been a source of confusion for AMG solvers, and various heuristics have been proposed. Instead, CMG uses a reduction known as double-cover [17] . Let $A = Ap + An + D$, where D is the diagonal of A and Ap is the matrix consisting only of the positive off-diagonal entries of A. It is easy to verify that

$$Ax = b \Leftrightarrow \begin{pmatrix} D + A_n & -A_p \\ -A_p & D + A_n \end{pmatrix} \begin{pmatrix} x \\ -x \end{pmatrix} = \begin{pmatrix} b \\ -b \end{pmatrix}$$

In this way, the original system is reduced to a Laplacian system, while at most doubling the size. In practice it is possible to exploit the obvious symmetries of the new system, to solve it with an even smaller space and time overhead.

Matrices of the form $A + D_e$, where A is a Laplacian and D_e is a positive diagonal matrix have also been addressed in various ways by different AMG implementations. In CMG, we again reduce the system to a Laplacian. If d_e is the vector of the diagonal elements of D, we have

$$Ax = b \Leftrightarrow \begin{pmatrix} A + D_e & 0 & -d_e \\ 0 & A + D_e & -d_e \\ -d_e^T & -d_e^T & \sum_i d_e(i) \end{pmatrix} \begin{pmatrix} x \\ -x \\ 0 \end{pmatrix} = \begin{pmatrix} b \\ -b \\ 0 \end{pmatrix}$$

Again it's possible to implement the reduction in a way that exploits the symmetry of the new system, and with a small space and time overhead work only implicitly with the new system.

A symmetric matrix A is called diagonally dominant (SDD), if $A_{i,i} \geq \sum_{i \neq j} |A_{i,j}|$. The two reductions above can reduce any SDD linear system to a Laplacian system. Symmetric positive definite matrices (SPD) with non-positive off-diagonals are known as *M*-matrices. It is well known that if A is an *M*-matrix, there is a positive diagonal matrix D such that $A = DLD$ where L is a Laplacian. Assuming D is known, an *M*-system can also be reduced to a Laplacian system via a simple change of variables. In many application D is given, or it can be recovered with some additional work [27].

There is a one-to-one correspondence between Laplacians and graphs, so we will be often using the terms interchangeably.

4.2.3 A graph decomposition algorithm

The crucial step for the construction of a good Steiner preconditioner is the computation of a group decomposition that satisfies, as best as possible, the requirements of Theorem 4.1. Before the presentation of the **Decompose-Graph** algorithm, that extends the ideas of [18], we need to introduce a couple of definitions. Let $volG(v)$ denote the total weight incident to node v in graph G. The $weighted\ degree$ of a vertex v is defined as the ratio

$$wd(v) = \frac{vol(v)}{max_{u \in N(v)} w(u,v)}$$

The average weighted degree of the graph is defines as

$$awd(G) = \left(\frac{1}{n}\right) \sum_{v \in N} wd(v)\ .$$

Algorithm 7 Algorithm Decompose Graph.

1: Algorithm **Decompose Graph**
2:
3: **Input**: Graph $A = (V, E, w)$
4: **Output**: Disjoint Clusters V_i with $V = \bigcup_i V_i$
5: Let $W \subseteq V$ be the set of nodes satisfying $wd(v) > \kappa \cdot awd(A)$, for some constant $\kappa > 4$.
6: Form a forest graph F, by keeping the heaviest incident edge of v for each vertex $v \in V$ in A.
7: For every vertex $w \in W$ such that $vol_T(w) < vol_G(w)/awd(A)$ remove from F the edge contributed by w in Step 2.
8: Decompose each tree T in F into vertex-disjoint trees of constant conductance.

Figure 4.4: Decompose Graph Algorithm

It is not very difficult to prove that the algorithm **Decompose-Graph** presented in Figure 4.4 produces a partitioning where the conductance of each cluster depends only on $awd(A)$ and the constant κ. In fairly general sparse topologies that allow high degree nodes, $awd(A)$ is constant and the number of clusters m returned by the algorithm is such that $n/m > 2$ (and in practice larger than 3 or 4).

4.2.4 The Multigrid algorithm

In this subsection we outline the intuition behind Steiner preconditioners and multigrid. Details and proofs can be found in [26]. Algebraically, any of the classic preconditioned iterative methods, such as the Jacobi and Gauss-Seidel iteration, is nothing but a matrix S, which gets applied implicitly to the current error vector e, to produce a new error vector $e' = Se$. For example, in the Jacobi iteration we have $S = (I - D^{-1}A)$. This has the effect that it reduces effectively only part of the error in a given iterate, namely the components that lie in the low eigenspaces of S (usually referred to as high frequencies of A). The main idea behind a two-level multigrid is that the current smooth residual error $r = b - Ax$, can be used to calculate a correction $R^T Q^{-1} Rr$, where Q is a smaller graph and R is an m×n restriction operator. The correction is then added to the iterate x. The hope here is that for smooth residuals, the low-rank matrix $R^T Q^{-1} Rr$ is a good approximation of A^{-1}. Algebraically, this correction is the application of the operator $T = (I - R^T Q^{-1} RA)$ to the error vector e. The choice of Q is most often not independent from that of R, as the ***Galerkin condition*** is employed:

$$Q = RAR^T$$

The Galerkin condition ensures that T is a projection operator with respect to the A-inner product. Two level convergence proofs are then based on bounds on the angle between the subspace $Null(P)$ and the high frequency subspace of S.

At a high level, the key idea behind CMG is that the provably small condition number $\kappa(A, B)$ where B is given in expression 4.2, is equal to the condition number $\kappa(\hat{A}, \hat{B})$ where $\hat{A} = D^{-1/2}AD^{-1/2}$ and $\hat{B} = D^{-1/2}BD^{-1/2}$. This in turn implies a bound on the angle between the low frequency of $\hat{A}$ and the high frequency of $\hat{B}$ [18]. The latter subspace is Null$(R^T D^{1/2})$. This fact suggests to choose $R^T D^{1/2}$ as the projection operator while performing relaxation with $(I - \hat{A})$ on the system $\hat{A}y = D^{-1/2}b$, with $y = D^{1/2}x$. Combining everything, we get the following two-level algorithm in Figure 4.5.

Two-level Combinatorial Multigrid

Input: Laplacian $A = (V, E, w)$, vector b, approximate solution x, $n \times m$ restriction matrix R
Output: Updated solution x for $Ax = b$

1. $D := diag(A)$; $\hat{A} := D^{-1/2}AD^{-1/2}$;
2. $z := (I - \hat{A})D^{1/2}x + D^{-1/2}b$;
3. $r := D^{-1/2}b - \hat{A}z$; $w := RD^{1/2}r$;
4. $Q := RAR^T$; Solve $Qy = w$;
5. $z := z + D^{1/2}R^T y$
6. $x := D^{-1/2}((I - \hat{A})z + D^{-1/2}b)$

Figure 4.5: Two-level Combinatorial Multigrid

The two-level algorithm can naturally be extended into a full multigrid algorithm, by recursively calling the algorithm when the solution to the system with Q is requested. This produces a hierarchy of graphs $A = A_0, \ldots, A_d$. The full multigrid algorithm we use, after simplifications in the algebra of the two-level scheme is as follows in Figure 4.6.

function $x := CMG(A_i, b_i)$
1. $D := diag(A)$
2. $x := D^{-1}b$
3. $r_i := b_i - A_i(D^{-1}b)$
4. $b_{i+1} := Rr_i$
5. $z := CMG(A_{i+1}, b_{i+1})$
6. **for** $i = 1$ **to** $t_i - 1$
7. $\quad r_{i+1} := b_{i+1} - A_{i+1}z$
8. $\quad z := z + CMG(A_{i+1}, r_{i+1})$
9. **endfor**
10. $x := x + R^T z$
11. $x := r_i - D^{-1}(A_i x - b)$

Figure 4.6: Full Combinatorial Multigrid

If $nnz(A)$ denotes the number of non-zero entries in matrix A, we pick

$$t_i = \max\{\left\lceil \frac{nnz(A_i)}{nnz(A_{i+1})} - 1 \right\rceil, 1\}$$

This choice for the number of recursive calls, combined with the fast geometric decrease of the matrix sizes, targets a geometric decrease in the total work per level, while optimizing the condition number.

As we can see at the above figure, the operation of sparse matrix-vector multiplication (SpMV) occurs in steps 3, 7 and 11 of the CMG algorithm. Those multiplications consist the worst bottleneck in CMG solver, so our implementation focuses on solving those bottlenecks accelerating the time required for those SpMV operations. The full Combinatorial Multigrid algorithm is called from PCG method every time we have to solve $Mz^{i-1} = r^{i-1}$ in preconditioner-solve step. Details on how we tried to improve the performance of the SpMV operations are given in Chapter 6.

Chapter 5

GPU Architecture and the CUDA Programming Model

5.1 Introduction

General-Purpose Graphics Processing Unit (GPGPU) Computing only became practical and popular after ca. 2001, with the advent of both programmable shaders and floating point support on graphics processors. GPGPU computing is the use of a GPU together with a CPU to accelerate scientific, analytics, engineering, consumer, and enterprise applications.

GPU-accelerated computing offers unprecedented application performance by offloading compute-intensive portions of the application to the GPU, while the remainder of the code still runs on the CPU as illustrated by Figure 5.1. From a user's perspective, applications simply run significantly faster.

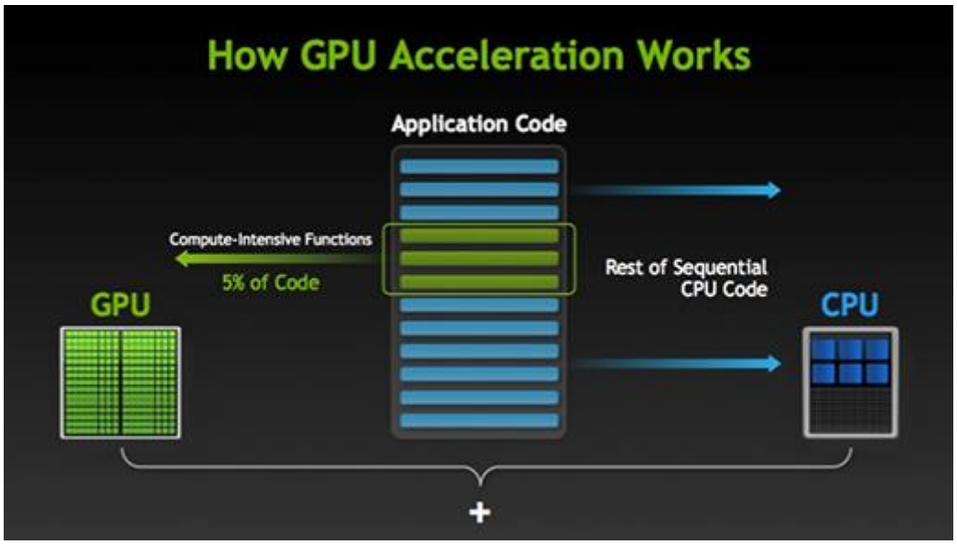

Figure 5.1: How GPU Acceleration Works

A simple way to understand the difference between a CPU and GPU is to compare how they process tasks. A CPU consists of a few cores optimized for sequential serial processing while a GPU has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously as shown in Figure 5.2.

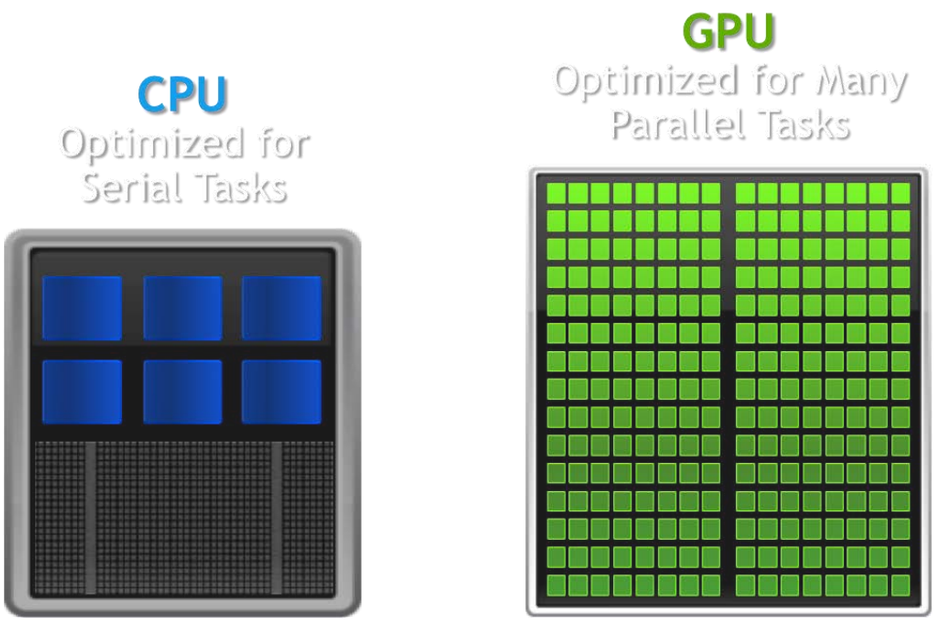

Figure 5.2: CPU vs GPU Architecture

5.2 Hardware Implementation

The NVIDIA GPU architecture is built around a scalable array of multithreaded Streaming Multiprocessors *(*SMs*)*. When a program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity. The threads of a thread block execute concurrently on one multiprocessor, and multiple thread blocks can execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors.

A multiprocessor is designed to execute hundreds of threads concurrently. To manage such a large amount of threads, it employs a unique architecture called SIMT (Single-Instruction, Multiple-Thread). The instructions are pipelined to leverage instruction-level parallelism within a single thread, as well as thread-level parallelism extensively through simultaneous hardware multithreading as detailed in Hardware Multithreading. Unlike CPU cores they are issued in order however and there is no branch prediction and no speculative execution.

5.2.1 SIMT Architecture

The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps. Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently. The term warp originates from weaving, the first parallel thread technology. A half-warp is either the first or second half of a warp. A quarter-warp is either the first, second, third, or fourth quarter of a warp.

When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps and each warp gets scheduled by a warp scheduler for execution. The way a block is partitioned into warps is always the same; each warp contains threads of consecutive, increasing thread IDs with the first warp containing thread 0. Thread hierarchy, which describes how thread IDs relate to thread indices in the block, is described in a later section.

A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths.

The SIMT architecture is akin to SIMD (Single Instruction, Multiple Data) vector organizations in that a single instruction controls multiple processing elements. A key difference is that SIMD vector organizations expose the SIMD width to the software, whereas SIMT instructions specify the execution and branching behavior of a single thread. In contrast with SIMD vector machines, SIMT enables programmers to write thread-level parallel code for independent, scalar threads, as well as data-parallel code for coordinated threads. For the purposes of correctness, the programmer can essentially ignore the SIMT behavior; however, substantial performance improvements can be realized by taking care that the code seldom requires threads in a warp to diverge. In practice, this is analogous to the role of cache lines in traditional code: Cache line size can be safely ignored when designing for correctness but must be considered in the code structure when designing for peak performance. Vector architectures, on the other hand, require the software to coalesce loads into vectors and manage divergence manually.

5.3 Device Memory Model

Threads may access data from multiple memory spaces during their execution as illustrated by Figure 5.3. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory.

There are also two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces. The global, constant, and texture memory spaces are optimized for different memory usages. Those memory spaces are persistent across kernel launches by the same application. Texture memory also offers different addressing modes, as well as data filtering, for some specific data formats.

An instruction that accesses addressable memory (i.e., global, local, shared, constant, or texture memory) might need to be re-issued multiple times depending on the distribution of the memory addresses across the threads within the warp. How the distribution affects the instruction throughput this way is specific to each type of memory and described in the following sections. For example, for global memory, as a general rule, the more scattered the addresses are, the more reduced the throughput is.

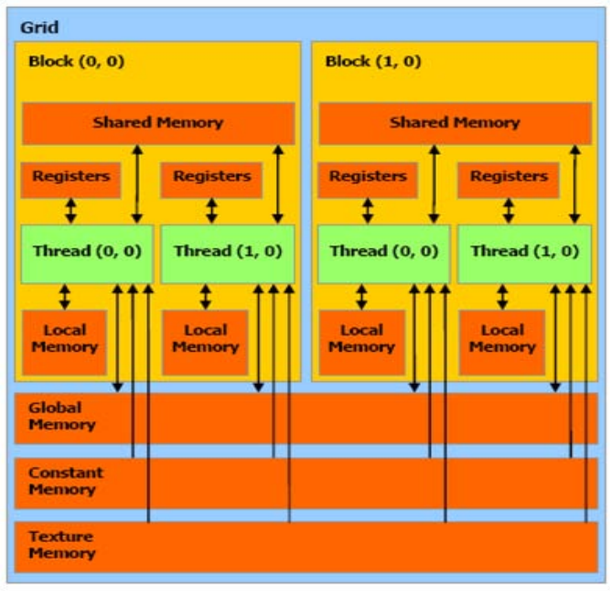

Figure 5.3: Memory Hierarchy

5.3.1 Global Memory

Global memory resides in device memory and device memory is accessed via 32-, 64-, or 128-byte memory transactions. These memory transactions must be naturally aligned: Only the 32-, 64-, or 128-byte segments of device memory that are aligned to their size (i.e., whose first address is a multiple of their size) can be read or written by memory transactions.

When a warp executes an instruction that accesses global memory, it coalesces the memory accesses of the threads within the warp into one or more of these memory transactions depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads. In general, the more transactions are necessary, the more unused words are transferred in addition to the words accessed by the threads, reducing the instruction throughput accordingly. For example, if a 32-byte memory transaction is generated for each thread's 4-byte access, throughput is divided by 8.

How many transactions are necessary and how much throughput is ultimately affected varies with the compute capability of the device. For devices of compute capability 1.1, the requirements on the distribution of the addresses across the threads to get any coalescing at all are very strict. For devices of compute capability 2.x, like the Tesla C2075 we use, and higher, the memory transactions are cached, so data locality is exploited to reduce impact on throughput.

To maximize global memory throughput, it is therefore important to maximize coalescing by:

- Following the most optimal access patterns based on the Compute Capability of the device being used
- Using data types that meet the size and alignment requirement
- Padding data in some cases, for example, when accessing a two-dimensional array

5.3.2 Local Memory

Local memory accesses only occur for some automatic variables. Automatic variables that the compiler is likely to place in local memory are:

- Arrays for which it cannot determine that they are indexed with constant quantities
- Large structures or arrays that would consume too much register space
- Any variable if the kernel uses more registers than available (this is also known as register spilling)

The local memory space resides in device memory, so local memory accesses have same high latency and low bandwidth as global memory accesses. Local memory is however organized such that consecutive 32-bit words are accessed by consecutive thread IDs. Accesses are therefore fully coalesced as long as all threads in a warp access the same relative address (e.g., same index in an array variable, same member in a structure variable).

5.3.3 Shared Memory

Because it is on-chip, shared memory has much higher bandwidth and much lower latency than local or global memory.

To achieve high bandwidth, shared memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously. Any memory read or write request made of n addresses that fall in n distinct memory banks can therefore be serviced simultaneously, yielding an overall bandwidth that is n times as high as the bandwidth of a single module.

However, if two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access has to be serialized. The hardware splits a memory request with bank conflicts into as many separate conflict-free requests as necessary, decreasing throughput by a factor equal to the number of separate memory requests. If the number of separate memory requests is n, the initial memory request is said to cause n-way bank conflicts.

5.3.4 Constant Memory

The constant memory space resides in device memory and is cached in the constant cache. A constant memory fetch costs one memory read from the device memory only on a cache miss, otherwise it costs one read from the constant cache. The memory bandwidth is best utilized when all instructions that are executed in parallel access the same address of the constant memory.

5.3.5 Texture Memory

The texture memory space reside in device memory and is cached in texture cache, so a texture fetch costs one memory read from device memory only on a cache miss, otherwise it just costs one read from texture cache. The texture cache is optimized for 2D spatial locality, so threads of the same warp that read texture addresses that are close together in 2D will achieve best performance. Also, it is designed for streaming fetches with a constant latency; a cache hit reduces DRAM bandwidth demand but not fetch latency.

Reading device memory through texture fetching present some benefits that can make it an advantageous alternative to reading device memory from global or constant memory:

- If the memory reads do not follow the access patterns that global or constant memory reads must follow to get good performance, higher bandwidth can be achieved providing that there is locality in the texture fetches
- Addressing calculations are performed outside the kernel by dedicated units
- Packed data may be broadcast to separate variables in a single operation
- 8-bit and 16-bit integer input data may be optionally converted to 32 bit floating-point values in the range [0.0, 1.0] or [-1.0, 1.0]

5.4 The CUDA Programming Model

This chapter introduces the main concepts behind the CUDA programming model by outlining how they are exposed in C.

CUDA stands for Compute Unified Device Architecture. It is a parallel programming paradigm released in 2007 by NVIDIA. It is used to develop software for graphics processors and is used to develop a variety of general purpose applications for GPUs that are highly parallel in nature and run on hundreds of GPU's processor cores.

CUDA C extends C by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions. A kernel is defined using the global declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new <<< … >>> execution configuration syntax (see C Language Extensions). Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through the built-in threadIdx variable.

For convenience, threadIdx is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one dimensional, two-dimensional, or three-dimensional thread block. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume. The index of a thread and its thread ID relate to each other in a straightforward way: For a one-dimensional block, they are the same; for a two-dimensional block of size (D_x, D_y), the thread ID of a thread of index (x, y) is $(x + y * D_x)$; for a three-dimensional block of size (D_x, D_y, D_z), the thread ID of a thread of index (x, y, z) is $(x + y * D_x + z * D_x * D_y)$.

There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. On current GPUs, a thread block may contain up to 1024 threads. However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks. Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks as illustrated by Figure 5.4 given in next page.

The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system, which it can greatly exceed. The number of threads per block and the number of blocks per grid specified in the <<< ... >>> syntax can be of type int or dim3. Two-dimensional blocks or grids can be specified as in the example above.

Each block within the grid can be identified by a one-dimensional, two-dimensional, or three-dimensional index accessible within the kernel through the built-in blockIdx variable. The dimension of the thread block is accessible within the kernel through the built-in blockDim variable.

A thread block size of 16x16 (256 threads), although arbitrary in this case, is a common choice. The grid is created with enough blocks to have one thread per matrix element as before. For simplicity, this example assumes that the number of threads per grid in each dimension is evenly divisible by the number of threads per block in that dimension, although that need not be the case.

Thread blocks are required to execute independently: It must be possible to execute them in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores, enabling programmers to write code that scales with the number of cores.

Threads within a block can cooperate by sharing data through some shared memory and by synchronizing their execution to coordinate memory accesses. More precisely, one can specify synchronization points in the kernel by calling the *syncthreads*() intrinsic function; *syncthreads*() acts as a barrier at which all threads in the block must wait before any is allowed to proceed.

For efficient cooperation, the shared memory is expected to be a low-latency memory near each processor core (much like an L1 cache) and *syncthreads*() is expected to be light-weight.

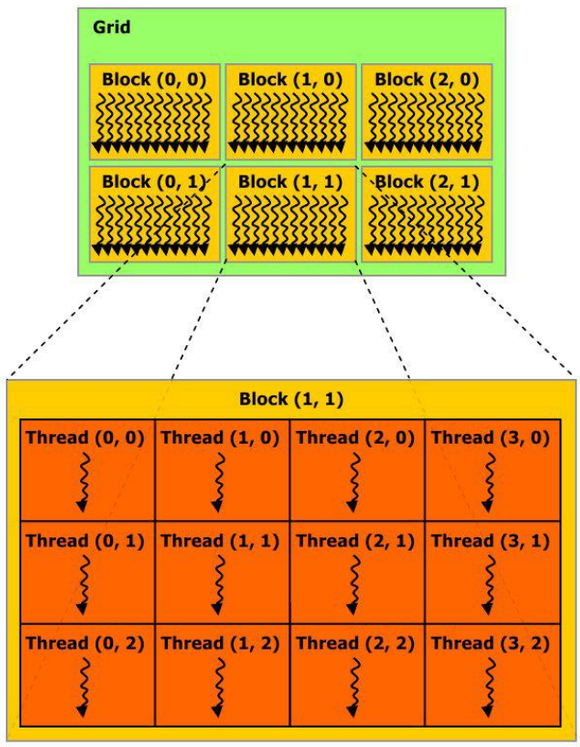

Figure 5.4: 2D Grid of Thread Blocks

5.5 NVIDIA® TESLA™ C2075

Based on the NVIDIA Fermi architecture, the TESLA™ C2075 computing processor has been engineered from the ground up for High Performance Computing, as is capable of reaching 1.03 TFLOPs and 515 GFLOPs peak performance for single and double precision floating-point operations respectively. This was a good reason for us to choose this GPU card for our implementation because our algorithms are based on double floating-point precision arithmetic operations.

5.5.1 Engine Specifications

The TESLA™ C2075 has 14 Multiprocessors with 32 CUDA cores each, which means 448 CUDA cores with 1.15 GHz clock rate per core. It has compute capability 2.0. The warp size is 32 and it can support up to 1536 threads per multiprocessor and 1024 threads per block. The maximum sizes of each dimension of a block and grid are 1024 x 1024 x 64 and 65535 x 65535 x 65535 respectively. Also it can support concurrent copy and kernel execution.

5.5.2 Memory Specifications

The total amount of global memory for this device is 6.144GB and with ECC support enabled the user's available memory is 5.376GB. The total amount of constant memory is 65KB. Each block has available 49KB of shared memory and 32768 registers. The memory clock rate is 1.57GHz. TESLA™ C2075 has available caching. The on-chip memory per multiprocessor is used for both L1 and shared memory, and how much of it is dedicated to L1 versus shared memory is configurable for each kernel call. Additionally, is has a unified L2 cache for all of the processor cores of 786KB. The maximum texture dimension size for 1D is (65536), for 2D is (65536, 65535) and for 3D is (2048, 2048, 2048).

Figure 5.5 shows the architecture of Fermi Streaming Multiprocessor. TESLA™ C2075 is consists of 14 such SMs.

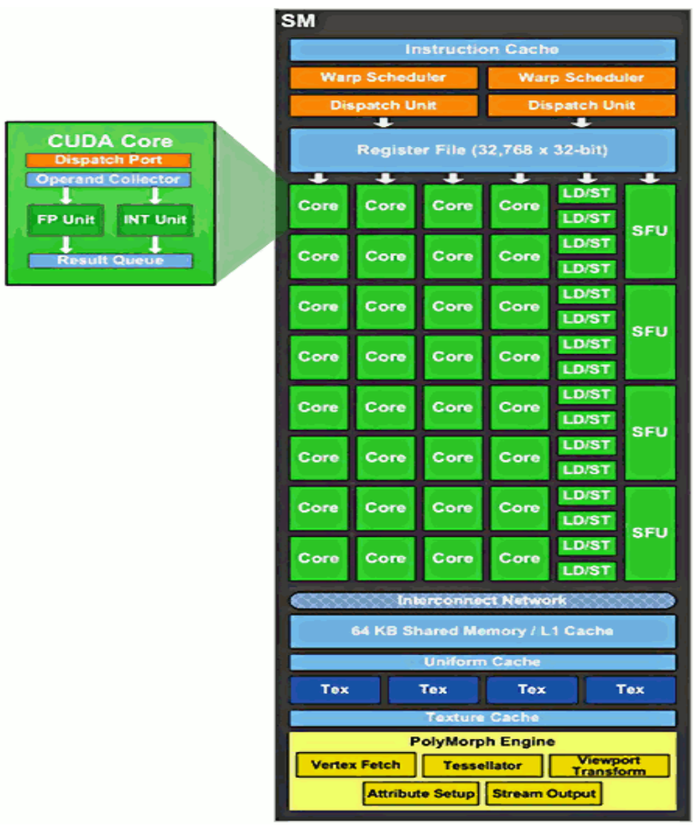

Figure 5.5: Fermi SM

Chapter 6

Improving The Performance of CMG

The CMG solver is an extension of the preconditioned conjugate gradient method (PCG), described in subsection 2.4.2 .Therefore, its core is based on sparse matrix-vector multiplication $Mz^{i-1} = r^{i-1}$ where M is the Steiner preconditioner. The PCG approximates the solution iteratively until the solution is satisfactory, so in the solve phase exist too many matrix-vector multiplications which are the biggest bottleneck of CMG. This bottleneck appeared also at the results of the performance profiling we done using Intel® VTune^{TM} Amplifier. In this section we represent our implementation of a new SpMV method on Nvidia GPUs. Before the description of our implementation, we review some relevant existing SpMV methods.

6.1 Review of existing SPMV methods on GPU

6.1.1 Introduction

Sparse matrix-vector multiplication (SpMV) is of crucial importance in sparse linear algebra as it plays an important role in many numerical and scientific computing applications such as finite difference and finite element based methods. SpMV operation represents the dominant computing cost in those problems and it is very important to improve the efficiency of the SpMV algorithms. As a result, several research efforts have been proposed for parallelizing SpMV on GPU platforms.

There are many sparse matrices formats such as DIA, ELL, CSR, HYB, PKT and COO for both structure and unstructured matrices [28] . The Combinatorial Multigrid Solver is based on Compressed Column Storage (CCS) and as it is designed for symmetric matrices, we focus on the CSR format. It is easy to see that CSR is equal to CCS for symmetric matrices with the difference that we use row major storing-access. The serial CPU kernel for the CSR SpMV is shown in Algorithm 6.1 .

```
double* spmv(int n, double *a, int *ia, int *ja, double *x, double *y){
      unsigned int i, j;

      /* Initialize y vector */
      for (i = 0; i < n; i++) {
        y[i] = 0;
      }

      for (i = 0; i < n; i++) {
        for (j = ia[i]; j < ia[i + 1]; j++)
            y[i] = y[i] + a[j] * x[ja[j]];
      }

      return (y);
}
```

Algorithm 6.1: Serial CPU kernel for the CSR SpMV

6.1.2 Relevant SpMV algorithms on GPU platforms

In this subsection, we describe some SpMV implementations on the GPU. The SpMV computing consists of two phases: the first *product phase*, which performs the element-element production between the matrix and the vector, the second *summation* phase adds the results for each row to get the final result. The existing methods are the mentioned below.

1. The ***row-based B&G method:*** Bell and Garland [28] first proposed a straightforward implementation, in which each row will take care of all the computation (multiplication and summation) by a single thread as shown in Figure 6.1. The algorithm only requires one kernel launch. The performance, however, turns out to be unsatisfying. A careful analysis reveals two major reasons leading to significant inefficiency. First, the memory access cannot be coalesced because one thread needs to load varying number of data words from global memory. Secondly, the load balance is poor since there could be hundreds of nonzero elements in some rows, while most other rows have only a few. Therefore, the GPU run time is dominated by those less sparse rows. The SpMV kernel using one thread per matrix row is given in Figure 6.2.

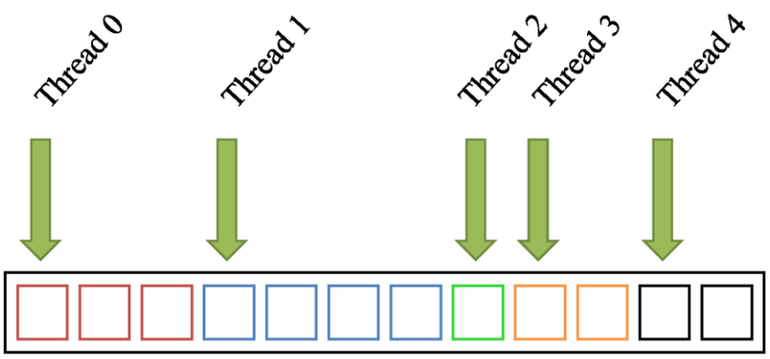

Figure 6.1: Row-Based B&G Method

```
__global__ void
spmv_csr_scalar_kernel(const int num_rows,
                       const int   * ptr,
                       const int   * indices,
                       const float * data,
                       const float * x,
                             float * y)
{
    int row = blockDim.x * blockIdx.x + threadIdx.x;

    if(row < num_rows){
        float dot = 0;

        int row_start = ptr[row];
        int row_end   = ptr[row+1];

        for (int jj = row_start; jj < row_end; jj++)
            dot += data[jj] * x[indices[jj]];

        y[row] += dot;
    }
}
```

Figure 6.2: SpMV kernel for the CSR sparse matrix format using one thread per matrix row

2. The ***warp-based B&G method***: The row-based B&G method was further improved by the warp-based B&G method in [28] in which one warp is assigned to each row of a matrix. After the multiplication phase, the intra-warp thread reduction is performed to compute the per-thread result. The algorithm is illustrated in Figure 6.3. Compared to the row-based B&G method, its memory accesses can be coalesced because 32 continuous threads in the same warp could work together to load the non-zero elements in one row. This approach is extremely efficient for those matrices with long strips of non-zeros and a throughput of over 10 GFLOPS can be achieved. However, it may suffers low performance when the number of nonzeros in each row is smaller than 32, which can be the case for many finite difference and finite element based methods. The SpMV kernel using one 32-thread warp per matrix row is given in Figure 6.4.

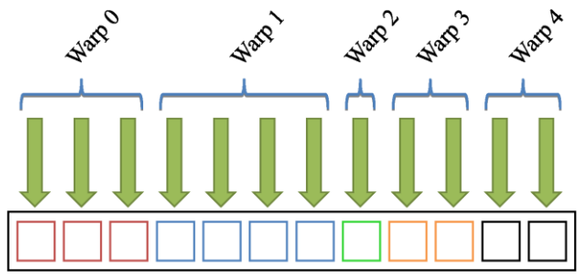

Figure 6.3: Warp-Based P&G Method

```
__global__ void
spmv_csr_vector_kernel(const int num_rows,
                       const int   * ptr,
                       const int   * indices,
                       const float * data,
                       const float * x,
                             float * y)
{
    __shared__ float vals[];

    int thread_id = blockDim.x * blockIdx.x + threadIdx.x;  // global thread index
    int warp_id   = thread_id / 32;                         // global warp index
    int lane      = thread_id & (32 - 1);                   // thread index within the warp

    // one warp per row
    int row  = warp_id;

    if (row < num_rows){
        int row_start = ptr[row];
        int row_end   = ptr[row+1];

        // compute running sum per thread
        vals[threadIdx.x] = 0;
        for(int jj = row_start + lane; jj < row_end; jj += 32)
            vals[threadIdx.x] += data[jj] * x[indices[jj]];

        // parallel reduction in shared memory
        if (lane < 16) vals[threadIdx.x] += vals[threadIdx.x + 16];
        if (lane <  8) vals[threadIdx.x] += vals[threadIdx.x +  8];
        if (lane <  4) vals[threadIdx.x] += vals[threadIdx.x +  4];
        if (lane <  2) vals[threadIdx.x] += vals[threadIdx.x +  2];
        if (lane <  1) vals[threadIdx.x] += vals[threadIdx.x +  1];

        // first thread writes the result
        if (lane == 0)
            y[row] += vals[threadIdx.x];
    }
}
```

Figure 6.4: SpMV kernel for the CSR sparse matrix format using one 32-thread warp per matrix row

3. The ***P&S method:*** Deng *et al* later proposed [29] an improved SpMV method, called P&S method for many electronic design automation (EDA) related problems. They realized that the SpMV problem actually consists of two phases with different available parallelism. In the first phase, every non-zero matrix element must be multiplied by a corresponding vector element. From this point of view, the multiplication operations are fully regular. The second phase, calculates the sum of the products on each row. Here the number of summations per row is determined by the distributions of non-zeros and thus cannot be regular for general cases. The approach will not directly operate on the CSR data structure. Instead, it creates a new vector, called $expanded\ vector$, of the same size of the $data\ vector$, as shown in Figure 6.5. The expanded vector consists of the elements from the multiplication vector $v = [b1, \dots, b3]$ and each element in the expanded vector has the value $v\_expanded[i] = v[col[i]]$ where $col[i]$ stores the column index of the element $data[i]$ .The two vectors ($v\_expanded$ and $data$) will be multiplied in the production phase. After the generation of expand vector, the remaining operations become two vector multiplication and partial summation over rows. The two phases can be organized as two succeeding GPU kernels as shown in Figure 6.6.

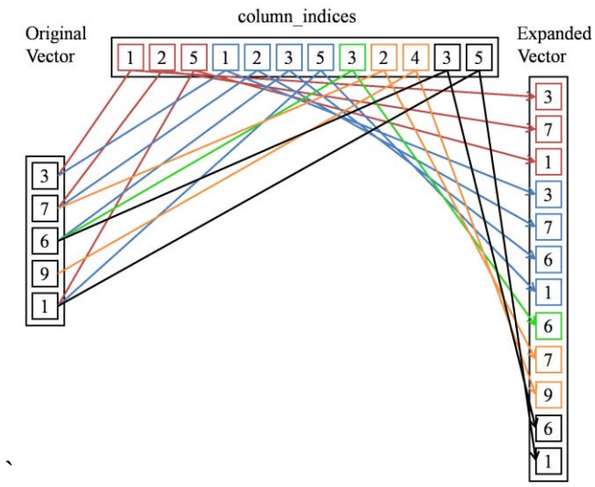

Figure 6.5: Vector expansion concept in P&S Method

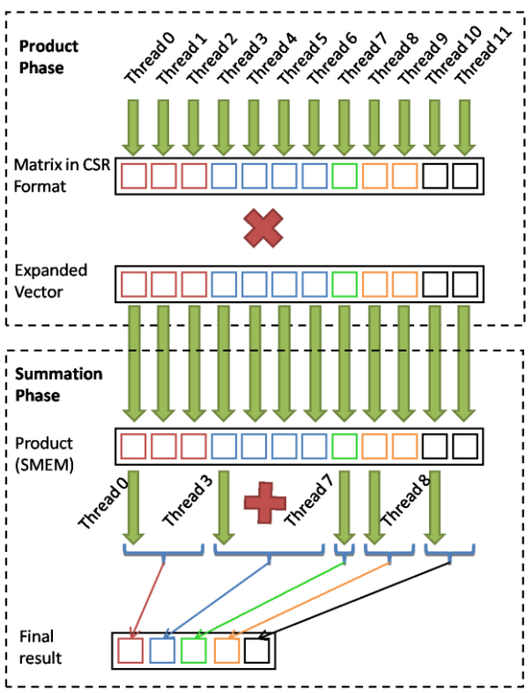

Figure 6.6: P&S Method

6.2 SegSpMV - A new SPMV method

In this section we present our method of choice to accelerate the SpMV operations performed into PCG in the solve phase of CMG. The new algorithm, called segSpMV [1] can overcome the aforementioned problems in the P&S method.

The new algorithm is also based on the expanded vector concept for the multiplication phase. But unlike the P&S method, the new algorithm can mitigate the irregular memory access problems in the summation phase, and thus lead to simpler implementation and better performance. The main idea is to partition the rows into a number of fixed-length regular segments before the operation. The length of the segment typically is selected to be just bigger than the average number of nonzero elements per row in a matrix and they also should be the power of 2 for easy reduction operation. For example, if the average number of nonzero elements is 18, then segment length $2^5 = 32$ is selected. For rows with more nonzeros than the average number, multiple segments will be needed.

After the segment length is determined, the each row is partitioned into a number of regular segments. If a segment does not fully filled by the elements from the given row, 0 is padded to the rest of the empty elements in the segment as shown in Figure 6.7. In this figure, one 0 is padded at the end of seg1 and seg4. This segment-based expansion is performed for both *original data* vector and the *expanded vector*. After this step, the two segment-expanded vectors are sent to GPU global memory for multiplication and addition phases with just one kernel launch as shown in Figure 6.7. Note that it takes O(#nonzeros) to do the zero padding. In the product phase, each thread first will read two elements from the two segment-expended vectors respectively via the coalesced memory access from the GPU global memory. Then each thread multiplies one pair of elements from the two segment-expanded vectors. A crucial fact for the performance of the algorithm is that it saves the product result immediately into the shared memory. In this case, all the partial product results from all threads are stored in shared memory, which is ready for the second phase of addition operation right away.

We note that the new method will never run out of shared memory as the amount of memory needed is 8 times of number of threads in each block as each thread takes care of one double precision element. So given 1K maximum thread allowed in each block in Tesla C2075 GPU, the maximum memory is just 8K, which is far less than the 49K shared memory in each block. This is also the case for other GPUs as well.

As a result, we do not need to write the product results back to global memory and then read them back again, which leads to one more kernel launch saving. In the addition phase, each thread sums the products in one segment and each block is responsible of the same number of segments. The number of non-zero elements in each row may be different, but all segments are with the same length. Therefore, according to this method we achieve better load balance than the P&S and we enjoy the coalesced memory access.

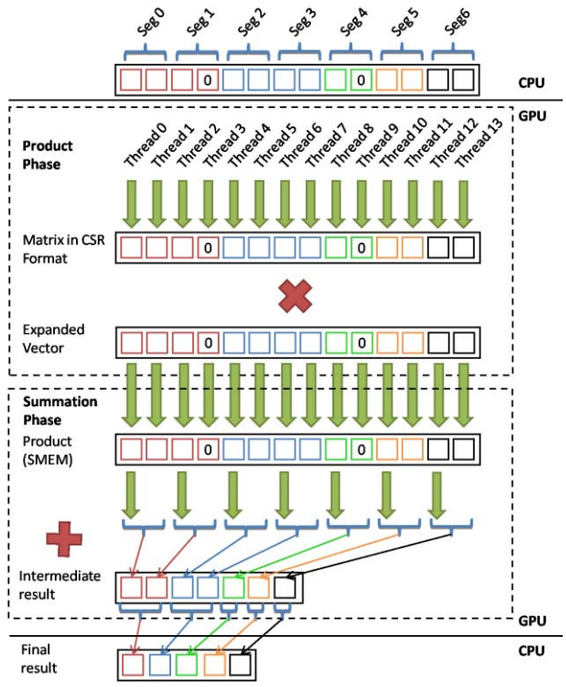

Figure 6.7: The new segSpMV method

In the summation phase, the new algorithm does not need to check the boundaries of each row any more, which causes the irregular memory access, as it can simply add all the results for each regular segment instead. Because the segment size is fixed, the summation can be very easily done by one thread or by multiple threads via reduction. Also the addition operation will take almost same time for all the threads. We add the $\_\_syncthreads()$ to ensure all the partial results from each segment finish first before they are written back into global memory using the coalesced memory access. Finally, *segSpMV* adds up the immediate results of segments corresponding to the same row in the CPU side to get the final results, which can be done very efficiently. The segSpMV kernel that we implemented is given at the next section.

Kai He et al proved that the segSpMV method constantly outperforms all published algorithms and the SpMV method in the recent cuSPARSE library [30] based on a set of public matrix benchmarks.

6.3 Implementation and Optimizations

This section presents and describes our implementation of segSpMV which we developed with the expectation of improving the performance of the CMG solve phase. We start by giving the hardware and software used in our performance study and the IBM benchmarks used in our experiments. We will then represent our implementation, the optimizations we done and our experimental results.

6.3.1 System Specifications and IBM Power Grid Benchmarks

Hardware and software specifications of our system are listed in Table 6.1. The host code is been compiled with Intel ICC compiler for better performance. For the compilation of our implementation which includes CUDA code, we use the NVCC compiler which make use of the GCC compiler to compile the host code together with the device code.

CPU	6 Intel(R) Xeon(R) E5645 @2.40GHz
GPU	Nvidia® Tesla™ C2075
MEMORY	24GB DDR3
OS	Kubuntu 12.10 Quantal(Linux 3.5.0-34)
CUDA	CUDA 5.5
INTEL COMPILER	ICC 13.1.1
GNU COMPILER	GCC 4.7.2
NVIDIA COMPILER	NVCC 5.5

Table 6.1: Test platform specifications

All the power grid benchmarks presented in this section are drawn from real designs, and vary over a reasonable range of size and difficulty. Those netlists are generated in Spice format. For more information for such Benchmarks we refer the reader to [31].Table 6.2 shows the IBM Power Grid Benchmarks we used for the DC Analysis.

Netlist	**#i**	**#n**	**#r**	#s	**#v**	**#l**
ibmpg1	10.774	30.638	30.027	14.208	14.308	2
ibmpg2	37.926	127.238	208.325	1.298	330	5
ibmpg3	201.054	851.584	1.401.572	461	955	5
ibmpg4	276.976	953.583	1.560.645	11.682	962	6

Table 6.2: IBM Power Grid Benchmarks for DC Analysis

- i for current sources
- n for nodes (total number, does not take shorts into account)
- r for resistors (include shorts)
- s for shorts (zero value resistors and voltage sources)
- v for voltage sources (include shorts)
- l for metal layers

For the MNA analysis of IBM netlists we used a software we had already implemented. This software parses the netlist file and creates the corresponding sparse MNA array "A" and right-hand side vector "b", which will be used later for solving the system $Ax = b$. Table 6.3 shows the dimensions and the number of non-zero elements of the MNA arrays corresponding to each IBM netlist.

Netlist	**Dimensions**	**Non-zeros**
ibmpg1	44.943 × 44.943	147.315
ibmpg2	127.565 × 127.565	544.545
ibmpg3	852.536 × 852.536	3.656.107
ibmpg4	954.542 × 954.542	4.058.866

Table 6.3: Matrix size and non-zero elements of MNA arrays

6.3.2 SegSpMV implementation and experimental results

The summation phase of the segSpMV can be implemented with 2 different ways as proposed at [1] . In the first one, the summation is done by one thread per segment while at the second version called segSpMV-r is formed by reduction. The average segment length of the matrices compose the hierarchy of Steiner preconditioners for all IBM benchmarks is very small as shown in Table 6.4.As a result we choose the first approximation. For matrices with large segment size (above 32), segSpMV-r would be faster than segSpMV. Table 6.4 shows the number of levels-matrices including in the Steiner preconditioner hierarchy of graphs for each benchmark, the average non-zeros per row and the average segment length for all that matrices.

Netlist	Hierarchy Levels	Average non-zeros per row	Average Segment Length
ibmpg1	5	4.8	8
ibmpg2	6	5.3	8
ibmpg3	7	5.3	8
ibmpg4	7	5.3	8

Table 6.4: Hierarchy levels, average non-zeros per row and average segment length for the IBM benchmarks

For our segSpMV, we set the number of thread per blocks to be 256, which is the best choice based on our observation. For shared memory configuration, the segSpMV method only requires (thread per block)*8 = 2048 bytes per block, so it is always satisfied for C2075 GPU.

The compilation of the original CPU code was made using the ICC with the optimization mode -O2 which gives as the best performance based on our observation. The compile command we used is:

“ icc -O2 *.c -o cmg -lm ”,

which compiles all the C files together and link them with the math library to produce the executable.

For our implementation we used the NVCC to compile host and device code together. The optimization mode for the host code which gave as the best execution times was -O4. The compile command we used is:

“ nvcc -O4 *.cu *.c -arch=sm_20 -o cmg -lm -lcuda -lcudart ”,

which compiles all the C and CU(CUDA) files together and link them with math and CUDA library and runtime, to produce the executable.

The CMG solver C code [12] was taking advantage of the symmetry occurred in our SDD matrices by storing only the upper triangular part of the matrix. This approach leads to less memory requirements. SpMV multiplications for all the levels of the preconditioner included inside the operation $Mz^{i-1} = r^{i-1}$ of the PCG method, are implemented as shown in Algorithm 6.1 with a small difference. The code for the approach where we are storing only the upper triangular part of the sparse matrix is called sspmv2 and is shown in Algorithm 6.2.

```
double* sspmv2(int n, double *a, int *ia, int *ja, double *x,double *y)
{
    unsigned int i, j, k;
    double sum;

    /* Initialize y vector */
    for (i = 0; i < n; i++) {
        y[i] = 0;
    }

    for (i = 0; i < n; i++) {
        sum = a[ia[i]] * x[i];
        for (j = ia[i] + 1; j < ia[i + 1]; j++) {
            k = ja[j];
            sum += a[j] * x[k];
            y[k] += a[j] * x[i];
        }
        y[i] += sum;
    }

    return (y);

}
```

Algorithm 6.2: Serial CPU SpMV kernel for the CSR sparse matrix format storing the upper triangular part

At this approximation, for each element of the sparse matrix that we stored, we compute the result for both the corresponding row and for the row which corresponds to the symmetric value (which is not stored) to get the correct solution.

However, this method causes problems when we try to implement it on a GPU architecture. The problem that occur is that the time where a thread with row index "i" adds a value to the current value $y[i]$ of the solution vector, at the same time another thread which has row index "z" will may also try to add a value to the current value $y[i]$ for the corresponding symmetric position at the primal matrix. This case appears when the column index of the second thread $k = ja[j]$ is equal to "i" and it is called race condition. This case can cause wrong results and it can be resolved using atomic operations. However, the atomic operations at the GPU and especially those that access the global memory are very expensive.

This fact led us to try storing the whole sparse matrices of each hierarchy at the memory and make the SpMV operation as described in Algorithm 6.1. The experimental results of this approximation is shown in Table 6.5 where we compare the execution times of those two methods. We also show the time spent for the segmentation of the hierarchy sparse matrices and the segmentation-expansion of the vectors.

Netlist	Storing full hierarchy matrices			Storing the upper part of hierarchy matrices		
	SpMV (sec)	PCG (sec)	SEGMENT-EXPAND (sec)	SpMV (sec)	PCG (sec)	SEGMENT-EXPAND (sec)
ibmpg1	2,34	7,70	4,00	2,21	7,00	3,00
ibmpg2	0,40	0,78	0,90	0,30	0,72	0,50
ibmpg3	3,56	7,00	5,50	3,15	6,50	4,43
ibmpg4	2,23	4,40	4,15	1,90	4,00	3,00

Table 6.5: Times when storing full hierarchy matrices vs times when storing the upper part of hierarchy matrices

The above results shows that when we take advantage of symmetry and we save memory resources, we only get a very small execution time speedup for the SpMV operations and consequently for the PCG solve phase of CMG. Also, we observe that the time we spend for the computation of non-zeros per row and segment length of all sparse arrays of each hierarchy, the expansion of all vectors to be multiplied with each matrix and for the segmentation of both hierarchy arrays and all vectors is not much better (Segmentation-Expansion speedup). We can see those small speedups in Table 6.6.

Netlist	SpMV speedup	PCG speedup	Segmentation-Expansion speedup
ibmpg1	1,06x	1,10x	1,38x
ibmpg2	1,13x	1,08x	1,24x
ibmpg3	1,33x	1,09x	1,80x
ibmpg4	1,17x	1,10x	1,33x

Table 6.6: Solve phase time speedups (Storing the upper part of hierarchy matrices over storing full hierarchy matrices)

Based on our experiments we decided that it would be more efficient to implement a GPU kernel that would made the Sparse Matrix-Vector Multiplication based on the full sparse matrices storage in order to achieve better performance from our kernel and exploit the GPU parallelism. Consequently, we will describe that kernel with the optimizations we tried and report our experimental results and the speedups we achieved with our implementation.

The first step in our segSpMV is the computation of segment length of the hierarchy matrices which only requires to find the number of non-zeros per matrix row. Also, we compute the size of zero padding required for each row and we create an array called "$segPtr$", which contains the number of the first segment of each row and it helps us to access all the segments of each row. We can then call our segmentation method to segment each sparse array of the Steiner preconditioner hierarchy.

In the second step we continue by solving the Steiner preconditioned system $Mz^{i-1} = r^{i-1}$ using the full multigrid algorithm described in Figure 4.6. As we mentioned in a previous section we focus on SpMV operations of steps 3, 7 and 11. It is obvious that the vectors are changing continuously, so we have to segment and expand the vector each time before we do the corresponding SpMV operation.

In the next step we have to copy the matrix A_i of current hierarchy level "i" and the corresponding vector to the GPU to have our data ready for multiplication and addition phases. The SpMV kernel works as described in section 6.2. The implementation of segSpMV kernel is given in Algorithm 6.3, where $data\_exp$ and x_exp stand for matrix and vector segmented-expanded data, $intermediate\_result$ stores the sum of all the elements for each segment and seg_length is the segment length.

```
__global__ void segSpmv(double *intermediate_result, double *data_exp,
double *x_exp, int seg_length){

      double sum; //segment's partial sum
      int i;

      int id=blockIdx.x * blockDim.x + threadIdx.x;

      __shared__ double product [256];

      product[threadIdx.x] = data_exp[id] * x_exp[id];

      __syncthreads();  /*ensure that all threads of the same segment
                        has finished before computing the partial
                        sums*/

      if( (id % seg_length) == 0){

            sum = 0.0;

            for(i=threadIdx.x;i<(threadIdx.x + seg_length);i++){
                  sum += product[i];
            }

            intermediate_result[id/seg_length] = sum;
      }

}
```

Algorithm 6.3: segSpmv kernel - The first approach

As shown in the above kernel each thread reads the two corresponding elements from global memory, multiplies this pair of elements and stores the result immediately into the shared memory. After all products of each segment is computed, the first thread of each segment can now proceed with the summation phase. For all elements of the segment, it sums the corresponding products and stores the sum in global memory ($intermediate\_result$).

Finally, we copy intermediate results back to the CPU where we use "$segPtr$" array to find the final result by adding all the intermediate results of the segments belonging to each row.

Table 6.7 shows the time speedups we achieved using our segSpMV kernel implementation without including the time spent transferring data between the CPU and GPU. For all benchmarks we get an average performance acceleration 5.8x and for the biggest netlist we achieved 7.9x speedup.

Netlist	SPMV (sec)	SegSpMV (sec)	Speedup
ibmpg1	2,34	0,60	3,84x
ibmpg2	0,40	0,08	4,65x
ibmpg3	3,56	0,52	6,85x
ibmpg4	2,23	0,28	7,88x

Table 6.7: GPU SegSpMV execution time speedup over CPU SpMV

However, the number of PCG iterations and the number of SpMV operations including in the recursive full multigrid algorithm leads to many memory copies of the sparse matrix A_i of current hierarchy level "i" and the corresponding vector to the GPU. As a result, the time spending for those memory copies has a negative effect to the total PCG time spent for the solution of the linear system. Table 6.8 shows the slowdown of PCG solve method comparatively with the CPU implementation using the serial version of SpMV.

Netlist	PCG with SPMV (sec)	PCG with segSpMV (sec)	Slowdown
ibmpg1	7,60	38,50	5,00x
ibmpg2	0,78	5,10	6,37x
ibmpg3	7,03	44,00	6,30x
ibmpg4	4,40	21,00	4,70x

Table 6.8: PCG slowdown using the SegSpMV

6.3.3 The next approach

It is obvious that if we want to improve the performance of PCG using our segSpMV kernel we have to reduce the transfers between the CPU and GPU.

As mentioned above, the hierarchy of preconditioner matrices, unlike the vectors, does not change over the SpMV operations in PCG iterations. Therefore, we tried to copy the full hierarchy to the GPU before the call of PCG method and do not copy the corresponding level at each SpMV operation. Of course before we copy the hierarchy we have already segmented-expanded each matrix. So, we create a structure which includes all the levels of the preconditioner and inside the kernel we access the level corresponding to the current SpMV multiplication. The kernel of our new approach is given in Algorithm 6.4 .

For this implementation we used two array of pointers, one for all the hierarchy matrices called *hierarchy_data* and one for the sum of each segment for all the hierarchy levels called *intermediate_result*. Also, we have to pass the current hierarchy level as argument to the kernel, so we can access the corresponding hierarchy preconditioner matrix and intermediate result.

```
__global__ void segSpmvNew(double **intermediate_result,
double **hierarchy_data, double *x_exp, int seg_length, int level){

    double sum;   //segment's partial sum
    int i;

    int id=blockIdx.x * blockDim.x + threadIdx.x;

    __shared__ double product[256];

      product[threadIdx.x] = hierarchy_data[level][id] * x_exp[id];


    __syncthreads();

    if( (id % seg_length) == 0){

      sum = 0.0;

      for(i=threadIdx.x;i<(threadIdx.x + seg_length);i++){
            sum += product[i];
      }

      intermediate_result[level][id/seg_length] = sum;
    }

}
```

Algorithm 6.4: segSpmvNew kernel – The second approach

Table 6.9 lists the full copy time speedups we achieved as we expected, using our second implementation. We can see that if we copy the hierarchy of preconditioner matrices once, we get an average speedup of 2x.

Netlist	COPY TIME with segSpMV (sec)	COPY TIME with segSpmvNew (sec)	Speedup
ibmpg1	7,91	3,95	2,00x
ibmpg2	0,98	0,53	1,85x
ibmpg3	7,82	3,66	2,14x
ibmpg4	4,47	2,25	1,99x

Table 6.9: segSpmvNew over segSpMV time speedup

The second implementation also improved the total time spent for the solve phase using the PCG method giving an average speedup of 2.2x. The time speedups of the PCG method when using our new kernel over using the previous segSpMV is shown in Table 6.10.

Netlist	PCG with segSpMV (sec)	PCG with segSpmvNew (sec)	Speedup
ibmpg1	38,50	15,30	2,52x
ibmpg2	5,10	2,37	2,15x
ibmpg3	44,00	18,70	2,35x
ibmpg4	21,00	11,80	1,78x

Table 6.10: PCG speedups when using our new approach of segSpMV

However, we see that the new kernel performs worse than our first segSpMV approach as shown in Table 6.11. This slowdown is caused by the problem of $pointer\ chasing$. Pointer chasing occurs when a thread tries to access an element of the two "$two-dimensional$" arrays ($hierarchy\_data[level][id]$ & $intermediate\_result[level][id/seg\_length]$). This happens because each thread has at first to access the global memory at address $hierarchy\_data[level]$ to find where the current level matrix elements are stored and then go to another location in the global memory to read the corresponding element. In the same way the first thread of each block writes to the address $ntermediate\_result[level][id/seg\_length]$.

Netlist	SegSpMV (sec)	segSpmvNew (sec)	Slowdown
ribmpg1	0,60	1,37	2,28x
ibmpg2	0,08	0,24	3,00x
ibmpg3	0,52	1,82	3,50x
ibmpg4	0,28	1,12	4,00x

Table 6.11: segSpMV slowdown caused by pointer chasing

Chapter 7

Conclusion

In conclusion, given the key role of circuit simulation in the design process, there has been a significant interest in accelerating the heart of simulation which is the solution of a very large system. This thesis reports our efforts to accelerate the performance of a linear system multi-grid-like solver called CMG using modern Nvidia GPUs.

Specifically, we focused on the solution phase of CMG and especially on the acceleration of sparse matrix-vector multiplications (SpMV). We have shown maximum speedups of 8x without counting memory transfers, using our new segmentation-based GPU-accelerated SpMV algorithm. Although, memory transfers do not allow to achieve a better total performance over the serial CPU version of solve phase.

The new SpMV algorithm tries to mitigate the low computing to communication ratio issues in the SpMV operations by regularizing the access patterns during the two operations of the SpMV. It is a very efficient multiplication method and it can give the best performance that can achieved for an SpMV multiplication on GPU. Of course, the proposed segSpMV method can be applied to a myriad other applications based on SpMV multiplications.

7.1 Future Work

Possible extensions in this project may be the following:

- **Implement the segment-expansion operations and the computations of the final SpMV result on the GPU**

  A first improvement of the segSpMV algorithm is simple and can be made by implementing both segment-expansion of matrix and vectors and the computation of final multiplication result inside our kernel, unlike our current implementation in which both are implemented in the CPU side.

- **Use asynchronous memory copy for the preconditioner hierarchy matrices**

  The time margin between the transfer of preconditioner hierarchy and the time we use it inside the kernel gives us the chance to make this copy asynchronous using *cudaMemcpyAsync* CUDA function. This function gives us the opportunity to hide the copy latency by overlapping with serial host code computations.

- **Port the entire PCG method on the GPU**

  This way memory transfers between CPU and GPU will be eliminated and we can possibly achieve a much better acceleration compared with the CPU version.

Bibliography

[1] K. He, S. X.-D. Tan, E. Tlelo-Cuautle, H. Wang and H. Tang, A New Segmentation-Based GPU-Accelerated Sparse Matrix-Vector Multiplication, 2014.

[2] F. N. Najm, Circuit Simulation, Wiley,IEEE, 2010.

[3] NVIDIA CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[4] Carnegie-Mellon University. Computer Science Dept and K.D. Gremban. Combinatorial Preconditioners for Sparse, Symmetric, Diagonally Dominant Linear Systems. Research paper. School of Computer Science, Carnegie Mellon University, 1996.

[5] Y. Saad, Iterative Methods for Sparse Linear Systems. Society for Industrial and Applied Mathematics, 2003.

[6] T. Davis, CSPARSE: a concise sparse matrix package.

[7] T. Davis, Direct Methods for Sparse Linear Systems.Fundamentals of Algorithms.Society for Industrial and Applied Mathematics, 2006.

[8] R. Barrett, Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. Miscellaneous Titles in Applied Mathematics Series No 43. Society for Industrial and Applied Mathematics, 1994.

[9] I. Koutis and G. Miler, The Combinatorial multigrid solver, in : Conference Talk,March, 2009.

[10] I. Koutis, G. L. Miller and D. Tolliver, Combinatorial Preconditioners and Multilevel Solvers for problems in computer vision and image processing. Computer Vision and Image Understanding, 115(12):1638–1646, 2011.

[11] I. Koutis, Matlab implementation of the combinatorial multigrid algorithm.

[12] D. Ntioudis, Development and Optimization of a combinatorial multigrid algorithm for large scale circuit simulation, 2013.

[13] E. G. Boman and B. Hendrickson, Support theory for preconditioning,SIAM J. Matrix Anal. Appl. 25 (3) (2003) 694-717.

[14] P. M. Vaidya, Solving linear equations with symmetric diagonally dominant matrices by constructing good preconditioners. A talk based on this manuscript was presented at the IMA Workshop on Graph Theory and Sparse Matrix Computation, October 1991.

[15] P. G. Doyle and J. L. Snell, Random Walks and Electric Networks, January 2000.

[16] M. Bern, J. R. Gilbert, B. Hendrickson, N. Nguyen and S. Toledo, Support-graph preconditioners. SIAM J. Matrix Anal. Appl., 27:930–951, 2005.

[17] K. Gremban, Combinatorial Preconditioners for Sparse, Symmetric, Diagonally Dominant Linear Systems. PhD thesis, Carnegie Mellon University, Pittsburgh, October 1996. CMU CS Tech Report CMU-CS-96-123.

[18] I. Koutis and G. L. Miller, Graph partitioning into isolated, high conductance clusters: theory, computation and applications to preconditioning. In Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures, SPAA ’08, pages 137–145, New York, NY, USA, 2008. ACM.

[19] L. Grady, A lattice-preserving multigrid method for solving the inhomogeneous poisson equaequations used in image analysis. ECCV, 5303:252–264, 2008.

[20] U. Trottenberg, A. Schuller and C. Oosterlee, Multigrid. Academic Press, 1st edition, 2000.

[21] A. Joshi, Topics in optimization and sparse linear systems. PhD thesis, Champaign, IL, USA, 1997. UMI Order No. GAX97-17289.

[22] D. A. Spielman and S.-H. Teng, Nearly-Linear Time Algorithms for Preconditioning and Solving Symmetric, Diagonally Dominant Linear Systems, May 2007.

[23] I. Koutis and G. L. Miller, A linear work, O(n1/6) time parallel algorithm for solving planar Laplacians. In Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms, SODA ’07, pages 1002–1011, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics, 2007.

[24] I. Koutis and G. L. Miller, Approaching optimality for solving, August 2010.

[25] I. Koutis, G. L. Miller και R. Peng, Solving sdd linear systems in time O(mlognlog(1/ε)), April 2011.

[26] I. Koutis, Combinatorial and algebraic tools for optimal multilevel algorithms.PhD thesis, Carnegie Mellon University, Pittsburgh, May 2007. CMU CS Tech Report CMU-CS-07-131, 2007.

[27] D. A. Spielman and S. I. Daitch, Faster approximate lossy generalized flow via interior point algorithms. In Proceedings of the 40th Annual ACM Symposium on Theory of Computing, May 2008.

[28] N. Bell and M. Garland, Efficient sparse matrix-vector multiplication on CUDA, NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.

[29] Y. Deng, B. Wang and S. Mu, Taming Irregular EDA Applications on GPUs. Computer-Aided Design - Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on, pp. 539–546, 2009.

[30] cuSPARSE library.
http://docs.nvidia.com/cuda/cusparse/

[31] IBM Power Grid Benchmarks.
http://dropzone.tamu.edu/~pli/PGBench/