ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ Η/Υ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ ΚΑΙ ΔΙΚΤΥΩΝ

Προσομοίωση μεγάλης κλίμακας κυκλωμάτων σε μαζικά παράλληλες αρχιτεκτονικές

Simulation of large scale circuits on massively parallel architectures

Διπλωματική Εργασία

Σαχπαλή Χαλήτ

Επιβλέποντες καθηγητές:

Ευμορφόπουλος Νέστωρ, Επίκουρος Καθηγητής

Μάρτιος 2017

Περίληψη

Γραμμικά συστήματα της μορφής Ax = b, για συμμετρικούς πίνακες με κυρίαρχη διαγώνιο, προκύπτουν πολύ συχνά σε προβλήματα προσομοίωσης πολύ μεγάλης κλίμακας κυκλωμάτων. Την τελευταία δεκαετία έχουν αναπτυχθεί ένα πλήθος εξειδικευμένων επιλυτών με σκοπό να αντιμετωπίσουν περιορισμένες περιπτώσεις από συστήματα τέτοιου είδους που προκύπτουν από μία συλλογή ποικίλων προβλημάτων. Σκοπός αυτής της διπλωματικής εργασίας είναι η επιτάχυνση της απόδοσης ενός παρόμοιου επιλυτή πάνω σε παράλληλες αρχιτεκτονικές για συστήματα τα οποία εμφανίζονται στην προσομοίωση κυκλωμάτων πολύ μεγάλης κλίμακας. Ο επιλυτής αυτός στηρίζεται στις αρχές της θεωρίας γράφων και επιτυγχάνει εξαιρετικά αποτελέσματα ενώ παράλληλα παρέχει ισχυρές εγγυήσεις για την ταχύτητα σύγκλισης.

Abstract

Linear systems of the form Ax = b, on symmetric diagonally dominant matrices, occur frequently in very large scale circuit simulation. In the past decade a multitude of specialized solvers have been developed to tackle restricted instances of SDD systems for a diverse collection of problems. In this thesis, we try to accelerate the performance of a similar solver on parallel architectures for systems that occur in very large scale circuit simulation. The solver is based on support graph theory principles and it achieves state of the art empirical results while providing robust guarantees on the speed of convergence.

Contents

1. Introduction 6
   1.1. Problem Description 6
   1.2. Thesis Contribution 6
2. Solving Linear Systems $\boldsymbol{Ax} = \boldsymbol{b}$ 7
   2.1. Sparsity Overview 7
   2.2. Iterative Methods 9
      2.2.1. Stationary Methods 9
         2.2.1.1. The Jacobi Method 9
         2.2.1.2. The Gauss-Seidel Method 10
      2.2.2. Non-stationary Methods 11
         2.2.2.1. Conjugate Gradient (CG) 11
         2.2.2.2. BiConjugate Gradient (BiCG) 12
   2.3. Computational Aspects of the Methods 13
   2.4. Multigrid Method 13
3. Preconditioners 15
   3.1. Jacobi Preconditioner 16
   3.2. SSOR Preconditioner 16
   3.3. Incomplete Factorization Preconditioners 17
4. Combinatorial Multigrid 18
   4.1. Related work on SDD solvers 18
   4.2. SDD linear systems as graphs 19
   4.3. The Multigrid algorithm 21
5. GPU Architecture 23
   5.1. Architecture of graphics processing units 25
   5.2. Programming Model 27
   5.3. Performance Optimization Methods 28
6. Improving the Performance of CMG 29
   6.1. Implementation and Optimizations 29

Bibliography 35

Chapter 1

Introduction

1.1 Problem Description

Circuit simulation is a technique to control and verify the design of electrical and electronic circuits, using mathematical models on the computer software. New designs can be tested and evaluated without actually constructing circuits or devices. It is used across a wide spectrum of applications, ranging from integrated circuits 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. Almost all integrated circuits design relies heavily on simulation.

1.2 Thesis Contribution

The circuit simulation is based on the solution of linear systems in the form Ax=b. These systems in circuit simulation arise after the Modified Nodal Analysis or MNA. Several algorithms are based on solving such sort of linear systems. The contribution of this thesis is implementation of the algorithm we ported the most time consuming part of the solver to a GPU, using the Compute Unified Device Architecture (CUDA) where is provided by NVIDIA, and addition of Combinatorial Multigrid (CMG) solver in the method preconditioned conjugate gradient (PCG) which is used as a preconditioner in solving symmetric diagonally dominant (SDD) systems.

Chapter 2

Solving Linear Systems $\boldsymbol{Ax = b}$

The methods used for solving linear systems $\boldsymbol{Ax = b}$ divided into direct and iterative methods. A direct method solving the system after a finite number of operations, which depends on the size N of the linear system. In contrast, iterative methods calculate an approximation of the real solution of the system, given a tolerance level.

2.1 Sparsity Overview

Consider the solution of linear systems of the form

$$Ax = b,$$

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. 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, a compressed-column matrix. 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.

For example, the matrix

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

is presented in zero-based triplet form below.

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

The triplet form is simple to create but difficult to use in most sparse matrix algorithms. The compressed-column (CCS) form 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 $nzmax$.

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

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 Iterative Methods

The iterative methods of solving linear systems of the form $Ax = b$ are divided into stationary and non-stationary.

2.2.1 Stationary Methods

Iterative methods that can be expressed in the simple form

$$x^{(k)} = Bx^{(k-1)} + c$$

(where neither B nor c depend upon the iteration count k) are called stationary iterative methods.

2.2.1.1 The Jacobi Method

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.

The definition of the Jacobi method can be expressed as

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

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

The Jacobi Method

2.2.1.2 The Gauss-Seidel Method

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.

The definition of the Gauss-Seidel method can be expressed as

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

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

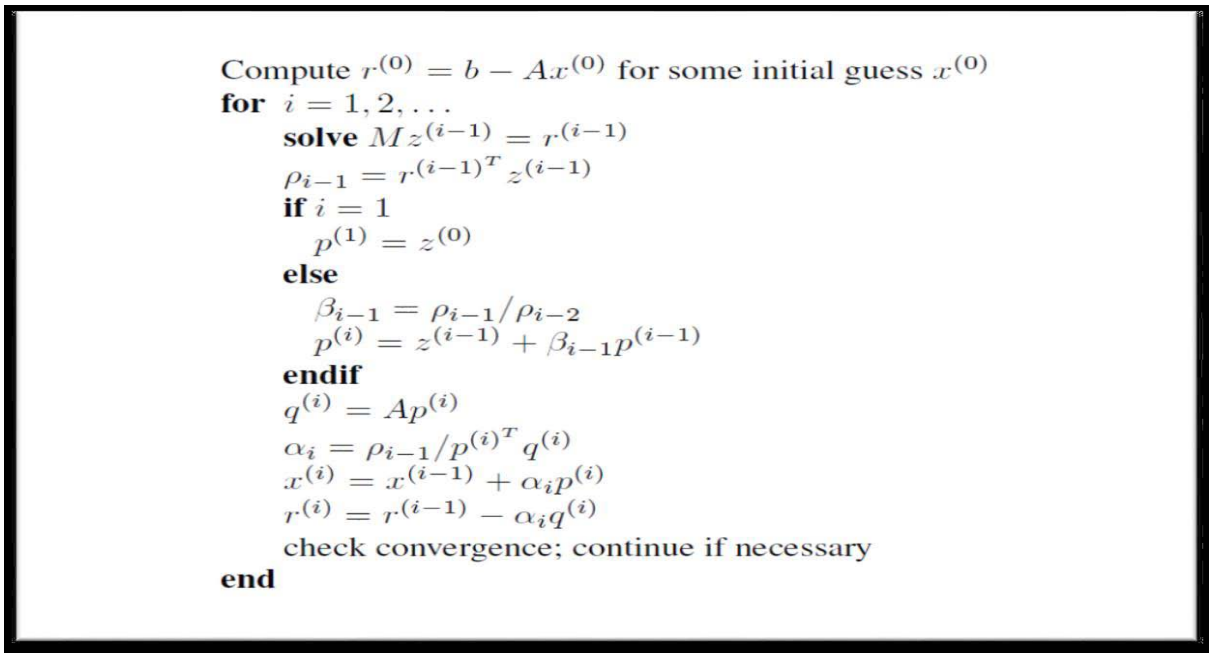
The Gauss-Seidel Method

2.2.2 Non-stationary 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.

2.2.2.1 Conjugate Gradient (CG)

The Conjugate Gradient method is an effective method for symmetric positive definite systems. The method proceeds by generating vector sequences of iterates, 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.



The Preconditioned Conjugate Gradient Method

2.2.2.2 BiConjugate Gradient (BiCG)

The BiConjugate Gradient method generates two CG-like sequences of vectors, one based on a system with the original coefficient matrix A, and one on A^T. This method, like CG, uses limited storage. It is useful when the matrix is nonsymmetric and nonsingular; however, convergence may be irregular. BiCG requires a multiplication with the coefficient matrix and with its transpose at each iteration.

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

The Preconditioned BiConjugate Gradient Method

2.3 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.

Method	Inner Product	SAXPY	Matrix-Vector Product	Precond Solve	Storage Reqmnts
Jacobi			1^a		Matrix+3n
Gauss Seidel			1^a		
CG	2	3	1	1	Matrix+6n
BiCG	2	5	2	2	Matrix+10n

2.4 Multigrid Method

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. 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.

Chapter 3

Preconditioners

The convergence rate of iterative methods depends on spectral properties of the coefficient matrix. 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 eigenvalues clustered around 1.

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

$$M^{-1}Ax = M^{-1}b$$

In the case of CG method, the preconditioned matrix $M^{-1}A$ not formed explicitly, as this would increase the computational requirements, due to the reversal of the matrix M. In contrast, the preconditioner is reduced to solving a number of linear systems (one in each iteration of CG method), described by the following equation:

$$\boldsymbol{Mz=r}$$

An efficient preconditioner should approximate well the system matrix so that the matrix $M^{-1}A \sim I$ and to accelerate the convergence rate of the method.

3.1 Jacobi Preconditioner

The preconditioner Jacobi is one of the simplest forms of preconditioning, in which the preconditioner is selected to be the diagonal elements of the matrix P=diag(A). Assuming $A_{ii} \neq 0$, we get

$$P_{ii}^{-1} = \frac{\delta_{ij}}{A_{ij}}$$

It is effective for diagonal dominant matrix A.

3.2 SSOR Preconditioner

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

$$M(\omega) = \frac{1}{2 - \omega}(\frac{1}{\omega}D + L)(\frac{1}{\omega}D)^{-1}(\frac{1}{\omega}D + L)^T$$

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.

3.3 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 very 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 m 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

Combinatorial Multigrid

4.1 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. The underlying geometry of the domain leads to a hierarchy of grids $A = A_0, ..., A_d$ 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 $2^{d-i} * 2^{d-i}$ for $i = 0, ..., d$. 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.

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) 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.

4.2 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

$$A_{ii} = \sum_{i \neq j} |A_{ij}|$$

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. 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

$$A_{ii} = \sum_{i \neq j} |A_{ij}|$$

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.

4.3 The Multigrid algorithm

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 $R^TQ^{-1}Rr$ 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 $R^TQ^{-1}Rr$ is a good approximation of A^{-1}. Algebraically, this correction is the application of the operator $T = (I - R^TQ^{-1}RA)$ to the error vector e. The choice of Q is most often not independent from that of R, as the Galerkin condition is employed:

$$Q = RAR^T$$

At a high level, the key idea behind CMG is that the provably small condition number $\kappa(A, B)$, is equal to the condition number $\kappa(\hat{A}, \hat{B})$ where $\hat{A} = D^{-1/2}AD^{-1/2}$ and $\hat{B} = D^{-1/2}BD^{-1/2}$.

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. $z := z + D^{1/2}R^Ty$
6. $x := D^{-1/2}((I - \hat{A})z + D^{-1/2}b)$

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 $A = A_0, ..., A_d$. The full multigrid algorithm we use, after simplifications in the algebra of the two-level scheme is as follows

```
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} := R r_i
5. z := CMG(A_{i+1}, b_{i+1})
6. for i = 1 to t_i − 1
7.       r_{i+1} := b_{i+1} − A_{i+1} z
8.       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)
```

Full Combinatorial Multigrid

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

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

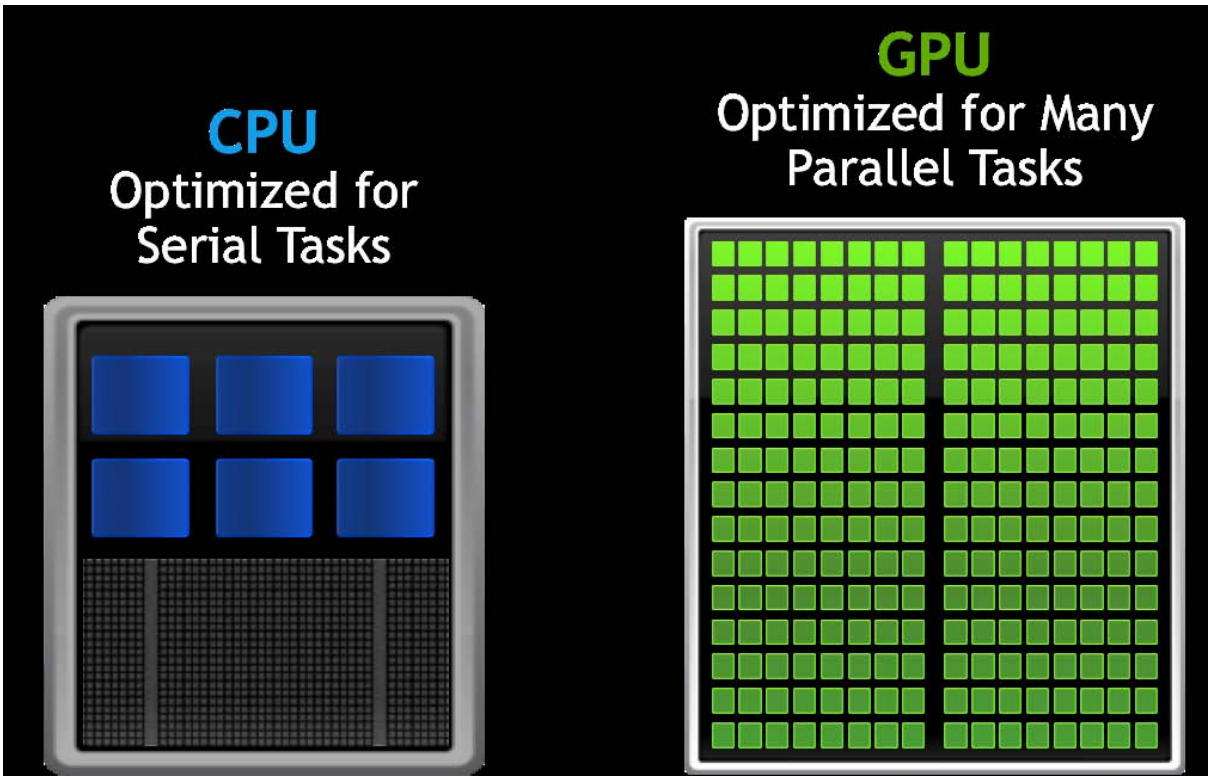
This choice for the number of recursive calls, combined with the fast geometric decrease of the matrix sizes, targets a geometric decrease in the total work per level, while optimizing the condition number.

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.

Chapter 5

GPU Architecture

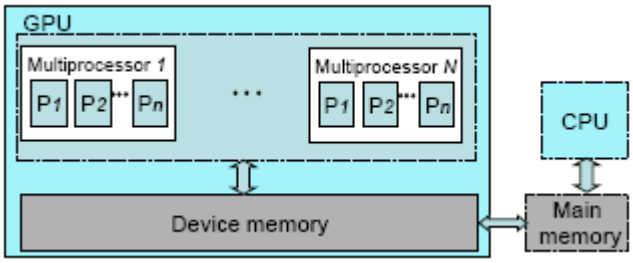
The Graphics Processing Units (GPUs) traditionally used for graphical applications mainly in computer games. The architecture of GPUs differ significantly from that of CPUs and especially newer multi-core CPUs. Typically GPUs are composed of hundreds of processors SIMD (Single Instruction Multiple Data) who offer the ability to perform parallel operations. Conversely the number of processors and even more on screened multi-core CPUs are dramatically lower.



CPU vs GPU Architecture

In addition, the transistor of GPUs devote themselves to computational units instead of the caches in the case of CPUs, with the caches of GPUs are typically 10 times smaller than those of the CPU. The following figure shows an example of GPU architecture model. Shows

how the GPU with multiple processors acting as a coprocessor for the CPU.

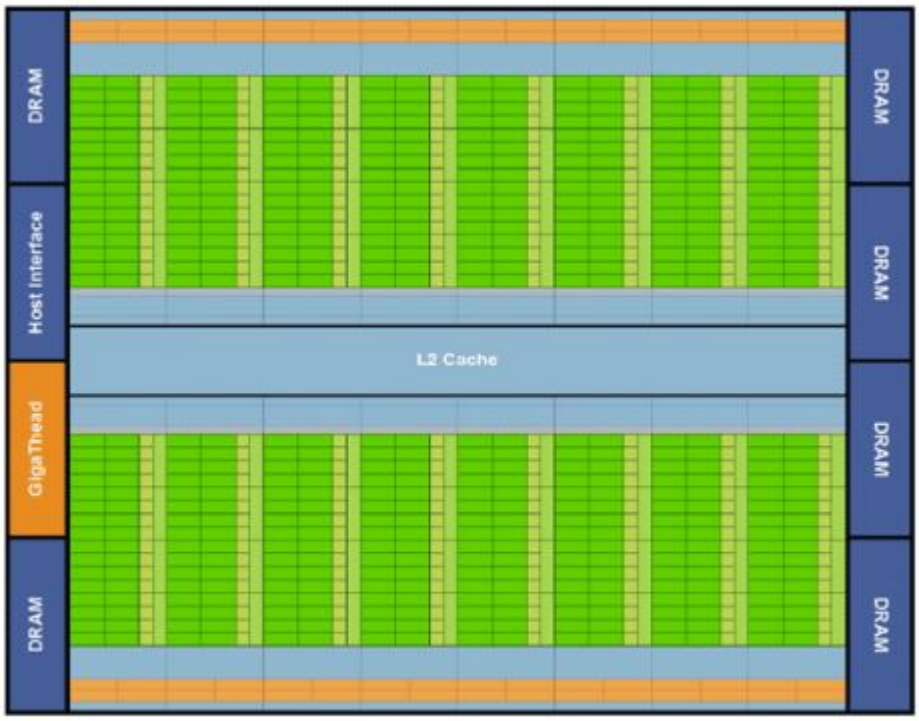


GPU Architecture

This special architectural style of GPUs soon led to the escape of the exclusively graphic applications and their use in general-use applications / calculations - General Purpose computation on GPU (GPGPU). The GPGPU represent for general purpose calculation on Graphics Processing Units, known as GPU computation. The GPUs with the use of many-core processors are able to achieve high performance calculations and data output. Today, GPUs are general purpose parallel processors to support interfaces to programming languages like C. Developers of applications on GPUs frequently achieve speedups versus optimized CPU applications.

5.1 Architecture of graphics processing units

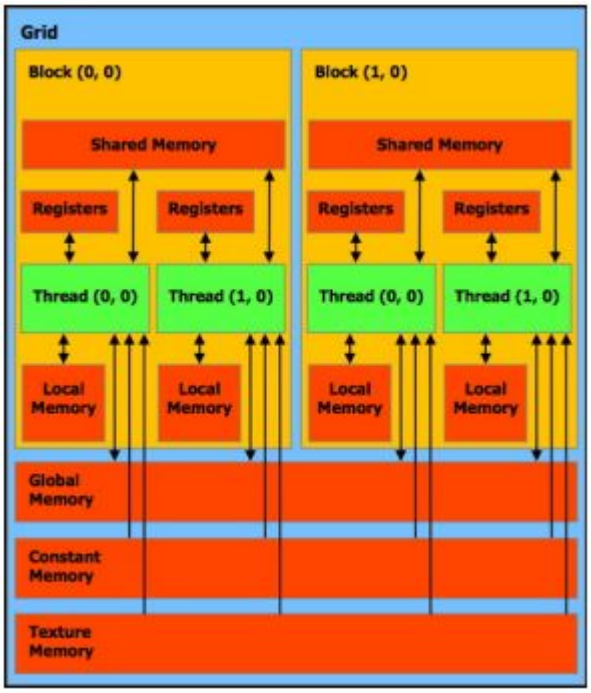
Modern graphics processing units consist of a multi-processor number, each of which has a specific number of cores. Each multi-processor contains 8, 32/48 or 192 single cores depending generation of the card. The GPU combines with the CPU via the PCI Express bus. Each processing unit has a different memory levels, global memory, texture memory and constant, the shared memory of each multi-processor and registers. The following figure shows the Fermi architecture.



Fermi's 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache).

As for the memory hierarchy in which graphics unit shown below. There is a global memory and the texture and constant memories. Also, each multi-processor has its own shared memory also an important number of registers. The global memory (Device Memory) is dynamic random access memory (DRAM), a very large capacity but very slow in response. The main memory response is from 400 to 600 clock cycles. Data transferred from the main memory of the computer can be stored in this memory. Also any outcome is required to return to the main

memory of the computer must be located in this memory. In global memory can write and read all the threads of a kernel. The shared memory is a fast memory, but very small capacity, each multi-processor having its own. The shared memory response time is 1-32 clock cycles. In the first generation card size was 16KB, while the Fermi architecture has 64KB,



Fermi architecture of NVIDIA GPUs

which share the shared memory and the cache (16KB, 48KB). Depending on the application can be selected by the programmer how many KB will be available for the cache and how much for the shared memory. Also important is the fact that the Fermi architecture exist a cache per multi-processor as mentioned above and a bigger for all multi-processors. The constant memory is a read-only memory, which has cache. The texture memory is an small memory which this also has cache, helping to improve the efficiency in the generation of the first card and in the third.

5.2 Programming Model

Basic element in executing programs GPUs is the thread. Modern GPUs require a very large number of threads to be executed in order to have maximum efficiency. This is due to the fact that switching between threads not aligns time and thus when a number of threads waiting to get data from the global memory immediately changed to another thread block having commands for execution.

To run a program in the graphics processing unit must first be allocate area in the global memory of card and transferred there the data required. At this point you can begin execution of the "core". The "core" is the part of code that will go for implementation in the GPU. When starting a "core" must apart from the usual arguments we give each function of the language C, to give the number of threads for execution.

The threads are organized into thread blocks with each 3 dimensions. The thread blocks organized into 3 dimensions in a grid of blocks. A thread blocks can have more than a specific number of threads for execution. This number depends on the generation of the graphics card and the "computing capacity". In the first generation the number of cards he was 512 and in next (Fermi, Kepler) 1024.

After the implementation of the core is completed, should the data be turned back into the computer's main memory so that it can be used by the central processing unit. However following be performed and other core which is needed data then do not need the data to be returned in the computer's memory. This happens many times when we want to synchronize threads of different blocks. While the threads of a block can be synchronized, there is no similar possibility of threads of different blocks.

5.3 Performance Optimization Methods

There are several details that one should watch to get the best performance from a single graphics card. First of all, we need data transfers to and from the card be minimized because they cost in time. Also, they should be reduced as much as possible accesses to main memory and those that would be to follow a particular pattern. Should the threads to read or write in contiguous memory locations. This way access to main memory is called coalesced and leads to better performance because the accesses of the threads are grouped into one. Still, you need to find the appropriate number of blocks and thread to be used as much as possible the cores of the card. This depends on the requirements in the registers of each thread and of the limitations of the material. Important advantage enables the use of shared memory, which is faster than the global, but usually can not hold all the data of the problem. Another feature that helps the good performance is to run as many thread as possible so as to hide the delays of memory. Something as important as the previous which it must noticed is that anyone should not, as far as possible, the threads have branch instructions. Because the threads running concurrently in a multi-processor execute the same instruction, a branch instruction would lead to a situation in which some of the threads remain inactive until you perform the next part of the branch. This leads to decline in performance.

We should note that in the last years the NVIDIA gives a very important tool for every programmer GPU. The Nsight that functions as add-on either the Eclipse or in Visual studio, allows easy debugging capabilities and performance check capabilities. An analysis is made of all parameters and are all points that delayed the program.

Chapter 6

Improving the Performance of CMG

The CMG solver is an extension of the preconditioned conjugate gradient method (PCG). 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.

6.1 Implementation and Optimizations

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.

System Specifications and IBM Power Grid Benchmarks

The hardware and software specifications of our system are described below

CPU	Intel(R) Core(TM) i7 CPU 870 @ 2.93GHz
GPU	GeForce GTX 690
MEMORY	32GB
GPU MEMORY	4GB
OS	Ubuntu 12.04.5 LTS

CUDA	Cuda 5.5
GNU COMPILER	gcc 4.7.2
NVIDIA COMPILER	nvcc 5.5

All the power grid benchmarks presented in this section are presented below.

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

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. The following table shows the dimensions and the number of non-zero elements of the MNA arrays corresponding to each IBM netlist.

Netlist	Dimension	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

Matrix size and non-zero elements of MNA arrays

Below are shows the number of hierarchy levels of matrices for each benchmark and the average non-zeros per row for all that matrices.

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

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

The CMG solver was taking advantage of the symmetry occurred in our SDD matrices by storing only the lower 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 below

```
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);

}
```

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 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. The experimental results of this approximation is shown in the below table, where we compare the execution times of those two methods.

```
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);
}
```

Netlist	Storing full matrix		Storing the lower part	
	SpMv	PCG	SpMV	PCG
ibmpg1	2,13	7,54	2,23	7,08
ibmpg2	0,38	0,71	0,21	0,69
ibmpg3	3.49	6,92	3,12	6,37
ibmpg4	2,24	4,42	1,87	4.01

The above results show that the only thing we gain taking advantage the symmetry is the storage space. By storing only the lower triangular part not we earn hardly any speed.

As indicated in the cusparse library of cuda, storage upper or lower triangular matrix into symmetric matrices do not offer us nothing more than storage space, we do not get more speed from the kernel of SpMV.

We mentioned that to solve this system on the GPU in parallel, we need to use atomic operations, as mentioned above atomic operations are very expensive on the GPU, or to solve the systems of $y = (L + D) * x$ and then solve $y = L^T * x + y$ as we have to solve the

transpose of the lower triangular matrix where costs 10x times of the normal SpMV.

Bibliography

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

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

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

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

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

[6] 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.

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

[8] 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.

[9] 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,

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

[11] 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,

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

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

[14] 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.

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

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