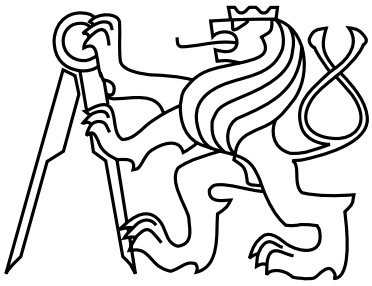
Czech technical University in Prague
Faculty of Electrical Engineering



Master Thesis

Iterative GPGPU Linear Solvers for Sparse Matrices

Bc. Filip Veselý

Supervisor: Ing. Ivan Šimeček

Computer Science and Engineering

Department of Computer Engineering

May 2008

Declaration

I hereby declare that I have completed this master thesis independently and that I have listed all the literature and publications used. I have no objection to usage of this work in compliance with the act §60 No. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

In Prague, May 20, 2008 .....

Abstract

Title: ***Iterative GPGPU Linear Solvers for Sparse Matrices***

The performance and the level of programmability of graphics processors (GPU) on current video cards offer new capabilities beyond the graphics applications for which they were designed. These are general-purpose computations which expose parallelism.

In this thesis, I describe the iterative methods for solving sparse linear systems: the Jacobi, Gauss-Seidel, Conjugate Gradient and BiConjugate Gradient method. I explain the architecture of the GPU, their capability of the general-purpose computations and current software platforms which provide an abstraction to the GPU programming: the BrookGPU and RapidMind Multi-core platform. I discuss the iterative methods in the GPU programming environment and describe the implementation for both platforms.

Then I made a performance tests between the sequential algorithms and the algorithms for Brook and Rapidmind. The results of my experiments show that the GPU is better suited for the computation with larger problem dimensions. For the iterative methods, the optimal problem size $\approx 10^6$ was observed, where most significant speedup compared to the sequential implementation was reached. Namely $3\times$ speedup for Brook and $1.5\times$ speedup for Rapidmind. Larger instances (10^7+) encountered a limitation of the graphics API. Brook has been found as faster platform than Rapidmind due to the necessary workarounds in Rapidmind implementation. The general Gauss-Seidel method has been found as not suitable for processing on the GPUs.

Abstrakt

Titul: ***Iterační řešiče pro grafické karty.***

Výkon a úroveň programovatelnosti grafických procesorů (GPU) na současných grafických kartách otevírá nové možnosti v jejich využití i mimo doménu jejich působnosti - renderování počítačové grafiky. Tím jsou numerické výpočty, které vyžadují paralelní zpracování.

V této práci popisuji iterační metody pro řešení řídkých lineárních soustav rovnic: Jakobiho metodu, Gauss-Seidelovu metodu, metodu konjugovaného gradientu a metodu bi-konjugovaného gradientu. Popisuji architekturu současných GPU, způsob jak se na ní dají mapovat numerické výpočty a současné platformy, které abstrahují programování GPU: BrookGPU a RapidMind Multi-core platform. Popisuji iterační metody v prostředí GPU a také jejich implementaci pro obě dvě platformy.

Následuje experimentální část, ve které srovnávám rychlosti sekvenční a obou paralelních implementací. Z mých experimentů vyplývá, že GPU je vhodná problémy, které zahrnují velký počet zpracovávaných dat. Pro iterační metody to znamená problém velikosti $\approx 10^6$, při kterém došlo k největšímu zrychlení paralelních implementací oproti sekvenční − konkrétně bylo pozorováno $3\times$ zrychlení pro Brook a $1.5\times$ pro Rapidmind. Větší instance (10^7+) narážejí na limity grafického API. Brook byl pozorován jako rychlejší platforma oproti platformě Rapidmind, kde bylo zapotřebí alternativní implementace redukce dvourozměrného pole. Obecná Gauss-Seidelova metoda byla shledána nevhodnou pro řešení na grafických kartách.

Table of Contents

List of figures

List of tables

List of abbreviations

BCRS	Block Compressed Row Storage
BiCG	BiConjugate Gradient method
BLAS	Basic Linear Algebra Subprograms
CCS	Compressed Column Storage
CDS	Compressed Diagonal Storage
CG	Conjugate Gradient method
COO	Coordinate sparse matrix storage format
CRS	Compressed Row Storage
CUDA	Compute Unified Device Architecture
FBO	Framebuffer Object
FEM	Finite Element Method
GLSL	OpenGL Shading Language
GPGPU	General-Purpose Computation Using Graphics Hardware
GPU	Graphics Processing Units
HLSL	High-Level Shader Language
JDS	Jagged Diagonal Storage
MIMD	Multiple Instruction Multiple Data
MVM	Matrix-vector multiply
PBO	Pixel Buffer Object
PD	Positive Definite
PDE	Partial Differential Equation
SAXPY	Scalar Alpha X Plus Y
SIMD	Single Instruction Multiple Data
SKS	Skyline Storage
SOR	Successive Overrelaxation
SPMD	Single Program Multiple Data
TJDS	Transposed Jagged Diagonal Storage

1 Introduction

The evolution of the graphic hardware (GPU) driven by the computer games business brought a graphics hardware as a high-performance, programmable and non-expensive chips. Nowadays, the graphic card has a truly programmable architecture which allows to process data with high parallelism and high memory access rate. That is the key motivation fact for using them for parallel processing.

The fast solution of linear equations with large sparse coefficient matrices is an essential requirement of advanced computations. The methods for solving sparse linear equations are suitable for processing on the hardware which enables parallelism and delivers better performance results.

1.1 Thesis Goal

This thesis goal is to study various iterative methods for sparse linear systems and implement them using the modern GPGPU languages which makes the programmability of the graphics hardware easier.

Sparse linear systems arise from many scientific fields and is essential to solve them efficiently. As the iterative methods for solving linear system of equations we consider usually: Jacobi method, Gauss-Seidel method, Conjugate Gradient method and the method of BiConjugate Gradient.

Further, we would like to implement the iterative methods for these GPGPU platforms to exploit the data-parallel processing they are providing. As the GPGPU platforms were chosen: the BrookGPU and RapidMind Multi-core platform.

Finally, we will proceed with an experimental part of our work: testing the algorithms on various sparse systems, measurement and comparing the result to the sequential algorithm. The speedup of the parallel algorithms will be observed and the conclusion presented.

1.2 Thesis Overview

In the initial chapters, we mention some background definitions including sparse matrices, vector algebra and linear systems that are necessary to understand before discussing the iterative methods for solving linear system of equations.

In Chapter 3, we introduce the iterative methods, we mention some sparse matrix properties that are essential to know for the linear systems convergence theory.

As processing exclusively sparse matrices, we mention in Chapter 4 the storage schemes of sparse matrices to make the iterative methods working more efficient.

In Chapter 5, we describe the modern GPU's architecture, the stream programming model, the availability to program GPUs as well as the bottlenecks in accessing their computational power. Further we discuss the abstraction platforms to program them.

In Chapter 6, we put the iterative methods in the context of the GPU programming. We mention the parallel algorithms too.

Chapter 7 is dedicated to description of implementation of the iterative methods on selected GPU platforms Brook and Rapidmind.

In Chapter 8, the various measurement results are discussed and justified.

Chapter 9 reviews the entire work on this thesis and presents the conclusion of our work.

The main document is followed with Bibliography and Appendixes with the content of enclosed CD and manual page of the application.

1.3 Related Work

During working on this thesis there several similar researches, publications and articles were found.

The work of [BCL07] is combining recent GPU programming techniques with supercomputing strategies (namely block compressed row storage and register blocking), and implemented a sparse general-purpose linear solver which outperforms leading-edge CPU counterparts.

In the publication of [BFGS05] the implementation of two basic, broadly useful, computational kernels is described: a sparse matrix conjugate gradient solver and a regular-grid multigrid solver. Further, they state that high-intensity numerical simulation computations can be performed efficiently on the GPU, which they regard as a full function streaming processor with high floating-point performance.

The article of [Göd00] describes some common techniques to improve performance of GPU-based implementations in linear algebra applications. The example presented here is a Jacobi iteration (commonly used as a smoother in multigrid scenarios) on a sparse matrix arising from Finite Element discretizations of standard operators.

2 Definitions

Before we introduce the iterative methods, we will mention some basic definitions including sparse matrices, vector algebra and linear systems.

2.1 Sparse Matrix

Sparse matrix is a variant of a matrix which has the minority of non-zero entries − is primarily populated with zeros. Sparse matrices arise typically from various real applications and scientific fields (engineering, modeling physical phenomena, etc.). Such matrices are generally large and sparse.

2.1.1 Density of a matrix

The density of a matrix is defined as the ratio of non-zero entries over the total number of entries.

$$d = \frac{nnz}{n} \tag{2.1}$$

where nnz is the number of the non-zero entries and n is the total number of entries in the matrix A.

A matrix with density around or less than 0.05 or 5% is usually considered to be sparse.

2.1.2 Types of sparse matrices

There are various sparse matrices types that differ in their sparsity structure and the application / scientific discipline they arise from.

Structured matrices have some regular sparsity structure (geometry) − e.g. rectangular with equally spaced grid points, or an irregular structure described as a graph, etc. This structure is brings benefits if applying a method that assumes such structure. According to [Dav07], structured matrices arise usually from disciplines such as structural engineering, computational fluid dynamics, model reduction, electromagnetics, semiconductor devices, thermodynamics, materials, acoustics, computer graphics/vision, robotics/kinematics, and other discretizations.

Unstructured matrices show no regular structure and therefore no special methods can be used. As observed by [Dav07] typically such scientific fields does not produce sparse matrices with much geometry: optimization, circuit simulation, networks and graphs (including web connectivity matrices), economic and financial modeling, theoretical and quantum chemistry, chemical process simulation, mathematics and statistics, and power networks. Further we discuss the sparsity structure see chapter 4. It is useful to represent the structure of a sparse matrix as a graph, where just the non-zero entries are highlighted. Various matrix types are depicted in Figure 2.1.

2.2 Vector algebra

In the *vector space*, there are mathematical quantities *vector* and *scalar*. These two categories can be distinguished from one another by their distinct definitions: *Scalars* are quantities which are fully described by a magnitude alone. *Vectors* are quantities which are fully described by both a magnitude and a direction.

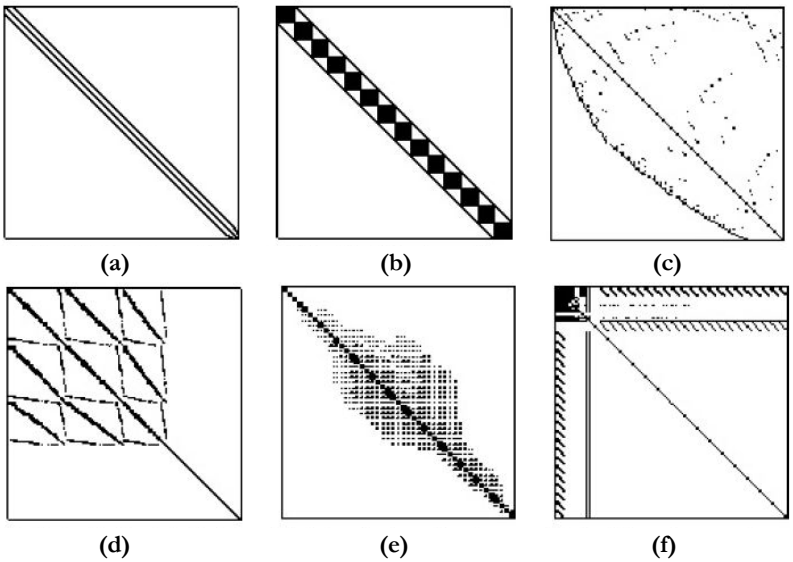


Figure 2.1: These graphs represent the structure of matrices from a variety of scientific and engineering areas. The matrices are from the matrix repository, the Matrix Market[Mar07]. (a) 2D fluid flow in a driven cavity. (b) Model of H2+ in an Electromagnetic Field. (c) Simulation of computer systems. (d) Air-traffic Control Model. (e) Astrophysics. (f) Jacobian of a nonlinear system of modeling a laser.

2.2.1 Inner product

The inner product (also called a scalar product) of two vectors $\vec{x} = (x_1, x_2, ..., x_n)$, $\vec{y} = (y_1, y_2, ..., y_n)$ in a vector space is a way to multiply vectors together, where the result of this multiplication is a scalar c:

$$c = x_1y_1 + x_2y_2 + \cdots + x_ny_n \tag{2.2}$$

2.2.2 Vector update

Vector update (also referred as SAXPY) involves a scalar-vector multiplication and a vector addition, where α is the scalar and $\vec{x}$ the multiplied vector

$$\vec{y} = \vec{y} + \alpha \cdot \vec{x} \tag{2.3}$$

2.2.3 Matrix-vector product

The matrix-vector product c_i for matrix A and vector $\vec{b}$, where the $a_{i,j}$ are the coefficients of the matrix A is

$$c_i = \sum_{r=1}^{n} a_{i,r}b_r = a_{i,1}b_1 + a_{i,2}b_2 + \cdots + a_{i,n}b_n. \tag{2.4}$$

2.3 Linear System

Linear system or system of linear equations is a collection of linear equations involving the same set of variables. In matrix terms, such system can be rewritten as follows:

$$\mathbf{A}\vec{x} = \vec{b} \tag{2.5}$$

where $\mathbf{A}$ is the coefficient matrix, $\vec{x}$ is the vector of unknown variables and $\vec{b}$ is the right-hand side vector.

$$\begin{pmatrix} a_{11} & a_{12} & \cdot\cdot & a_{1n} \\ a_{21} & a_{22} & \cdot\cdot & a_{2n} \\ : & : & & : \\ a_{m1} & a_{m2} & \cdot\cdot & a_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ : \\ x_m \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ : \\ b_m \end{pmatrix}$$

2.3.1 Solution of a linear system

The solution of a system (2.5) is an assignment of values to the variables $x_1, x_2, ..., x_n$ such that each of the equations is satisfied. The set of all possible solutions is called the solution set.

A linear system may behave in any one of three possible ways:

1. The system has infinitely many solutions.
2. The system has a single unique solution.
3. The system has no solutions.

2.3.2 Sparse linear system

Sparse linear system is such linear system which involves a coefficient matrix which is sparse and large. In the real-world applications large linear system means system with the dimension n is thousands to millions.

Fortunately, the large linear systems found in practice are generally sparse and symmetric and have additional properties which allow for more efficient solution.

3 Iterative methods

Iterative method is a technique that attempts to solve a problem − in our case finding solution of a linear system of equations (see equation 2.5) − by finding successive approximations to the solution starting from an initial guess. We can rewrite it into following form:

$$||A\tilde{x} - b|| \leq \varepsilon ||b|| \quad , \tag{3.1}$$

where $\tilde{x}$ is the approximate solution, ε is given and $0 < \varepsilon < 1$. This is one of the most fundamental of all computational problems.

This approach is in contrast to *direct methods*, which attempt to solve the problem by a finite sequence of operations, and, in the absence of rounding errors, would deliver an exact solution (like solving a linear system of equations by *Gaussian elimination*).

Iterative methods are useful for solving linear problems involving a large number of variables, where the direct methods would be too expensive or impossible to solve even with the best available computing power. These methods for solving linear systems of equations are at the heart of many computational science applications. Examples include science domains such as astrophysics, biology, chemistry, fusion energy, power system networks, and structural engineering, employing a diverse set of modeling approaches, such as computational fluid dynamics, finite element modeling, and linear programming.

In this thesis we will describe and later implement, two types of iterative methods for each *stationary* and *nonstationary* methods.

Stationary methods are older, simpler to understand and implement, but usually not as effective. Nonstationary methods are a relatively recent development; their analysis is usually harder to understand, but they can be highly effective [BBC$^+$94].

3.1 Convergence of the Iterative Methods

The rate at which an iterative method converges depends greatly on the spectrum of the coefficient matrix [BBC$^+$94]. The methods may involve a second matrix that transforms the coefficient matrix into one with a more favorable spectrum. The transformation matrix is called a *preconditioner*. A good preconditioner improves the convergence of the iterative method, sufficiently to overcome the extra cost of constructing and applying the preconditioner. More about preconditioner is mentioned in Section 3.5.

Let's mention first some of the special matrix properties that are essential to decide about the convergence of an iterative method on a linear system applied. In the discussion about the convergence further in the text we will refer these definitions often:

Symmetric matrix A matrix is symmetric if

$$A = A^T \quad , \tag{3.2}$$

where A^T is the transpose of A.

Positive definite matrix A matrix A is positive-definite if, for every nonzero vector x,

$$x^T A x > 0.$$

A is Symmetric Positive Definite if A is symmetric and Positive Definite.

(Weakly) diagonally dominant matrix The matrix A is weakly diagonally dominant if, for all matrix coefficients a_{ij} of A:

$$|a_{ii}| \geq \sum_{i \neq j} |a_{ij}| \quad i = 1, ..., n. \tag{3.3}$$

Strictly diagonally dominant matrix (non-singular matrix) The matrix A is strictly diagonally dominant if, for all matrix coefficients a_{ij} of A:

$$|a_{ii}| > \sum_{i \neq j} |a_{ij}| \quad i = 1, ..., n. \tag{3.4}$$

Irreducibly diagonally dominant matrix Matrix A is irreducibly diagonally dominant matrix if

1. A is weakly diagonally dominant
2. A is strictly diagonally dominant in at least one row

To unify the presentation of all methods, let introduce $\boldsymbol{D}$, $\boldsymbol{L}$, and $\boldsymbol{U}$ that denote the diagonal, lower triangular and upper triangular parts of a matrix $\boldsymbol{A}$:

$$D_{ij} = \begin{cases} A_{ij} & \text{for} \quad i = j \ , \\ 0 & \text{otherwise;} \end{cases} \quad L_{ij} = \begin{cases} A_{ij} & \text{for} \quad i < j \ , \\ 0 & \text{otherwise;} \end{cases} \quad U_{ij} = \begin{cases} A_{ij} & \text{for} \quad i > j \ , \\ 0 & \text{otherwise;} \end{cases}$$

3.2 Complexity of the Iterative Methods

The analysis of the complexity of iterative methods is not trivial. All the iterative methods depend on the convergence criteria that depends on the coefficient matrix A and the right-hand side vector b. However, we can state at least the complexity of one iteration pass and compare the methods from this point.

Overview of the methods

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.

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.

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 iterates. They are also the gradients of a quadratic functional, the minimization of which is equivalent to solving the linear system. CG is an extremely effective method when the coefficient matrix is symmetric positive definite.

BiConjugate Gradient (BiCG) The BiConjugate Gradient method generates two CG-like sequences of vectors, one based on a system with the original coefficient matrix , and one on . 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.

Let's discuss mentioned methods in details.

3.3 Stationary methods

Following form expresses an iterative method

$$x^{(k)} = Bx^{(k-1)} + c \tag{3.5}$$

Stationary iterative methods are such methods where neither B nor c depends upon count k in this form.

3.3.1 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 remain fixed, we obtain

$$x_i = \frac{(b_i - \sum_{i \neq j} a_{i,j}x_j)}{a_{i,i}} \tag{3.6}$$

This suggests an iterative method defined by

$$x_i^{(k)} = \frac{(b_i - \sum_{i \neq j} a_{i,j}x_j^{(k-1)})}{a_{i,i}} \tag{3.7}$$

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 [BBC+94]. Figure 3.1 illustrates the method.

In matrix terms, the definition of the Jacobi method can be expressed as

$$x^k = \mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})x^{k-1} + \mathbf{D}^{-1}b \tag{3.8}$$

Algorithm Note that an auxiliary storage vector σ is used in the Algorithm 3.3.1. It is not possible to update the vector σ in place, since values from x^{k-1} are needed throughout the computation of x^k.

Convergence of the Jacobi method Iteration does not always converge. The method will always converge if the matrix A is diagonally dominant, if is a non-singular matrix (strictly diagonally dominant) or irreducibly diagonally dominant. For definitions see section 3.1.

The Jacobi method sometimes converges even if this condition is not satisfied. It is necessary, however, that the diagonal terms in the matrix are greater (in magnitude) than the other terms.

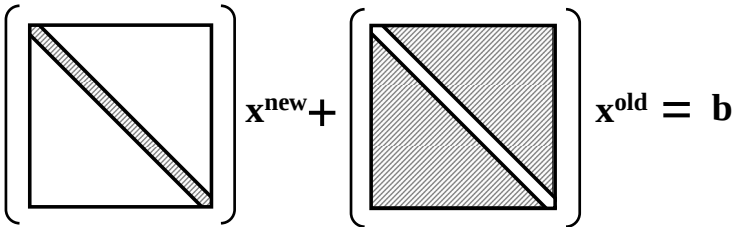


Figure 3.1: Scheme of the Jacobi method

Algorithm 1 Jacobi method

Choose an initial guess $x^{(0)}$ to the solution x
repeat
  for $i = 1$ to n **do**
    $\sigma = 0$
    for $j = 1$ to n **do**
      if $j \neq i$ **then**
        $\sigma = \sigma + a_{i,j}x_j^{(k-1)}$
      end if
    end for
    $x_i^{(k)} = \frac{(b_i - \sigma)}{a_{ii}}$
  end for
until convergence is reached
$x \approx x^{(k)}$

3.3.2 Gauss-Seidel method

If we proceed as with the Jacobi method, but now assume that the equations are examined one at a time in sequence, and that previously computed results are used as soon as they are available, we obtain the Gauss-Seidel method:

$$x_i^k = \frac{(b_i - \sum_{j<i} a_{i,j}x_j^{(k)} - \sum_{j>i} a_{i,j}x_j^{(k-1)})}{a_{i,i}} \tag{3.9}$$

This derives the Algorithm 3.3.2. The main difference to the Jacobi methods lies in a more efficient use of Equation 3.6. 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. However, 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. This will be discussed futher in the Chapter 6. The scheme of the method is in Figure 3.2.

In matrix terms, the definition of the Gauss-Seidel method can be expressed as:

$$x^k = (\mathbf{D} - \mathbf{L})^{-1}(\mathbf{U}x^{k-1} + b) \tag{3.10}$$

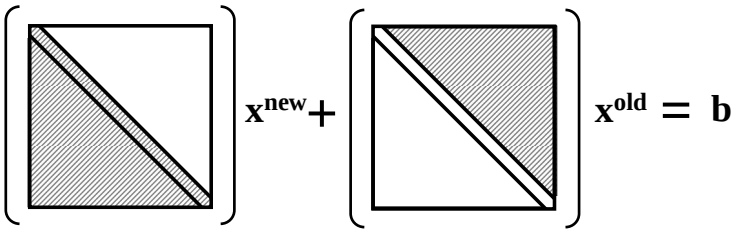


Figure 3.2: Scheme of the Gauss-Seidel method

Algorithm 2 Gauss-Seidel Method

Choose an initial approximation x^0
repeat
  for $i = 1$ to n **do**
    $\sigma = 0$
    for $j = 1$ to $(i-1)$ **do**
      $\sigma = \sigma + a_{ij}x_j^{(k)}$
    end for
    for $j = (i+1)$ to n **do**
      $\sigma = \sigma + a_{ij}x_j^{(k-1)}$
    end for
    $x_i^{(k)} = \frac{(b_i - \sigma)}{a_{ii}}$
  end for
until convergence is reached
$x \approx x^{(k)}$

Convergence of the Gauss-Seidel method Since the Gauss-Seidel method comes from Jacobi method just the equations are examined one at a time in sequence, convergence is the same as the Jacobi's (see Section 3.3.1). Variations of Gauss-Seidel methods described further in text differ in the terms of convergence. A poor choice of ordering can degrade the rate of convergence; a good choice can enhance the rate of convergence.

A routine which check the dominancy should be used before any call to Jacobi iteration or Gauss-Seidel iteration is made. There exists a counter-example for which Jacobi iteration converges and Gauss-Seidel iteration does not converge.

3.3.3 The Successive Overrelaxation Method

The Successive Overrelaxation Method (SOR) is the Gauss-Seidel method with an extrapolation applied. This extrapolation has the form of a weighted average between the previous iterate and the computed Gauss-Seidel iterate successively for each component:

$$x_i^{(k)} = \omega \tilde{x}_i^{(k)} + (1-\omega) x_i^{(k-1)}, \tag{3.11}$$

where $\tilde{x}$ is the Gauss-Seidel iterate, and ω is the extrapolation factor.
In matrix terms, the SOR algorithm can be written as

$$x^{(k)} = (\mathbf{D} - \omega \mathbf{L})^{-1}[\omega \mathbf{U} + (1-\omega)\mathbf{D}]x^{(k-1)} + \omega(\mathbf{D} - \omega \mathbf{L})^{-1}\vec{b} \tag{3.12}$$

The idea is to choose a value for ω that will accelerate the rate of convergence of the iterates to the solution.

Convergence of Successive Overrelaxation Method The choice of relaxation factor ω is depends on the properties of the coefficient matrix A.

For symmetric, positive definite matrices, *Kahan's theorem* states that the SOR method will converge only if ω is chosen in the interval $0 < \omega < 2$. Note that for $\omega = 1$ that the iteration reduces to the Gauss-Seidel iteration. As with the Gauss-Seidel method, the computation may be done in place, and the iteration is continued until the changes made by an iteration are below some tolerance.

We are generally interested in faster convergence rather than just convergence. Using SOR with the optimal relaxation parameter significantly increases the rate of convergence. Note however that the convergence acceleration is obtained only for ω very close to ω_{opt}. If ω_{opt} cannot be computed exactly, it is better to take ω slightly larger rather than smaller [GHM07].

There are various methods that adaptively set the relaxation parameter ω based on the observed behavior of the converging process however this is beyond the topic of our work.

3.4 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.

3.4.1 Conjugate Gradient

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.

Steepest Descent One of the simplest of the *gradient methods*, is the method of "Steepest Descent". We will mention this method here before we proceed to the CG.

In steepest descent method we construct a functional which when extremized will deliver the solution of the problem. The functional will have convexity properties, so that the vector which extremizes the functional is the solution of the algebraic problem. This means that we search for the vector for which the gradient of the functional is zero, and this can be done in an iterative fashion.

We assume that A is symmetric positive definite. Then solving $Ax = b$ is equivalent to problem of minimizing a quadratic function of vector x, so-called *quadratic form* (3.13):

$$f(x) = \frac{1}{2}x^T Ax - x^T b \tag{3.13}$$

The quadratic form appears in Figure 3.3. Note that if the matrix is not symmetric positive-definite, the convexity properties of the quadratic form are different.

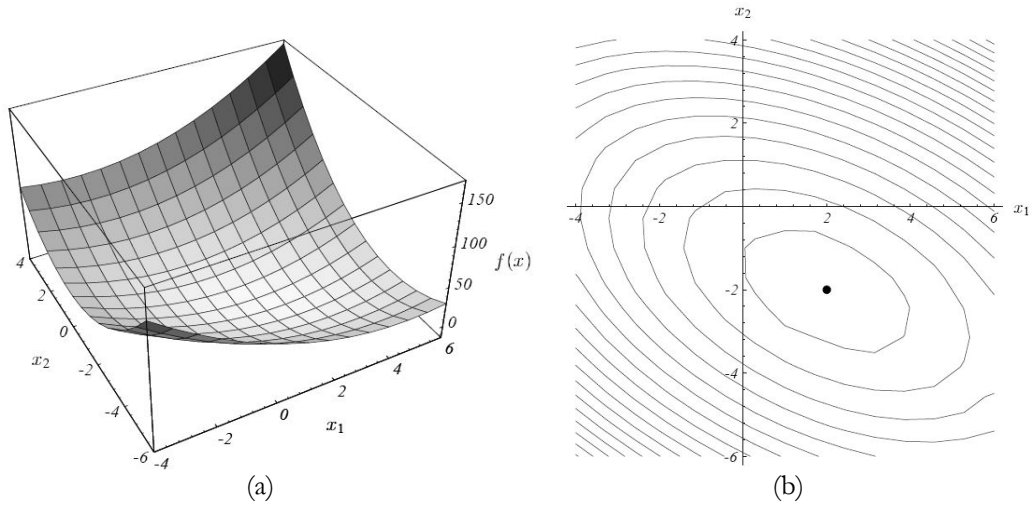


Figure 3.3: (a) Graph of a quadratic form $f(x)$. (b) The minimum point of this surface is the solution to $Ax = b$. This Figure is taken from the article [She94].

The choice of direction is where $f(x)$ decreases most quickly, which is in the direction opposite to $f'(x_i)$. The search starts at an arbitrary point x_0 and then slide down the gradient, until we are close enough to the solution. In other words, the iterative procedure is (let's assume just first step $x_0 \to x_1$):

$$x_{(1)} = x_{(0)} + \alpha r_{(0)} \quad , \tag{3.14}$$

where $r_0 = -f'(x_0)$ is the gradient at one given point. Now the question is what is the value of α; A *line search* is a procedure that chooses α to minimize f along a line. We want to move to the point where the function $f(x)$ takes on a minimum value, which is where the *directional derivative* is zero (note that the chain rule is applied):

$$\frac{d}{d\alpha}f(x_1) = f'(x_1)^T \frac{d}{d\alpha}x_1 = f'(x_1)^T r_0. \tag{3.15}$$

Setting this expression to zero, we find that α should be chosen so that $f'(x_1)$ and r_0 are orthogonal. The next step is then taken in the direction of the negative gradient at

this new point and we get a *zig-zag* pattern as illustrated in Figure (3.5a). This iteration continues until the extremum has been determined within a chosen accuracy ε. For more details about Steepest Descent see [She94].

Steepest Descent converges slowly is that it has to take a right angle turn after each step, and consequently search in the same direction as earlier steps (Figure 3.5a). The method of Conjugate Gradients is an attempt to mend this problem by "learning" from experience.

Conjugate Directions *Conjugate* or *A-orthogonal* means that two unequal vectors d_i and d_j are orthogonal with respect to symmetric positive definite matrix A:

$$d_i^T A d_j = 0 \tag{3.16}$$

The idea of Conjugate Directions is to let each search direction d_i be dependent on all the other directions searched to locate the minimum of $f(x)$ through equation (3.16). A set of such search directions is referred to as a A-orthogonal, or conjugate set, and it will take a positive definite quadratic function as given in (3.13) to its minimum point in, at most, n exact linear searches. In the thesis of [Hjo99] is shown a best way to visualize the Conjugate Directions is by comparing the space we are working in with a "stretched" space. An example of this "stretching" of space is illustrated in Figure 3.4.

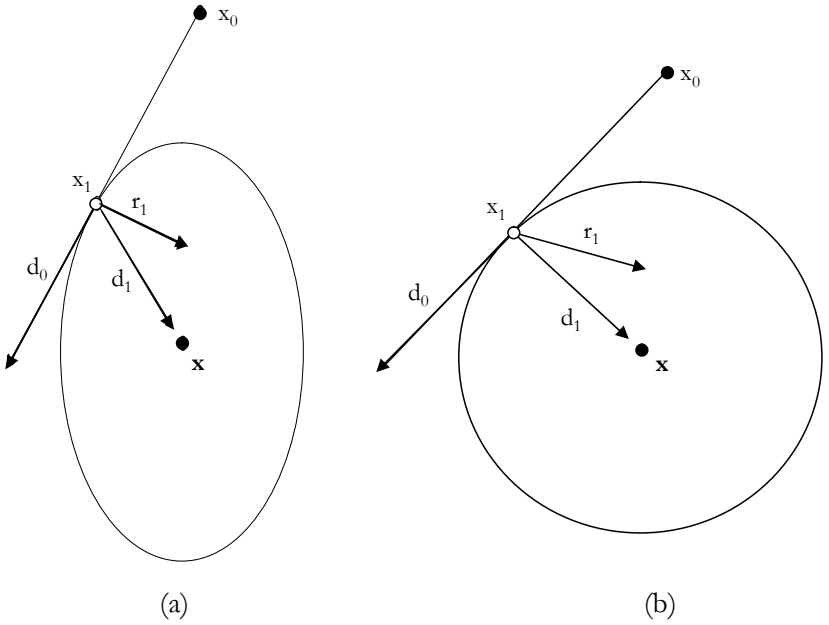


Figure 3.4: Optimality of the method of Conjugate Directions. (a) Lines that appear perpendicular are orthogonal. (b) The same problem in a "stretched" space. Lines that appear perpendicular are A-orthogonal.

The search for a minimum of the quadratic functions starts at x_0 in Figure 3.4a, and takes a step in the direction d_0 and stops at the point x_1 . This is a minimum point along that direction, determined the same way as for the Steepest Decent method in the previous section: the minimum along a line is where the directional derivative is zero (equation 3.15). The essential difference between the Steepest Descent and the Conjugate Directions lies in the choice of the next search direction from this minimum point. While the Steepest Descent method would search in the direction r_1 in Figure 3.4a, the Conjugate Direction method would chose d_1. How come Conjugate Directions manage to search in

the direction that leads us straight to the solution x? Since the next search direction d_1 is constrained to be A-orthogonal to the previous, they will appear perpendicular in this modified space. Hence, d_1 will take us directly to the minimum point of the quadratic function $f(x)$.

The Conjugate Gradients method is a special case of the method of Conjugate Directions, where the conjugate set is generated by the gradient vectors. For a quadratic function, as given in equation (3.13), we proceed as follows: The initial step is in the direction of the steepest descent:

$$d_0 = -f'(x_0) = -g_0 \tag{3.17}$$

Subsequently, the mutually conjugate directions are chosen so that:

$$d_{(k+1)} = -g_{(k+1)} + \alpha_k d_k \tag{3.18}$$

where the coefficient α_k is given by, for example, the so called *Fletcher-Reeves formula*:

$$\alpha_k = \frac{{g_{(k+1)}}^T g_{(k+1)}}{{g_k}^T g_k} \tag{3.19}$$

For details, see [Hjo99]. The step length along each direction is given by

$$\alpha_k = \frac{{d_{(k)}}^T g_{(k)}}{{d_k}^T A d_k} \tag{3.20}$$

The resulting iterative formula is identical to 3.14. In Figure 3.5, we can observe the speed of "hitting" the minimum for the methods of Steepest Descent and Conjugate Gradient.

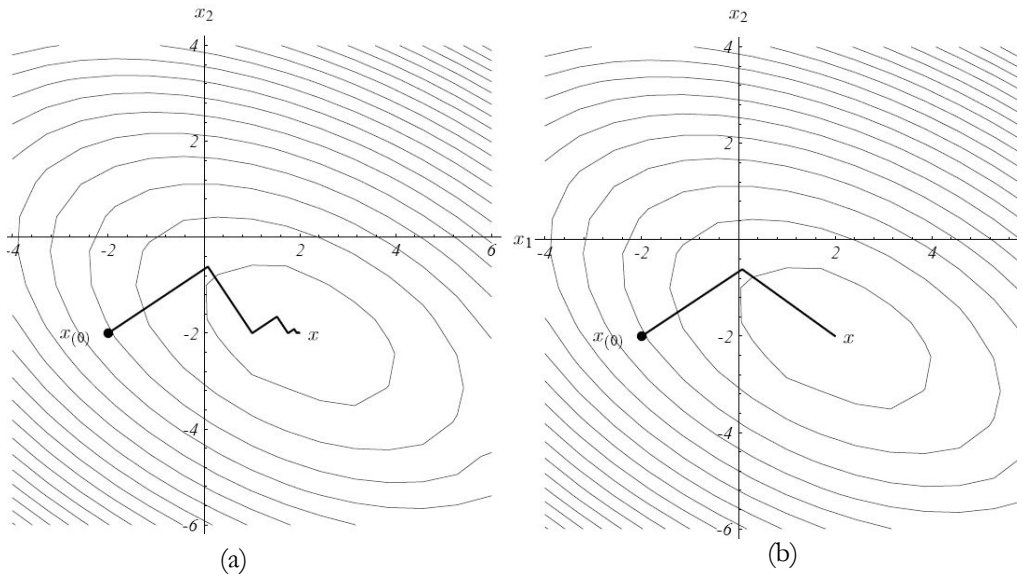


Figure 3.5: Comparison of the search directions: (a) Steepest Descent (b) Conjugate Gradient. Image taken from the thesis of [She94].

After some simplifications, this results in the following algorithm (3.4.1) for solving $Ax = b$ where A is a real, symmetric, positive-definite matrix.

The Conjugate Gradients method is apart from being an optimization method, also one of the most prominent iterative method for solving sparse systems of linear equations. It is fast and uses small amounts of storage since it only needs the calculation and storage

Algorithm 3 Conjugate Gradient Method

$r_0 := b - Ax_0$
$p_0 := r_0$
$k := 0$
repeat
  $\alpha_k := \frac{r_k^\top r_k}{p_k^\top A p_k}$
  $x_{k+1} := x_k + \alpha_k p_k$
  $r_{k+1} := r_k - \alpha_k A p_k$
  $\beta_k := \frac{r_{k+1}^\top r_{k+1}}{r_k^\top r_k}$
  $p_{k+1} := r_{k+1} + \beta_k p_k$
  $k := k + 1$
until convergence is reached
$x \approx x_{k+1}$

of the second derivative at each iteration. The latter becomes significant when n is so large that problems of computer storage arises. But, everything is not shiny; the less similar f is to a quadratic function, the more quickly the search directions lose conjugacy, and the method may in the worst case not converge at all [Hjo99] .

Convergence of the Conjugate Gradient method The predictions of the convergence of Conjugate Gradient is not a trivial topic. In general we assume involving matrix that is symmetric positive-definite. However, matrices with other sparsity structure and properties could be observed to converge too. For more about convergence see [BDD$^+$00].

3.4.2 BiConjugate Gradient

The Conjugate Gradient method is not suitable for non-symmetric systems because the residual vectors cannot be made orthogonal with short recurrences. 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 . Thus we update two sequences of residuals:

$$r_i = r_{(i-1)} - \alpha_{(i)} A p_{(i)}, \quad \tilde{r}_{(i)} = \tilde{r}_{(i-1)} - \alpha_i A^T \tilde{p}_{(i)} \tag{3.21}$$

and two sequences of search directions

$$p_i = r_{(i-1)} + \beta_{(i-1)} p_{(i-1)}, \quad \tilde{p}_{(i)} = \tilde{r}_{(i-1)} + \beta_{(i-1)} \tilde{p}_{(i-1)} \tag{3.22}$$

The choices

$$\alpha_{(i)} = \frac{\tilde{r}_{(i-1)}^T r_{(i-1)}}{\tilde{p}_i^T A p_{(i)}}, \quad \beta_{(i)} = \frac{\tilde{r}_{(i)}^T r_{(i)}}{\tilde{r}_{(i-1)}^T r_{(i-1)}} \tag{3.23}$$

ensure the *bi-orthogonality relations* given by:

$$\tilde{r}_{(i)}^T r^j = \tilde{p_{(i)}}^T A p^j = 0, \text{ if } i \neq j. \tag{3.24}$$

The resulting method, similar to the scheme of CG (Algorithm 3.4.1) is in Algorithm 3.4.2.

Algorithm 4 BiConjugate Gradient Method

$r_0 := b - Ax_0$
$p_0 := r_0$
$\tilde{p}_0 := \tilde{r}_0$
$k := 0$
repeat
  $\alpha_k := \frac{r_k^T \tilde{r}_k}{p_k^T A \tilde{p}_k}$
  $x_{k+1} := x_k + \alpha_k p_k$
  $r_{k+1} := r_k - \alpha_k A p_k$
  $\tilde{r}_{k+1} := \tilde{r}_k - \alpha_k A^T \tilde{p}_k$
  $\beta_k := \frac{r_{k+1}^T \tilde{r}_{k+1}}{r_k^T \tilde{r}_k}$
  $p_{k+1} := r_{k+1} + \beta_k p_k$
  $\tilde{p}_{k+1} := \tilde{r}_{k+1} + \beta_k \tilde{p}_k$
  $k := k + 1$
until convergence is reached
$x \approx x_{k+1}$

Convergence of BiCG method Few theoretical results are known about the convergence of the BiConjugate Gradient method. For symmetric positive definite systems, the method returns the same results as the conjugate gradient method, but at twice the cost per iteration. For more about the method's convergence see [BDD+00].

3.5 Preconditioners

Preconditioning is a technique for improving the condition number of a matrix. Suppose that M is a symmetric, positive-definite matrix that approximates A, but is easier to invert. We can solve $Ax = b$ indirectly by solving

$$M^{-1}Ax = M^{-1}b \tag{3.25}$$

This linear system has the same solution as the original system $Ax = b$, but the spectral properties of its coefficient matrix may be more favorable.

If we are searching a preconditioner, we are faced with a choice between:

1. finding a matrix M that approximates A, and for which solving a system is easier than solving one with A, or
2. finding a matrix M that approximates A^{-1}, so that only multiplication by M is needed.

The majority of preconditioners falls in the first category; For more about preconditioners see [BBC+94].

4 Sparse Matrix Storage Schemes

The efficiency of most of the iterative methods is determined primarily by the performance of the matrix-vector product and therefore on the storage scheme used for the matrix. Often, the storage scheme used arises naturally from the specific application problem.

As said in Section 2.1, if most of the entries of a matrix are 0, then the matrix is said to be sparse. In such a case, it may be very expensive to store zero values for two reasons: the actual storage, and useless multiplication product − we will be multiplying $x \times 0$ many times. There is no single "best" method to represent a sparse matrix. The selection of the possible storage format is dependent on the algorithm being used, the original sparsity pattern of the matrix, the underlying computer architecture, together with the considerations such as in what format the data already exists, and so one.

In this chapter we will describe common schemes to store sparse matrices of different sparsity structure, such as tridiagonal, diagonal, random sparse, etc.

4.1 Coordinate format

The coordinate format is the most flexible and simplest format for the sparse matrix representation. Only non-zero elements are stored, and the coordinates of each non-zero element are given explicitly. The coordinate format is specified by three arrays: values, rows, and column, and a parameter nnz which is number of non-zero elements in A. All three arrays have dimension nnz:

- **Values** An array of real numbers that contains the non-zero elements of A in any order.
- **Rows** Element i of the integer array rows is the number of the row in A that contains the i-th value in the values array.
- **Columns** Element i of the integer array columns is the number of the column in A that contains the i-th value in the values array .

For instance the official exchange storage format called Matrix Market (see [Mar07]) uses the Coordinate format.

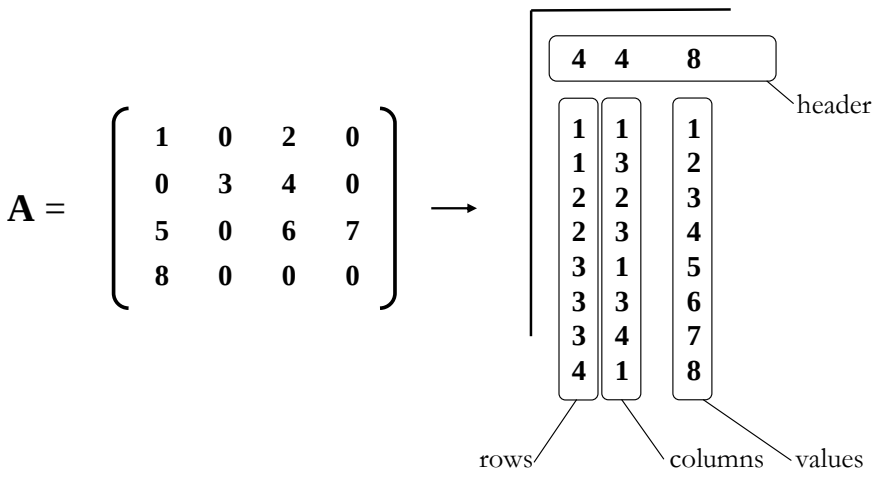


Figure 4.1: Scheme of the Coordinate Storage format

4.2 Compressed Column Storage / Harwell-Boeing format

The compressed column (CCS), in literature referred also as Harwell-Boeing format, and row (see 4.3) storage formats are general formats − they make no assumptions about the sparsity structure of the matrix, and they don't store any unnecessary elements. On the other hand, they are not very efficient, needing an indirect addressing step for every single scalar operation in a matrix-vector product.

Assuming we have a non-symmetric sparse matrix A, we create three vectors (Figure 4.2):

- One for floating point numbers (val). The val vector stores the values of the nonzero elements of the matrix A as they are traversed in a column-wise fashion. This elements are stored in the contiguous memory locations.
- Second for integer (row_{ind}). The row_{ind} vector stores the column indexes of the elements in the val vector. That is, if $val(k) = a_{i,j}$, then $row_{ind}(k) = j$.
- Third for integer (col_{ptr}). The col_{ptr} vector stores the locations in the val vector that start a column; that is, if $val(k) = a_{i,j}$, then $col_{ptr}(i) \leq k < col_{ptr}(i+1)$. By convention, we define $col_{ptr}(n+1) = nnz+1$, where nnz is the number of non-zeros in the matrix A.

The CCS format for this matrix A in Figure 4.2 is then specified by the arrays val, row_{ind}, col_{ptr}.

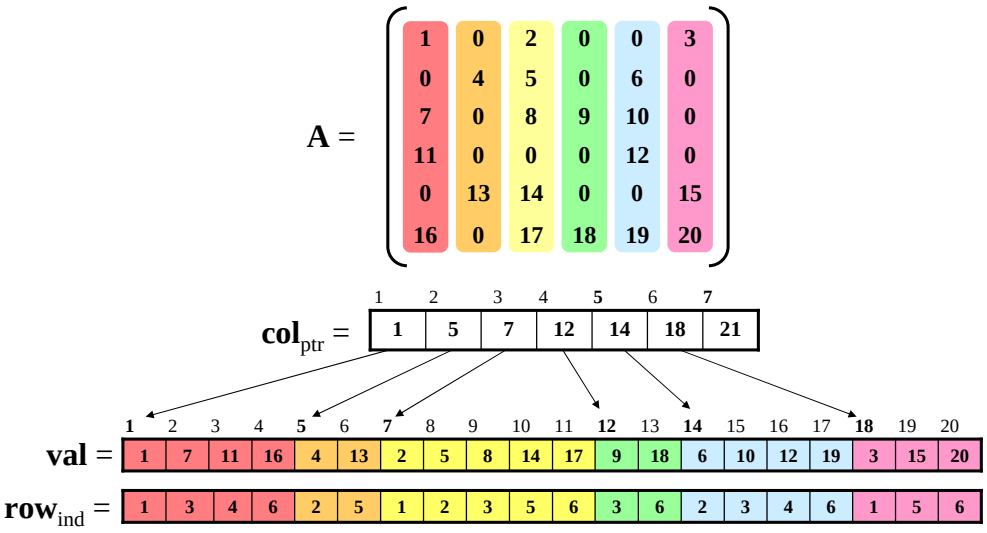


Figure 4.2: Scheme of the Compressed Column Storage format

The storage savings for this approach is significant. Instead of storing n^2 elements, we need only $2nnz + n + 1$ storage locations.

4.3 Compressed Row Storage

The Compressed Row Storage (CRS) is analogous to CCS. Instead of column-wise fashion, the non-zero entries are traversed in the row-wise fashion.

If we want to convert matrix A in Compressed Column format into Compressed Row format we in fact need to construct a traversed matrix A^T as:

$$CCS(A) = CRS(A^T). \tag{4.1}$$

The CRS format is specified, similarly like the CCS format, by the 3 arrays: val, col_{ind}, row_{ptr}, where col_{ind} stores the column indices of each non-zero, and row_{ptr} stores the index of the elements in val which start a row of A.

4.4 Block Compressed Row Storage

If the sparse matrix A is composed of square dense blocks of non-zeros in some regular pattern, we can modify the CRS (or CCS) format to exploit such block patterns. Block matrices typically arise from the discretization of partial differential equations (PDE) in which there are several *degrees of freedom* associated with a grid point. We then partition the matrix in small blocks with a size equal to the number of degrees of freedom and treat each block as a dense matrix, even though it may have some zeros.

Assume that n_b is the dimension of each block and $nnzb$ is the number of nonzero blocks. Similar to the CRS format, we require three arrays for the Block Compressed Row Storage (BCRS) format:

- a rectangular array for floating point numbers: $val(1..nnzb, 1..n_b, 1..n_b)$ which stores the non-zero blocks in (block) row-wise fashion,
- an integer array, $col_{ind}(1..nnzb)$ which stores the actual column indices in the original matrix A of the $(1,1)$ elements of the nonzero blocks, and
- a pointer array, $row_{blk}(1..(n_d+1))$ whose entries point to the beginning of each block row in val and col_{ind}. Note that the block dimension n_d is defined by $n_d = n/n_b$. The savings in storage locations and reduction in time spent doing indirect addressing for BCRS over CRS can be significant for matrices with a large n_b.

The total storage needed is $nnz = nnzb \times n_b^2$.

4.5 Compressed Diagonal Storage

This storage is especially useful for *banded* matrices with constant number non-zero entries on the diagonal − on each non-zero diagonal. Banded matrix is if all matrix elements are zero outside a diagonally bordered band whose range is determined by constants $k1$ and $k2$:

$$a_{i,j} = 0 \quad \text{if} \quad j < i - k_1 \quad \text{or} \quad j > i + k_2; \quad k_1, k_2 \geq 0. \tag{4.2}$$

The quantities $k1$ and $k2$ are the left and right *half-bandwidth*, respectively. The bandwidth of the matrix is $k1 + k2 + 1$ (in other words, the smallest number of adjacent diagonals to which the non-zero elements are confined).

We can allocate for the matrix A an array $val(1..n, -k1..k2)$. We can pack the nonzero elements in such a way as to make the matrix-vector product more efficient. Let's assume a nonsymmetric matrix A (see Figure 4.3). In CDS format, we store this matrix A in an array of dimension $(6, -1:1)$ using the mapping

$$val(i, j) = a_{i,i+j} \tag{4.3}$$

Obviously, this storage makes assumptions about the matrix sparsity structure, but is useful for special applications matrices − particularly, if the matrix arises from a finite element or finite difference discretization [BDD+00].

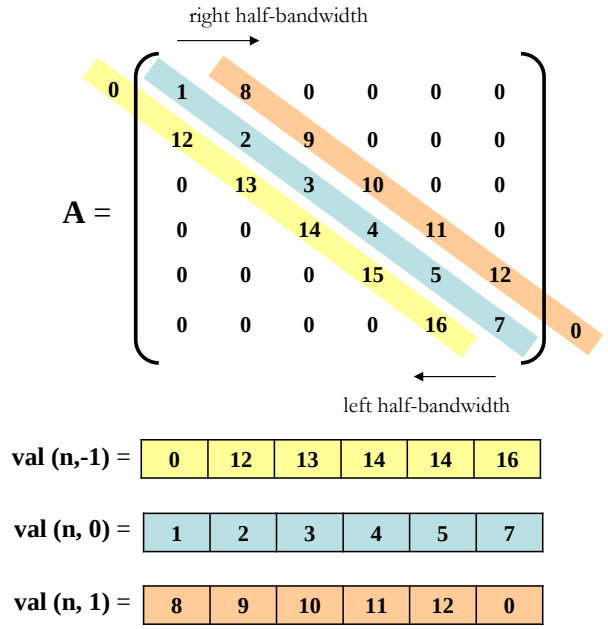


Figure 4.3: Compressed Diagonal Storage format.

4.6 Padded ITPACK Storage

Padded ITPACK format is a representation that is easy to stream. This representation consists of two rectangular arrays, one real and one integer. Both arrays are of size at least n by max_nz, where n is the number of linear equations and max_nz is the maximum number of non-zeros in the matrix.

Each row in Anz will contain the non-zeros of the respective row in the full matrix A, and the corresponding row in $cIdx$ will contain its respective column numbers. For example, the matrix A (see figure 4.4) would be represented in the Anz and $cIdx$ arrays.

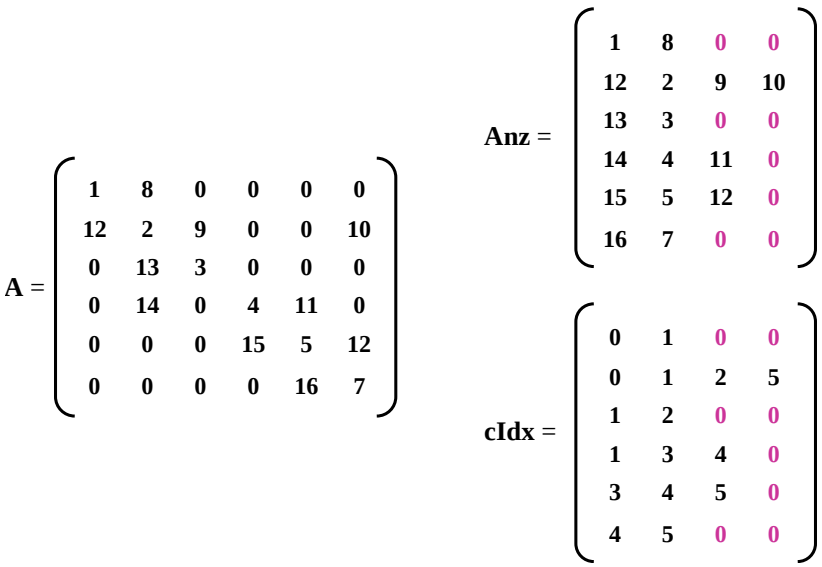


Figure 4.4: ITPACK sparse matrix format. Padding zeros in the matrices Anz and $cIdx$ are highlighted.

Note that:

1. A parameter nnz_per_row (maximum nonzero entries per row) should be provided, to obtain the matrix optimal savings.

2. If a row in the matrix A has fewer than nnz_per_row non-zeros, the corresponding rows in Anz and $cIdx$ should be padded with zeros.

3. All non-zero matrix entries must be present even if the matrix is symmetric. It does not suffice to store just the upper or lower triangle of the matrix A.

Instead of storing n^2 elements, we need $2(n \times nnz\_per\_row)$ storage locations. It is clear that the padding zeros (depending on nnz_per_row) in this structure may be a disadvantage, especially if the bandwidth of the matrix differs strongly.

4.7 Jagged Diagonal Storage

Jagged Diagonal Storage is the Padded ITPACK Storage (see Section 4.6) with a row compression applied − analogically as in the Compressed Row Storage format (see Section 4.3).

Transposed Jagged Diagonal Storage as similar to JDS, but the ITPACK representation is constructed in the column-wise fashion instead. Then the row compression is applied again.

The jagged diagonal storage (JDS) format can be useful for the implementation of iterative methods on parallel and vector processors. Like the CDS format, it gives a vector length of essentially the same size as the matrix. It is more space-efficient than CDS at the cost of a gather/scatter operation.

4.8 Skyline

The skyline storage format (SKS) is important for the *direct* sparse solvers. This format is specified by two arrays: values and pointers:

- a real array of *values*. For a lower triangular matrix it contains the set of elements from each row of the matrix starting from the first non-zero element to and including the diagonal element. For an upper triangular matrix it contains the set of elements from each column of the matrix starting with the first non-zero element down to and including the diagonal element. Encountered zero elements are included in the sets.

- an integer array of *pointers* with dimension $(m + 1)$, where m is the number of rows for lower triangle (columns for the upper triangle).

If the matrix is symmetric, we only store its lower or upper triangular part (Figure 4.5). For a nonsymmetric skyline matrix, we store the lower triangular elements in SKS format, and store the upper triangular elements in a column-oriented SKS format (transpose stored in row-wise SKS format). These two separated substructures can be linked in a variety of ways. One approach is to store each row of the lower triangular part and each column of the upper triangular part contiguously into the array *val*. An additional pointer is then needed to determine where the diagonal elements, which separate the lower triangular elements from the upper triangular elements, are located.

4.9 Diagonal Storage

Diagonal Storage is dedicated to store just the diagonal entries in the main diagonal of the matrix. The storage saving is n entries instead of n^2. This storage format is

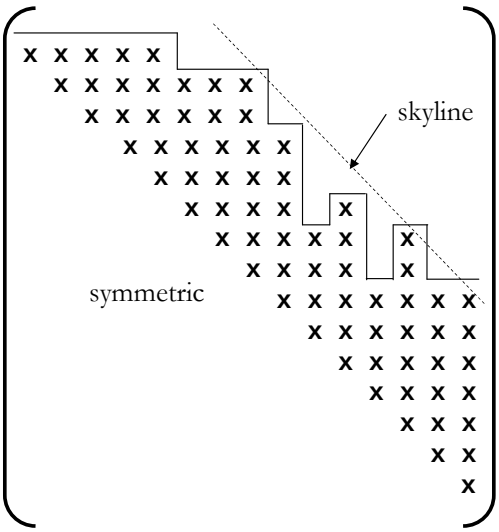


Figure 4.5: Profile of a nonsymmetric skyline or variable-band matrix.

particularly useful in many applications where the matrix arises from a finite element or finite difference discretization. However, this storage makes strong assumptions about the sparsity structure and provides no useful scheme for our work.

4.10 Summary

The research of [SU07] concludes, that between five storage formats including Coordinate Storage, Compressed Row Storage, Compressed Column Storage, Jagged Diagonal Storage and Transposed Jagged Diagonal Storage (TJDS), the TJDS achieve the high performance on distributed memory parallel architecture.

However, for stream programming model and the GPGPU platforms used for implementation of our work, we considered as the best the Padded ITPACK Storage format (see Section 4.6) as the inner sparse matrix representation, which gives a good compromise between performance, accessing data and storage savings. For more about the storage format we will discuss in Chapter 7.

Note that Coordinate (Matrix Market) and Compressed Column (Harwell-Boeing) format are usually used for exchanging between the scientists and researchers in the internet matrices repository like Matrix Market [Mar07] or University of Florida Sparse Matrix Collection [Dav07]. Many matrix format converters like [BG] exists too.

5 GPU Programming

Graphics Processing Units (GPU) were primarily invented exclusively for tasks of computer graphics. Such tasks are mainly producing a visualization on the display: that involves geometry creation and transformation, mapping the textures, producing the lights, clipping, coloring, interpolations, rasterization, etc. The computer graphics hardware was hard-wired to these specific tasks.

In the last decade GPUs has developed significantly in the terms of programmability and performance − so we can state that the current GPUs are *truly programmable* as summarizes [OLG+07]. As a result we can use them for any general computation and use their power of parallelism and wide throughput to achieve better system performance. The motivation for using GPU for general-purpose calculations illustrates table 5.

	CPU	**GPU**
Model	3.0 Ghz Intel Core2 Quad	NVIDIA GeForce 8800GTX
Computation	96 GFlops	330 GFlops
Memory bandwidth	21 GB/s	55.2 GB/s
Price	1100$	550$

Table 5.1: Comparison of current CPU and GPU -model as a motivation for GPGPU. Note that the performance values at the CPU are *peak* values. On the GPU the performance values are *typical*.

Currently, it is a big challenge in migrating CPU algorithms to the GPU. There are hundreds of non-graphic applications ported for the GPUs from many scientific fields − including scientific computing, geometric computing, raytracing, audio signal processing, digital image processing, bioinformatics, etc.

In practice this parallel algorithm in combination with the GPU achieves better performance than the optimized non-parallel CPU algorithms.

5.1 GPU Architecture

The fundamental part of any graphics processing hardware is its pipeline. We will describe the concept of the older pipeline (we will call it here "traditional" pipeline) consisting of three major stages − Vertex, Rasterization and Fragment stage. This traditional concept was used from the late 1990s, in the year 2000, the first chip with added programmability of Vertex and Fragment Stages was introduced and this concept was used up to recent years. However, the latest development aims towards unification of the pipeline stages into one, to provide more general programming model and to unify and balance the processing resources. New model of so-called Unified Shaders pipeline architecture was first introduced in ATI Xenos chip for the Xbox 360 and GeForce 8800 chip for PC and brings more general approach to program GPUs. More about Unified Shaders pipeline is mentioned in Section 5.1.2.

5.1.1 Traditional GPU Pipeline

The graphics pipeline is well suited to the rendering process because it allows the GPU to function as a stream processor since all processed elements (vertices and fragments) can be thought of as independent. This allows all stages of the pipeline to be used simultaneously for different vertices or fragments as they work their way through the pipe. In addition

to pipelining vertices and fragments, their independence allows graphics processors to use parallel processing units to process multiple vertices or fragments in a single stage of the pipeline at the same time.

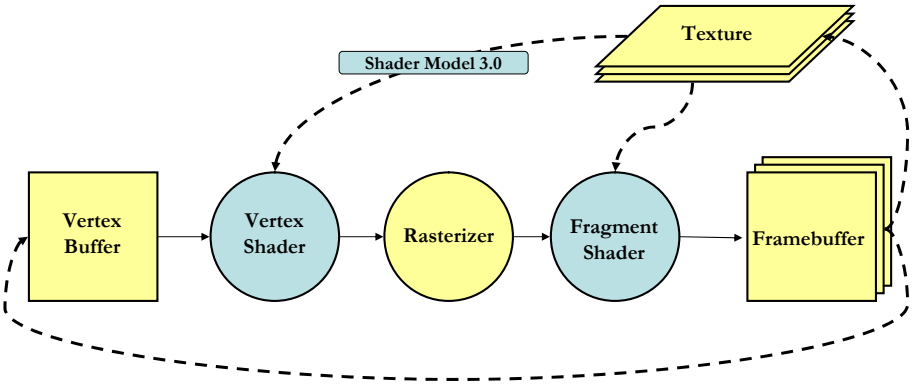


Figure 5.1: The traditional pipeline – consists of programmable stages (vertex, fragment). The arrows show the direction of data flow, also the possibility of reading the data back into the pipeline. By the introduction of the Shader Model 3.0 the data could be loaded from Framebuffer directly to the Vertex Shader.

Vertex Processing Stage The vertex processing stage of the rendering pipeline is responsible for taking an input set of vertex attributes (such as model-space positions, vertex normals, texture coordinates, etc.) and producing a set of attributes suitable for clipping and rasterization (such as homogeneous clip-space position, vertex lighting results, texture coordinates, and more). Naturally, performance in this stage is a function of the work done per vertex, along with the number of vertices being processed.

Rasterization Stage In the rasterization stage the vertices coming from vertex transformation stage are assembled into geometry primitives such as points, lines or triangles. Here, these primitives are transformed into fragments. The data are turned into a raster format with reference to its original position from original 3D vector representation. Unlike Vertex Processing stage, this stage is not programmable.

Fragment Processing Stage The Fragment Processing Stage (also called Pixel Shader) is the last stage of the GPU pipeline and processes fragments coming out from the rasterization stage. The fragment shader can discard any fragment due to depth or alpha conditions, however it can not generate any new fragments. It is responsible for applying textures, fog, blending; the color of each rendered pixel is being defined and then the pixel is stored in the framebuffer. From here, the final picture is displayed on the screen. For purposes of GPGPU computations we need to keep the results in the pipeline to apply another computations. To achieve this we use a feature called *render to texture* (Section 5.4.2), which redirects the output of the shader to some other location than the regular framebuffer. The location is a provided texture – and therefore we keep the results in the pipeline. After we read the data back to the main memory to get the result.

5.1.2 GPU Pipeline with Unified Shaders

The latest step in the evolution from hardwired pipeline to flexible computational fabric is the introduction of unified shaders. Instead of separate custom processors for vertex shaders, rasterization shaders, and fragment shaders, a unified shader architecture provides one large grid of data-parallel floating-point processors general enough to run all these shader workloads. As Figure 5.2 shows, vertices, triangles, and pixels recirculate through the grid rather than flowing through a pipeline with stages of fixed width.

This configuration leads to better overall utilization because demand for the various shaders varies greatly between applications, and indeed even within a single frame of one application. For example, a computer game might begin an image by using large triangles to draw the sky and distant terrain. This quickly saturates the pixel shaders in a traditional pipeline, while leaving the vertex shaders mostly idle. One millisecond later, the game might use highly detailed geometry to draw intricate characters and objects. This behavior will saturate the vertex shaders and leave the pixel shaders mostly idle.

For example, a GeForce 8800 might use 90 percent of its 128 processors as pixel shaders and 10 percent as vertex shaders while drawing the sky, then reverse that ratio when drawing a distant character's geometry. The result is a flexible parallel architecture that improves GPU utilization and provides much greater flexibility for game designers as well as for GPGPU programmers [LH07].

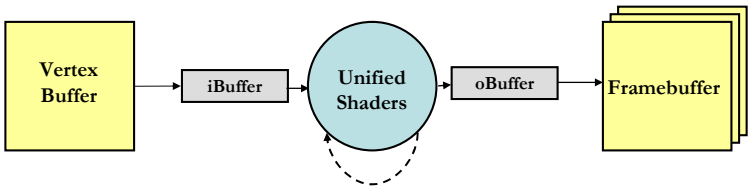


Figure 5.2: Loop oriented processing in pipeline with unified shaders.

5.1.3 DirectX 10 and Shader Model 4.0

The current major generation of GPUs are widely supporting Microsofts Direct3D 10 API, part of Microsofts DirectX multimedia APIs, and appeared in early 2007. The major hardware and software changes that characterize these GPUs are:

- The geometry shader, a new programmable unit to the pipeline, that is placed after the vertex shader (see Figure 5.3). The input to the GS is an entire primitive. The major difference between the GS and the previous vertex/fragment shaders is that it can output anywhere from 0 to many primitives. This ability to procedurally create new elements is useful in both graphics tasks and general-purpose tasks.

- GPGPU application developers have long requested more flexible operations on memory buffers. Operations such as *render-to-texture* and *render-to-vertex-array* have partially met these requests, however one new operation in DX10 hardware is the *stream output*, allowing the output of the geometry shader to be directly stored into a memory buffer.

- The new shader model 4.0 associated with DX10 hardware unifies the basic instruction set between the programmable shader units (though each programmable shader

still has stage-specific specializations). Along with increases in a variety of shader limits, such as instruction count, register space, and render targets, shader hardware now supports 32-bit integers. This integer capability is expected to both enhance current GPGPU applications (particularly in memory address calculations) as well as enable new ones [OLG+07].

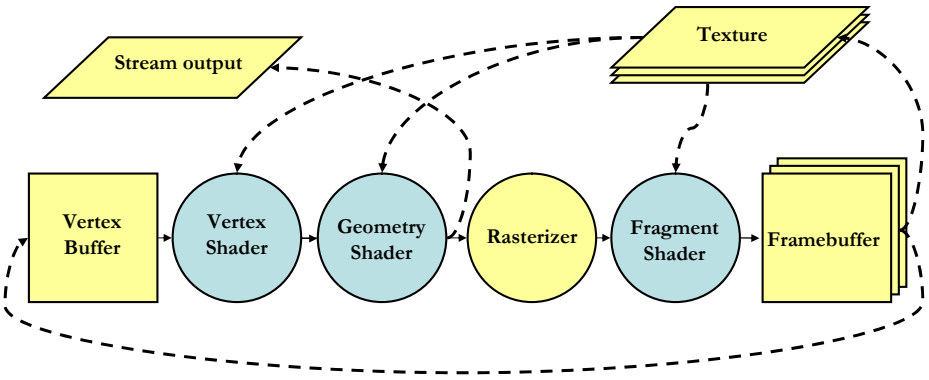


Figure 5.3: The pipeline with new Geometry Shader programmable stage.

5.2 Stream programming model

Programming model is an approach how to look at data and instructions in a program. Stream programming model is the natural approach to program modern GPUs.

In the stream programming model, the data primitive is a stream, an ordered set of data of an arbitrary datatype (vertices, fragments, etc.). Operations in the stream programming model are expressed as operations on entire streams. These operations include stream loads and stores from memory, stream transfers, and computation in the form of kernels [Owe02]. The most significant drawback of the parallelism in the stream processing is the unknown order of processing of the data, which requires the data to be independent of each other. Consequently, any data dependency requires additional synchronization, which costs some processing delay.

Let's summarize how stream programming model exposes parallelism:

- **Instruction level parallelism** Kernels typically perform a complex computation of tens to hundreds of operations on each element in a data stream. Many of those operations can be evaluated in parallel.

- **Data level parallelism** Kernels that operate on an entire stream of elements can operate on several elements at the same time. For example, the transformation of each point in the stream could be calculated in parallel. Even calculations that have dependencies between adjacent elements of a stream can often be rewritten to exploit data parallelism.

- **Task level parallelism** A stream of data could be divided in half and each half could be transformed on separate stream nodes. Or, if another kernel was necessary after the transformation, it could be run on a second stream node performing the transformation. This work partitioning between stream nodes is possible and straightforward because the stream programming model makes the communication between kernels explicit.

The publication of [Owe02] summarizes several common characteristics of typical computer graphics applications, a class of applications collectively termed "media applications":

High Computation Rate Many media applications require many billions of arithmetic operations per second to achieve real-time performance. Modern graphics processors advertise computation rates of over one trillion operations per second.

High Computation to Memory Ratio Media applications tend to achieve a high computation to memory ratio: they perform tens to hundreds of arithmetic operations for each necessary memory reference.

Producer-Consumer Locality with Little Global Data Reuse The typical data reference pattern in media applications requires a single read and write per global data element. Little global reuse means that traditional caches are largely ineffective in these applications. Intermediate results are usually produced at the end of a computation stage and consumed at the beginning of the next stage. Graphics applications exhibit this producer-consumer locality: the graphics pipeline is divided into tasks which are assembled into a feed-forward pipeline, with the output of one stage sent directly to the next.

Parallelism Media applications exhibit parallelism at the instruction, data, and task levels. Efficient implementations of graphics workloads certainly take advantage of the parallel nature of the graphics pipeline.

5.3 Stream programming model on the GPU

At this point we can observe the common features in stream programming model and the graphics (or media) application tasks. Let's have a look at the table 5.3 which illustrates mapping of the stream programming model to the GPUs:

Stream model	**GPU model**
Data Stream	Texture
Kernel program	Shader program

Table 5.2: Mapping the stream programming concept to the GPU.

Textures On the GPU, the data represents the textures. A texture is a chunk of memory where each cell has one to four subcomponents for storing the full information of color, normal vector, position, or any other property. The physical allocation is optimized for accessing cells with two indexes, rather than by one.

Kernels Kernels perform computation on entire streams, usually by applying a function to each element of the stream in sequence. Kernels operate on one or more streams as inputs and produce one or more streams as outputs. On the GPU, kernel program is called Shader program or Fragment program and this program runs on Vertex or Fragment Shader. It is programmed with a dedicated languages, so-called high-level shading languages. There are following shading languages: Cg programming language from NVIDIA [NVIa], DirectX High-Level Shader Language (HLSL) from Microsoft and OpenGL shading language (GLSL) [Ope].

Flow Control Since the GPUs are strongly focused on the performance typical program flow control structures were in the older GPUs not allowed. Currently, the programmability of the GPUs is becoming essential and the manufacturers of the GPUs provide their

cards with flow control techniques almost similar to those on the CPU Currently, the programmability of the GPUs is becoming essential and the manufacturers of the GPUs provide their cards with flow control techniques almost similar to those on the CPU. The article [OLG+07] describes the three basic implementations of data-parallel branching in use on current GPUs: predication, MIMD branching, and SIMD branching. Architectures that support only predication do not have true data-dependent branch instructions. Instead, the GPU evaluates both sides of the branch and then discards one of the results based on the value of the Boolean branch condition. The disadvantage of predication is that evaluating both sides of the branch can be costly, but not all current GPUs have true data-dependent branching support. The compiler for high-level shading languages like Cg or the OpenGL Shading Language automatically generates predicated assembly language instructions if the target GPU supports only predication for flow control.

At this point we will introduce a short step-by-step guide for mapping a problem to the GPU. We already have the *stream analysis* and we are ready to map it to the GPU:

1. We divide the data into independent chunks that can be processed in the data-parallelism.

2. Array = texture. Data are mapped to the textures which invokes the data transfer from the main memory onto the GPU memory. There are mainly two approaches how to access the texture elements − via integer indexes or normalized texture coordinations. Indexes are the absolute way of addressing a point in texture, normalized coordinations are addressing the texture relatively.

3. Kernel program = shader program. We implement kernel program which will process the data defined in step 1. This program runs on programmable chips (probably fragment shader). The inputs for kernel are the textures defined in the step 2.

4. Data processing = drawing. Invoking of the whole process consist of geometry drawing. This could be for instance a rectangle which represents the output array. This process causes loading the kernel program to the programmable processor and data transfer from the main memory to the GPU memory.

5. Rasterizer generates fragment for each pixel.

6. For each fragment from rasterizer, the kernel program from fragment shader is invoked.

7. Output from fragment shader is a pixel and his value (or vector of values). This value could be stored to the framebuffer directly (will be displayed) or to the texture for next passes through the pipeline. This technique is called multipassing.

5.4 GPGPU Programming Techniques

Techniques for the traditional pipeline differ from the programming techniques for the Unified Shaders. Unified Shaders brought a significant change into GPGPU techniques. We can say that if using an the latest graphics hardware with CUDA (Section 5.5.3), we don't need any particular technique anymore. However, this thesis covers mainly the usage of more general GPU programming languages like Brook (see 5.5.2) and Rapidmind (see 5.5.1) that are still strongly bounded to the traditional architecture, so we are obliged to discuss these techniques.

5.4.1 Scatter and Gather

Two fundamental memory operations are write and read. If the write and read operations access memory indirectly, they are called *scatter* and *gather* respectively. A scatter operation looks like the code $d[a] = v$ where the value v is being stored into the data array d at address a. A gather operation is just the opposite of the scatter operation. The code for gather looks like $v = d[a]$. The GPU implementation of gather is essentially a dependent texture fetch operation. A texture fetch from texture d with computed texture coordinates a performs the indirect memory read that defines gather. While gathering data is easily implementable in both shaders, scattering can be achieved only by advanced indirect techniques. Vertex shader can for example adjust the position of vertices, which allows the programmer to define where the data will be stored. For an illustration see Figure 5.4.

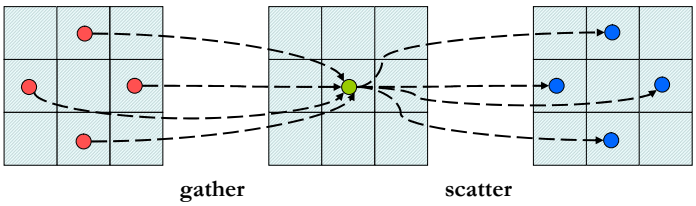


Figure 5.4: Gathering data is possible by the texture look-ups, direct scattering is on traditional GPUs impossible

5.4.2 Render To Texture

The Framebuffer Object extension (FBO) defines a simple interface for drawing to rendering targets other than the framebuffer. These rendering destinations are called framebuffer-attachable images and are bound to one of the standard logical buffers, such as color, depth or stencil buffer. This allows us to access the output of the fragment shader again, to use it for multi-passing through the pipeline

5.4.3 Render To Vertex Buffer

The are two indirect method to render the output from the fragment shader directly into the vertex buffer: The first uses the *pixel buffer object* (PBO), which is an extension to the vertex buffer, but the data are stored in pixel fashion. The second approach is to render the data into a texture and then use the texture as an input parameter to the shader program.

5.5 GPGPU Platforms

Nowadays, the evolution of high-level programming GPU languages and platforms brought two main approaches. First is an effort to develop a general high-level language such as BrookGPU[BFH+] or RapidMind[Rap] that run on different-branded GPUs or multi-core processors. Second approach is development of languages which are bounded to a specific hardware - such as CUDA [NVIb], bounded to NVIDIA hardware, CTM [ATIa] bounded to ATI hardware, which can fully exploit the potential and functionality of the dedicated hardware.

5.5.1 RapidMind Multi-core Software Platform

The commercially developed RapidMind Multi-Core Platform (shortly Rapidmind) aims towards accessing the "mainstream developers" the computing power of the general multi-core systems. That means the API is simple to use and no deep knowledge about parallel processing and underlying architecture is necessary.

Rapidmind is designed to target multiple hardware platforms − enables single threaded applications to access different branded multi-core processors. Currently supported multi-core backends are: x86 Intel/AMD processors, ATI/NVIDIA GPUs, Cell Blade and Playstation3.

The RapidMind programming model was inspired by work on abstract parallel computational models as well as previous languages and notations for parallel computation. Its computational model has also been based in part on the model of stream programming used on GPUs. It uses an SPMD (Single Program Multiple Data) model, not a pure SIMD (Single Instruction Multiple Data) model: program objects may include control flow. Its use of an SPMD execution model enables it to emulate certain forms of task parallelism, even though it is primarily a data-parallel model [McC06].

In Figure 5.5 is the Rapidmind software architecture depicted. Let's describe the three components: API, Platform and Support Modules.

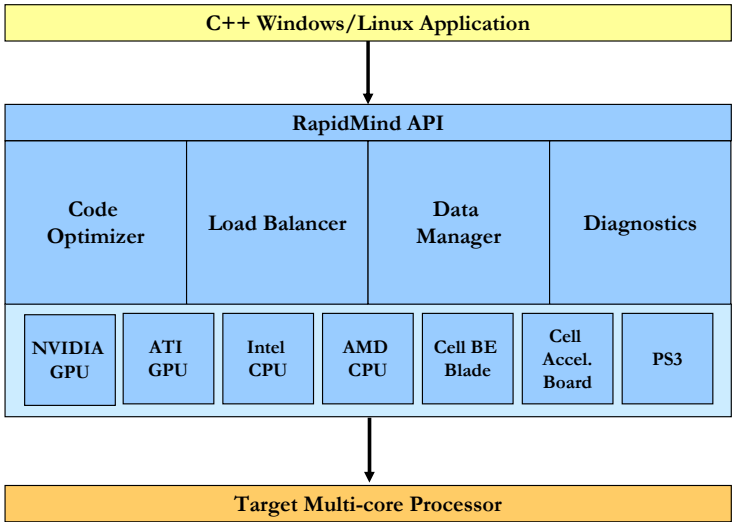


Figure 5.5: The Rapidmind Software Architecture

Rapidmind API The application interface integrates with existing C++ compilers, and requires no new tools or workflow. The interface is based on three main C++ types: Containers for data as $Value$ and $Array$, and a container for operations $Program$. Parallel computations are invoked by applying programs to arrays to create new arrays, or by applying a parallel collective operation that can be parameterized with a program object, such as a reduction.

The $Value < N, T >$ type is an N-tuple. Instances of this type holds N values of type T, where T can be a basic numerical type: single-precision floating point, double precision floating point, or signed and unsigned integers of various lengths. The $Array < D, T >$ type is also a data container. However, it is multidimensional and potentially variable in size. The integer D is the dimensionality and may be 1, 2, or 3; the type T gives the type of the elements. Currently, element types are restricted to instances of the $Value < N, T >$

type.

Before we will introduce the type Program we mention the *retained* and *intermediate* mode. Every operation has two implementations. In immediate mode, when you ask for two numbers to be added, the computation is performed and the result returned at that time. At retained mode, all the operations switch from performing a computation to recording a computation. All the operations you specify in a particular instance of retained mode are collected together into a sequence of operations forming a program. In retained mode it looks like you are writing operations in C++, but those operations are really compiled at runtime into a program by the compiler portion of Rapidmind that targets several GPU and CPU backends.

The fundamental type is *program object* that stores a sequence of operations. These operations are specified by switching to retained mode, which is indicated by using the BEGIN keyword macro. Normally the system operates in immediate mode. In immediate mode, operations on a value tuple take place when specified as in a normal matrix-vector library: the computation will be performed on the same processor as the host program and the numerical result will be stored in the output value tuple. While in retained mode, operations on value tuples are not executed; they are instead symbolically evaluated and stored in the program object. Retained mode is exited using the END macro, which closes the program object and marks it as being ready for compilation. A completed program object can then be used for computation. Program objects are compiled using a dynamic compiler that can take advantage of the low-level features of the target machine. An example of program object is in Figure 5.6.

```
float f;
float a[w*h*3], b[w*h*3];

for (int y = 0; y < h; y++)
  for (int x = 0; x < w; x++)
    for (int e = 0; e < 3; e++){
      a[(y * w + x) * 3 + e] +=
      b[(y * w + x) * 3 + e] * f;
}
```

→

```
Value1f f;
Array<2,Value3f> a(w,h), b(w,h);

Program mul_add = BEGIN {
  In<Value3f> i1, i2;
  Out<Value3f> o;

  o = i1 + i2 * f;
}END;

a = mul_add(a, b);
```

Figure 5.6: Parallelizing a sequential code to Rapidmind program object

Rapidmind Platform The Rapidmind Platform consists of Code Optimizer that analyzes and optimizes computations to remove overhead. The Load Balancer plans and synchronizes work to keep all cores fully utilized, Data Manager reduces data bottlenecks and Diagnostics detects and reports performance bottlenecks.

Support Modules Since this thesis is focused on the GPU, we will describe only the GPU support module. The GPU implementation is based on top of the OpenGL API, and works on both NVIDIA and ATI GPUs. Data is stored in textures and render-to-texture and multiple render target extensions are automatically used when the output of a program is assigned to an array. Several optimizations specific to the OpenGL API are used to obtain good performance. For instance, it is relatively expensive to allocate and deallocate textures under OpenGL, so the system automatically reuses previously allocated textures whenever possible.

Just-in time compilation Converting program definition into OpenGL codes at runtime. First, it decides which backend should be responsible for the program executions-

backend form the connection between the Rapidmind platform and a particular piece of target hardware, e.g. OpenGL-based GPUs or a fallback backend. Once suitable backend has been chosen it is asked to execute the program under the given conditions: The first time this is done generally causes the program object to be compiled for that particular backend. Once a program has been compiled, it is not recompiled.

5.5.2 BrookGPU

BrookGPU (shortly Brook) is an extension of standard C and is designed to incorporate the ideas of data parallel computing and arithmetic intensity into a familiar, efficient language. The general computational model, referred to as *streaming* (see Section 5.3), provides two main benefits over traditional conventional languages: *Data Parallelism* that allows the programmer to specify how to perform the same operations in parallel on different data and *arithmetic intensity* that encourages programmers to specify operations on data which minimize global communication and maximize localized computation [BFH+].

Brook API The application interface of Brook integrates with standard ANSI C. The interface is based on two main types *stream* and *kernel*. A *stream* is a new data type addition which represents a collection of data which can be operated on in parallel. Each stream is made up of stream elements. The shape of the stream refers to the dimensionality of the stream.

Kernels are special functions that operate on streams. A kernel is a parallel function applied to every element of the input streams. Calling a kernel function on a set of input streams performs an implicit for loop over the elements of streams, invoking the body of the kernel for each element. With GPU hosted kernels, the streams are pushed into video memory and the kernels compiled down to fragment programs that "render" from them. An illustration of kernel program is in Figure 5.7.

Brook provides support for parallel reductions on streams. A reduction is an operation from one stream to another stream of smaller dimensions or to a single value. The operation can be defined by a single, two-input operator that is both associative and commutative. Other features like Multi-dimensional Reductions, Stream Shape functions and Iterator Streams are mentioned in [BFH+].

```
float f;
float a[w*h*3], b[w*h*3];

for (int y = 0; y < h; y++)
  for (int x = 0; x < w; x++)
    for (int e = 0; e < 3; e++){
      a[(y * w + x) * 3 + e] +=
      b[(y * w + x) * 3 + e] * f;
}
```

→

```
float f;
float3  a<w,h>;
float3  b<w,h>;

kernel void mul_add
  (float a<>,float b<>,float f, out float c<>) {
  c = a + b * f;
}

mul_add(a, b, f, a);
```

Figure 5.7: Parallelizing a sequential code to Brook kernel program

Brook Platform The Brook compilation and runtime architecture consists of a two components (see Figure 5.8). BRCC is the Brook compiler is a source to source meta-compiler which translates Brook source files into C++ files. The compiler converts Brook primitives into legal C++ syntax with the help of the Brook Runtime library.

The BRT is an architecture independent software layer which implements the backend support of the Brook primitives for particular hardware. The BRT is a class library

which presents a generic interface for the compiler to use. The implementation of the class methods are customized for each hardware supported by the system. The backend implementation is choosen at runtime based on the hardware available on the system or at request of the user. The backends include: DirectX, OpenGL and C++ reference.

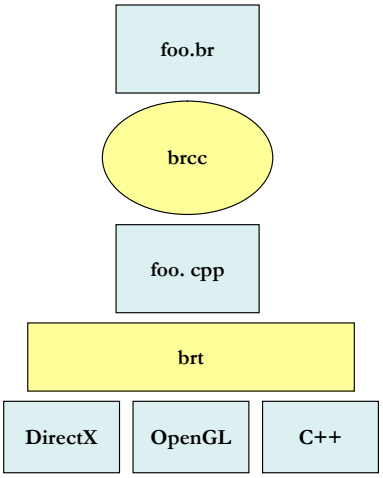


Figure 5.8: The Brook layered architecture

There are certain constraints in programming in Brook: Scatter operation is not hardware supported and there are not allowed global memory reads or writes within a kernel program.

We should mention also Brook+[ATIb], an implementation by AMD of the Brook GPU specifications on AMD's compute abstraction layer with some enhancements. The architecture is similar, however kernels, written in C, are compiled to AMDs IL code for the GPU or C code for the CPU. AMD IL is a portable *immediate language* that sits between high level languages and the assembly code of the underlying graphics hardware. Unlike the former Brook, the scatter function is hardware implemented already.

5.5.3 Compute Unified Device Architecture (CUDA)

Compute Unified Device Architecture is a new hardware and software solution for data-parallel computing from NVIDIA. The difference from the older generation GPUs is that for issuing and managing computations on the GPU as a data-parallel computing device, there is no need of mapping them to a graphics API. The new simplified architecture is introduced − a pipeline with unified shaders (see Section 5.1.2).

In CUDA, the GPU is viewed as a compute device suitable for parallel data applications. It has its own device random access memory and may run a very high number of threads in parallel. Threads are grouped in blocks and many blocks may run in a grid of blocks. Such structured sets of threads may be launched on a kernel of code, processing the data stored in the device memory. Threads of the same block share data through fast shared on-chip memory and can be synchronized through synchronization points. An important aspect of CUDA programming is the management of the memory spaces that have different characteristics and performances.

The proper choice of the memory to be used in each kernel depends on many factors such as the speed, the amount needed, and the operations to be performed on the stored data. An important restriction is the limited size of shared memory, which is the only available read-write cache. Unlike the CPU programming model, here the programmer needs to explicitly copy data from the global memory to the cache (shared memory) and

backwards. But this new architecture allows the access to memory in a really general way, so both scatter and gather operations are available. The unavailability of scatter was one of the major limitations of OpenGL when applied to GPGPU applications. The main point in approaching CUDA is that the overall performance of the applications dramatically depends on the management of the memory, which is much more complex than in the CPUs.

A part of CUDA is also a BLAS library implementation, but only for dense matrices (so-called *CuBLAS* library).

6 Iterative methods in the parallel environment

In Chapter 3, we described iteration methods form the sequential point of view. After introducing the GPU programming model in Chapter 5, we can take a deeper look at these methods in the parallel programming environment. We will go through each method and discuss the parallel implementation, constraints, also a new approaches that have emerged in the parallel environment of the GPU.

6.1 Jacobi Method

Since the Jacobi method treats all the equations independently, method suits perfectly to the stream programming model, where all the n equations could be resolved concurrently. The parallel version of Jacobi method is in Figure 3.3.1.

Algorithm 5 Parallel Jacobi method

Choose an initial guess $x^{(0)}$ to the solution x
repeat
  for $\forall i, j$ **do in parallel**
  $\sigma = \sigma + \mathbf{A}x_j^{(k-1)}$
  $x_i^{(k)} = \frac{(b_i - \sigma)}{a_{ii}}$
  do in parallel end
until convergence is reached
$x \approx x^{(k)}$

Complexity Jacobi method requires one matrix-vector product, two vector updates and one inner product.

6.2 Gauss-Seidel Method

The updates in the Gauss-Seidel method cannot be done simultaneously as in the Jacobi method (see 3.3.2) since each iteration depends on the new as well as on the old values. There is the typical data dependency which makes this method not suitable for stream processing on the GPU. However, modern GPU program control flow allows us to avoid at least the inner loop of the algorithm 3.3.2 where the new x is being computed. Note that this improvement is time-expensive, since needs branching within the kernel program and is not eliminating the main outer program loop. As already mentioned in each iteration method uses some "new" values and therefore converges faster than Jacobi method.

If the coefficient matrix A is a particular sparse matrix, i.e result of a Partial Differential Equation discretization as discussed in Chapter 2, a workaround scheme which removes the data-dependency could be applied; this is the variant of the Gauss-Seidel Method called Red-Black Gauss-Seidel Method, see Section 6.3.

Other methods for parallel Gauss-Seidel algorithm exists too. In [KRF94] is described the implementation and performance of an efficient parallel Gauss-Seidel algorithm for irregular, sparse matrices. A two-part matrix ordering technique is used - first to partition the matrix into block-diagonal-bordered form, then using multi-coloring technique (see Section 6.3).

Another publication (see [ABHT]) describes *Processor Block Gauss-Seidel* method a true parallel Gauss-Seidel implementation, which is possible to combine with Jacobi's method.

Algorithm 6 Parallel Gauss-Seidel Method

Choose an initial guess $x^{(0)}$ to the solution x
repeat
  for $i = 1$ to n **do**
    for $\forall i, j$ **do in parallel**
    $\sigma \Leftarrow \sigma + a_{ij}x_j^{(k)}$ **for** $j = 1$ to $(i-1)$
    $\sigma \Leftarrow \sigma + a_{ij}x_j^{(k-1)}$ **for** $j = (i+1)$ to n
    $x_i^{(k+1)} = \frac{(b_i - \sigma)}{a_{ii}}$
    do in parallel end
  end for
until convergence is reached
$x \approx x^{(k+1)}$

Complexity Each iteration of the Gauss-Seidel method requires n sequential passes each consisting of one matrix-vector product, two vector updates and one inner product. Note that n is the dimension of the coefficient matrix A.

6.3 Red-Black Gauss-Seidel Method

The Red-Black Gauss-Seidel Method comes from the idea of multi-coloring, or graph coloring. This approach is used in the context of relaxation techniques. The 2-color case is called *red-black* ordering.

The problem addressed by multi-coloring is to determine a coloring of the nodes of the adjacency graph of a matrix such that any two adjacent nodes have different colors. For a 2-dimensional finite difference grid, this can be achieved with two colors. Assume that the unknowns are labeled by listing the red unknowns first together, followed by the black ones. The new labeling of the unknowns is shown in figure 6.1.

Since the red nodes are not coupled with other red nodes and, similarly, the black nodes are not coupled with other black nodes, the system that results from this reordering will have the structure:

$$\begin{pmatrix} D_1 & F \\ E & D_2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

For more about multicoloring applied for accelerating Gauss-Seidel method see [Saa00] and [KRF94].

Algorithm 7 Red-Black Gauss-Seidel Method

repeat
  $\forall$ **red points** relax();
  $\forall$ **black points** relax();
until $x \approx x^{(k)}$

6.4 Conjugate Gradient Method

The Conjugate Gradient method, ass seen on algorithm 3.4.1, does not need any significant change for the parallel environment. There is no data-dependency in the algorithm and therefore the CG method is well suited to the paralllel processing. We leave the algorithm without change, but the matrix-vector product, parallel reduction, two vector updates and inner product will be processed in parallel instead.

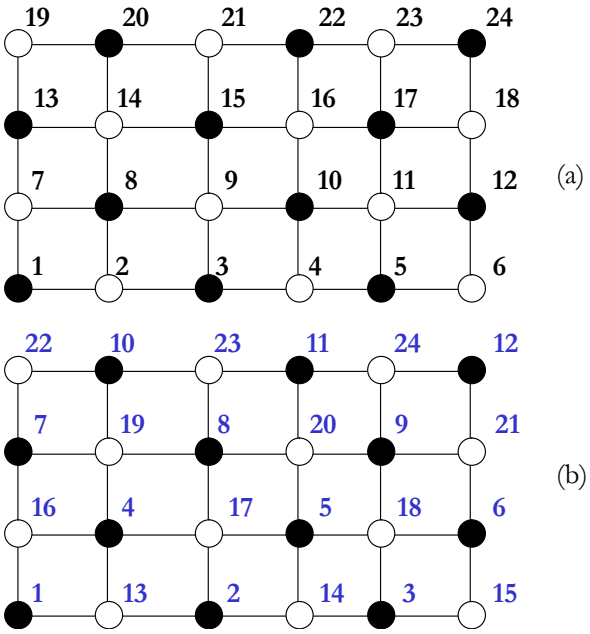


Figure 6.1: Red-Black coloring of 6 × 4 grid. Black numbers indicate the natural numbering, blue numbers the red-black numbering.

Complexity The Conjugate Gradient method involves one matrix-vector product, three vector updates, and two inner products per iteration. Some slight computational variants exist that have the same structure.

6.5 Biconjugate Gradient Method

Similarly as with the CG method, BiCG does not need significant change since there is no data dependency. However, BiCG requires computing a matrix-vector product Ap^k and a transpose product $A^T\tilde{p}^k$. In some applications the latter product may be impossible to perform, for instance if the matrix is not formed explicitly.

Complexity BiCG method is the most time-consuming method in comparison with the other methods. There are following computational kernels involved: two matrix-vector products, four vector updates and two inner product. All the mentioned function will be processed in parallel.

6.6 Summary

Let's discuss here the complexity of mentioned iterative methods. All the iterative methods depend on the convergence criteria, which remains unknown − depends on the coefficient matrix A and the right-hand side vector b. Therefore we can only state the complexity of one iteration. The methods are basically share most of their computational kernels: matrix-vector product, parallel reduction, vector update and inner product and differ in the quantity of these kernels applied. We consider as the most time-consuming kernel the matrix-vector product so we can state that one iteration is approximately that time-expensive as one matrix-vector product for Jacobi, Gauss-Seidel and CG method; the BiCG method performs two matrix-vector products per iteration.

7 Implementation

In this chapter, we describe our experience with the implementation of various iterative methods that we discussed in Chapters 3 and 6: Jacobi method, Gauss-Seidel method, Successive Overrelaxation, Red-Black Gauss-Seidel method, Conjugate Gradient method and BiConjugate Gradient method.

We implemented this methods using GPU hardware abstraction languages (shortly "platforms"): the compiler and runtime Brook (see Section 5.5.2) from the Stanford University and commercially developed platform Rapidmind (see Section 5.5.1). Indeed, we needed also a plain sequential implementation these methods to compare all these approaches. To compare these distinct implementations we performed a measurement and the results are provided in Chapter 8.

7.1 Kernel programs

The fundamental part of the code for a streaming programming language is the kernel program. As said in Chapter 5, kernel program is a program which operates on the entire stream of data (see 5.3).

Both platforms are incorporating the same idea, concept − stream computing (data parallel computing), although they differ in their software architecture how to achieve that. Further we observe that each platform has its own constraints which we will mention further in text.

Basic Linear Algebra Subprograms (BLAS) is a standard library to perform basic linear algebra operations such as vector and matrix multiplication. Our implementation of the iterative methods require only a few functions from this library. These functions belong to vector-vector operations of BLAS, namely BLAS Level 1: the vector update and the inner product. Also the function that performs a sparse matrix-vector product that belongs to matrix-vector operations (BLAS Level 2).

The sparse matrix-vector product requires a suitable sparse matrix data structure and an associated kernel program to execute the multiply. The inner product computation requires a sum-reduction. We implemented all the mentioned functions as the kernel programs to expose parallelism and achieve better system performance.

7.2 Sparse Matrix Data Structure

First, we need to mention the data structure to store the sparse matrix. In Chapter 4, we described various sparse matrix storage schemes. As the good compromise we considered, for the inner representation, the Padded ITPACK Storage format (see Section 4.6), which gives a good compromise between performance, accessing data and storage savings. Further, both platforms can access matrix coefficient efficiently − further described in Section 7.3. As the outer representation of the sparse matrix we have chosen the Harwell-Boeing format since it is a popular format within the researchers and scientists and could be obtained many testing instances from the real-world applications on the internet resources (see [Mar07] or [Dav07]).

7.3 Matrix-vector product

Let's describe on the first place, the most time-consuming kernel program that our implementation needs − a multiplication of matrix-vector. Assume we have matrix A and vector x. In the sequential programming environment it involves an inner product of every

row of matrix A and the vector v.

As we use the ITPACK representation, the sparse matrix is stored in two arrays (streams): Anz, where the non-zero entries are stored and $cIdx$, where the column pointer of corresponding value in Anz are stored. What we need to do is first to map the vector x as a matrix and then perform a matrix-matrix multiply; as we mentioned before, the kernel program works with data as the data streams of the same size; this is achieved easily, because both platforms are performing this operation automatically. In other words, the platforms resize the input/output streams as soon as they get as parameters streams of different size.

Second step is a gather operation (Section 5.4.1); we need to get the the right entries from the temporary matrix, composed of vectors x as we described in first step. This is done analogically like in the sequential environment, can be written like: $v = d[a]$, where the value v is being retrieved from the data array d, from address a; the difference is that all the variables are streams of the same size instead of single variables.

Note, that this operation of matrix-vector product looks straightforward, because we have taken advantage of the ITPACK sparse matrix representation which makes the sparse matrix easy to stream. The entire operation is depicted in the Figure 7.1, where all the steps are highlighted. The last step is to perform a sum-reduction over the rows that is not shown.

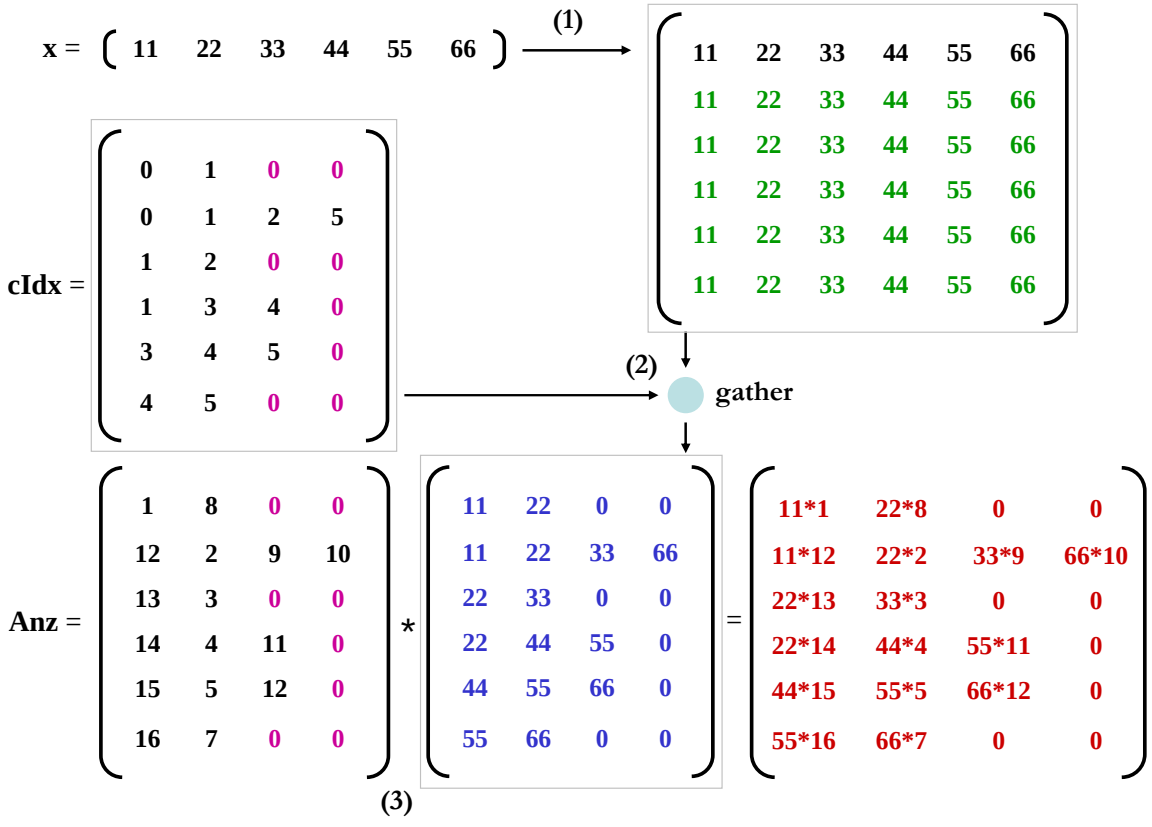


Figure 7.1: Scheme sparse matrix-vector multiplication with the matrix stored in Padded ITPACK format. Step (1): extends vector x to the dimensions of the matrix A. Step (2): Gather proper values. Step (3) Perform multiplication of corresponding elements.

7.4 Vector update

Vector is updated when all the vector members are updated in the same way. This is fundamental computational kernel of any stream programming language, so the implementation is trivial. In our implementation the function performing vector-update (or SAXPY) is called *scaleAdd*.

Following other kernel programs that perform an update of the vector members are used: *add, subtract, mult, square, divide, scale*. The functionality of these kernel programs is straightforward so we leave them without further description.

7.5 Inner product

Inner product involves an update of all vector members and a sum-reduction over the entire vector. Reduction is performed by a reduction operators which apply a binary associative operator to all elements of a vector returning the result $r = v1 \circ v2 \circ \cdots \circ v_n$. At this point the GPU takes advantage of the fact that addition is commutative, no particular order is required. Typically, stream programming language are providing support for this operation as well as the graphics API and the hardware, therefore this kernel is considered as less time consuming. Figure 7.2 illustrates the parallel approach at the stream sum-reducing.

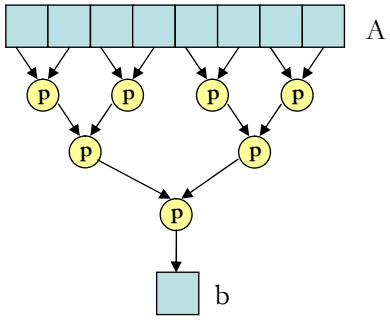


Figure 7.2: Parallel sum-reduction.

The sum-reduction operation is supported by both platforms, although Rapidmind does not support natively the reduction of multi-dimensional arrays - only one dimensional. Since we need to sum-reduce a two-dimensional array sometimes, a workaround in the Rapidmind code has been implemented. However, it has been observed that this workaround causes a timing penalty.

The workaround illustrates Algorithm 8. It requires a cycle in order to construct the sum-reduced array items from the "sliced" vectors. The function *slice*, "cuts" vector given as argument into a new one of the dimension given by the rest of arguments. Since we are accessing arrays that is packed in the ITPACK format we need the constant nnz_per_row. The function *sum* then performs the parallel reduction and we have one member of the desired 2D sum-reduction array.

Brook supports well both reductions and therefore no workarounds were needed.

7.6 Summary

In this section we will summarize the experience with the implementation of iterative methods using particular functions and methods described above. Generally, we achieved

Algorithm 8 Workaround for 2D array sum-reduction

for $i = 0$ to $SIZE\_M$ **do**
  $temp \Leftarrow$ **slice**$(product, i \cdot nnz\_per\_row, (i+1) \cdot nnz\_per\_row - 1)$;
  $tempSum \Leftarrow$ **sum**$(temp)$;
  $reduced[i] \Leftarrow tempSum$;
end for

to implement the majority of mentioned methods, except of those that were considered as non-efficient using the platforms or the platform API was not well-suited for those methods. We successfully implemented Jacobi method, Gauss-Seidel method, Successive Overrelaxation, Conjugate Gradient method, BiConjugate Gradient method.

We attempted to implement the Red-Black Gauss-Seidel method, but later the effort has been dropped. The method requires additional non-trivial operations. For instance, gathering the "surrounding" values from the matrix (as described in Section 6.3) which was achieved using a special stencil matrix. Further, method needs rearrange the coefficients to avoid the data dependency. This would be done by additional matrix-matrix multiplication that takes some extra time. Due to this facts and the fact that Red-Black Gauss-Seidel method is suited just for a matrix with certain sparsity (i.e. Poisson discretization matrix) that brings no general method for solving linear system of equations, we dropped this implementation and focused towards the more general methods.

The method of Successive Overrelaxation uses a parameter ω (see 3.3.3), which influences the rate of convergence of the iterates to the solution. Since the study of the relationship between the convergence rate and ω was beyond the scope of this thesis, we used a default value $\omega = 1.25$.

Our parallel implementation for the Brook platform did not bring any constraint, since all the necessary function were available in the platform API and no time-consuming workarounds were needed. The Rapidmind implementation brought a constraint at the sum-reduction over 2D array. However, as we got informed from the official development sources of Rapidmind, the support of the "native" sum-reduction over multi-dimensional fields is on the development roadmap.

General implementation notes Since both platforms are using C/C++ syntax we could reuse many of the utility functions code, i.e. parsing input file, convert sparse matrix representation, etc. All the three implementation take as input a file with Harwell-Boeing sparse matrix format (CCR) and the right-hand side vector b, if provided.

For BiCG, we need a transposition of the matrix A. The function which transposes matrix is provided, but does not try to exploit the stream computing capabilities of the GPGPU platforms. Although the problem of matrix transposition is generally a case study for parallel algorithms, reshuffling data from CSR to CCR format (or transposition, see 4.1) is not well suited for stream programming model on our GPGPU platforms. Second, the focus of this thesis is not optimizing such utility function for GPGPU platforms rather than exploring and measuring the "real" performance of solving a sparse linear system.

Stopping Criteria The output of an iterative method is a sequence of vectors $x^{(i)}$ converging (or not) and satisfying the system $Ax = b$. A method must decide when to stop. In our implementation, two stopping criteria must be provided by the user:

1. The number of maximum iterations (max_it), for the case that the method fails to converge.

2. The real number ε which measures how small the user wants the residual $r^{(i)} = Ax^{(i)} - b$. In each iteration, a function which checks whether the new residual vector is reaching the convergence threshold is applied. For example, choosing $\varepsilon = 10^{-6}$ means that the user considers the entries of A and b to have errors within the range $\pm 10^{-6}A$ and $\pm 10^{-6}b$, respectively.

As all the algorithms of the iterative methods can be structured similarly, we propose a common structure which share all the iterative methods (see 9). Note that $convergeThresh = \varepsilon^2 \cdot r_{init}$, where ε is the user submitted desired error measure and r_{init} is the initial residual norm.

Algorithm 9 Common iterative method template

$i = 0$;
repeat
  $i++$;
  Compute the approximate solution $x^{(i)}$.
  Compute the residual $r^{(i)} = \mathbf{A}x^{(i)} - b$
  Compute $||r^{(i)}||$ and $||x^{(i)}||$
until (it < it_max) && ($||r^{(i)}|| > convergeThresh$)

Floating point precision Currently, Brook provides just single floating point precision (32-bit) as it is bounded with the older GPUs. In the newer Brook+ is also double precision supported, however the implementation was not tested on Brook+. Rapidmind supports both standard single and double precision (64-bit). For the testing purposes was on both platforms just the single floating point precision used.

8 Results

In this chapter, we will describe the results of our experiments. We have two parallel implementations of the same algorithm for different software platforms and a plain sequential algorithm so we can compare the distinct implementations as well as the efficiency of the platforms and at the end we will be able to emphasize what is the most efficient approach to perform the iterative methods for solving linear equations for which platform.

8.1 Testing Environment

For our work, which was focused for using GPGPU platforms, we needed a GPU supported by each Brook[1] and Rapidmind[2]. We used the NVIDIA GeForce 8600 GPU[3]. The work was therefore tested only on NVIDIA GPUs and it is very likely that the code runs on another GPUs (ATI, different GeForce Series) since both Brook and Rapidmind support each.

As the testing system was used a laptop[4] computer with the operating system Microsoft Windows XP[5]. Note that the CPU has two executing units and the algorithm is running only in one thread. All the testing programs were developed at the Microsoft Visual Studio[6] development platform. Further, Brook requires for its compilation and runtime graphics API OpenGL[7], DirectX[8] and the shader program compiler Cg[9].

8.2 Testing Matrices

For testing my implementation of iterative methods I used matrices from the Matrix Market repository [Mar07] and Collection of University of Florida [Dav07], where matrices from variety of real-world applications can be found. This databases provide matrices in Harwell-Boeing exchange format (see Section 4.2) that we used as the input format. I have chosen set of matrices with different properties to test my implementation: symmetric positive definite, unsymmetric positive definite, diagonally dominant, not diagonally dominant matrices and random sparse matrices too. Further, I have chosen matrices with different number of non-zero elements − to observe the relationship between performance and problem size.

8.3 Testing Methodics

The benchmark measures the "clean" time, just the pure algorithm time. All the benchmarks were executed multiple times and as the results the average has been taken. In the benchmarking of the platforms performance the transfer time between CPU-GPU memory, respectively, was measured too. Since the DirectX backend runtime for Brook has slightly better performance than OpenGL backend, the DirectX backend was used exclusively.

[1] Brook v5.0

[2] RapidMind Platform v2.1

[3] NVIDIA GeForce 8600M GS, Memory 512MB, 16 Stream Processors, Core Clock 600MHz, Shader Clock 1200MHz, Memory Clock 700MHz

[4] CPU Intel Core Duo 2 T5550 @ 1.830GHz, Cache, Procesor x86 Family 6 Model 15 Stepping 13 GenuineIntel 1828 Mhz, 2GB RAM

[5] Microsoft Windows XP Professional, 5.1.2600, Service Pack 2, 2600

[6] Microsoft Visual Studio 2005 Version 8.0.50727.42 (RTM.050727-4200), Microsoft Visual C++ 2005 77626-009-0000007-41314

[7] OpenGL Utility Toolkit 3.7

[8] DirectX 9.0 SDK, April 2007

[9] NVIDIA Cg 2.0

8.4 Performance comparison

In this section we will show the performance timing between sequential and parallel implementation of the iterative methods in graphs. These graphs show the time needed for solving the given linear system. The linear system is referred by a name which originates from the application this matrix arises from. We start listing the graphs according to the problem size. Note that the Conjugate Gradient method is some time not listed since with test problem instance the method does not converged. First we proceed with the three methods − Jacobi, CG and BiCG which had the runtime in the similar scale, later we will discuss the remaining Gauss-Seidel method.

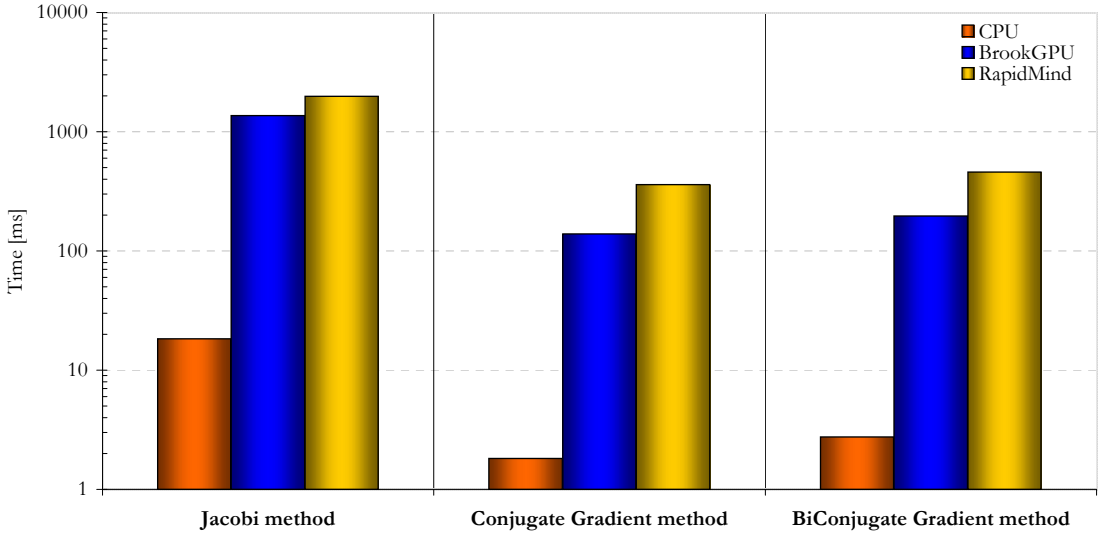


Figure 8.1: BCSSTM22: problem size $\approx 10^2$.

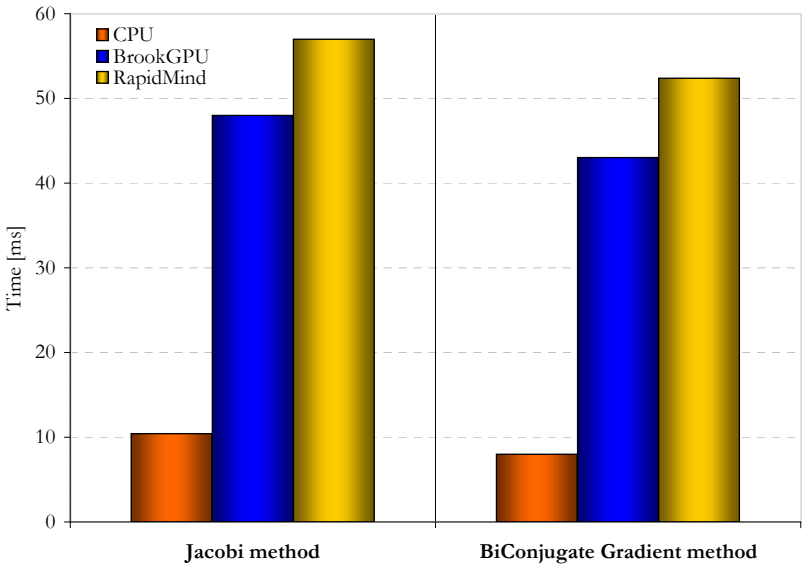


Figure 8.2: E05R0000: problem size ≈ 5000.

The Figure 8.1 represents the smallest problem instance $\approx 10^2$ non-zero entries. We observe that the CPU runtime achieves the best results. The CG and BiCG methods are

faster than Jacobi method. The cost per iteration is lower for Jacobi method, but the convergence rate is slower − the method needs more iterations and therefore delivers the result slower.

In Figure 8.2 is a bigger instance depicted − has over 5000 non-zero entries. The sequential implementation delivers still the fastest result. The parallel implementations stand similarly like in the previous configuration (8.1).

Figure 8.3 illustrates problem size about 10^4 non-zeros. Unlike the two previous graphs, we can see that in this situation the distance between sequential and parallel implementation become to be "closer", however CPU runtime is still the fastest one.

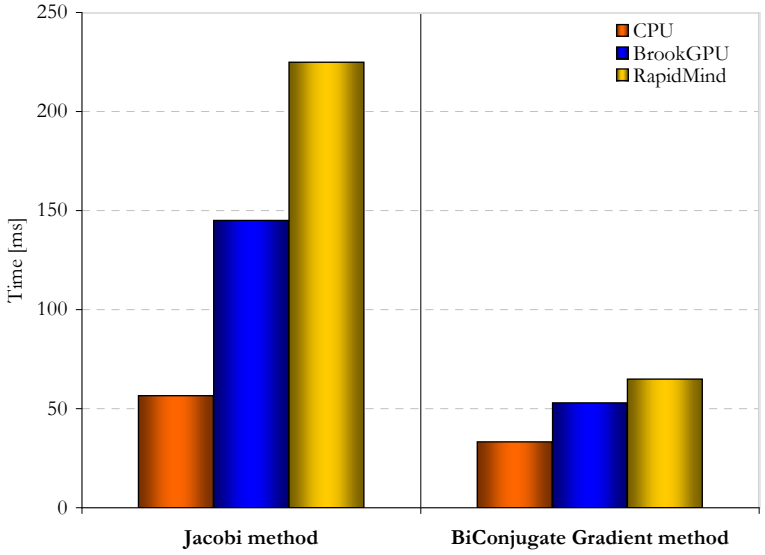


Figure 8.3: ADD32: problem size 10^4.

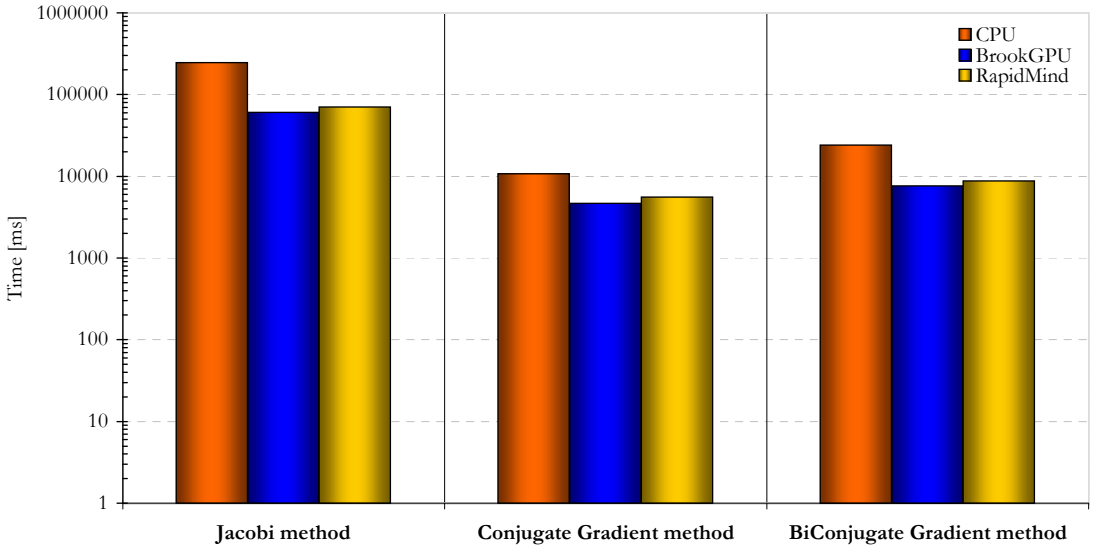


Figure 8.4: E40R0000: problem size 10^6.

In the Figure 8.4 we can observe an instance with 10^6 processsed elements where finally the parallel implementation outperformed the sequential one. Similar situation is depicted on graph 8.5, where is problem size $6{\cdot}10^6$ involved and the parallel implementation performed better.

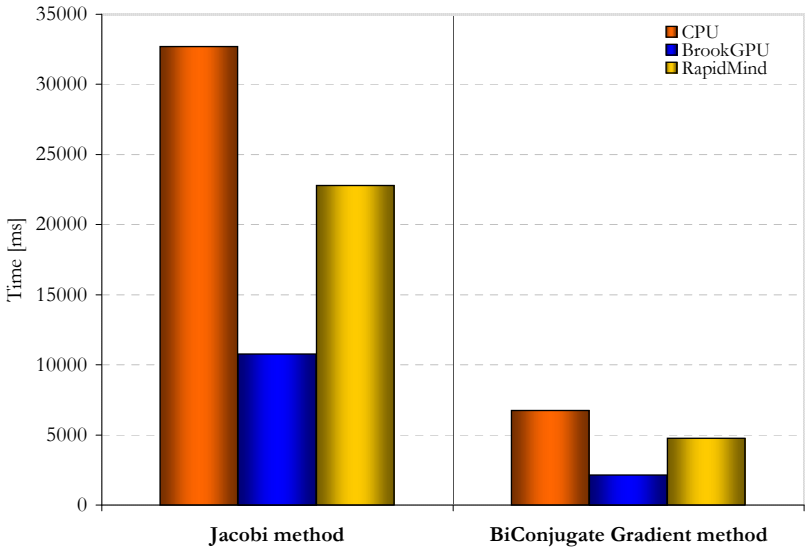


Figure 8.5: MEMPLUS: problem size 6000000+.

We summarize the results for Jacobi, CG and BiCG method in the terms of given platform. Following situations were observed: The small problem instances, $10^2 - 10^4$ elements, the CPU outperformed both parallel implementation. However, the break-through configuration was when the number of processing non-zeros were about 10^6. Both platforms outperformed the sequential implementation for this configuration. Further, the Brook implementation was faster than Rapidmind. In the terms of the methods applied, the Conjugate Gradient and BiConjugate Gradient method performed better than the Jacobi method since CG and BiCG has faster convergence.

The general Gauss-Seidel method is not well suited for parallel processing as discussed in Chapter 6. However, we measured the performance and compared it to the sequential solution too. Since the performance did not reach results like the other methods did, we will mention two graphs so far, where some slight change could be observed − see Figure 8.6. As the parallel algorithm is forced to work sequentially (except of the special kernel program described in Chapter 7) the method is gaining only minor any speedup so far. Note, that we did not measure the SOR method since the method has similar properties as the Gauss-Seidel method.

We compare the speedup to the sequential algorithm. The Figure 8.7 shows the speedup comparison, for the generally best performing method in all the mentioned problem instances. As we compared many different problems, sometimes the Conjugate Gradient method failed to converge since the involved matrix was not symmetric. Therefore we took as the representative method for exposing the speedup the BiConjugate Gradient method, which performed better than Jacobi's and we can observe a $3\times$ speedup for the instance of size 10^6 processed elements.

8.5 CPU-GPU data transfer

The time to transfer the data from CPU to GPU, respectively was measured. For better idea about the transfer speed, let's mention the benchmark timings. The upload rate was about 200 MB/s ($GPU \rightarrow CPU$), the download rate ($CPU \rightarrow GPU$) about 350 MB/s. The separate timings during the testing instances runtime were performed and supported the measurement above. We can conclude that for the iterative method application the

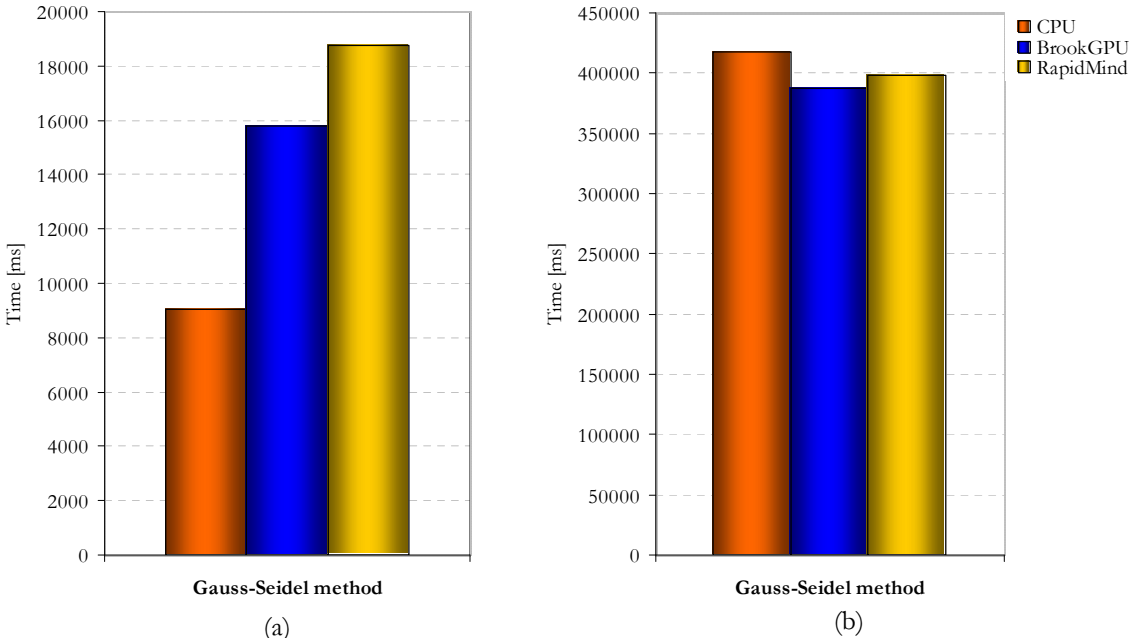


Figure 8.6: Gauss-Seidel method: (a) ADD32, problem size 10^4;(b) e40r0000, problem size 10^6

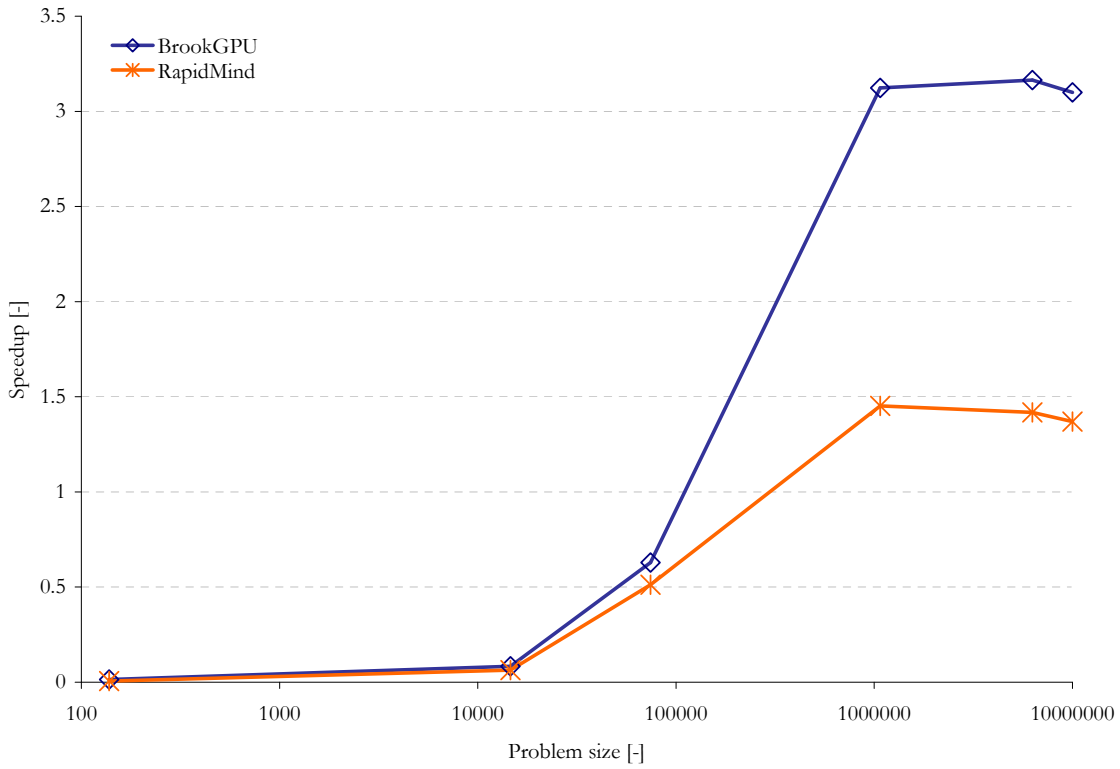


Figure 8.7: Speedup of the parallel implementation

transfer time is generally minor compared to the solving time.

8.6 Summary

Here, we will review and justify the results. While analyzing the results, we must consider the platform architecture, the processes within, the application layers (i.e. thresholds of the graphics APIs) as well as the hardware limits. Note also, that our testing graphics hardware belongs to the newer generation, however it is not a cutting edge hardware. Our testing hardware owns 16 stream processor compared to the 128 processors on a cutting edge hardware[10].

Generally, the sequential algorithm performs better for the smaller instances ($10^2 - 10^4$ elements). The parallel implementation, especially the "platforms" software architecture, their initial setup cause an overhead which does not payoff when processing the small instances. When working with the bigger instances, processing large number of elements, the overhead caused by the platform is minor compared to the runtime. It has been observed in our experiments that the large number of elements means approximately 10^6 processed entries. The break-through speedup is estimated to be around $8 \cdot 10^5$ processed elements.

The Rapidmind platform performed worse than Brook; the architecture of Rapidmind evaluates some of the kernel code in just-in-time mode and the platform supports different brands multi-core processor − there are more abstraction layers than in the Brook; A significant penalty is causing also our "workaround" of the sum-reduction of 2D array described in 8.

The Brook performs better than Rapidmind. In our Brook implementation there was no workaround needed. Further, the architecture of Brook is closely bounded to the hardware since it generate directly the given shader code.

However, the bottleneck observed on both platforms, when working with larger instances (10^7), is caused by the texture size limit of the graphics APIs[11] used, and the algorithms which map the streams to the textures. The maximum texture size is 2048^2 that corresponds to the number of processed elements where a boundary in the speedup curve is observed (Figure 8.7).

[10] i.e. NVIDIA GeForce 8800 Ultra

[11] DirectX 9 for Brook, OpenGL for Rapidmind.

9 Conclusion

In the first part of this thesis, we presented the methods for solving linear systems, their theoretical convergence and the outline of their sequential algorithm. Next we discussed various common sparse storage formats and their pros and cons.

We introduced the modern GPUs; their computational performance as the key motivation fact, their architecture and the possibility to program them. We introduced the stream programming concept and model, put it into the context of GPUs and compared these two approaches.

After we mentioned all above, we discussed the iterative methods in the stream programming environment. We focused on the standard methods, also introduced new schemes which expose parallelism.

We implemented the iterative methods for the GPGPU platforms Brook and RapidMind, we provided also a plain sequential implementation. After, we elected a set of testing matrices from Matrix Market database, performed a benchmarking test for linear system with different properties, various dimensions, and settings of the solver (ε, max_it).

Several observations were made. Processing small problem size (up to 10^4 elements) causes an overhead of both the parallel platforms used and makes the parallel implementation performance many times slower than the sequential one. While processing larger instance (10^6 elements) the parallel algorithms are gaining a speedup $3\times$ for Brook and $1.5\times$ for Rapidmind compared to the sequential implementation and the overhead caused by the platforms becomes to be minor.

We also encountered a probable limitation of our parallel implementation for both platforms and graphics APIs. When processing larger instance (10^7 elements) the speedup is lower than we expected, therefore we can consider the problem size 10^6 elements as the optimal problem size for our parallel implementation. The limitation is caused by the maximum texture size that defines the graphics APIs[1] used. The maximum texture size is 2048^2 that corresponds to the number of processed elements where a boundary in speedup curve is observed.

The time to transfer data to the graphics hardware has been found as minor compared to the solving time for each test case considered.

The general Gauss-Seidel method which gained so far $1.1\times$ speedup compared to the sequential algorithm and the method was found as ineffective for running on the GPU.

9.1 Future Work

The possible directions for the future work on our iterative solvers implementation are following.

- Test our parallel implementation on a cutting egde graphics hardware, which has the unified pipeline architecture i.e. GeForce 8800GTX and ATI brands GPUs. A useful test would be to run the Rapidmind implementation on a different backends (e.g. Cell BE, PS3).
- Structure the processed data differently to avoid the graphics API texture size limitations.
- Implement all the iterative methods for NVIDIA CUDA since currently it is considered as the future of GPGPU programming.

[1] DirectX for Brook, OpenGL for Rapidmind; maximum texture size 2048×2048

- Implement a variant of the Gauss-Seidel method which is better suited for parallel processing: Red-Black Gauss-Seidel or Processor Block Gauss-Seidel which describes [ABHT]

10 References

[ABHT] Mark Adams, Marian Brezina, Jonathan Hu, and Ray Tuminaro. Parallel multigrid smoothing: Polynomial versus gauss-seidel.

[ATIa] ATI. CTM closer to metal. `http://ati.amd.com/companyinfo/researcher/Documents.html`.

[ATIb] ATI AMD. Brook+. `http://ati.amd.com/technology/streamcomputing/`.

[BBC+94] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[BCL07] Luc Buatois, Guillaume Caumon, and Bruno Lévy. *Concurrent Number Cruncher: An Efficient Sparse Linear Solver on the GPU*. Nancy Université, France, Nancy Université, France, 2007.

[BDD+00] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst. *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. SIAM, Philadelphia, 2000.

[BFGS05] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schroeder. *Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid*. CALTECH, 2005.

[BFH+] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Pat Hanrahan, Mike Houston, and Kayvon Fatahalian. Brook for GPUs compiler and runtime implementation of the brook stream program language for modern graphics hardware. `http://graphics.stanford.edu/projects/brookgpu`.

[BG] Berkeley Benchmarking and Optimization Group. BeBOP sparse matrix converter. `http://bebop.cs.berkeley.edu/smc`.

[Dav07] Tim Davis. University of florida sparse matrix collection, 2007. `http://www.cise.ufl.edu/research/sparse/matrices`.

[Göd00] Dominik Göddeke. *GPGPU Performance Tuning An illustrated example*. University of Dortmund, Germany, 2000.

[GHM07] James E. Gentle, Wolfgang Härdle, and Yuichi Mori. *1. Computational Statistics: An Introduction*. Springer, 2007.

[Hjo99] Trond Hjorteland. The action variational principle in cosmology, 1999.

[KRF94] D. P. Koester, S. Ranka, and G. C. Fox. *A Parallel Gauss-Seidel Algorithm for Sparse Power Systems Matrices*. School of Computer and Information Science and The Northeast Parallel Architectures Center (NPAC), Syracuse University, Syracuse, NY 13244-4100, 1994.

[LH07] David Luebke and Greg Humphreys. How GPUs work. 2007.

[Mar07] Matrix Market. Matrix market, 2007.
http://math.nist.gov/MatrixMarket.

[McC06] Michael D. McCool. *Data-Parallel Programming on the Cell BE and the GPU using the RapidMind Development Platform.* RapidMind Inc., 470 Weber St N, Waterloo Ontario, Canada, N2L 6J2, 2006.

[MMF] Matrix market format definition.
http://math.nist.gov/MatrixMarket/formats.html.

[NVIa] NVIDIA. Cg shading language.
http://developer.nvidia.com/object/cg_toolkit.html.

[NVIb] NVIDIA. CUDA C language development environment for CUDA-enabled GPUs.
http://www.nvidia.com/object/cuda_home.html.

[OLG+07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[Ope] OpenGL. Opengl shading language.
http://www.opengl.org/documentation/glsl/.

[Owe02] John Douglas Owens. Computer graphics on a stream architecture, 2002.

[Rap] RapidMind. Rapidmind multi-core software platform.
http://www.rapidmind.net.

[Saa00] Yousef Saad. *Iterative methods for sparse linear systems.* SIAM, 2000.

[She94] Jonathan Richard Shewchuck. An introduction to the conjugate gradient method without the agonizing pain. 1994.

[SU07] Rukhsana Shahnaz and Anila Usman. An efficient sparse matrix-vector multiplication on distributed memory parallel computers, 2007.

A Content of the CD

The content of the CD is organized into five main directories.

- `\doc`
  contains some of the referred literature
- `\prog`
  includes the implementation
- `\matrices`
  contains the testing matrices
- `\script`
  contains the testing scripts
- `\thesis`
  contains the pdf version of this thesis

The directory tree of the CD is shown in Figure A.1.

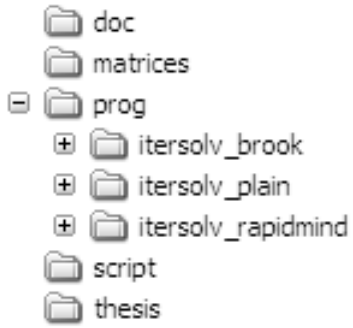


Figure A.1: Directory tree of the included CD.

B Usage of the application

The application provides the command-line interface and several arguments must be provided.

```
SYNOPSIS
        itersolv.exe [method] [file] [it_max] [eps] [size_m] [size_n] [nnz] [verb]
METHOD
        jcbi    Jacobi method
        gs      Gauss-Seidel method
        gssor   Successive Overrelaxation method
        cg      Conjugate Gradient method
        bicg    BiConjugate Gradient method
FILE
        Harwell-Boeing sparse matrix file format
IT_MAX
        maximum number of iterations (in the case method fails to converge)
EPS
        the desired accuracy of the solution.
SIZE_M, SIZE_N
        the dimensions of the matrix
NNZ
        nnz_per_row. Number of non-zero entries per row.
VERB
        0   no debug output.
        1   some debug output.
        2   total debug output.
```