Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft Institute of Applied Mathematics

Implementation of the Deflated Preconditioned Conjugate Gradient Method for Bubbly Flow on the Graphical Processing Unit(GPU)

A thesis submitted to the
Delft Institute of Applied Mathematics
in partial fulfillment of the requirements

for the degree

MASTER OF SCIENCE
in
COMPUTER ENGINEERING

by

Rohit Gupta

Delft, the Netherlands
August 2010

MSc THESIS COMPUTER ENGINEERING

"Implementation of the Deflated Preconditioned Conjugate Gradient Method for Bubbly Flow on the Graphical Processing Unit(GPU)"

ROHIT GUPTA

Delft University of Technology

Daily supervisors

Prof. dr. ir. C. Vuik
Ir. C.W.J. Lemmens

Responsible professor

Prof. dr. ir. H.J. Sips

Other thesis committee member(s)

Dr. ir. M.B. van Gijzen

August 2010

Delft, the Netherlands

Abstract

In this work we have implemented the Iterative Method of Conjugate Gradients with two levels of Preconditioning to solve a System of Linear Equations on Graphical Processing Unit(GPU). This system represents the discretized Pressure equation resulting from the Level Set Method Solution of the Incompressible Navier Stokes Equation used to compute Bubbly Flows. We have tried to explore the problem space with different grid sizes, number of preconditioning blocks and deflation vectors. The results show that when the methods for preconditioning are chosen so that they can exhibit ample parallelism we can achieve considerable performance up to 20 times better than the CPU version. We show in our analysis that we are very close to maximum achievable speedup. We also report on the accuracy of our results and argue that GPUs can be beneficial in solving such problems efficiently.

Acknowledgments

I thank God :). I thank My Family for their blessings and I thank my supervisors for helping me and trusting me to complete this work.

Contents

1 Introduction

Computations of Bubbly flows is the main application for this implementation. Understanding the dynamics and interaction of bubbles and droplets in a large variety of processes in nature, engineering, and industry are crucial for economically and ecologically optimized design. Bubbly flow occur, for example, in chemical reactors, boiling, fuel injectors, coating and volcanic eruptions.

Two phase flows are complicated to simulate, because the geometry of the problem typically varies with time, and the fluids involved have very different material properties. Following from the previous work [Tang, 2008] we consider stationary and time-dependent bubbly flows, where the computational domain is always a unit square or unit cube filled with a fluid to a certain height. The bubbles and droplets in the domain are always chosen such that they are located in a structured way and have equal radius, at the starting time.

Mathematically bubbly flows are modeled using the Navier Stokes equations including boundary and interface conditions, which can be approximated numerically using operator splitting techniques. In these schemes, equations for the velocity and pressure are solved sequentially at each time step. In many popular operator-splitting methods, the pressure correction is formulated implicitly, requiring the solution of a linear system (3) at each time step. This system takes the form of a Poisson equation with discontinuous coefficients (also called the 'pressure(-correction) equation') and Neumann boundary conditions, i.e.,

$$- \bigtriangledown . \left(\frac{1}{\rho(x)} \bigtriangledown p(x) \right) = f(x), x \in \Omega, \tag{1}$$

$$\frac{\partial}{\partial \mathbf{n}} p(x) = g(x), x \in \partial\Omega, \tag{2}$$

where Ω, p, ρ, x and $\mathbf{n}$ denote the computational domain, pressure, density, spatial coordinates, and the unit normal vector to the boundary, $\partial\Omega$, respectively. Right-hand sides f and g follow explicitly from the operator-splitting method, where g is such that mass is conserved, leading to a singular but compatible linear system (3).

In this work we look at the implementation of a numerical solution of a Linear Partial Differential Equation (PDE), resulting from the mathematical modeling of bubbly flows. The PDEs have been discretized through the use of finite differences. A Linear Sytem arises from such a discretization. We are interested in systems of the form,

$$Ax = b, A \in \mathbb{R}^{n \times n}, n \in \mathbb{N} \tag{3}$$

where n is the number of degrees of freedom and is also called the dimension of A. Also A is symmetric positive definite(SPD), i.e.,

$$A = A^T, y^T A y > \ 0 \ \forall \ \mathrm{y} \ \in \mathbb{R} \ \mathrm{y} \ \neq 0. \tag{4}$$

The linear system given by (3) is usually sparse and ill-conditioned. This means that there are few non-zero elements per row of A and also that the condition number $\kappa(A)$ is usually large. Put in other words, the ratio of the largest eigenvalue to the smallest is large and this leads to slow convergence of the Conjugate Gradient Method.

$$\kappa(A) : \ = \frac{\lambda_n}{\lambda_1} \tag{5}$$

where $0 < \lambda_1 \leq \lambda_2 \leq ... \leq \lambda_n$ are eigenvalues of matrix A.

Solving the system (3) by direct methods is also an option but it is usually not memory-wise or computationally efficient. Though these methods are robust and generally applicable but they also tend to be prohibitively expensive. The sparsity of the matrix A

necessitates the use of efficient storage methods and computation with 'iterative methods'. The term 'iterative method' refers to a wide range of techniques that use iterates, or successive approximations to obtain more accurate solutions to a linear system at each iteration step.

Krylov subspace methods, especially the Conjugate Gradient Method is the prominent choice for solving such systems. However the convergence of this method depends heavily on $\kappa(A)$. In order to avoid more and more iterations, as $\kappa(A)$ rises with increasing problem sizes, the matrix A is *preconditioned* to bring down the condition number form $\kappa(A)$ to $\kappa(M^{\frac{-1}{2}}AM^{\frac{1}{2}})$ which is equivalent to $\kappa(M^{-1}A)$. The coefficient matrix A is multiplied by M^{-1}, the preconditioner. The original system (3) then looks like,

$$M^{-1}Ax = M^{-1}b, \tag{6}$$

where M is symmetric and positive definite just like A. M^{-1} is chosen in such a way that the cost of the operation $M^{-1}y$ with a vector y is computationally cheap. However, sometimes preconditioning might also not be enough. In that case we use second level of preconditioning or Deflation in order to reduce $\kappa(A)$. In this work some previous results [Tang, 2008] are used for implementation. The focus is to implement these methods on the Graphical Processing Unit (GPU).

Recently Scientific Computing has largely benefited from the data parallel architecture of graphical processors. Many interesting problems which are computationally intensive are ideally suited to the GPU, especially matrix calculations. It is only intuitive to use them for solution of discretized partial equations. With the advent of the Component Unified Device Architecture (CUDA) paradigm of computing available on NVIDIA GPU devices, it has become easier to write such applications. More time can be spent in exploring the 'What If?' scenarios that are of scientific importance rather than understanding device specifics. We briefly define the problem of Bubbly flows followed by a background on Iterative Methods. Subsequently we introduce the Parallelization of Iterative methods. After introducing some of the Architectural details of the GPU and and a glimpse of the programming techniques we discuss the recent work that has been done on the GPU with respect to solving iteratively the linear systems emerging from discretization of PDEs. Then we present an overview of the Implementation followed by the results from the Numerical Experiments. Further, we analyze our results and we end this report with a conclusion.

2 Problem Definition

Solving the linear system (3), that is a discretization of (1), within an operator splitting approach is a bottleneck in the fluid-flow simulation, since it typically consumes the bulk of the computing time. In order to accelerate the calculation of the solution by the Preconditioned Conjugate Gradient Algorithm with deflation we propose to use the GPU. The challenge lies in optimizing the computation on the GPU in such a way so as to extract the maximum computation throughput it can deliver. This requires exploring Algorithmic enhancements and Architectural provisions to achieve the best possible performance speedup.

2.1 Formulating the Problem for Two-Phase Flow

To solve the Pressure equation's (1) discretization (3) we have to work on a matrix that represents the grid in Figure 72 in the Appendix B.5. This is a standard 5-point Laplacean stencil for a square grid. We use such a matrix for all the grid sizes we conduct our experiments with. We start with Conjugate Gradient and step by step add Preconditioning

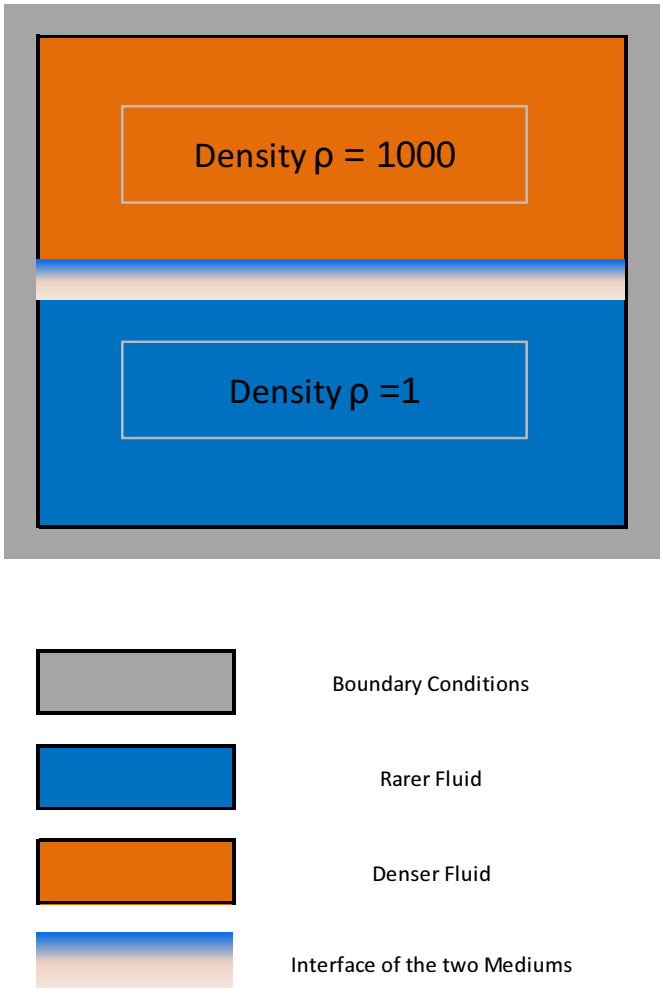

Figure 1: Two phase Flow Computational Model

and Deflation to it. For the 5-point Laplacean matrix we do not see any change in diagonal preconditioning since the coefficients are not having any discontinuities.

However when we consider the matrix for a two phase problem then instead of the smooth coefficients on the diagonals we have big jumps or discontinuities in the matrix. Table B.5.2 in the Appendix B.5 lists a part of the Two Phase Matrix (Density Contrast 1000 : 1). This is attributed to the large difference in the densities of the two fluids we are trying to model. Now we explain how we arrive on this kind of a matrix.

First let us take a look at the picture (Figure 1) of the problem in terms of an interface and two mediums with Neumann Boundary Conditions.

When we discretize the problem, a grid results which has coefficients placed on the 5 diagonals with the jumps appearing at the interface region. We follow a mass conserving approach (as shown in Figure 2) while calculating the coefficients on the interface.

There is some flux that enters horizontally and vertically and some of it leaves a cell. By taking the cell-centered approach (wherein the discretization point is at the center of the cell) and taking into account the contribution of all the flows through that point we arrive on a stencil.

The stencils for the cells which are not on the interfaces are straightforward to calculate.

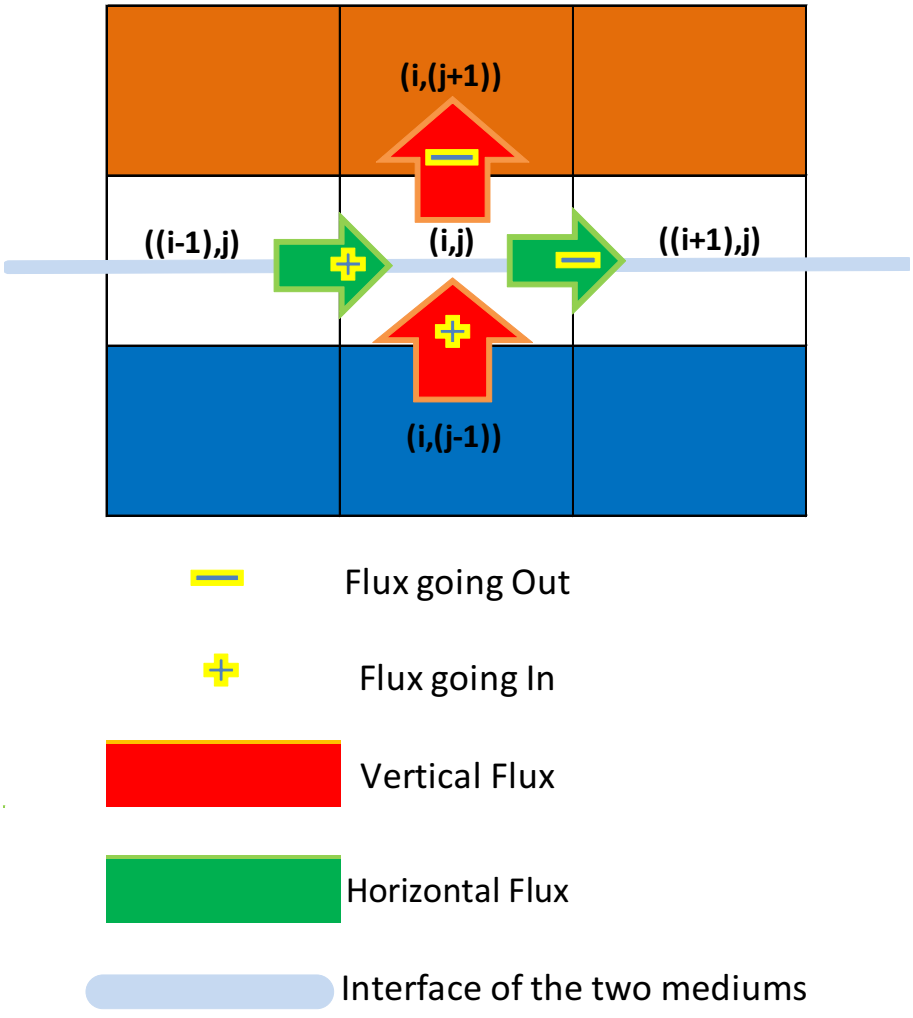

Figure 2: Flux entering and leaving a discretized cell

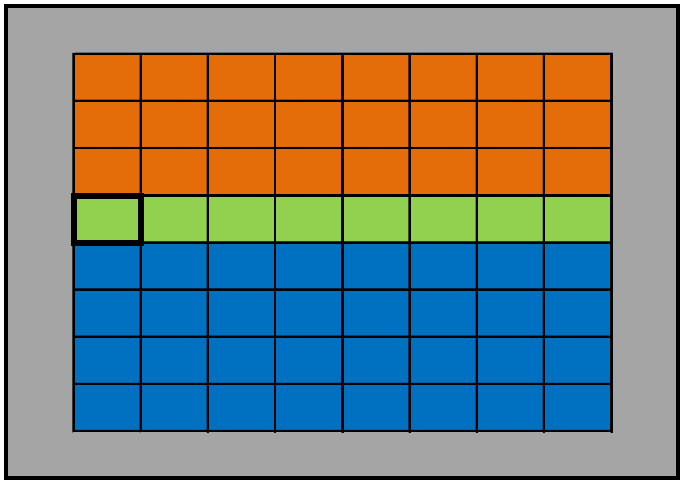

Figure 3: 8×8 grid with two mediums and an interface.

Below we briefly talk about the points on the interface and how we calculated their stencils.

Consider the model grid of dimension 8×8 in which we have a point that lies on the interface and also adjoining the boundary. The cell is highlighted with a thick black border in the Figure 3.

Refer to the Flux picture as shown in Figure 2. For example if we consider the left cell with co-ordinates $((i-1), j)$ to be on the boundary (the gray colored section in Figure 1). Then there is no flux coming in from the horizontal direction however since the cell $((i+1), j)$ is inside (not adjoining the boundary) we have some flux going out. In the vertical direction we have flux coming in from the the point $(i, (j-1))$ and flux going out from the cell $(i, (j+1))$. Keeping these in mind and assuming that the density ratio of the rarer medium (orange) to the denser medium is ϵ. We can write for this point

$$\frac{1}{h}[-P_{i,j-1} - \epsilon P_(i, j+1) - (\frac{1}{2} + \frac{1}{2}\epsilon)P_{i+1,j} + (1\frac{1}{2} + 1\frac{1}{2}\epsilon)P_{i,j}] \tag{7}$$

where h is the dimension of a cell i.e. $\frac{1}{n}$ for a $n \times n$ grid. Further we can get the stencil

$$[-1 \qquad 0 \qquad (1\frac{1}{2} + 1\frac{1}{2}\epsilon) \qquad (-\frac{1}{2} - \frac{1}{2}\epsilon) \qquad -\epsilon]. \tag{8}$$

Deriving stencils in this way the complete matrix (as provided in Table B.5.2 in the Appendix B.5) for this 8×8 grid can be worked out.

3 Iterative Methods for Linear Systems

There are a variety of methods to approximate a solution to the system

$$Ax = b \tag{9}$$

where x is an unknown vector, b is a known vector, and A is a known matrix of coefficients. To begin we consider two basic methods.

These methods might take a large number of iterations to converge to a solution and might not be useful as standalone solvers.

Jacobi Method

The Jacobi iteration is based on the idea of splitting up A into D, E and F.

$$A = D - E - F \tag{10}$$

in which D is the diagonal of A, $-E$ its strict lower part, and $-F$ its strict upper part, It is always assumed that the diagonal entries of A are all nonzero.

The Jacobi iteration determines the $i-$th component of the next approximation so as to annihilate the $i-$th component of the residual vector. In the following, ξ_i denotes the $i-$th component of the iterate x_k and β_i the $i-$th component of the right-hand side b. Thus, writing

$$(b - Ax_{k+1})_i = 0 \tag{11}$$

in which $(y)_i$ represents the $i-$th component of the vector y, yields

$$a_{ii}\xi_i^{(k+1)} = -\sum_{j=1, j\neq 1}^{n} a_{ij}\xi_j^{(k)} + \beta_i, \tag{12}$$

or

$$\xi_i^{(k+1)} = \frac{1}{a_{ii}}\left(\beta_i - \sum_{j=1 j\neq 1}^{n} a_{ij}\xi_j^{(k)}\right) i = 1, ..., n \tag{13}$$

This is a component-wise form of the Jacobi iteration. All components of the next iterate can be grouped into the vector x_{k+1}. The above notation can be used to rewrite the Jacobi iteration (13) in vector form as

$$x_{k+1} = D^{-1}(E + F)x_k + D^{-1}b \tag{14}$$

Gauss-Seidel

The Gauss-Seidel iteration corrects the $i-$th component of the current approximate solution, in the order $i = 1, 2, ..., n$, again to annihilate the $i-$th component of the residual. However, this time the approximate solution is updated immediately after the new component is determined. The newly computed components $\xi_i^{(k)}, i = 1, 2, ..., n$ can be changed within a working vector which is redefined at each relaxation step. Thus, since the order is $i = 1, 2, ..., n$, the result at the $i-$th step is

$$\beta_i - \sum_{j=1}^{i-1} a_{ij}\xi_j^{(k+1)} - a_{ii}\xi_i^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}\xi_j^{(k)} = 0, \tag{15}$$

which leads to the iteration,

$$\xi_i^{(k+1)} = \frac{1}{a_{ii}}\left(-\sum_{j=1}^{i-1} a_{ij}\xi_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}\xi_j^{(k)} + \beta_i\right), i = 1, ..., n \tag{16}$$

the defining equation (15) can be written as

$$b + Ex_{k+1} - Dx_{k+1} + Fx_k = 0, \tag{17}$$

which leads immediately to the vector form of the Gauss-Seidel iteration

$$x_{k+1} = (D - E)^{-1}Fx_k + (D - E)^{-1}b. \tag{18}$$

Computing the new approximation in (14) requires multiplying by the inverse of the diagonal matrix D. In (18) a triangular system must be solved with $D - E$, the lower triangular part of A. Thus, the new approximation in a Gauss-Seidel step can be determined either by solving a triangular system with the matrix $D - E$ or from the relation (15).

A backward Gauss-Seidel iteration can also be defined as

$$(D - F)x_{k+1} = Ex_k + b, \tag{19}$$

which is equivalent to making the coordinate corrections in the order $n, n-1, ..., 1$. A Symmetric Gauss-Seidel Iteration consists of a forward sweep followed by a backward sweep. The Jacobi and the Gauss-Seidel iterations are both of the form

$$Mx_{k+1} = Nx_k + b = (M - A)x_k + b, \tag{20}$$

in which

$$A = M - N \tag{21}$$

is a splitting of A , with $M = D$ for Jacobi, $M = D - E$ for forward Gauss-Seidel,and $M = D - F$ for backward Gauss-Seidel.

Writing in terms of the solution vector x_k at the k^{th} iteration we have the Jacobi and Gauss-Seidel iterations look like the form

$$x_{k+1} = Gx_k + f, \tag{22}$$

in which

$$G_{JA}(A) = I - D^{-1}A \tag{23}$$

$$G_{GS}(A) = I - (D - E)^{-1}A, \tag{24}$$

for the Jacobi and Gauss-Seidel iterations, respectively.

3.1 Conjugate Gradient

The Conjugate Gradient method is an important method for solving sparse linear systems. It is based on the idea of using a projection method on Krylov Subspaces $\mathcal{K}_m$ to find an approximate solution x_m to

$$Ax = b \tag{25}$$

This is in turn done by imposing a Petrov Galerkin condition

$$b - Ax_m \perp \mathcal{L}_m, \tag{26}$$

where $\mathcal{L}_m$ is another subspace of dimension m. Here, x_0 represents an arbitrary initial guess to the solution. The subspace $\mathcal{K}_m$ is written as

$$\mathcal{K}_m(A, r_0) = span\{r_0, Ar_0, A^2r_0, ..., A^{m-1}r_0\}, \tag{27}$$

where $r_0 = b - Ax_0$.

When there is no ambiguity, $\mathcal{K}_m(A, r_0)$will be denoted by $\mathcal{K}_m$. The different versions of Krylov subspace methods arise from different choices of the subspace $\mathcal{L}_m$ and from the ways in which the system is preconditioned.

Two broad choices for $\mathcal{L}_m$ give rise to the best known techniques. The first is simply $\mathcal{L}_m = \mathcal{K}_m$ and the minimum-residual variation $\mathcal{L}_m = A\mathcal{K}_m$.

An important assumption, for The Conjugate Gradient Method, is that the coefficient matrix A is Symmetric Positive Definite(SPD).

3.1.1 Arnoldi Orthogonalization

The Arnoldi method is an orthogonal projection method onto $\mathcal{K}_m$ for general non-Hermitian matrices. The procedure was introduced in 1951 as a means of reducing a dense matrix into Hessenberg form. Arnoldi presented his method in this manner but hinted that the eigenvalues of the Hessenberg matrix obtained from a number of steps smaller than n could provide accurate approximations to some eigenvalues of the original matrix. It was later discovered that this strategy leads to an efficient technique for approximating eigenvalues of large sparse matrices.

Algorithm 1 Arnoldi Orthogonalization

1: Choose a vector v_1 of norm 1
2: **for** For $j = 1, 2, ...m$ **do**
3: Compute $h_{ij} = (Av_j, v_i)$ for $i = 1, 2, ..., j$
4: Compute $w_j := Av_j - \sum_{i=1}^{j} h_{ij} v_i$
5: $h_{j+1,j} = \|w_j\|_2$
6: **if** $h_{j+1,j} = 0$ **then**
7: Stop
8: **end if**
9: $v_{j+1} = w_j / h_{j+1,j}$
10: **end for**

At each step, the Algorithm 1 multiplies the previous Arnoldi vector v_j by A and then orthonormalizes the resulting vector w_j against all previous v_is by a standard Gram-Schmidt procedure. It stops if the vector w_j vanishes.

3.1.2 Lanczos Method

The symmetric Lanczos algorithm can be viewed as a simplification of the Arnoldi method for the particular case when the matrix is symmetric. When A is symmetric, then the Hessenberg matrix H_m becomes symmetric tridiagonal. This leads to a three-term recurrence in the Arnoldi process and short-term recurrences for solution algorithms such as FOM and GMRES. The standard notation used to describe the Lanczos algorithm is obtained by setting

$$\alpha_j \equiv h_{ij}, \qquad \beta_j \equiv h_{j-1,j}, \tag{28}$$

and if T_m denotes the resulting H_m matrix, it is of the form,

$$T_m = \begin{pmatrix} \alpha_1 & \beta_2 & & & \\ \beta_2 & \alpha_2 & \beta_3 & & \\ & \cdot & \cdot & \cdot & \\ & & \beta_{m-1} & \alpha_{m-1} & \beta_m \\ & & & \beta_m & \alpha_m \end{pmatrix}. \tag{29}$$

This leads to the Lanczos algorithm specified in Algorithm 2.

It is rather surprising that the above simple algorithm guarantees, at least in exact arithmetic, that the vectors $v_i, i = 1, 2, ...$, are orthogonal. In reality, exact orthogonality of these vectors is only observed at the beginning of the process. At some point the v_is start losing their global orthogonality rapidly. The major practical differences with the Arnoldi method are that the matrix H_m is tridiagonal and, more importantly, that only three vectors must be stored, unless some form of re-orthogonalization is employed.

Algorithm 2 Lanczos Algorithm

Choose an initial vector v_1 of norm unity. Set $\beta_1 \equiv 0, v_0 \equiv 0$
for $j = 1, 2, ..., m$ **do**
  $w_j := Av_j - \beta_j v_{j-1}$
  $\alpha_j := (w_j, v_j)$
  $w_j := w_j - \alpha_j v_j$
  $\beta_{j+1} := \|w_j\|_2$
  if $\beta_{j+1} = 0$ **then**
    Stop
  end if
  $v_{j+1} := w_j/\beta_{j+1}$.
end for

3.1.3 Conjugate Gradient Algorithm

The Conjugate Gradient method is a realization of an orthogonal projection technique onto the Krylov Subspace $\mathcal{K}_m(A, r_0)$ where, r_0 is the initial residual.

First we derive the Arnoldi Method for the case when A is symmetric. Given an initial guess x_0 to the linear system $Ax = b$ and the Lanczos vectors $v_i, i = 1, ..., m$ together with the tridiagonal matrix T_m, the approximate solution obtained from an orthogonal projection method onto $\mathcal{K}_m$, is given by

$$x_m = x_0 + V_m y_m, y_m = T_m^{-1}(\beta e_1). \tag{30}$$

We now have the Lanczos method for linear systems in Algorithm 3. We can write LU

Algorithm 3 Lanczos Method for Linear Systems

Compute $r_0 = b - Ax_0$, $\beta := \|r_0\|_2$, and $v_1 := r_0/\beta$
for $j = 1, 1, ..., m$ **do**
  $w_j = Av_j - \beta_j v_{j-1}$(If $j = 1$ set $\beta_1 v_0 \equiv 0$)
  $\alpha_j = (w_j, v_j)$
  $w_j := w_j - \alpha_j v_j$
  $\beta_{j+1} = \|w_j\|_2$
  if $\beta_{j+1} = 0$ **then**
    set $m := j$ and go to 9
  end if
  $v_{j+1} = w_j/\beta_{j+1}$
end for
Set $T_m =$ tridiag $(\beta_i, \alpha_i, \beta_{i+1})$, and $V_m = [v_1, ..., v_m]$.
Compute $y_m = T_m^{-1}(\beta e_1)$ and $x_m = x_0 + V_m y_m$

factorization of T_m as $T_m = L_m U_m$. The matrix L_m is unit lower bi-diagonal and U_m is upper bi-diagonal. Thus, the factorization of T_m is of the form

$$T_m = \begin{pmatrix} 1 & & & & \\ \lambda_2 & 1 & & & \\ & \lambda_3 & 1 & & \\ & & \lambda_4 & 1 & \\ & & & \lambda_5 & 1 \end{pmatrix} \times \begin{pmatrix} \eta_1 & \beta_2 & & & \\ & \eta_2 & \beta_3 & & \\ & & \eta_3 & \beta_4 & \\ & & & \eta_4 & \beta_5 \\ & & & & \eta_5 \end{pmatrix}. \tag{31}$$

The approximate solution is then given by,

$$x_m = x_0 + V_m U_m^{-1} L_m^{-1}(\beta e_1). \tag{32}$$

Letting

$$P_m \equiv V_m U_m^{-1} \tag{33}$$

and

$$z_m = L_m^{-1} \beta e_1, \tag{34}$$

then,

$$x_m = x_0 + P_m z_m. \tag{35}$$

Note that, p_m, the last column of P_m, can be computed from the previous p_i's and v_m by the simple update

$$p_m = \eta^{-1}[v_m - \beta_m p_{m-1}]. \tag{36}$$

Here β_m is a scalar computed from the Lanczos Algorithm, while η_m results from the m-th Gaussian Elimination step on the tridiagonal matrix, i.e.,

$$\lambda_m = \frac{\beta_m}{\eta_{m-1}}, \tag{37}$$

$$\eta_m = \alpha_m - \lambda_m \beta_m. \tag{38}$$

Also from the structure of L_m we have

$$z_m = \begin{bmatrix} z_{m-1} \\ \zeta_m \end{bmatrix}, \tag{39}$$

in which $\zeta_m = -\lambda_m \zeta_{m-1}$. As a result, x_m can be updated at each step as

$$x_m = x_{m-1} + \zeta_m p_m \tag{40}$$

where p_m is defined above.

This brings us to the direct version of Lanczos algorithm for linear systems.

Algorithm 4 D-Lanczos

1: Compute $r_0 = b - Ax_0, \zeta_1 := \beta := \|r_0\|_2$, and $v_1 := r_0/\beta$
2: Set $\lambda_1 = \beta_1 = 0, p_0 = 0$
3: **for** $m = 1, 2, ..., until convergence$: **do**
4: Compute $w := Av_m - \beta v_{m-1}$ and $\alpha_m = (w, v_m)$
5: **if** $m > 1$ **then**
6: compute $\lambda_m = \frac{\beta_m}{\eta_{m-1}}$ and $\zeta_m = -\lambda_m \zeta_{m-1}$
7: **end if**
8: $\eta_m = \alpha_m - \lambda_m \beta_m$
9: $p_m = \eta_m^{-1}(v_m - \beta_m p_{m-1})$
10: $x_m = x_{m-1} + \zeta_m p_m$
11: **if** x_m has converged **then**
12: Stop
13: **end if**
14: $w := w - \alpha_m v_m$
15: $\beta_{m+1} = \|w\|_2, v_{m+1} = w/\beta_{m+1}$
16: **end for**

This algorithm computes the solution of the tridiagonal system $T_m y_m = \beta e_1$ progressively by using Gaussian elimination without pivoting. However, partial pivoting can also be implemented at the cost of having to keep an extra vector. In fact, Gaussian elimination with partial pivoting is sufficient to ensure stability for tridiagonal systems. Observe

that the residual vector for this algorithm is in the direction of v_{m+1} due to equation (30). Therefore, the residual vectors are orthogonal to each other. Likewise, the vectors p_i are $A-$orthogonal, or conjugate orthogonal.

A consequence of the above proposition is that a version of the algorithm can be derived by imposing the orthogonality and conjugacy conditions. This gives the Conjugate Gradient algorithm which we now derive.

The vector x_{j+1}, the solution at iteration $j+1$, can be written as,

$$x_{j+1} = x_j + \alpha_j p_j. \tag{41}$$

Therefore the residual vectors must satisfy the recurrence

$$r_{j+1} = r_j - \alpha_j A p_j. \tag{42}$$

If the r_j's are to be orthogonal, then it is necessary that $(r_j - \alpha_j A p_j, r_j) = 0$ and as a result

$$\alpha_j = \frac{(r_j, r_j)}{(Ap_j, r_j)} \tag{43}$$

Also, it is known that the next search direction p_{j+1} is a linear combination of r_{j+1} and p_j, and after rescaling the p vectors appropriately, it follows that

$$p_{j+1} = r_{j+1} + \beta_j p_j. \tag{44}$$

Thus, a first consequence of the above relation is that

$$(Ap_j, r_j) = (Ap_j, p_j - \beta_{j-1} p_{j-1}) = (Ap_j, p_j) \tag{45}$$

because Ap_j is orthogonal to p_{j-1}. Then, (43) becomes $\alpha_j = (r_j, r_j)/(Ap_j, p_j)$. In addition, writing that p_{j+1} as defined by (44) is orthogonal to Ap_j yields

$$\beta_j = -\frac{(r_{j+1}, Ap_j)}{(p_j, Ap_j)} \tag{46}$$

β_j can also be written as

$$\frac{1}{\alpha_j}\frac{(r_{j+1}, (r_{j+1} - r_j))}{(Ap_j, p_j)} = \frac{(r_{j+1}, r_{j+1})}{(r_j, r_j)} \tag{47}$$

Putting these together we have the algorithm for Conjugate Gradient.

Algorithm 5 Conjugate Gradient Algorithm

1: Compute $r_0 := b - Ax_0, p_0 := r_0$.
2: **for** $j = 0, 1, ...,$ until convergence **do**
3: $\quad \alpha_j := (r_j, r_j)/(Ap_j, p_j)$
4: $\quad x_{j+1} := x_j + \alpha_j p_j$
5: $\quad r_{j+1} := r_j - \alpha_j Apj$
6: $\quad \beta_j := (r_{j+1}, r_{j+1})/(r_j, r_j)$
7: $\quad p_{j+1} := r_{j+1} + \beta_j p_j$
8: **end for**

3.2 Preconditioning

Efficiency of iterative techniques can be improved by using *preconditioning*. Preconditioning is simply a means of transforming the original linear system into one which has the same solution, but which is likely to be easier to solve with an iterative solver.

Consider a matrix A that is symmetric and positive definite and assume that a preconditioner M is available. The preconditioner M is a matrix which approximates A in some yet-undefined sense. It is assumed that M is also Symmetric Positive Definite. From a practical point of view, the only requirement for M is that it is inexpensive to solve linear systems $Mx = b$. This is because the preconditioned algorithms will all require a linear system solution with the matrix M at each step. Then, for example, the following preconditioned system could be solved:

$$M^{-1}Ax = M^{-1}b \tag{48}$$

or

$$AM^{-1}u = b \tag{49}$$

$$x = M^{-1}u \tag{50}$$

These two systems are no longer symmetric in general. To preserve symmetry one can decompose M in its Cholesky factorization, that is:

$$M = LL^T, \tag{51}$$

Then a simple way to preserve symmetry is to split the preconditioner between left and right, i.e., to solve

$$L^{-1}AL^{-T}u = L^{-1}b, x = L^{-T}u \tag{52}$$

which involves a Symmetric Positive Definite matrix. However, it is not necessary to split the preconditioner in this manner in order to preserve symmetry. Observe that $M^{-1}A$ is self-adjoint for the M -inner product,

$$(x, y)_M \equiv (Mx, y) = (x, My) \tag{53}$$

since

$$(M^{-1}Ax, y)_M = (Ax, y) = (x, Ay) = (x, M(M^{-1}A)y) = (x, M^{-1}Ay)_M \tag{54}$$

Therefore, an alternative is to replace the usual Euclidean inner product in the Conjugate Gradient (CG) algorithm by the M-inner product. If the CG algorithm is rewritten for this new inner product, denoting by the original residual and by $r_j = b - Ax_j$ the original residual and by $z_j = M^{-1}r_j$ the residual for the preconditioned system the following sequence can be written .

Algorithm 6 Preconditioned method of calculating new search direction.

1: $\alpha_j := (z_j, z_j)_M/(M^{-1}Ap_j, p_j)_M$
2: $x_{j+1} := x_j + \alpha_j p_j$
3: $r_{j+1} := r_j - \alpha_j Ap_j$ and$z_{j+1} := M^{-1}r_{j+1}$
4: $\beta_j := (z_{j+1}, z_{j+1})_M/(z_j, z_j)_M$
5: $p_{j+1} := z_{j+1} + \beta_j p_j$

Algorithm 7 Preconditioned Conjugate Gradient Algorithm.

1: Compute $r_0 := b - Ax_0, z_0 = M^{-1}r_0,$ and $p_0 := z_0$
2: **for** $j = 0, 1, ...$ until convergence **do**
3: $\alpha_j := (r_j, z_j)/(Ap_j, p_j)$
4: $x_{j+1} := x_j + \alpha_j p_j$
5: $r_{j+1} := r_j - \alpha_j Ap_j$
6: $z_{j+1} := M^{-1}r_{j+1}$
7: $\beta_j := (r_{j+1}, z_{j+1})/(r_j, z_j)$
8: $p_{j+1} := z_{j+1} + \beta_j p_j$
9: **end for**

Since $(z_j, z_j)_M = (r_j, z_j)$ and $(M^{-1}Ap_j, p_j)_M = (Ap_j, p_j)$, the M -inner products do not have to be computed explicitly.

We have the preconditioned iteration for the CG algorithm as follows in Algorithm 7. A host of Preconditioning methods are known in the literature like [Saad, 2003]. ILU Preconditioning and Incomplete Cholesky being the most prominent. In our implementations we use Block version of the Incomplete Cholesky Preconditioning and also a novel algorithm called the Incomplete Poisson Preconditioning.

3.2.1 Incomplete Cholesky Preconditioning

Incomplete Cholesky Preconditioning involves a preconditioner of the variety

$$M = LL^T \tag{55}$$

where L is lower triangular. It is made 'incomplete' by dropping off some of the elements. In our case we simply follow the sparsity pattern of A. So if A has two diagonals on either side of the main diagonal we make L such that it also has the same structure i.e. two sub-diagonals.

3.2.2 Incomplete Poisson Preconditioning

Although Incomplete Cholesky Preconditioning is very effective in achieving convergence for the Conjugate Gradient Method. It is highly sequential within the block. Since in this study we implement Preconditioned Conjugate Gradient on a Data Parallel Architecture we also consider a recently suggested method of preconditioning called the Incomplete Poisson Preconditioning [Ament, Knittel, Weiskopf, and Straβer, 2010].

As we will show later in our results and the authors also hint that there is a price to pay for this 'parallelism' in terms of convergence speed. However, our experiments show that it is still at least as fast(rate of convergence) or comparable to the Block Incomplete Cholesky version (for a particular grid size) when the number of blocks is the maximum possible.

We now present a brief discussion as presented by the authors in the original publication.

While the SpAI and AINV algorithms are reasonable approaches for arbitrary matrices to find an inverse of the Coefficient Matrix, the authors of this method sought an easier way for the Poisson equation. For this reason, they developed a heuristic preconditioner that approximates the inverse of the 5-point Laplacean matrix. Their goal was to find an algorithm that is as easy to implement as a Jacobi preconditioner and that is also well suited for GPU processing. They provide an analytical expression for the preconditioner and show that it satisfies the requirements for convergence in the Preconditioned Conjugate Gradient (PCG) method. Secondly, they enforce the sparsity pattern of A on their

preconditioner and sketch an informal proof that the condition of the modified system is improved compared to the original matrix.

Just like SSOR their preconditioner depends on sum decomposition of A into its lower triangular part L and its diagonal D. The approximation of the inverse then is

$$M^{-1} = (I - LD^{-1})(I - D^{-1}L^T) \tag{56}$$

where L is the lower triangular part of A and D is the diagonal matrix containing diagonal elements of A. In this expression D^{-1} can be calculated by the reciprocal operation on the diagonal of A. Applying a preconditioner to the PCG algorithm requires that the modified system is still symmetric positive definite, which in turn requires that the preconditioner is a symmetric real-valued matrix.

$$\begin{aligned}
(M^{-1})^T &= ((I - LD^{-1})(I - D^{-1}L^T)^T && (57)\\
&= (I - D^{-1}L^T)^T(I - LD^{-1})^T && (58)\\
&= (I^T - (D^{-1}L^T)^T)(I^T - (LD^{-1})^T) && (59)\\
&= (I - L^{T^T} - D^{-1^T})(I - D^{-1^T}L^T) && (60)\\
&= (I - LD^{-1})(I - D^{-1}L^T) && (61)\\
&= M^{-1} && (62)
\end{aligned}$$

Therefore one can write the preconditioner as,

$$M = KK^T \tag{63}$$

where

$$K = I - LD^{-1} \qquad \textit{and} \qquad K^T = I - D^{-1}L^T. \tag{64}$$

This shows that PCG Algorithm converges when this preconditioner is applied. Considering a two-dimensional regular discretization and taking the case of an inner grid cell, the stencil of the i-th row of A is

$$\begin{aligned}
row_i(A) &= (a_{y-1}, a_{x-1}, a, a_{x+1}, a_{y+1}) && (65)\\
&= (-1, -1, 4, -1, -1) && (66)
\end{aligned}$$

In order to demonstrate that the introduced method is advantageous, they provide a short abstract as to why the condition of the modified system improves. To make things clearer, they use a two-dimensional regular discretization and only focus on an inner grid cell. In this case, the stencil of the i-th row of A is

$$\begin{aligned}
row_i(A) &= (a_{y-1}, a_{x-1}, a, a_{x+1}, a_{y+1}) && (67)\\
&= (-1, -1, 4, -1, -1) && (68)
\end{aligned}$$

Hence, the stencils for L, D^{-1}, and L^T

$$\begin{aligned}
row_i(L) &= (-1, -1, 0, 0, 0) && (69)\\
row_i(D^{-1}) &= (0, 0, 0.25, 0, 0) && (70)\\
row_i(L^T) &= (0, 0, 0, -1, -1) && (71)
\end{aligned}$$

In the next step, after performing the operations for (64).

$$\begin{aligned}
row_i(K) &= (0.25, 0.25, 1, 0, 0) && (72)\\
row_i(K^T) &= (0, 0, 1, 0.25, 0.25) && (73)
\end{aligned}$$

The final step is the matrix-matrix product KK^T, which is the multiplication of a lower and an upper triangular matrix. Each of the 3 coefficients in $row_i(K)$ hits 3 coefficients in K^T but in different columns. The interleaved arrangement in such a row-column product introduces new non-zero coefficients in the result. The stencil of the inverse increases to

$$row_i(M^{-1}) = (m_{y-1}, m_{x+1,y-1}, m_{x-1}, m, m_{x+1}, m_{x-1,y+1}, m_{y+1}) \tag{74}$$
$$= (0.25, 0.0625, 0.25, 1.125, 0.25, 0.0625, 0.25) \tag{75}$$

Without going into too much detail here, the stencil enlarges to up to 13 non-zero elements in three dimensions for each row, which would almost double the computational effort in a matrix-vector product compared to the 7-point (in 3-D) stencil in the original matrix. By looking again at the coefficients in $row_i(M^{-1})$ it can be observed that the additional non-zero values are rather small compared to the rest of the coefficients. Furthermore, this nice property remains true in three dimensions, so they use an incomplete stencil assuming that these small coefficients only have a minor influence on the condition. They set them to zero and obtain the following 5-point stencil in two dimensions

$$row_i(M^{-1}) = (0.25, 0, 0.25, 1.125, 0.25, 0, 0.25) \tag{76}$$

Another important property of the incomplete formulation is the fact that symmetry is still preserved as the cancellation always affects two pair-wise symmetric coefficients namely

$$(m_{x+1,y-1}, m_{x-1,y+1}) \tag{77}$$

in two dimensions and

$$(m_{x+1,z-1}, m_{x-1,z+1})(m_{x+1,y-1}, m_{x-1,y+1})(m_{y+1,z-1}, m_{y-1,z+1}) \tag{78}$$

in three dimensions. From this it follows that the CG method still converges and hence the *Incomplete Poisson* (IP) *preconditioner*. They provide a heuristic approach to demonstrate the usefulness of their preconditioner. A perfectly conditioned matrix is the identity matrix, hence they simply try to evaluate how close this modified (Incomplete Poisson Preconditioned) system reaches this ideal. For this purpose, they calculate AM^{-1}. For an inner grid cell, this leads to the following not trivially vanishing elements

$$row_i(AM^{-1}) = (-0.25, 0.0, 0.0, -0.50, -0.125, -0.50, -0.125, 3.5, \tag{79}$$
$$-0.125, -0.50, -0.125, -0.50, -0.125, -0.50, 0.0, 0.0, -0.25). \tag{80}$$

They argue that this row represents the whole band in the matrix with $\frac{7}{2}$ as the diagonal element. All elements to the left and right of this tuple are zero. This result still does not look like the identity but there is another property of the condition number that allows to multiply the system with an arbitrary scalar value, except zero, because multiplying a matrix with a scalar does not affect the ratio of the maximum and minimum eigenvalues. For example, $\alpha \times AM^{-1}$ also scales all of the eigenvalues of AM^{-1} with α

$$\kappa(\alpha AM^{-1}) = \frac{\alpha \times \lambda_{max}}{\alpha \times \lambda_{min}} = \frac{\lambda_{max}}{\lambda_{min}} = \kappa(AM^{-1}). \tag{81}$$

Hence by arbitrarily choosing the α to be the constant $\frac{2}{7}$ the tuple changes to,

$$\frac{2}{7}.row_i(AM^{-1}) = (-0.071, 0.0, 0.0, -0.142, -0.0357, -0.142, -0.071, -0.0357, 1, \tag{82}$$
$$0.0357, -0.071, -0.142, -0.0357, -0.142, 0.0, 0.0, -0.071). \tag{83}$$

The result shows that the element-wise signed distance to the identity is much smaller than with the original Poisson system, suggesting a lower condition number. By calculating the

product AM^{-1} and comparing the element-wise distance of this product with the Identity Matrix they further show that this preconditioner effectively reduces the condition number of the resulting matrix.

Talking about the parallel properties of this algorithm, they are very well aligned to the GPU. Since we have to calculate values (in the inverse of the Preconditioning matrix, M^{-1}) only for the non-zero elements of A. The structure and number of elements in the preconditioner is the same and hence the method of storage and matrix vector multiplication will be the same. This means that we can exploit the same amount of parallelism in the Preconditioning($y_k = M^{-1}r_k$) as in case of Sparse Matrix vector ($Ax = b$), since they are identical operations. The degree of parallelism being N. Each row can be separately computed for the resulting vector Ax or y.

3.3 Deflation

Deflation is an attempt to treat the bad eigenvalues resulting in the preconditioned matrix

$$M^{-1}Ax = M^{-1}b \tag{84}$$

$M^{-1}A$, where M^{-1} is a symmetric positive definite (SPD) preconditioner and A is the symmetric positive definite (SPD) coefficient matrix. This operation reduces the convergence iterations for the Preconditioned Conjugate Gradient (PCG) method.

The original linear system

$$Ax = b \tag{85}$$

can be solved by employing the splitting

$$x = (I - P^T)x + P^T x \tag{86}$$

Simplifying we get

$$\begin{aligned} x = (I - P^T)x + P^T x \Leftrightarrow x &= Qb + P^T x && (87)\\ \Leftrightarrow Ax &= AQb + AP^T x && (88)\\ \Leftrightarrow b &= AQb + PAx && (89)\\ \Leftrightarrow Pb &= PAx, && (90) \end{aligned}$$

where

$$P := I - AQ, Q := ZE^{-1}Z^T, E := Z^T AZ. \tag{91}$$

where

$E \in \mathbb{R}^{k\times k}$ is the invertible Galerkin Matrix, $Q \in \mathbb{R}^{n\times n}$ is the correction Matrix, and $P \in \mathbb{R}^{n\times n}$ is the deflation matrix.

Also it is given that A is an SPSD coefficient matrix as given in (85) and $Z \in \mathbb{R}^{n\times k}$, with full rank and $k < n - d$ is given. k is the number of columns of Z and is also called as the 'number of deflation vectors'.

The x at the end of the expression is not necessarily a solution of the original linear system (85), since it might consist of components of the null space of PA, $\mathcal{N}(PA)$. Therefore this 'deflated' solution is denoted as $\hat{x}$ rather than x. The deflated system is now,

$$PA\hat{x} = Pb \tag{92}$$

The Preconditioned deflated version of the Conjugate Gradient Method can now be presented. The deflated method (92) can be solved using a symmetric positive definite (SPD) preconditioner, M^{-1}. We therefore now seek a solution to

$$\tilde{P}\tilde{A}\hat{\tilde{x}} = \tilde{P}\tilde{b}, \tag{93}$$

where

$$\tilde{A} := M^{-\frac{1}{2}}AM^{-\frac{1}{2}}, \hat{\tilde{x}} := M^{\frac{1}{2}}\hat{x}, \tilde{b} := M^{-\frac{1}{2}}b, \tag{94}$$

and

$$\tilde{P} := I - \tilde{A}\tilde{Q}, \tilde{Q} := \tilde{Z}\tilde{E^{-1}}\tilde{Z^T}, \tilde{E} := \tilde{Z^T}\tilde{A}\tilde{Z}, \tag{95}$$

where $\tilde{Z} \in \mathbb{R}^{n*k}$ can be interpreted as a preconditioned deflation-subspace matrix. The resulting method is called the Deflated Preconditioned Conjugate Gradient (DPCG) method [Tang, 2008].

Algorithm 8 Deflated Preconditioned Conjugate Gradient Algorithm

1: Select x_0. Compute $r_0 := b - Ax_0$ and $\hat{r_0} = Pr_0$, Solve $My_0 = \hat{r_0}$ and set $p_0 := y_0$.
2: **for** j:=0,..., until convergence **do**
3: $\hat{w_j} := PAp_j$
4: $\alpha_j := \frac{(\hat{r_j}, y_j)}{(p_j, \hat{w_j})}$
5: $\hat{x}_{j+1} := \hat{x}_j + \alpha_j p_j$
6: $\hat{r}_{j+1} := \hat{r}_j - \alpha_j \hat{w}_j$
7: Solve $My_{j+1} = \hat{r}_{j+1}$
8: $\beta_j := \frac{(\hat{r}_{j+1}, y_{j+1})}{(\hat{r}_j, y_j)}$
9: $p_{j+1} := y_{j+1} + \beta_j p_j$
10: **end for**
11: $x_{it} := Qb + P^T x_{j+1}$

This method is numerically more stable than the method discussed in [Saad, Yeung, Erhel, and Guyomarc'h, 2000]. It can be seen that $\tilde{P}$ or $M^{\frac{1}{2}}$ are never calculated explicitly. Hence the linear system is often denoted by

$$M^{-1}PA\hat{x} = M^{-1}Pb \tag{96}$$

Some Observations:

- All known properties of Preconditioned Conjugate Gradient (PCG) also hold for DPCG, where PA can be interpreted as the coefficient matrix A in (48). Moreover if $P = I$ is taken the algorithm above reduces to the PCG algorithm.
- Careful selection of Deflation vectors is required for this method to prove useful. Two methods, one based on eigenvector (of $M^{-1}A$)based subspace for Z and the other based on an arbitrary choice of the deflation subspace, are worth to mention.

However to calculate the eigenvectors itself could be computationally intensive so an arbitrary choice which closely resembles the part of the eigenspace is the way out. In short the ideal deflation method should satisfy the following criteria:

- The deflation-subspace matrix Z must be sparse;
- The deflation vectors approximate the eigenspace corresponding to the unfavorable eigenvalues;

- The cost of constructing deflation vectors is relatively low;
- The method has favorable parallel properties;
- The approach can be easily implemented in an existing PCG code.

Subdomain Deflation based choice for Deflation vectors emerges as a close match.

3.3.1 Subdomain Deflation

In Subdomain deflation, the deflation vectors are chosen in an algebraic way. The computational domain is divided into several subdomains, where each subdomain corresponds to one or more deflation vectors. Consider application of Subdomain deflation to Poisson Equation with discontinuous coefficients as listed in Equation 1

Now assume that the computational domain, Ω is divided into several subdomains, Ω_j, where each Ω_j corresponds to one deflation vector, consisting of ones for grid points in the interior of the discretized subdomain, Ω_{h_j}, and zeroes for other grid points. Then, subdomain deflation is effective, if each subdomain, Ω_j, corresponds to exactly one constant part of the coefficient, ρ. In this case, the subspace spanned by the deflation vectors is proved to be almost equal to the eigenspace associated with the smallest eigenvalues.

4 GPU for Scientific Computing

GPUs commercially available of the shelf, promise up to 900 gigaflops (single precision) of compute power. The cost at which this performance is available is a couple of hundred euros. This makes it an attractive option already compared to setting up or sharing a cluster that might be hard to get access to or costlier to put up in the first place. Scientific computing problems could be restructured with some effort to work on the GPUs and it is not uncommon to get up to 10x speed-up (compared to a sequential implementation on the CPU) with simple first implementation. Talking about implementation, it has become a lot easier with the advent of CUDA (Compute Unified Device Architecture) from NVIDIA, to write C code, that runs on the GPU. Earlier this was not the case. Before the advent of CUDA, to harness the capabilities of the GPU, C programs had to be adapted to the shader languages like Cg. The programmer had to understand the way how a GPU interprets textures and objects in a rendering environment and he had to explicitly mold the application code as if it were a rendering operation.

4.1 Device Architecture

The GPU is a SIMD processor (Single Instruction Multiple Data). What this means is that there is an army of processors waiting to crunch the problem computations, all the processors execute the exact same instructions but they do so in parallel without any dependence. The result is that if one of them is able to say execute only at a clock rate of 1 Ghz, and they can do one floating point operation (FLOP) per clock cycle then the number of flops (FLOP/second) equal the number of processors multiplied by the clock rate. A GPU has some fixed number of processors within, these are called streaming multiprocessors (SMs). Each multiprocessor further has eight scalar processors (SPs). Each of the scalar processors executes a piece of code called a kernel. This is the basic unit of execution at the level of Scalar Processors. So if there are say 240 (scalar) processors then they all run the same kernel at the same time (this also depends on the execution configuration and the kernel itself).

4.1.1 Execution Configuration

To run the kernels on the GPU one has to define an execution configuration in terms of grid of blocks. A grid has blocks arranged like the elements of a matrix. Each block contains threads arranged like elements in a matrix. Each thread runs the kernel. Before launching these threads the GPU must be told how many of these threads have to be run. For example we have a kernel that adds two elements of two different matrices and stores the sum in the corresponding element of a third matrix. Then if we want all sums to be done in parallel we would need to launch $m \times n$ kernels (if we have architectural provisions to launch that many threads) if each of the matrices are $m \times n$ matrices.

Now once we have decided on a number of kernels we need to divide them amongst blocks and also calculate how the blocks have to be arranged in a grid (as in Figure 4).

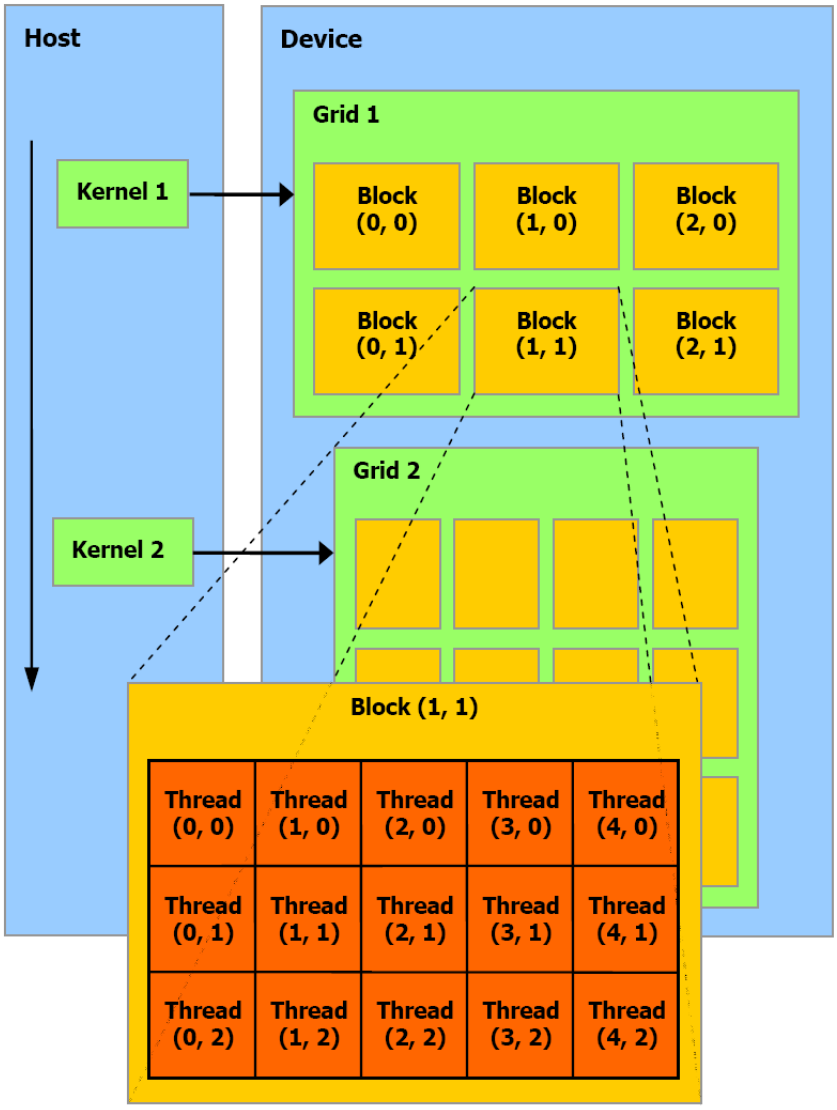

Figure 4: Grids of Blocks with threads

This can be done using variables of the type $dim3$ provided by NVIDIA CUDA environment. The $dim3$ variables are structures with three individual elements that contain the number of threads in x, y and z direction. Such an arrangement makes it easier to

address individual elements. Threads can be individually addressed by their x, y and z co-ordinates corresponding to the co-ordinates in the matrix.

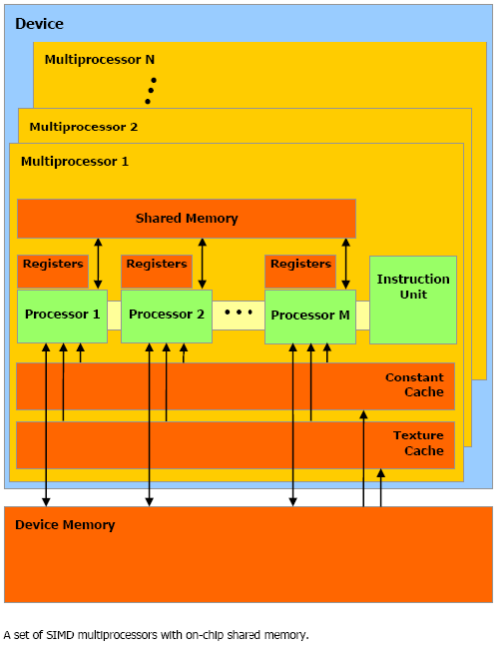

Figure 5: GPU Architecture

4.1.2 Execution of Threads

Threads inside a block are grouped into warps. A multiprocessor on the GPU is assigned some number of blocks. The scheduler that picks up threads for execution, does so in granularity of a warp. So, if the warp size is say, 32, it will pick 32 threads with consecutive thread Ids and schedule them for execution in the next cycle.

Each thread executes on one of the scalar processors. The Streaming Multiprocessors (SMs) are capable of executing a number of warps simultaneously. This number can vary from 512 up to 1024 on the GPUs depending on the type of card. At the time of issue from the scheduler the SMs are handed over a number of blocks to execute. These can vary on the requirement that each thread imposes in terms of registers and shared memory since they are limited on each multiprocessor.

For example suppose that maximum number of threads that can be scheduled on a multiprocessor is 768. Further if each warp is composed of 32 threads then we can have maximum of 24 warps. Now many possible schemes for division could be laid down:

1. 256 threads per block * 3 blocks
2. 128 threads per block * 6 blocks
3. 16 threads per block * 48 blocks

Now each SM has a restriction on the number of blocks that can simultaneously run on it. So if the maximum number of (active) blocks is say 8 then only the 1st and 2nd schemes could form a valid execution configuration.

4.1.3 Memory Model

Each multiprocessor has a set of memories associated with it. These memories have different access times. It must be noted that these memories are on the device (the GPU) and are different from the DRAM available with the CPU. They are:

1. Register Memory

   There are fixed number of registers (per block) that must be divided amongst the number of threads (in a block) that are configured (by the previously discussed allocation of threads in blocks and grids). Registers are exclusive to each thread.

2. Shared Memory

   Shared memory is accessible to all the threads within a block. It is the next best thing after registers since accessing it is cheaper than the global memory.

3. Texture Memory

   Texture memory is read-only and could be read by all the threads across blocks on a single multiprocessor. It also has a local cache on the MultiProcessor.

4. Global Memory

   This memory is the biggest in size and is placed farthest from the threads executing on the multiprocessors. Its access times compared to the shared memory (access) latency might be up to 200 times more.

An important consequence of the different access times of the available memories is that it often becomes important to hide the latency with available computation. Register usage can also decide how many blocks (consequently how many threads) might execute in parallel. So suppose if we have 8192 registers and each thread (in a 16×16 block) uses 10 registers then the maximum number of blocks that can execute on a SM is 3 since $10 * 256 = 2560$ and $\frac{8192}{2560}$ gives 3 (integer) as result. Increasing by only one register per thread reduces the number of blocks by 1.

Global memory access if done in a regular fashion (regularly spaced or aligned) and if it is consistent within a warp (all accesses are at equal strides or increments) then the data can be brought from global memory in a lower number of instructions. This alignment called coalescing could yield performance benefits when data is neatly aligned. Aligned Global Memory Access is the key to exploiting the Huge Memory Bandwidth available on a GPU. More details can be found out in [NVIDIA prog, 2009].

4.2 Language Extensions

NVIDIA provides Language extensions in form of a package called CUDA(Compute Unified Device Architecture). This package enables programmers to write parallel code for all the CUDA-enabled GPUs that NVIDIA has to offer.

The availability of CUDA APIs is instrumental in rapidly parallelizing an application for performance on the GPU. Without getting into the details of how it is mapped to the Graphics hardware we can say that it allows space to focus on the Computational Aspect of the problem.

We have used CUDA 2.3 for our implementation on a Tesla C1060 card.

4.3 Methods of reducing code execution times

The Best Practices Guide [NVIDIA best prac, 2009] lists in definite steps (with examples) how one can achieve speed-up on an NVIDIA GPU card. However the following points are worth mentioning as they form the core of all optimization on the GPU.

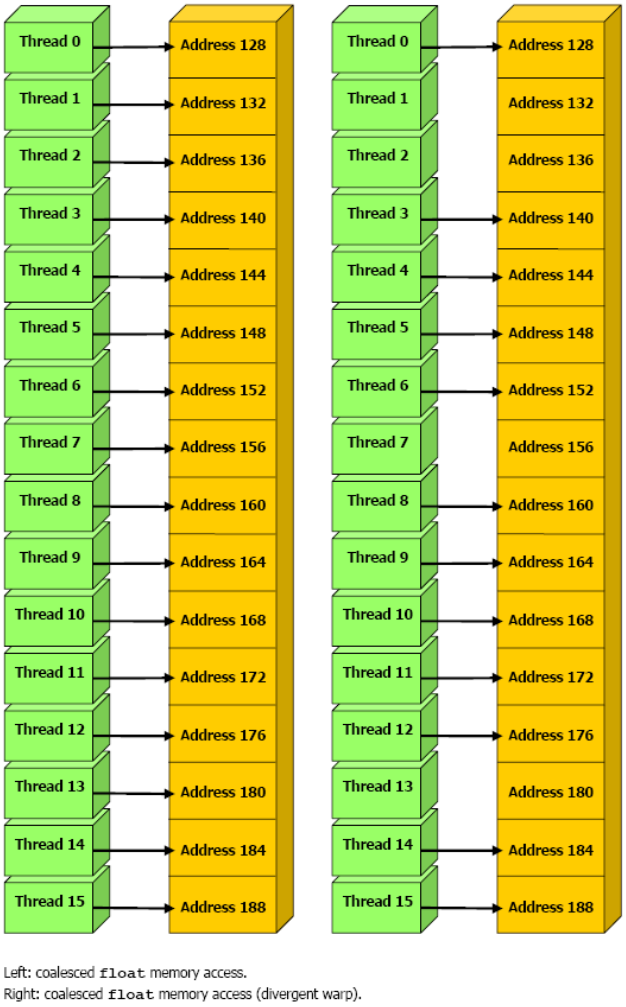

Figure 6: Coalesced Memory Accesses in a GPU

- Coalesced Memory Access
- Minimization of Host-Device Memory Transfers
- Use of Shared Memory Wherever Possible
- Minimal or No Thread Divergence
- Enough Blocks to keep all Multi-Processors at work

Other than these it is also possible to fine tune the application for even more speed-up compared to the host code. Though at some point one has to choose between readability of the code and the fractional increase in performance. It must also be kept in mind that newer revisions of hardware can make the clever hacks obsolete or, at least not worth the effort, in no time. More about how to achieve speedup by using different architectural provisions could be found out in [NVIDIA best prac, 2009].

5 Previous Work

Graphics cards were very early on [Bolz, Farmer, Grinspun, and Schröoder, 2003] identified by researchers as suitable for parallel processing in their own right. At that time it was

required to write applications in shader languages. The applications had to be adapted by understanding the graphics pipeline.

With the advent of CUDA things have changed. Primarily because it has made software development for the GPUs in the domains of Scientific Computing very accessible. The internal architecture has been abstracted so that it fits the application rather seamlessly. Many people have already utilized the GPU to their advantage in achieving accelerated performance for their application. In this section we list a number of efforts in the direction of solving the system of our interest which is,

$$Ax = b \tag{97}$$

The linear system when subjected to Iterative methods for approximating the solution up to a required accuracy has some challenges when mapped to the GPU. When the matrix A is sparse the problem is even more tricky to implement on the GPU.

5.1 Solving Linear Systems on the GPU

When subjecting a system of linear equations to an iterative method, in our case Conjugate Gradient method, some of the basic building blocks become important to optimize. This can be further verified from the Profiler outputs on the host as well as on the device. We now discuss the already available body of work for each of these building blocks.

5.1.1 Prefix Scan for calculating sum

In our implementation we often had to sum up the partial products of a matrices' row elements and the corresponding elements in the vector. This summing operation could be done in parallel by assigning threads that sum it up in turns. Such a method is called the Prefix-Scan based sum operation. [SenGupta, Harris, Zhang, and Owens, 2007] discuss the Scan based operations and we can borrow from one of the phases of their suggested solution to perform summing of the partial products. However this method is useful only when there are many such partial products per row. For our matrix which is a 5-point Laplacean this method could be of little benefit. However later in the Deflation operation ($Z^T x$) we utilize this approach since the number of elements is always greater than 16 and can go upto 512 or more.

The idea is to build a balanced binary tree on the input data and sweep it to and from the root to compute the prefix sum. A binary tree with n leaves has $d = log_2 n$ levels, and each level d has 2^d nodes. If we perform one add per node, then we will perform $O(n)$ adds on a single traversal of the tree. The algorithm consists of two phases: the *reduce* phase (also known as the *up-sweep* phase) and the *down-sweep* phase. In the *reduce* phase, we traverse the tree from leaves to root computing partial sums at internal nodes of the tree, as shown in Figure 7. This is also known as a parallel reduction, because after this phase, the root node (the last node in the array) holds the sum of all nodes in the array.

This is followed by a *down-sweep* phase if one wishes to find the Prefix Sum, however we can stop at this step and already have the sum of all the elements in the element with the highest index.

Details can be found out in the GPU Gems article available online [Harris, Sengupta, and Owens, 2007]. In the same document one can also find methods suggested by the authors to optimize the scan for the GPU by taking into account memory bank conflicts, shared memory and provisions for handling arbitrary array sizes (i.e. where n is not a power of 2).

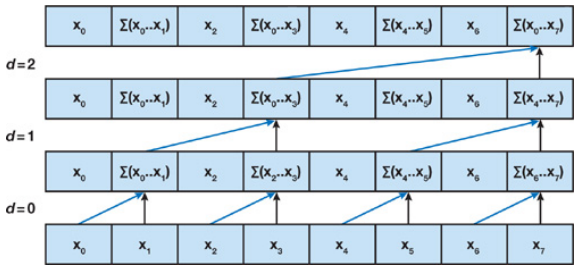

Figure 7: An Illustration of the Up-Sweep, or Reduce, Phase of a Work-Efficient Sum Scan Algorithm

5.1.2 Sparse Matrix Vector Products- SpMVs

Sparse Matrix Vector Multiplication takes bulk of the time when executing CG method. Optimizing it becomes the first step in getting performance out of the many core system. NVIDIA released a study [Bell and Garland, 2008], which they later made available as the CUSP library, for optimizing Sparse Matrix Vector Multiplication. They try out a host of different storage formats and also report on which formats deliver the best possible utilization of GPU resources. They use a hybrid storage format for sparse matrices. It stores the matrices in a ELLpack-COO hybrid format, wherein the rows with more than a threshold number of non-zero elements are stored in the COO format. The rows with less than the threshold are stored in ELLPack format. Details can be found in [Bell and Garland, 2008]. In this extensive study they compare the performance of the kernels they have developed for the GPU $vis - a \cdot - vis$ other architectures like STI Cell, and CPUs like Xeon, Opteron etc. We adopt their idea of storing our Laplacean matrix in the DIA format and then use the suggested kernel for doing SpMV operations.

In a recent study by [Monakov and Avetisyan, 2009] they suggest a hybrid use of a blocked method and the ELL-COO format of the storing the sparse matrix. Then they perform vector multiply in order to extract more performance on both the fronts. Of course their method relies on an initial sweep on the matrix to find out the number of non-zero elements and the decision to divide the matrix into two different formats for storage. They suggest a dynamic programming approach to calculate optimal selection of blocks and also an heuristic approach based on greedy block selection.

The CUDA library can also be enriched with CUDAPP [Harris, Sengupta, Owens, Tseng, Zhang, and Davidson, 2009] which provides a routine *cudppSparseMatrix*, for sparse matrix vector multiply routine which comes in handy when solving through iterative methods. To use a method the user first declares a *Plan* in which he/she specifies the input output arrays, the number of elements etc.

[M. Baskaran and Bordawekar, 2008] demonstrate improvements over the methods discussed above ([Bell and Garland, 2008] [Harris, Sengupta, Owens, Tseng, Zhang, and Davidson, 2009]) by exploiting some of the architectural optimizations to the Sparse Matrix-Vector Multiplication code. In particular they center the optimization efforts on the following four points:

- Exploiting Synchronization-Free Parallelism,
- Optimized Thread Mapping,
- Aligned Global Memory Access;
- Data-Reuse.

5.2 Conjugate Gradient

[Georgescu and Okuda, 2007] discuss how conjugate gradient methods could be aligned to the GPU architecture. They also discuss the problems with precision and implementing preconditioners to accelerate convergence. In particular they state that for double precision calculations problems having condition numbers less than 10^5 may converge and give a speed-up also. They however warn that above a threshold value of the condition number the Conjugate Gradient Method will not converge. This last observation relates to the limited double precision performance available on the GPU.

[Buatois, Caumon, and Levy, 2009] discuss their findings on implementing single precision iterative solvers on the GPU and show that for Jacobi preconditioning and a limited number of iterations the GPU is able to provide a solution of comparable accuracy but as the iterations increase the precision drops in comparison to the CPU. They use the Conjugate Gradient method exploiting some of the techniques like register blocking, vectorization and the Block Compressed Row storage(BCRS) to extract parallel performance on the GPU.

They try to maximize the throughput for the memory transactions by using 4×4 blocks in the BCRS format. This format later proves beneficial for strip mining operations. Using such an arrangement complemented with the vector data types available on the GPU (for example $float4$ that allows storing 4 32-bit floats to be stored at one index). By making arrays of such aggregate data types one can access data in chunks thereby saving address calculation. For e.g. in the case of a 4×4 block storage as shown in Figure 8, an array of 4 $float4$'s can be used and accessing all elements is possible with a single address that of the array.

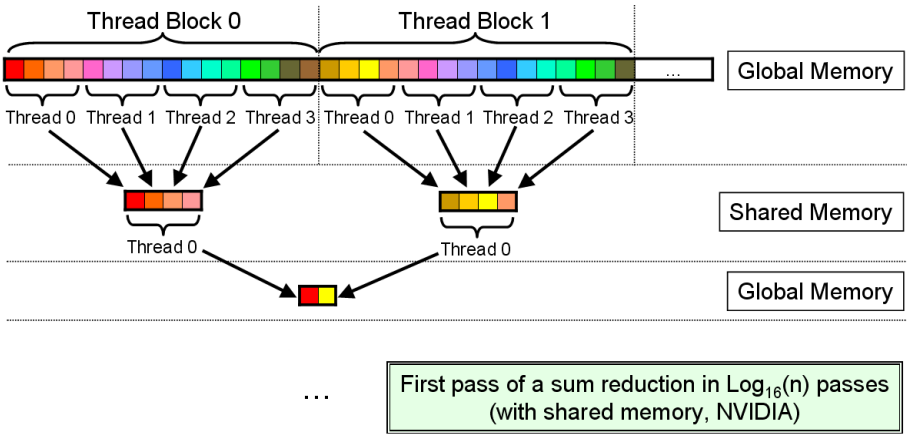

Figure 8: Sum reduction on a Scalar architecture with shared Memory

Further they suggest that individual elements within an aggregate data type could be allocated to multiple threads and such a pattern could be followed among the other elements of this 4 $float4$ element array. Thereby providing speedups in reduction operations. An important finding that is indicated in their results is that reordering of the type Cuthill-McKee did not show any influence on the implementations they executed for the GPU and the CPU.

5.3 Preconditioning

Techniques that are basically dependent on the Sparse Matrix Vector Multiply discussed in previous sections have been suggested in literature for accelerating Preconditioning of Iterative Solvers like GMRES and Conjugate Gradient. [Wang, Klie, Parashar, and Sudan, 2009] use an ILU Block Preconditioner, which has poor convergence qualities but is easier

to parallelize, for solving a sparse linear system by the GMRES method. Coefficient matrix

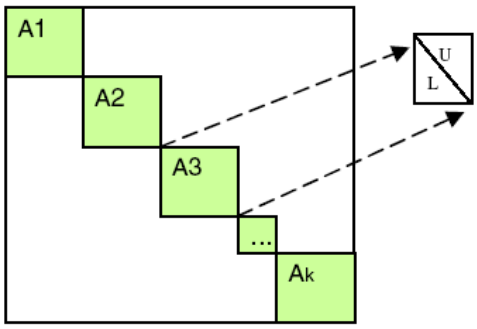

Figure 9: Block ILU preconditioner

A is divided into equal sized sub-matrices which are then locally decomposed using ILU, as shown in Figure 9. The blocks shown in Figure 9 do not communicate to each other during the decomposition and also in solving it, this scheme fits well in the data parallel paradigm. A stream now is a collection of sub-matrices along the main diagonal.

In work published by [Asgasri and Tate, 2009] they discuss how the use of a Chebyshev polynomial based preconditioner could be utilized for achieving speedups in the Conjugate Gradient method for solving a linear system. The said preconditioner effectively reduces the condition number of the coefficient matrix thereby achieving convergence quickly. It approximates the inverse of the coefficient matrix with linear combinations of matrix-valued Chebyshev polynomials. This method uses only matrix multiplication and addition to compute the approximate inverse of the coefficient matrix, which makes it suitable for parallel platforms. In the implementation described in the paper, they use linear combinations of the first few Chebyshev polynomials to build a preconditioner. The combination of Chebyshev preconditioner and Krylov subspace linear solver leads to a highly efficient solver on parallel platforms. It must noted however that Chebyshev preconditioner is useful only if inner products are very important.

[Ament, Knittel, Weiskopf, and Straβer, 2010] suggest a new kind of preconditioning called the Incomplete Poisson Preconditioning. We have discussed it previously in Section 3.2.2. This preconditioner introduces some fill-in (other than the normal sparsity pattern of A) in the multiplication of K with K^T. Their experiments suggest that it could be dropped due to it's comparatively small effect on the preconditioning process. The convergence might suffer, however the method then maps well to the GPU. Hence the incomplete nature of the stencil that emerges gives the method it's name.

5.4 Precision Improvement

The GPUs can suffer a substantial performance hit if considered for Double Precision Computing. If an application can deal with the single precision math on the GPU it is viable. However, if double precision accuracy is desired then the algorithm has to be balanced and some steps could be done in double precision and others in single precision.

[Baboulin, Buttari, Dongarra, Kurzak, Langou, Langou, Luszczek, and Tomov, 2008] suggest that it is possible to use double precision calculations for some part of the iterative method and use single precision for others thereby achieving a trade off that meets precision criteria and converges as good as the double precision case. At the same time the rate of convergence is also not affected very much. They use it for for direct and iterative solution methods for sparse systems where in they pose the problem as the refinement of

the solution x_{i+1} which can be written as:

$$x_{i+1} = x_i + M(b - Ax_i), \tag{98}$$

where M is the preconditioner and approximates S^{-1}. If we use right preconditioning then the system $Ax = b$ reduces to the following

$$AMu = b, \tag{99}$$

$$x = Mu \tag{100}$$

Further they suggest calculating M using an iterative method and for the solution of the original system a Krylov Subspace method is employed. This system of Inner and Outer Iteration is now ready for Mixed Precision use. The idea here is that a single precision arithmetic matrix-vector product is used as a fast approximation of the double precision operator in the inner iterative solver. They have reported the results for a non-symmetric solver wherein the outer iteration is of a FMGRES and the inner one is a GMRES cycle.

5.5 Other Important Approaches

It is also possible to use multiple GPUs to boost the performance of an application. Multiple GPUs stacked through the NVIDIA SLI interface or across workstations can collaborate via MPI. [Ament, Knittel, Weiskopf, and Straβer, 2010] use a redundant buffer based scheme coupled with the availability of overlapping computation with memory transfers through the use of *streams* on the GPU. They divide the 3-D domains into tiles and each GPU handles a subset of the tiles. This necessitates the inter-GPU communication when the boundary layers are being updated in an iteration. They use synchronization primitives available on the GPU and the CPU for handling boundary cases.

6 Implementation

For our experiments we use a single core of a quad core Intel CPU (Q9650) running at 3.0 GHz and having 12 Mb of L2 cache. The GPU we use is a Tesla machine with 30 Streaming Multi-Processors running at 1.3 GHz and a memory bandwidth of 102 GB/sec. We first wrote the the Deflated Preconditioned CG Method on the host(CPU). An important consideration while writing the code was to make it modular so that it could be analyzed and optimized step-by-step. For this the following points were kept in mind.

1. Identifying Kernels of Computation.
2. Organizing code in form of kernels.
3. Prioritized Optimization of Kernels after subjecting the code to the profiler.

On the CPU we have used the Meschach BLAS Library for Dot Products and Saxpys. The kernels that were hand-coded are

1. Sparse Matrix Vector Multiply Kernel
2. Preconditioning Kernel(s)
3. Deflation Kernels

We (primarily) ran the kernels on grid sizes with 16000 to $260,000$ unknowns. Also we subjected these to Block sizes varying from 256 to $4,096$ unknowns per block. As a final step deflation vectors were varied between 256 and $4,096$ as well. After the results were in line with the expectations which means

1. No. of Iterations decrease with Block-IC Preconditioning compared to plain Conjugate Gradient,

2. With increasing block sizes the number of Iterations decrease further, and

3. With increasing number of deflation vectors number of iterations further decrease.

We proceeded with implementing the algorithm on the GPU. On the GPU the *CUBLAS* library provided some useful functions for saxpy, dot products which we have used.

6.1 SpMV Kernel

Our matrix has a regular pattern that of a 5-point Laplacean Matrix in two dimensions. So there are 5 diagonals which carry the complete matrix. It is important to mention here that the matrix represents a grid $n \times n$ in size and hence the matrix has $N = n \times n$ rows. The storage format that we choose is called the Diagonal Storage format. All the diagonals are stored in a 1-D array, starting from the lowest sub-diagonal(with offset $-n$) followed by sub-diagonal with offset (-1), then the main diagonal and then the two super-diagonals with positive offsets. Also an important feature is that they all have the same length. This kind of uniformity of size makes coalesced access possible. So for example, if say the sub-diagonal with offset -1 has one element less, then at that position a zero fills in to make it equal in size to the main diagonal.

Once stored in this way for each row of the matrix we have 5 fetches from the array holding the 5 diagonals and 5 from the vector. On the GPU we assign one thread to compute one element of the resulting Matrix-Vector Product. Additional optimizations include using shared memory and texture memory. The $offsets$ array is accessed by every thread and hence we store it on the shared memory to optimize the SpMV Kernel. This gives a very coalesced access pattern since threads in a half warp access contiguous elements in the array albeit each of those 5 accesses(per thread) are at offsets of n elements.

6.2 Preconditioning Kernel

The preconditioning kernel is the most sequential part of the entire algorithm. We initially begin with the Block Incomplete Cholesky Preconditioning. The Block Variant of Incomplete Cholesky Preconditioning basically exposes the parallelism at the block level. However each block has considerable serial amount of work to be done. This includes fetching three times block size number of elements from the array holding the preconditioner (this array is stored in a similar fashion as the coefficient matrix A) and also fetching block size number of elements from the right-hand side vector. The computation per block is completely serial. A subsequent row needs a value from the previous row.

One technique that we have employed is to break down the steps of preconditioning into three.

1. Forward Substitution

2. Diagonal Scaling.

3. Back Substitution

The second step of diagonal scaling can be heavily optimized using shared memory. This is possible since it is inherently parallel with two reads every thread followed by one multiplication and all the calculations(N) are independent. For the first and the final steps we can also use shared memory. The trick is to load the elements using a number of threads(number same as the block size) in parallel and then work on them and store

them back in global memory. Later in the development process we used Incomplete Poisson Preconditioning to maximize benefits of parallelism. It has been discussed earlier in Section 3.2.2.

6.3 Deflation Kernels

For deflation we sub-divide the tasks into a couple of kernels at the outset. Namely,

1. Calculate $b = Z^T \mathrm{x}$
2. Calculate Matrix-Vector Product of E^{-1} with b.
3. Calculate Matrix-Vector Product of AZ with the result of the previous step and subtract from x.

For the first kernel $b = Z^T \mathrm{x}$ we have used the parallel sum approach. For the other two kernels it is useful to tailor the matrix multiplication example and use shared memory instead. This is better than the *cublasSegmv*(for some grid sizes) since we do not have an additional vector scaling and addition as required by *cublasSgemv*.

The decision to calculate E^{-1} explicitly is instrumental since it greatly reduces the time for the iterations. Though the setup time for the algorithm is affected but the overall gain in the running time of the method more than compensate the costly operation. Although it is not sparse and we understand that if the number of deflation vectors become very high then this approach might not be very efficient.

For the calculation of $AZ\times$ result of $E^{-1} \times b$ we used the *cublasSgemv* call. The final calculation $x_{it} = Qb + P^T x$ can also utilize the kernels discussed here and also the *cublasSgemv*. In the later stages of development we optimize the storage of AZ and re-write the kernels for calculations involving AZ.

7 Optimizations and Results

In this section we list the data resulting from the experiments conducted. We start with simple Conjugate Gradient Method and successively add Preconditioning and Deflation to it. First we present results from preliminary versions with minimal to no optimizations present. Further we apply optimizations and discuss the approaches. In all the results below we work with a Poisson type Matrix with a 5-point Laplacean stencil. Hence it has 5 diagonals and is a regular sparse matrix.

7.1 Conjugate Gradient - Vanilla Version

As a first step we have implemented Conjugate Gradient(CG) on the CPU as well as on the GPU. For the GPU we only have one kernel, namely, the Sparse Matrix Vector Product Kernel. On the host side also we use a similar kernel. Albeit, it runs serially for all rows one by one. We now provide a brief Code Commentary that outlines our code design.

7.1.1 Code Commentary

We use the Conjugate Gradient iteration as discussed in Algorithm 5. The convergence condition being that the $2 - norm$ of the residual at the $k-$th step with respect to the original residual r_0 is minimized to a tolerance ϵ or less.

$$\frac{\| r_k \|_2}{\| r_0 \|_2} \leq \epsilon. \tag{101}$$

For the the CG code we have kept the tolerance at 10^{-6}. The matrix A is kept in the form of diagonals and another matrix named $offsets$ notes the offsets of these diagonals from the main diagonal. Since for a $5-$point Laplacean in $2-D$ we have 5 diagonals and it is symmetric. So the offsets are $-n, -1, 0, 1, n$ where grid is $n \times n$ and the offset for main diagonal is 0. Even though it is intuitive to use 3 diagonals for storage we use 5 since that makes the code simpler and on the GPU there is an added advantage of coalesced access that such a storage pattern provides.

As previously discussed the diagonals are of the same length $N = n \times n$ which is the number of unknowns for an $n \times n$ grid. This is useful when accessing these elements in the GPU since then each thread picks one element from each of the arrays and the memory access becomes regular. This is explained in a simple example in [Bell and Garland, 2008].

The host code utilizes the Meschach library. It is an optimized BLAS library. The reason we chose Meschach lies more in ease of use than anything else. An implementation of a more known library like gotoBLAS could be also interesting to compare. Another option is to use Intel's MKL library. All these Libraries provide functions for Dot Products, Saxpy, scaling and norm operations.

Sparse Matrix-Vector Products were done by summing up the product of elements of each row(of the matrix) with the corresponding vector element. This was repeated for all rows of the array inside a for loop.

For the GPU we mostly used $CUBLAS$ as provided by $NVIDIA$. The dot product was implemented using the *cublasSdot* and the saxpy similarly using *cublasSaxpy*. Also useful were the *cublasScal* for doing scaling and *cublasSnrm2* for calculation of $2-norm$. We developed three different flavors(versions) for Sparse Matrix Vector Product Kernels on the GPU

1. Vectorization - we simply copied the kernel from the Host and removed the for loop. Instead now each row is calculated by a separate thread.

2. Vectorization with the offsets array stored in the Shared Memory.

3. Vectorization with the offsets array and x stored in the Shared Memory.

Putting the offsets array in shared memory gave us a significant performance boost. Since the offsets array is accessed by every element in every row so it was imperative to put it in shared memory.

Throughout the inner loop for CG (steps 2 to 8 of Algorithm 5) there is no transfer of information between the host and the device (except for transferring results of dot-products from the GPU to the host). On basis of the parameter $OPT\_THREADS$ the optimum number of threads [usually 256 but can be 512 for bigger grid sizes] are assigned first and the block size is calculated as $\frac{N}{OPT\_THREADS}$. If the block size is more than that supported by the device then the number of threads is increased and the number of blocks is updated accordingly.

We initially tested this implementation with three different storage formats(CSR,DIA and Matrix-Free). Details about the CSR and DIA format can be found in [Bell and Garland, 2008]. The matrix free format eliminates use of any matrices to store the coefficient array A. Instead all operations on the matrix can be specifically coded. For e.g. for the 5-point Laplacean matrix the diagonal matrix $diag(A)$ is 4 repeated N times. Similarly for other diagonals similar rules can be defined and operations can be written without any storage space for the matrix.

The DIA format gives us the best results in terms of speedup so we performed comparison of the SpMV kernels on DIA format only. We chose the DIA structure to move ahead since CSR is not well suited for our kind of matrix and matrix-free is too specific.

As previously mentioned we have largely relied on $CUBLAS$ and shared memory based optimizations. Although texture memory could be useful when an array is accessed in an unordered way we have not used it for x since x changes in our application. Texture memory could be useful for smaller sizes of A but since it cannot be useful for larger grid sizes we have not used it. We tried it both for storing the coefficient matrix A and the vector x in separate versions of the code.

7.1.2 Comparisons with GPU versions

In Figure 10 we show a comparison of the Conjugate Gradient Algorithm's run time on the Host and Device across three Grid sizes.

1. 128×128, 16384 unknowns (16,000)
2. 256×256, 65536 unknowns (65,000)
3. 512×512, 262144 unknowns (260,000)
4. 1024×1024, 1048576 unknowns (1,000,000)
5. 2048×2048, 4194304 unknowns (4,000,000)
6. 4096×4096, 16777216 unknowns (16,000,000)

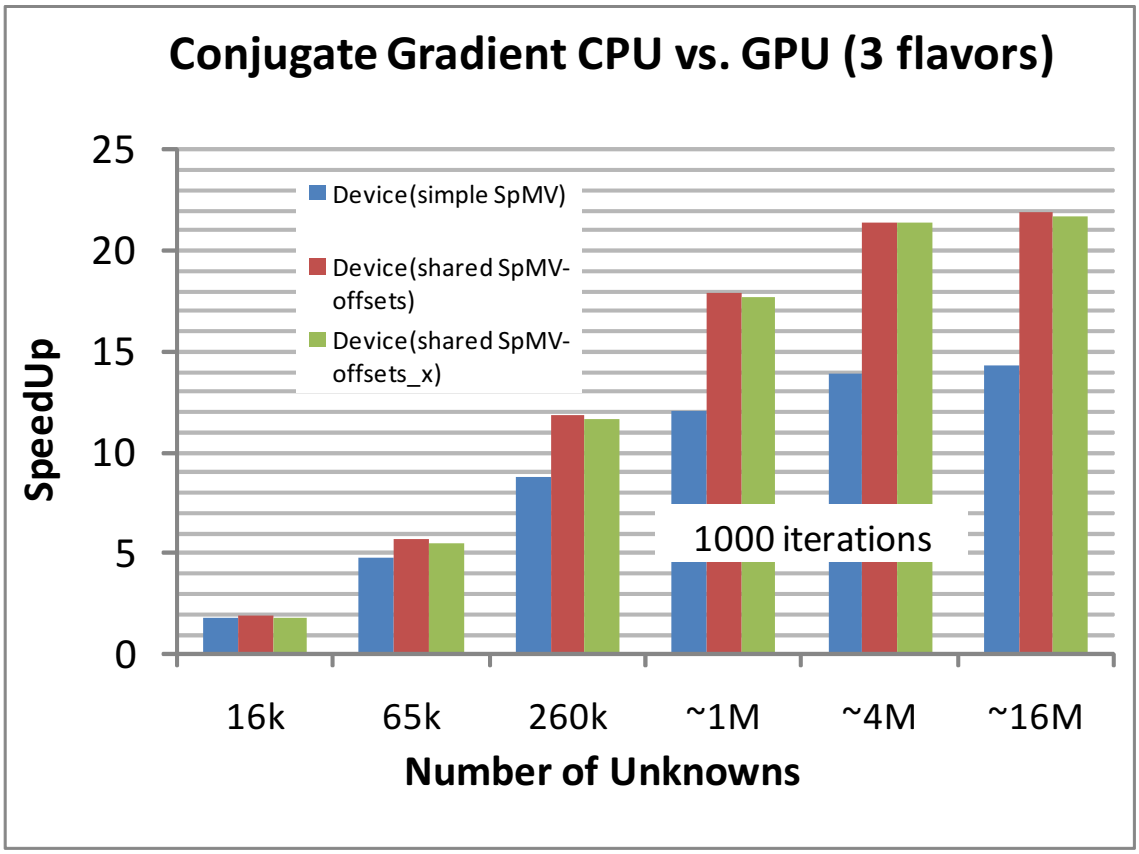

Figure 10: Conjugate Gradient on Host vs Device

Before we delve into the result analysis it is important to talk about divergence on the GPU. Divergence could be misleading as divergence of the CG method. However divergence in terms of CUDA means presence of an $If - then - else$ construct. This

retards the throughput of the kernel since it forces all the threads in a block to first take the if path and then take the $else$ path. In our case, suppose each block has 256 threads. Now since the offset array has five entries in it $-n, -1, 0, 1, n$ five threads are needed to load it in parallel. However the rest of the 251 threads have nothing to do which wastes some of the device time. This phenomena is termed as divergence on the GPU. The second flavor of our tests as listed in 7.1.1 has this kind of divergence in it however the effects of divergence are more than compensated by the reduction in repeated global memory access.

Looking at the results in Figure 10, we get a speed-up of around 21 times (for the largest grid size)for the shared memory kernel which only caches offsets. The version with the vector x also in the shared memory comes a close second and that is because of the divergence that such a design has. It has to access some elements of x from the shared memory and some from the global memory. So the advantage of shared memory is greatly reduced. For this reason we choose the variant of SpMV kernel that only utilizes shared memory for storing offsets only.

It must be noted that here for sizes greater than $260k$ we haven't run the method up till convergence as the code stops at 1000 iterations.

7.1.3 Profiler Picture

The SpMV kernel takes the maximum amount of time for the execution as far as GPU execution is concerned. The shared memory tweak of loading the offsets though has divergence but it reduces the amount of GPU time SpMV kernel takes by half. In Figure 11 and Figure 12 this can be observed. It can also be seen that as the number of unknowns increases, more and more time is consumed by these three kernels of SpMV, saxpy and sdot.

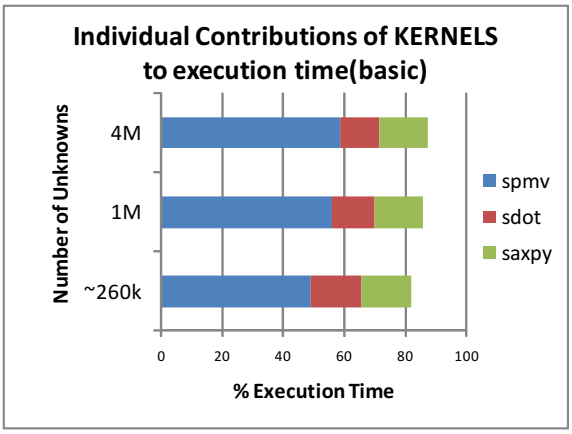

Figure 11: Conjugate Gradient with basic kernel

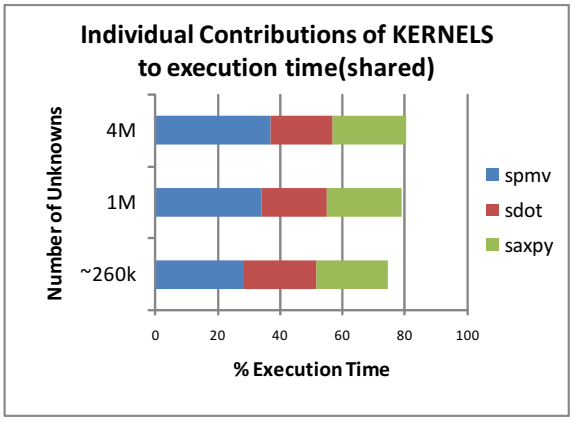

Figure 12: Conjugate Gradient with shared memory kernel

From Figure 10 it can be seen that at the size of 16M the speedup seems to become constant. The reason for this is that the kernels on the GPU become bandwidth-bound. The $CUBLAS$ calls are taking around 85Gb/s of the available bandwidth. The only hand-written kernel of SpMV is consuming more than 85 Gb/s. This kernel is at 100% occupancy. The rest of the calls are to $CUBLAS$ functions and they are also at 100% occupancy.

7.2 Diagonal Preconditioning

Since our matrix for now is a simple 5-point Laplacean matrix (with similar elements on the diagonal) we did not expect any changes and our results stayed the same. The preconditioner step only involved scaling the main diagonal by $\frac{1}{4}$ which was rather trivial to implement using a *cublasSscal* call. On the host-side this is implemented by a scalar-vector multiplication routine.

7.3 Conjugate Gradient with Preconditioning

7.3.1 Code Commentary

Extending CG with the Incomplete Cholesky Factorization based preconditioning is the next step in our implementation. For this another kernel comes in place for the operation

$$z_j = M^{-1} r_j \tag{102}$$

where,

$$M = KK^T \tag{103}$$

and K is the (incomplete) lower Cholesky triangle. In order to keep it "incomplete" K follows the sparsity pattern of A. We solve the system $KK^T z_j = r_j$ in three steps in the following way:

1. $Ky = r_j$ Forward Substitution
2. $y = y * diag(K)$ Diagonal Scaling
3. $K^T z_j = y$ Back Substitution

This step occurs inside the iteration as well, as shown on line number 6 of the Preconditioned Conjugate Gradient Iteration listing in Algorithm 7.

It must be noted that block sizes are always at least double the number of grid points in one direction. This is important since if they are between the grid dimension in say, the X direction, and $2\times$ that size then they do not really suit the GPU (multiples of 32, 16 only suit) and if they are less than the grid size in any one dimension they completely remove the outer most diagonals in A. Recalling that A resembles the 5-point Laplacean in cases of our interest. This is better explained by a picture given in the Appendix B.2.

We have followed these rules for the host also for a similar comparison. As can be noticed in Figure 13 preconditioning reduces the number of iterations required for convergence drastically. For grid sizes $4M$ and above the convergence does not occur within 1000 iterations. However the norm of the result at the 1000th iteration decreases with bigger preconditioner block sizes. This can be observed in Figure 14

7.3.2 Comparisons with GPU versions

Comparing the CPU version of the code in Figure 15 to 20 with the GPU versions we see that for smaller grid sizes the GPU version lags behind the CPU. However some gain is induced for bigger grid sizes. This could be partly explained by the relatively large block sizes which might become overwhelming for the closest (and the fastest) level of cache inside the CPU. Also from these figures we can see that as the block sizes increase (and number of blocks decrease) the CPU gains a bit and this is because the number of iterations in the for loop is less, so less jumps in the CPU code hence better performance.

The most simple version of the kernel code for preconditioning would have been putting all the three steps (forward substitution, diagonal scaling and backward substitution) in

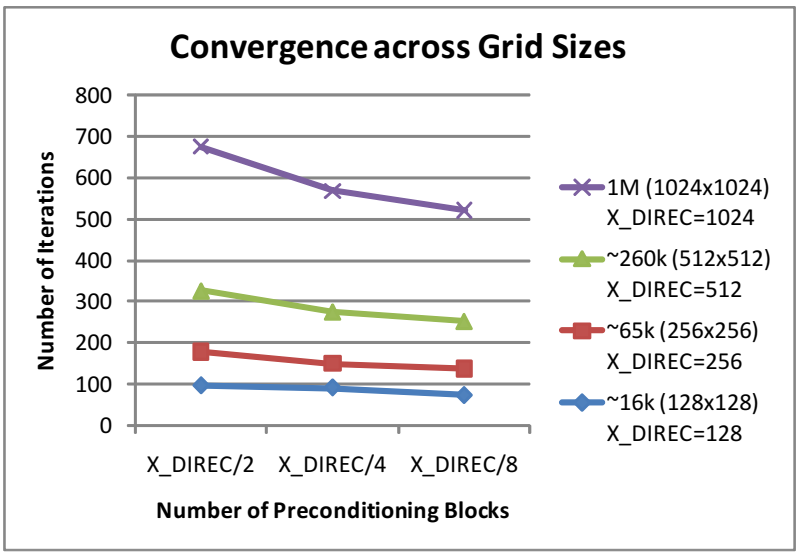

Figure 13: Decreasing iterations across block sizes (CG with Block-IC Preconditioning)

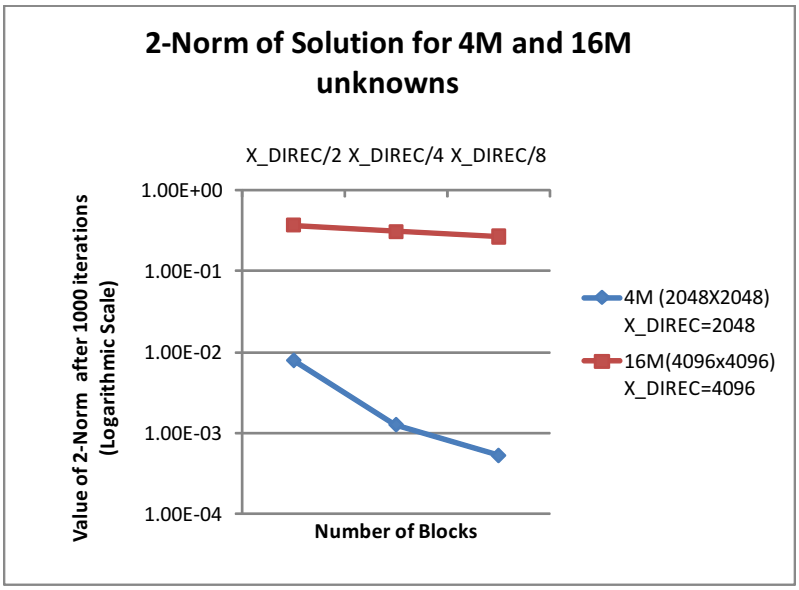

Figure 14: Norm decreasing with decrease in number of blocks (CG with Block-IC Preconditioning)

one kernel but that takes up a lot of registers. So we broke them down in three parts. Out of these three parts diagonal scaling is embarrassingly parallel so it consumes less than 1% of the total time required for preconditioning but forward substitution and backward substitution are inherently sequential and not suited for the GPU. However computing them on the CPU every cycle and then transferring back and forth between the GPU-CPU is not a good solution either.

Hence as much as Block-IC permits we used shared memory for temporary calculation of the new values in each block and once the calculations are complete we copy them to the global memory. This means that for each block of the Block-IC method we store 'size of block' number of floats in which we store the result $y = K^{-1}r_j$ for the back substitution and the forward substitution steps. This seems natural since in the implementation of forward and back substitution y is written at most three times in the whole iteration that spans across a block, so keeping it in shared memory and writing only the final value improves performance.

As promising as this approach sounds it fails as soon as block sizes approach 4096 which happens for a grid size 1024×1024 in our experiments. Then we have to come back to the naive kernel that launches one thread per block that calculates the new vector per block.

This approach also might loose on bigger grid sizes (if and when we have that much shared memory) since then a lot of copying will have to be done by a single thread from shared memory to global memory.

We also tried using as many threads in a block as there were elements in the block and tried to load all elements in parallel and store them back in parallel but the intermediate step of calculating all elements of y was done by a single thread. This version also fails for the same reason as the previous approach does. There can be a maximum of 512 threads per block so beyond that it is not possible to use this idea of co-operative loading and writing back.

An even more aggressive version (using shared memory) for the forward and back substitution steps is to store the matrix K's three diagonals also in shared memory, though it becomes unusable even faster than the previous approach.

Preconditioning of the Block-Incomplete Cholesky type is best when there are more and more blocks that can all be computed in parallel. However as the block sizes increase the amount of the time that GPU is under-utilized also increases. Since inside the forward/back substitution kernels only one thread works per block which is wasteful for the GPU.

Figure 15 to 20 show how how speedup slowly crawls up to 2.5 times from a fractional level for large grid sizes. In these graphs the blue color shows the kernel in which shared memory can be used for storing the intermediate result y and the green color shows the naive kernel which *has* to be used after a certain point since shared memory is limited to 16384 bytes or 4096 single precision floats per block.

7.3.3 Profiler Picture

Preconditioning dominates more than 90% of the execution time (across all grid sizes and preconditioning block sizes) in on the device versions of the code. This was expected because of the serial nature of the code. Saxpy, Sdot and SpMV come a close second. On the host the majority of the time is taken up by the SpMV operation and the Block-IC Preconditoning.

On the host this percentage rises much more slowly as compared to the device. Since for the device(GPU) serial code is not suited at all.

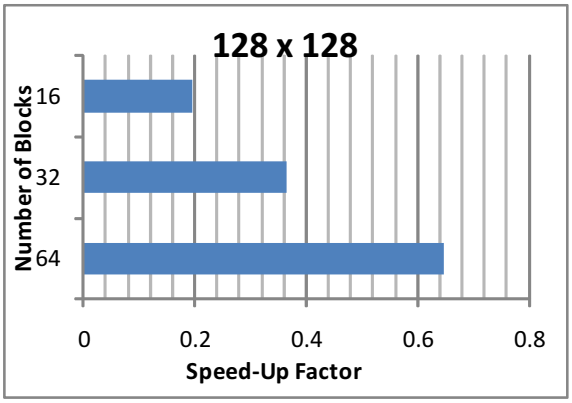

Figure 15: SpeedUp across block sizes for 16k unknowns

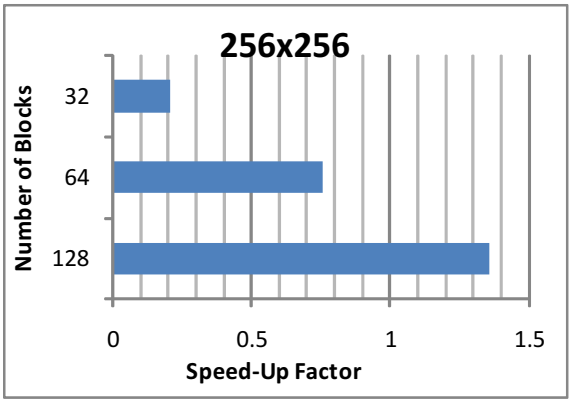

Figure 16: SpeedUp across block sizes for 65k unknowns

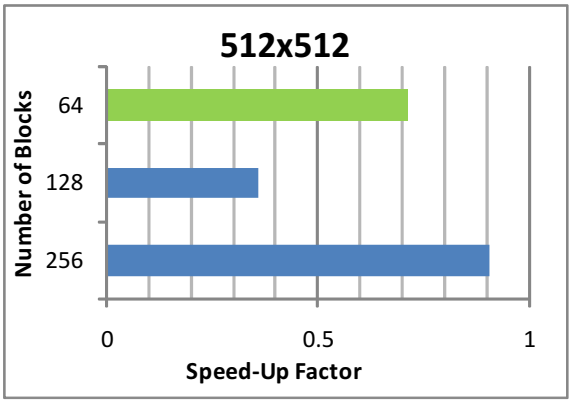

Figure 17: SpeedUp across block sizes for 260k unknowns

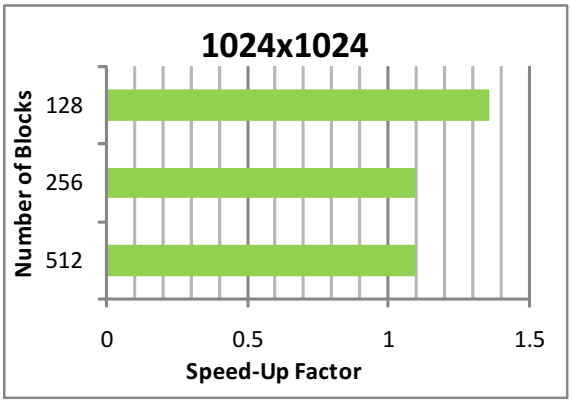

Figure 18: SpeedUp across block sizes for 1M unknowns

7.4 Conjugate Gradient with Deflation and Preconditioning-Block IC

Deflation is applied to the existing preconditioned iteration. Deflation provides much more scope for exploiting parallelism and that will be evident with the results discussed henceforth. First we put the optimized kernels on the device, then we added change the way we calculate and store AZ. While improving the performance of the code on the GPU we also gave equal attention to optimization of the CPU code.

7.4.1 Code Commentary

We implemented deflation on the Block Incomplete Cholesky version of the code discussed in the previous section. Sub-domains were introduced in the grid which could be handled by separate processors. The Sub-domains have been assigned stripe-wise (for details refer Appendix B.4) so that in the matrix a set of rows (one domain) could be assigned to one processor. The new operations required for Deflated Preconditioned Conjugate Gradient can be written down in separate kernels.

1. Calculation of AZ (happens once).
2. Calculation of E (happens once).
3. Calculation of E^{-1} from E (happens once).
4. Calculation of $m = Z^T x$ (happens every iteration).

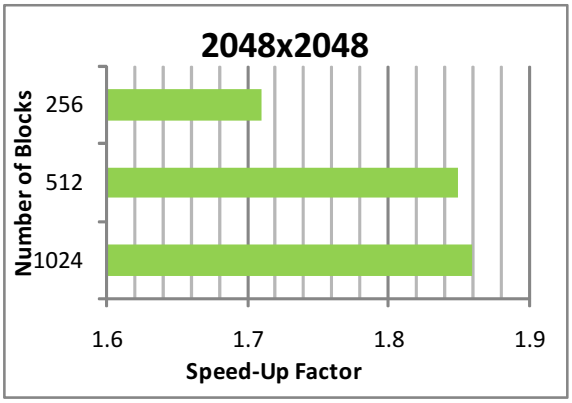

Figure 19: SpeedUp across block sizes for 2M unknowns

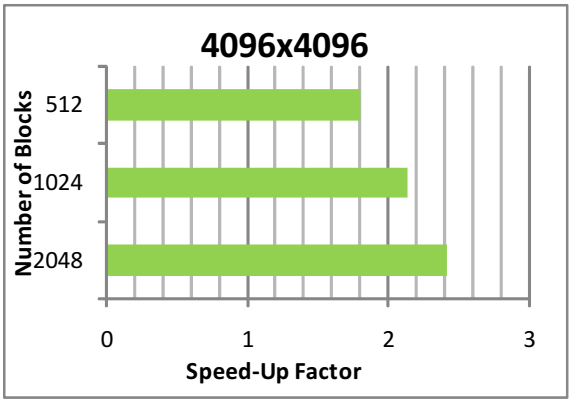

Figure 20: SpeedUp across block sizes for 4M unknowns

5. Calculation of $y = E^{-1}m$ (happens every iteration).
6. Calculation of result$= x - AZy$ (happens every iteration).
7. Final Update of x, the solution with Qb and $P^T x$ (happens once).

Now AZ, E^{-1} can be calculated on the CPU itself. Since the setup of the kernel is not included in our analysis of speedups. This part is common to both the host and the device kernels.

After we have calculated E^{-1} the 5^{th} step in the list above is a Dense Matrix Vector Multiplication. Similarly the 6^{th} step also since we store AZ as a full matrix of dimensions $N \times d$ where N is the number of unknowns (also $N = n \times n$, where n is grid size in one dimension) and d is the number of deflation vectors.

This limits our experiments to a grid size of a maximum of 512×512 with 4096 domains since at this size AZ alone requires 4 gigabytes of storage which is a limit for the GPU which has only 4GB of Global Memory storage. In order to overcome this limitation, later we store AZ in a format where only $5 \times N$ storage is required.

For doing Dense Matrix Vector Multiplication we use the $CublasSgemv$ function. For the 4^{th} step, $Z^T x$, we write a simple kernel that sums up N/d elements in each thread.

For the final step which involves correcting the solution x. We have to do $AZ^T x$ which can again be computed using a $CublasSgemv$ call. Also we have to calculate $E^{-1}AZ^T x$ and $E^{-1}Z^T b$.

7.4.2 Comparisons with GPU versions

Figure 21 to 23 present the speedup on three different grid sizes and across three different numbers of deflation vectors and three different numbers of preconditioning blocks.

Detailed results with Error norms for device and host and the number of iterations are available in the Appendix C.1.1. A couple of things are worth noting in these results.

- The number of iterations decrease with increasing block size for any number of deflation vectors.
- Increasing the number of deflation vectors decreases the number iterations required for convergence.
- The relative error norm is same across the device and the host

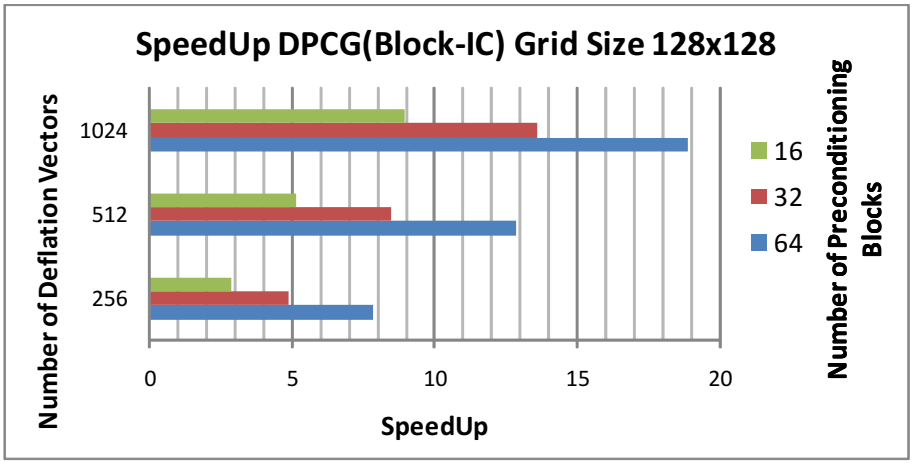

Figure 21: Deflated Block-IC Preconditioned Conjugate Gradient (Simple CUDA kernel) 16k unknowns

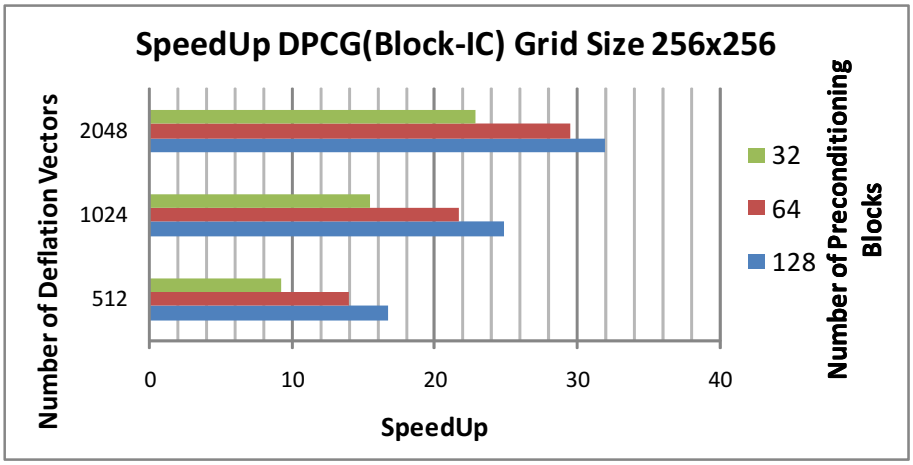

Figure 22: Deflated Block-IC Preconditioned Conjugate Gradient (Simple CUDA kernel) 65k unknowns

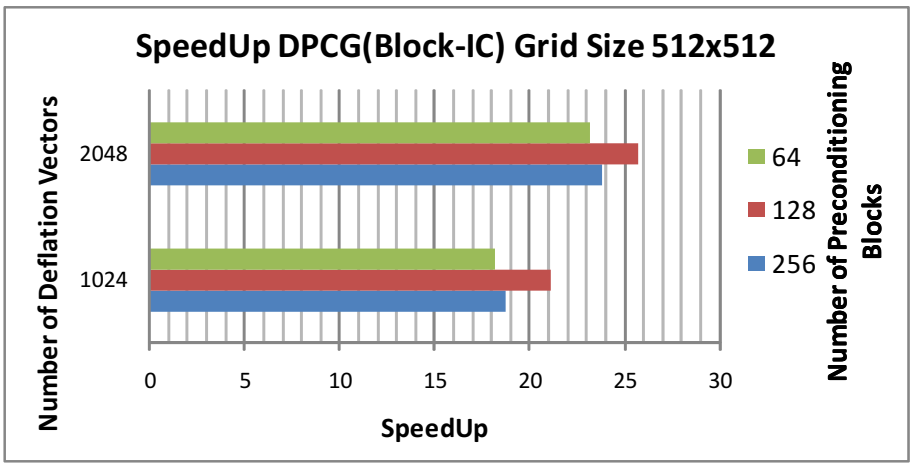

Figure 23: Deflated Block-IC Preconditioned Conjugate Gradient (Simple CUDA kernel) 260k unknowns

- The Speed-Up is maximum for the largest number of deflation vectors and the smallest Block Size (largest number of blocks).

The first three observations confirm that the results are correct. The last observation confirms that as the parallelism increases in the code the usefulness of the GPU appears in the execution times. They are reduced considerably.

However there is one interesting thing to be noted in the speedups for the grid size 512×512 (Figure 23). The maximum speedup is for the number of blocks at 128 and when the number of blocks increases to 256 the speedup declines.

This has something to do with the hardware we have. We use a Tesla C1060 GPU with 240 scalar processors. They are divided amongst 30 Multiprocessors in groups of 8.The preconditioning kernel we use now is the naive kernel, in which the Forward and Backward Substitution kernels form the bulk of the computing time. This remains true for deflated Preconditioned Conjugate Gradient as well as we will discuss in the Profiler Picture section that follows. These kernels (forward and backward substitution) launch as many CUDA blocks (CUDABLOX from hereon for clarity's sake) as there are preconditioning blocks. So 128 CUDABLOX are launched when there are 128 preconditioning blocks. Now in this case we have enough scalar processors to process all of these blocks simultaneously. Since only one thread calculates the whole preconditioning block in both kernels.

However in case of the 256 blocks we have a limitation that first 240 blocks will be scheduled and then once they are done the remaining 16 blocks are scheduled. This can also be verified from the execution times for this particular case. As can be seen in Figure 24 and 25, the execution times rise only on the GPU whereas across block sizes 64 and 128 they are more or less constant. The occupancy in both cases is the same and the bandwidth is more (50%) for the case when the number of blocks is 128. This couldn't be compared in the previous version of the code (Section 7.3) since there we had two different kernels for different block sizes.

7.4.3 Profiler Picture

At this stage we have 24 (3 grid sizes, 3 preconditioning block sizes, 3 different deflation vector counts (2 for the last grid size)) different instances of the solution we are trying to implement so we only display a couple of profiler pictures that make the point clear. Specifically we discuss the case from the 128×128 matrix for 1024 deflation vectors across all preconditioning block sizes and for the grid size 512×512 we do the same.

From Figure 24 we can infer that the calculation of $AZ \times x$ is crucial at all preconditioning block sizes however its contribution is shadowed by the preconditioning kernels (forward substitution and back substitution) as the preconditioning block sizes increases or the number of preconditioning blocks decreases. This is because each thread on the GPU (which calculates one complete block) has to work more and more.

Careful examination of figure 25 , which is for grid size 512×512, we can see that from the number of blocks 128 and 256 there is a slight increase in the preconditioning time which accounts for the decrease in speedup as shown in the earlier in Section 7.4.2.

7.5 Conjugate Gradient with Deflation and Preconditioning - AZ storage optimized

As seen in the previous two sections the $CublasSegmv$ call for calculating $AZ \times x$ is the second most time consuming kernel after preconditioning (considering forward and back substitution as one and ignoring the diagonal scaling step since it is inherently parallel). Keeping the preconditioner same we try to optimize this kernel.

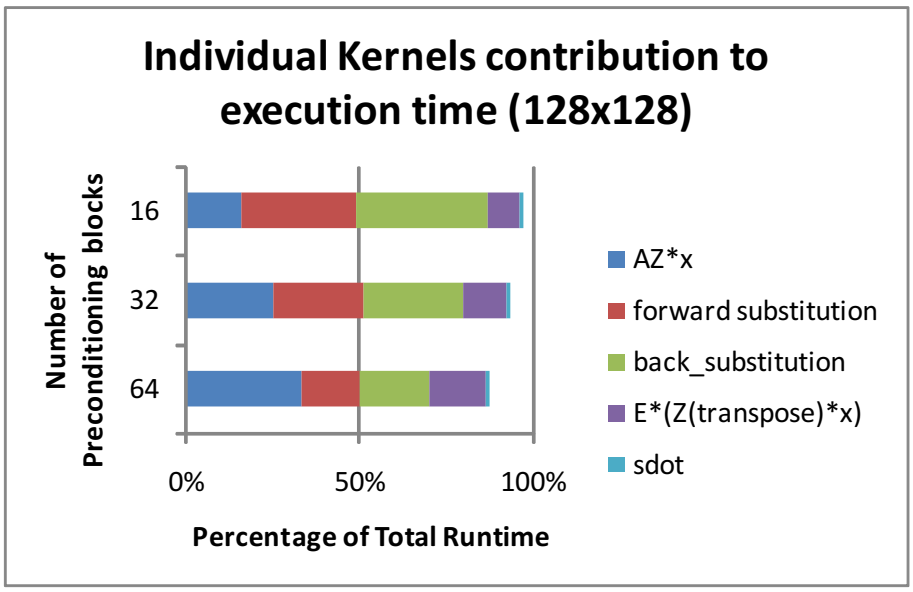

Figure 24: Conjugate Gradient with Deflation and Block-IC Preconditioning (Simple Version) Deflation Vectors=1024

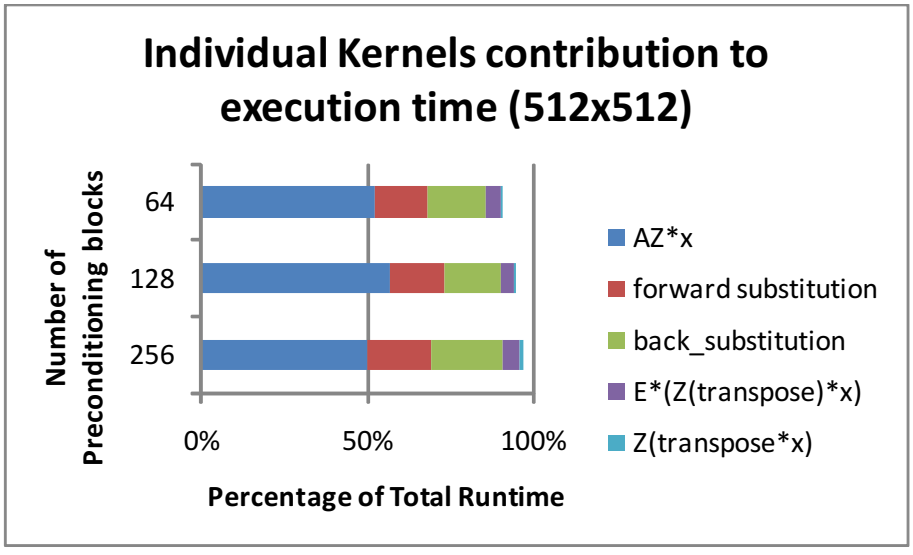

Figure 25: Conjugate Gradient with Deflation and Block-IC Preconditioning (Simple Version) Deflation Vectors=1024

Figure 26: 5 diagonals in AZ. N=64, n=8 and d=16

The reason why this kernel is taking so much time is the order of computations involved. AZ is an $N \times d$ matrix to which a vector of size N is multiplied in this kernel. Now we examine the structure of AZ in Figure 26.

The blue areas form the diagonals with offsets $\pm d/n$ and width N/d per column. The main diagonal is also N/d wide per column. The green areas form the diagonals with offsets ± 1. So the blue areas have in all $N - n$ elements. The red area has N elements since $N/d \times d = N$. The green area has $d - 1$ elements on each side of the main diagonal.

Now to make the storage structure uniform we use a block of $5 * N$ elements. Out of which one chunk of N is assigned to the main diagonal. The diagonals (in blue) with offset $\pm d/n$ can be stored in chunks of N each with padding of zeros before$(-n/d)$ or after$(+n/d)$. For the green areas, i.e. the diagonals with offset ± 1, there is padding required at every element which is a zeros pattern N/d long. This way all diagonals can occupy N spaces and the data structure can now be used. One of the main reasons, other than saving space, for storing AZ like this is also that such an arrangement makes it very

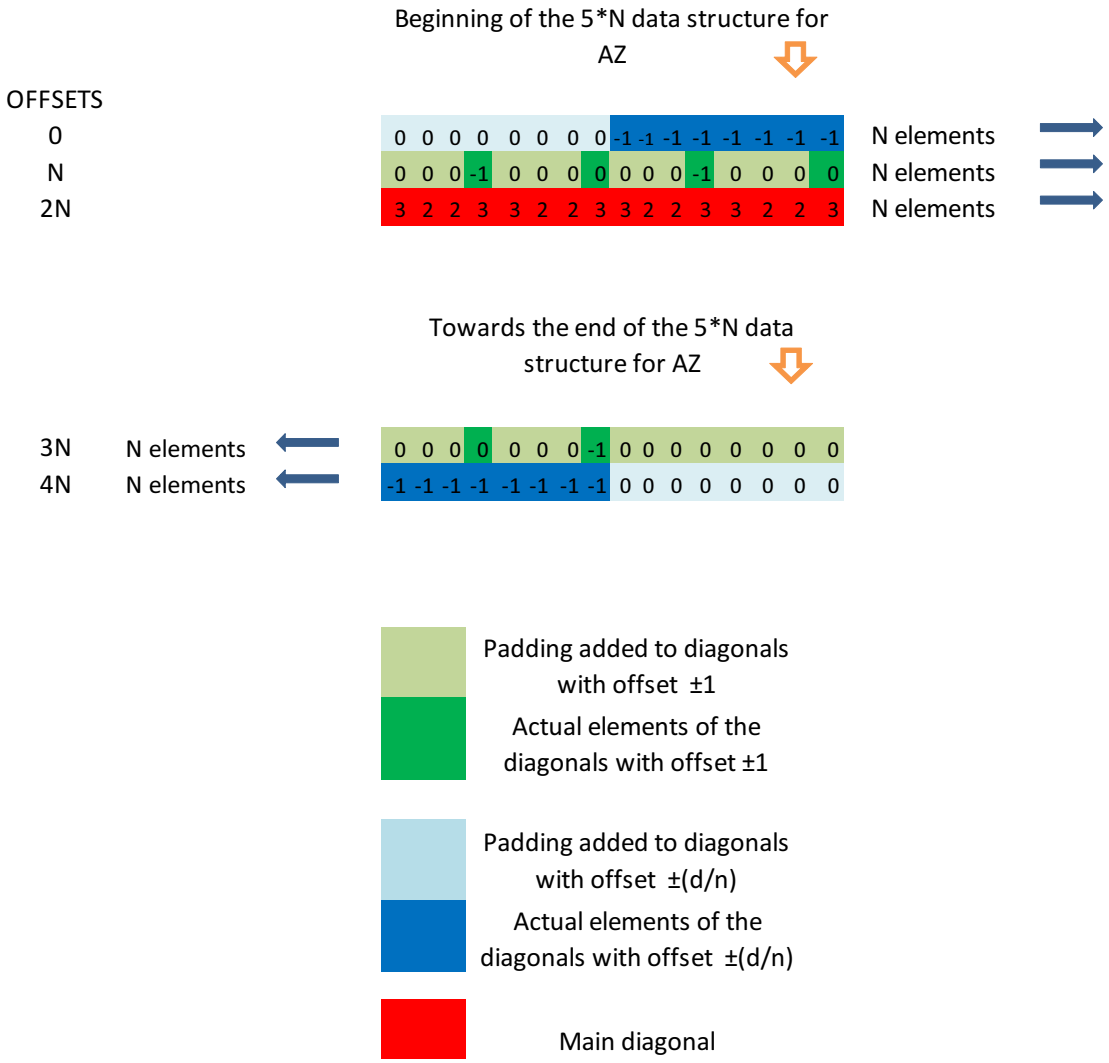

Figure 27: AZ stored as 5 diagonals(padding). N=64, n=8 and d=16.

close to how A is stored and hence promises better performance on the GPU.

After reorganization the AZ data structure looks like Figure 27. Given this organization of AZ we developed new kernels for $AZ \times b$ and $AZ^T \times x$ where b is a $d \times 1$ vector and x is a $N \times 1$ vector. It is important to note here that $AZ \times b$ operation happens every iteration whereas $AZ^T \times x$ happens once at the end of convergence to correct x.

The kernel for $AZ \times b$ is similar to the SpMV kernel we write for A. The kernel for $AZ^T \times x$ works in 5 calls. One call for every diagonal. Since in this case we have to take care of a couple of things like,

1. The row offset of the current column being read from AZ,
2. The actual place in the $5*N$ array in which we store AZ from where we start reading (taking into account padding),
3. The offset of the diagonal in the storage array for AZ. This is in multiples of N; and
4. The offset where we start writing the results in the output array.

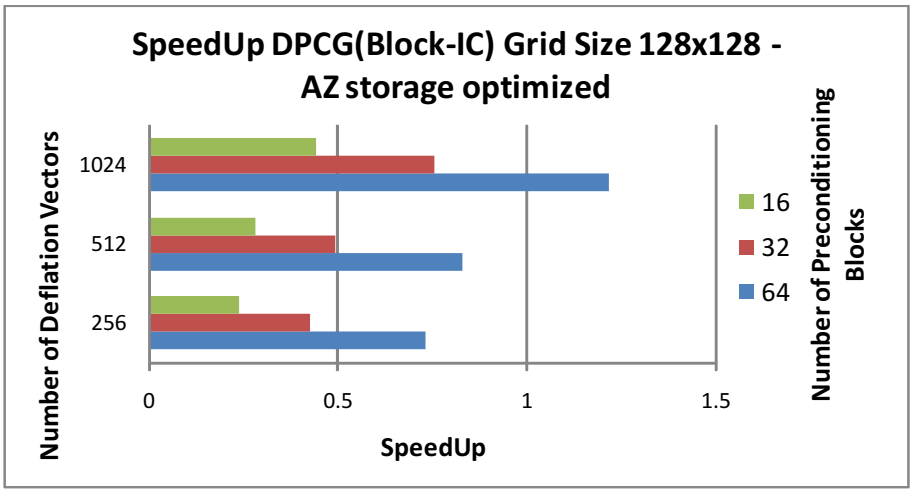

Figure 28: Deflated Conjugate Gradient with Block-IC Preconditioning. 128×128 grid.

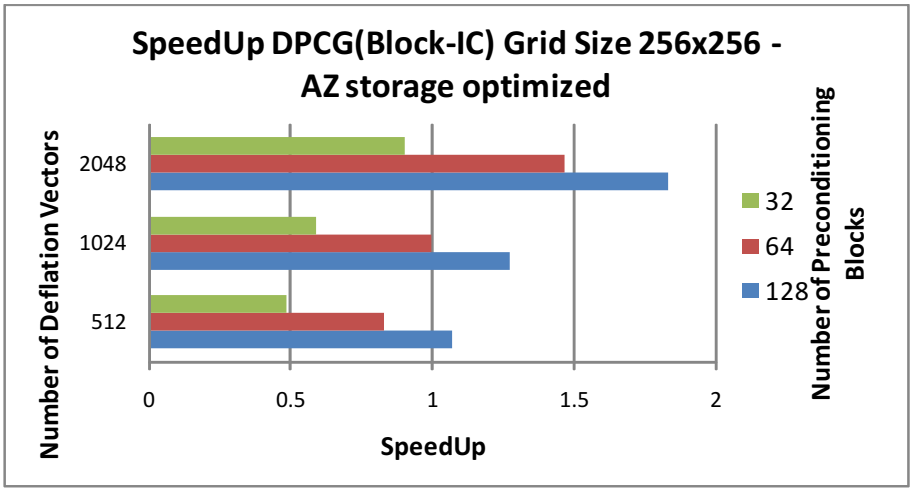

Figure 29: Deflated Conjugate Gradient with Block-IC Preconditioning. 256×256 grid.

The kernels after re-organization of AZ have been written for the host side as well. We now show the comparisons of the two codes running on the host and the device.

7.5.1 Comparisons with GPU versions

With the optimized storage of AZ we reduce the effective number of multiplications by around two orders of magnitude or more. This is a mixed blessing for the GPU version of the code since previously (when we had full AZ) more calculations meant more parallelism, but now that reduces and furthermore due to the smaller size of the AZ matrix it fits better in the CPU memory (cache). Result being that the GPU gains in speed by 2x whereas the CPU gains 20x. Effectively the speedups are decimated. Figure 28 to 30 show this effect.

7.5.2 Profiler Picture

Since we have optimized one of the heavyweights from the previous version of the code as in Section 7.5. Now the majority of the time is spent in preconditioning and as the grid size increases preconditioning dominates more and more.

This is visible in Figure 31 to 33. In the following section we will try to address the

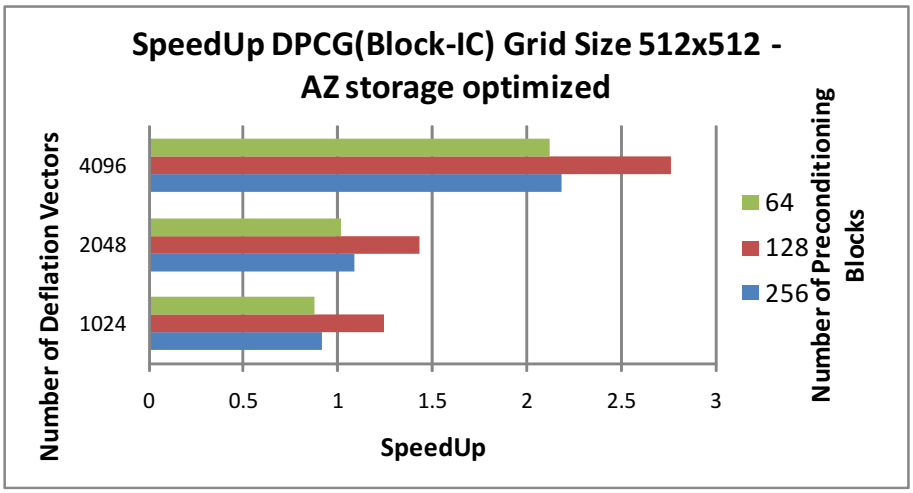

Figure 30: Deflated Conjugate Gradient with Block-IC Preconditioning. 512×512 grid.

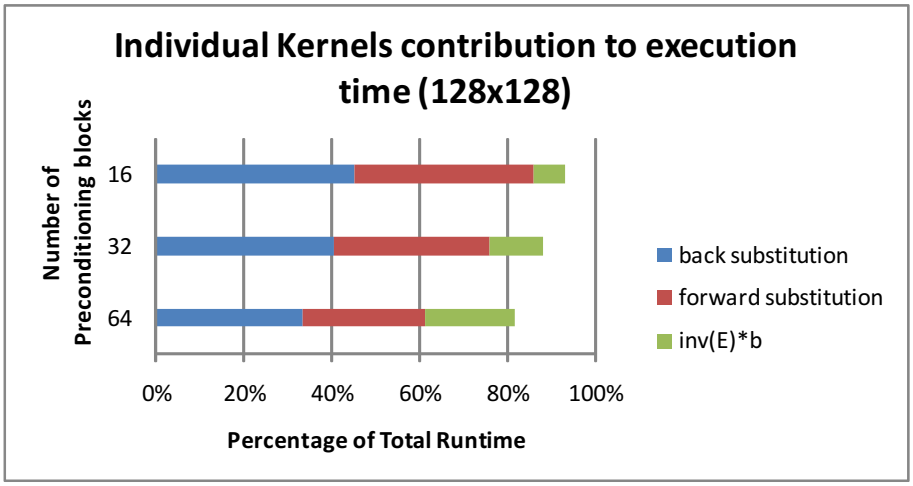

Figure 31: Deflated Preconditioned Conjugate Gradient(AZ storage optimized). Grid Size (128×128)

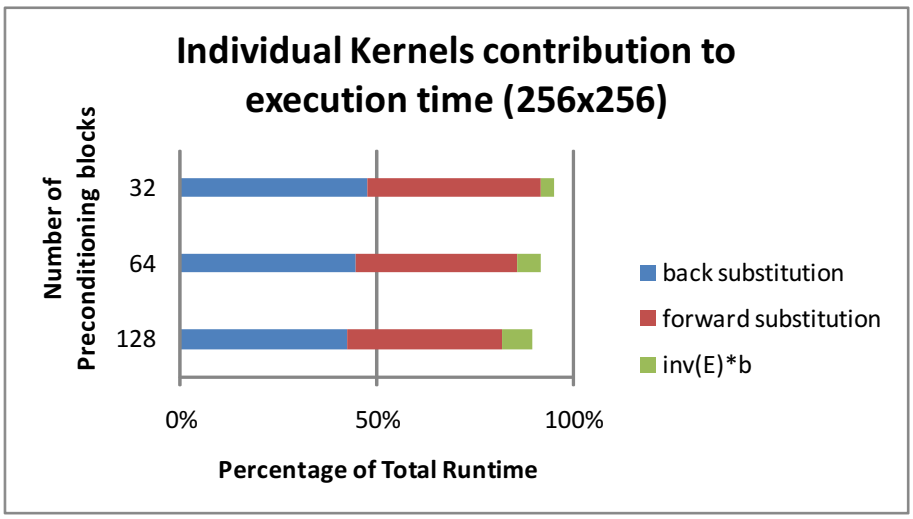

Figure 32: Deflated Preconditioned Conjugate Gradient(AZ storage optimized). Grid Size (256×256)

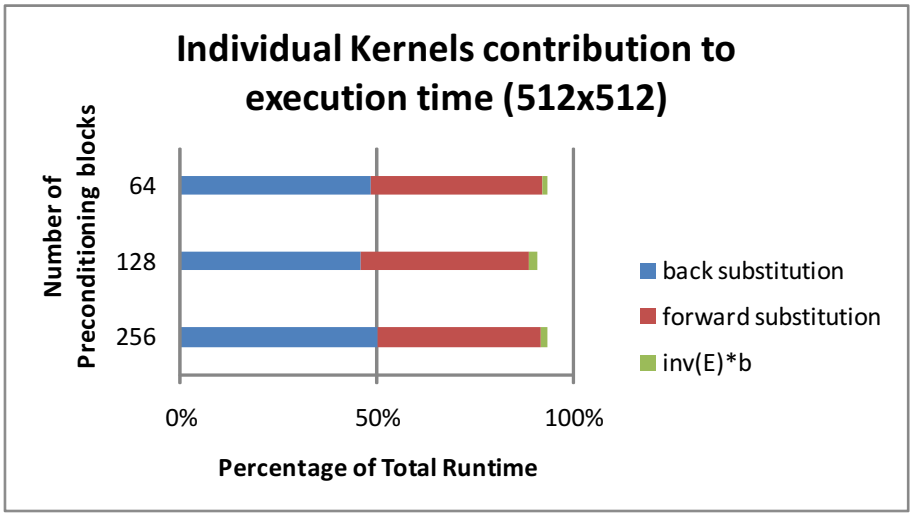

Figure 33: Deflated Preconditioned Conjugate Gradient(AZ storage optimized). Grid Size (512×512)

problem with sequential nature of Block-IC preconditioning with a new kind of Preconditioning that gives ample scope for parallelism.

7.6 Conjugate Gradient with Deflation and IP Preconditioning - AZ storage optimized

After optimizing AZ storage most of the time is spent in the preconditioning kernels, namely the forward and back substitution steps. In order to get a better speedup we experimented with the IP preconditioning (discussed in Section 3.2.2) approach suggested by [Ament, Knittel, Weiskopf, and Straβer, 2010].

This preconditioner is inherently parallel since it(M^{-1}) is also composed of 5 diagonals. In order to calculate the updated vector y we need to do a sparse matrix vector multiplication just like we do for Ax. This dramatically affects the run time of the algorithm as we show below in the results.

7.6.1 Comparisons with GPU versions

We made a CPU version for Deflated IP Preconditioned CG with optimizations to AZ and compared it with an equivalent GPU version written in CUDA. After optimizing the preconditioning we again get upto an order of magnitude speedups. See Figure 34.

It is also useful to put into perspective the speedups we had with the previous version that used Block-IC Preconditioning.

From Figure 35 to 37 we can see the improvement in speedup that IP Preconditioning offers for our problem. Also it is important to compare the number of iterations taken in the Block-IC variants for convergence to those required for the Incomplete Poisson version to converge. In Figure 38, 40 and 42 we can see that we pay a small cost for the parallelism by having a slightly higher iteration count for IP Preconditioning based solution compared to the highest number of blocks based Block-IC Solution.

Finally if we compare the error norms (Figure 39, 41 and 43) of the solution for all the cases, they are also comparable between all variants of Block-IC based DPCG and IP based DPCG.

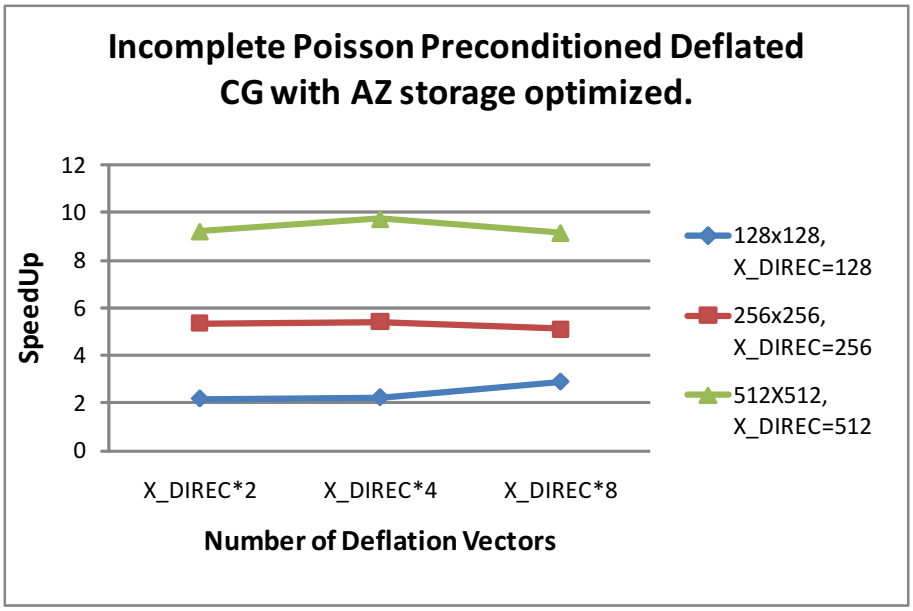

Figure 34: Deflated Preconditioned Conjugate Gradient with IP Preconditioning across three grid sizes

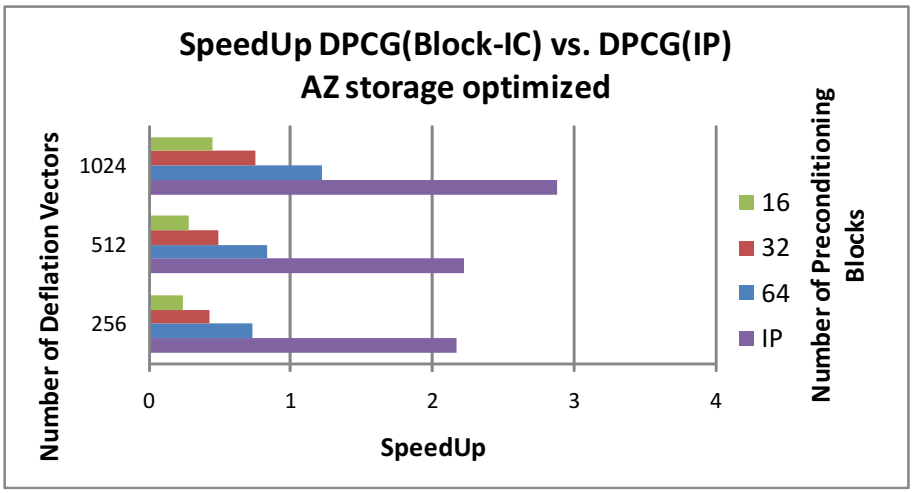

Figure 35: Grid Size 128×128

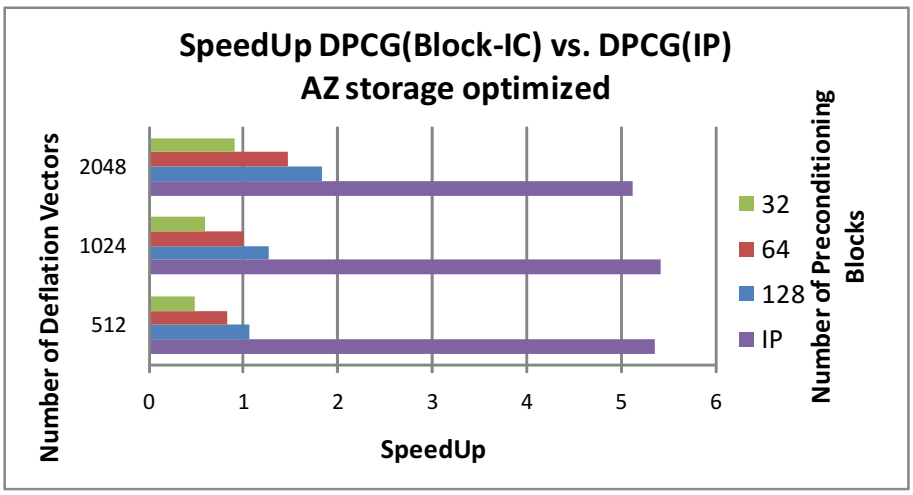

Figure 36: Grid Size 256×256

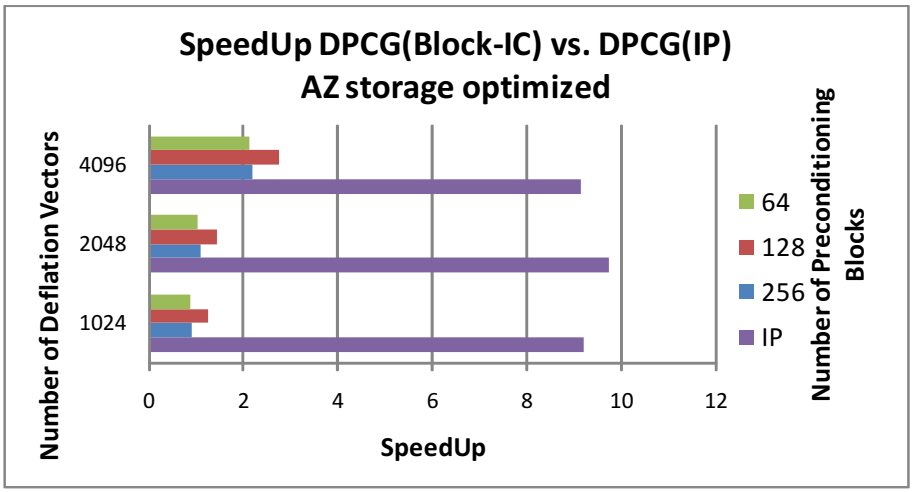

Figure 37: Grid Size 512×512

7.6.2 Profiler Picture

Looking at the profiler output in Figure 44 to 46 now we see that it is dominated by the the *cublasSgemv* call for the operation $E^{-1}b$. for larger number of deflation vectors. However at smaller number of deflation vectors the execution time is almost evenly distributed.

The important thing that emerges out of these figures is that the kernel for $E^{-1}b$ must be optimized since it is taking the majority of the time. For smaller number of deflation vectors however the percentage of time taken for $E^{-1}b$ comes down due to a decrease in the number of operations involved and the dot product kernels also take a significant amount of time.

7.7 Conjugate Gradient with Deflation and IP Preconditioning - AZ storage optimized and optimized Matrix Vector ($E^{-1}b$) Multiplication

After optimizing storage of AZ and preconditioning the dominant part of the kernel becomes the the matrix vector product $E^{-1}b$ as shown in the results in the Profiler Picture of previous section 7.6.2. As previously (Section 6.3) discussed this method (of calculating E^{-1} explicitly) was chosen for its suitability for the GPU since it could be otherwise be calculated using the substitution methods which are completely serial.

For optimizing this operation better than with $CUBLAS$ (*cublasSgemv*) we use the Magma Library for CUDA version 0.2. The memory throughput of the Magma *gemv* function is much better than *cublasSgemv*. It often (for many matrix sizes) is 2.5 times faster than $CUBLAS$.

In addition to this major change we also club some of the existing functions that involved saxpys and copy/scaling operations on the device involving α and β for calculating the new search direction and updating the residual vector(see steps 4 and 7 of Algorithm 5). These deliver close to $8-9\%$ improvement over all the grid sizes in addition to the improvment due to the use of $MAGMA$.

7.7.1 Comparisons with GPU versions

In Figure 47 to 49 we present the results of using the the blas routine *gemv* from the $MAGMA$ library. As one can notice the effect is significant when compared to the previous best version we have that of Incomplete Poisson Preconditioning (with AZ storage

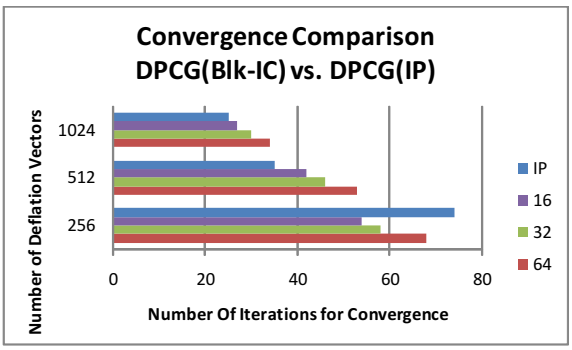

Figure 38: Grid Size (128×128)

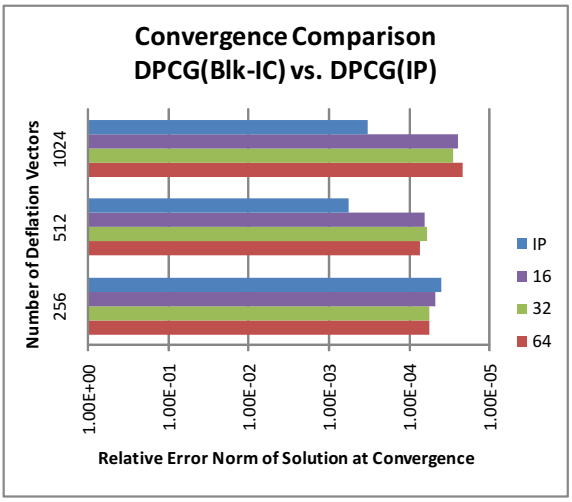

Figure 39: Grid Size (128×128)

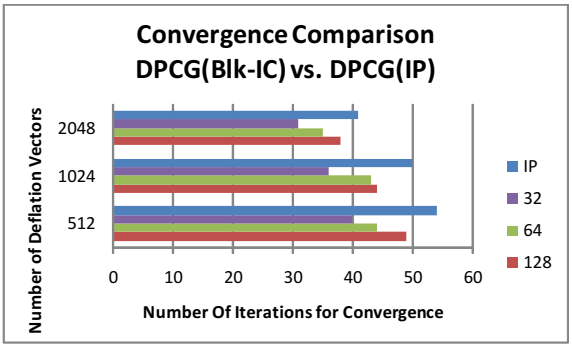

Figure 40: Grid Size (256×256)

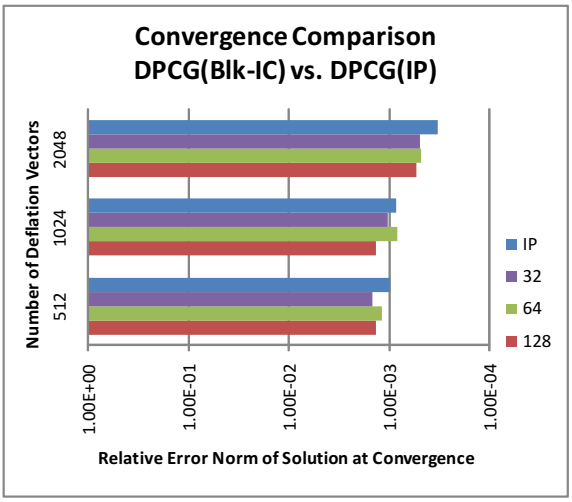

Figure 41: Grid Size (256×256)

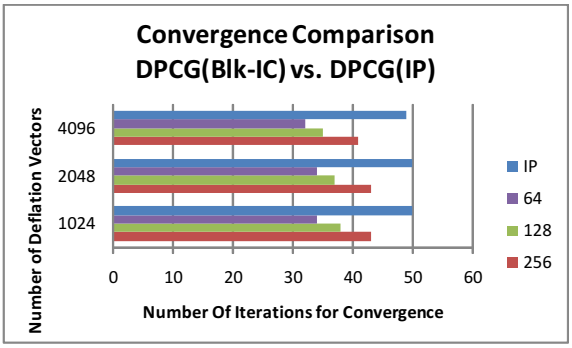

Figure 42: Grid Size (512×512)

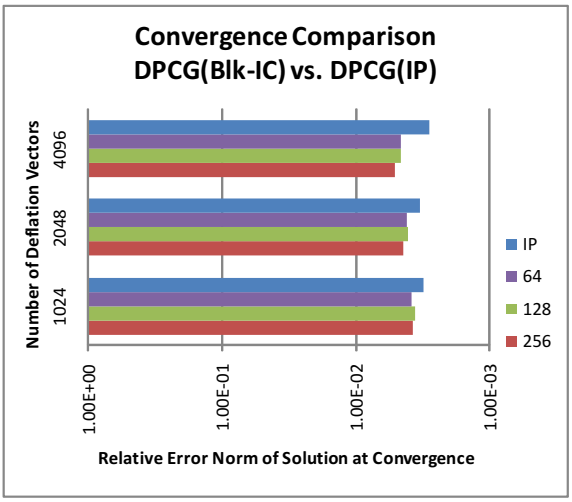

Figure 43: Grid Size (512×512)

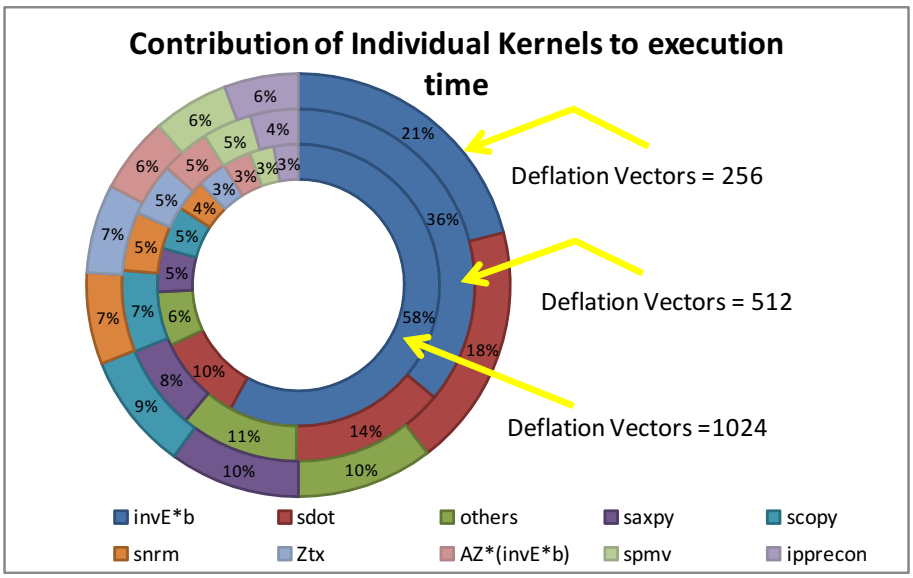

Figure 44: Deflated IP Preconditioned Conjugate Gradient. Grid Size (128×128)

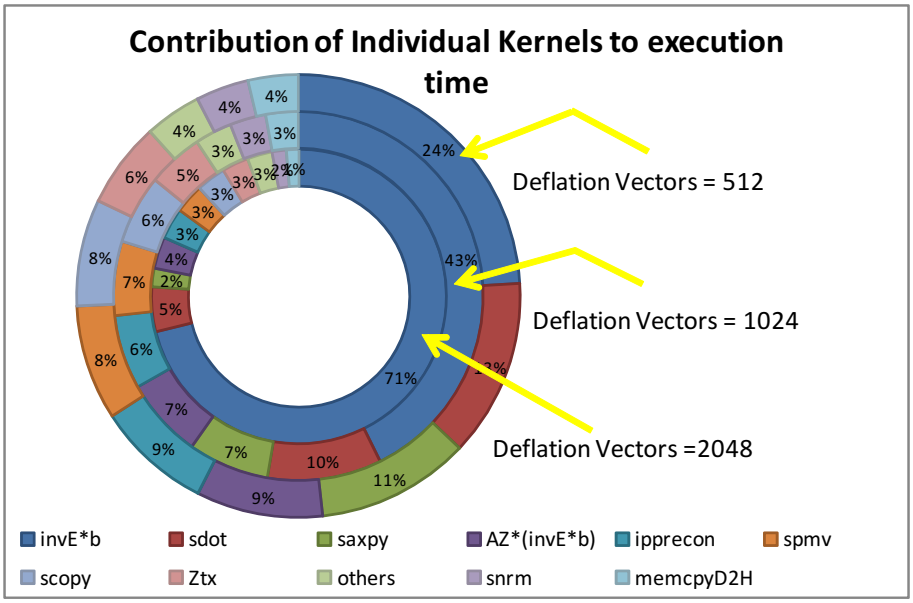

Figure 45: Deflated IP Preconditioned Conjugate Gradient. Grid Size(256×256)

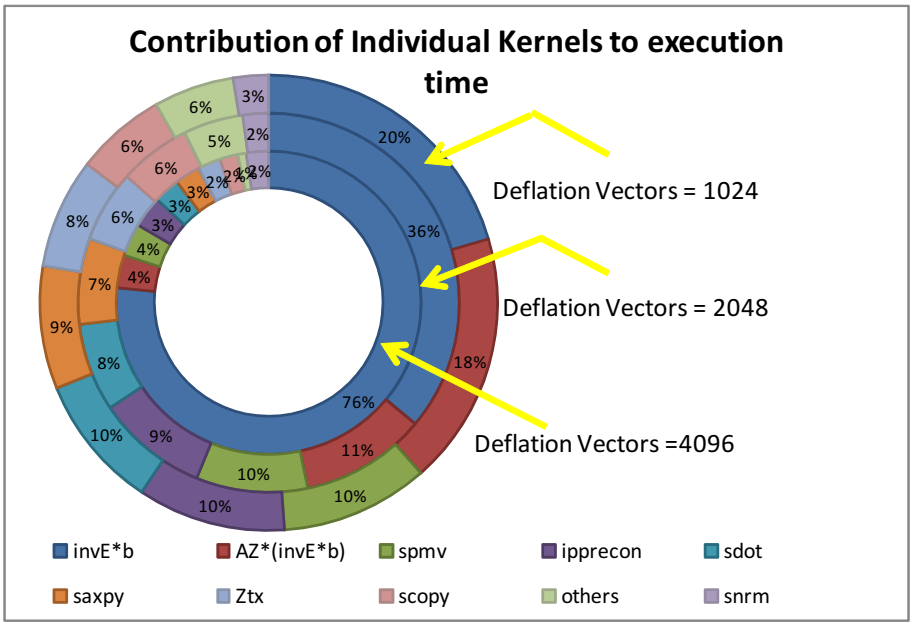

Figure 46: Deflated IP Preconditioned Conjugate Gradient. Grid Size(512×512)

optimized) and *CUBLAS*. In some cases the speedup has also doubled. The host version has been kept the same as was in the previous (Section 7.6) experiment.

7.7.2 Profiler Picture

The *MAGMA* blas library has levelled the execution times to a large extent. In the Figure 50 through 52 it can be noticed that the $E^{-1}b$ operation execution time now scales with the number of deflation vectors almost linearly. Also worth noting is that for the biggest problem size in the Figure 52 for a Grid Size of 512×512 and 4096 deflation vectors we get the bandwidth for the *gemv* kernel at 86Gb/s which is substantial since Tesla hardware supports a maximum of 101 Gb/s of Memory Bandwidth.

8 Experiments with Two Phase Flow Matrix

Till now all the results we have shown had the 5-point Laplacean Matrix as the coefficient matrix A. However the real intent of this study is to solve the multi-phase flow problems. By picturing two phases in the coefficient matrix we have a problem that is just a hair removed from a potential multi-phase/bubbly flow problem.

All that changes are the coefficients in the matrix A. Its structure remains the same. However since we are picturing fluids with very different densities there is a significant discontinuity between the coefficients now. As discussed in Section 2.1 we work with a matrix of the same structure, however with a very different condition number. The two phase flow matrix in its original form is not invertible. That is a problem if we want to find a solution to $Ax = b$. We force invertibility by setting $A(N, N) = 2$, where $N = n \times n$ and n =grid size in one dimension.

8.1 Conjugate Gradient -Vanilla Version and with Block- IC and Diagonal Preconditioning

We ran Conjugate Gradient Algorithm with Block-IC Preconditioning (size of blocks = $2 \times n$ where $N = n \times n$) and Diagonal Preconditioning and record the variation in the

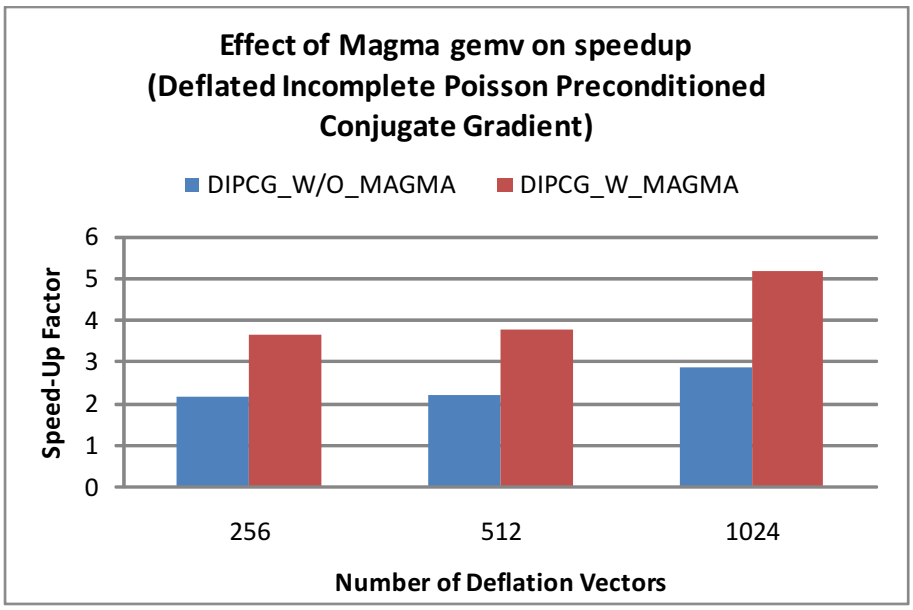

Figure 47: Grid Size(128×128)

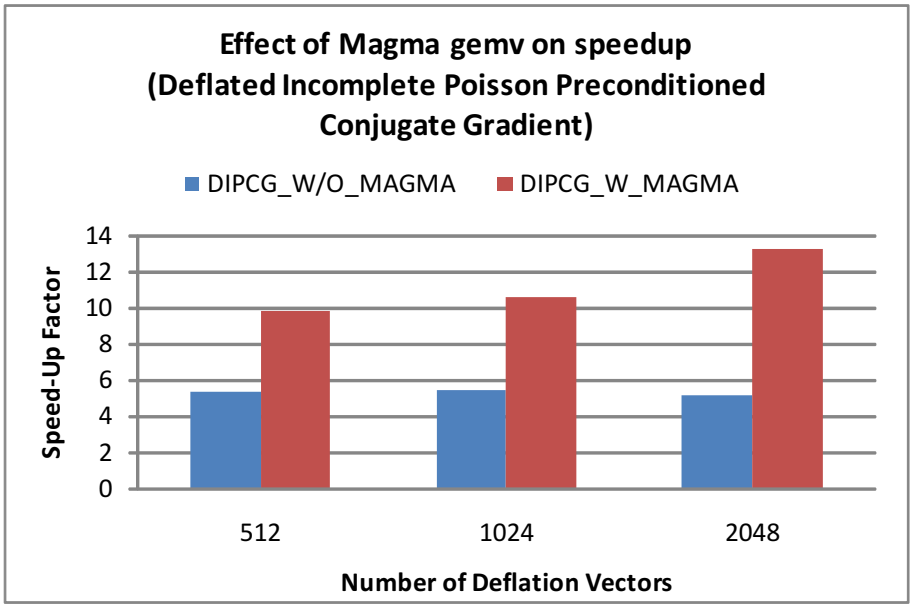

Figure 48: Grid Size(256×256)

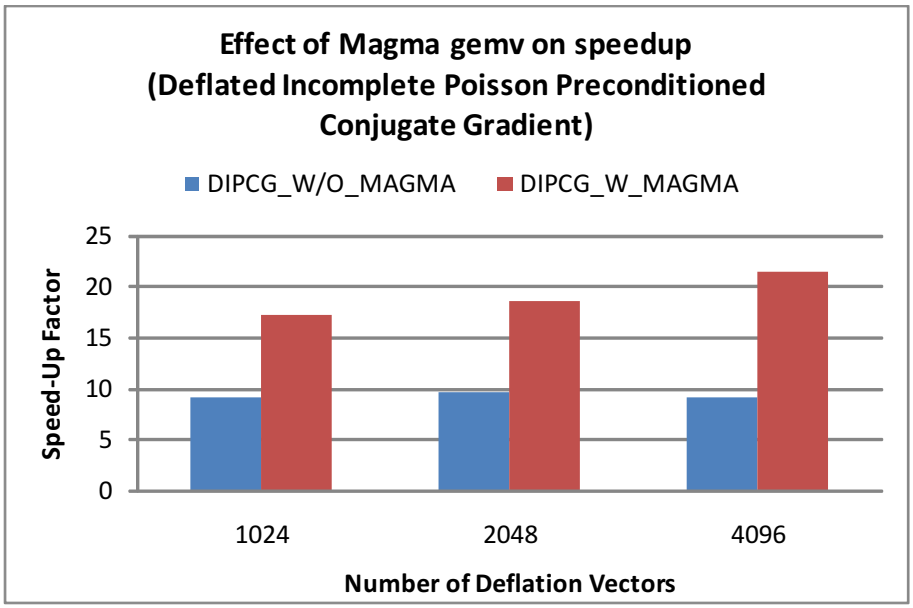

Figure 49: Grid Size(512×512)

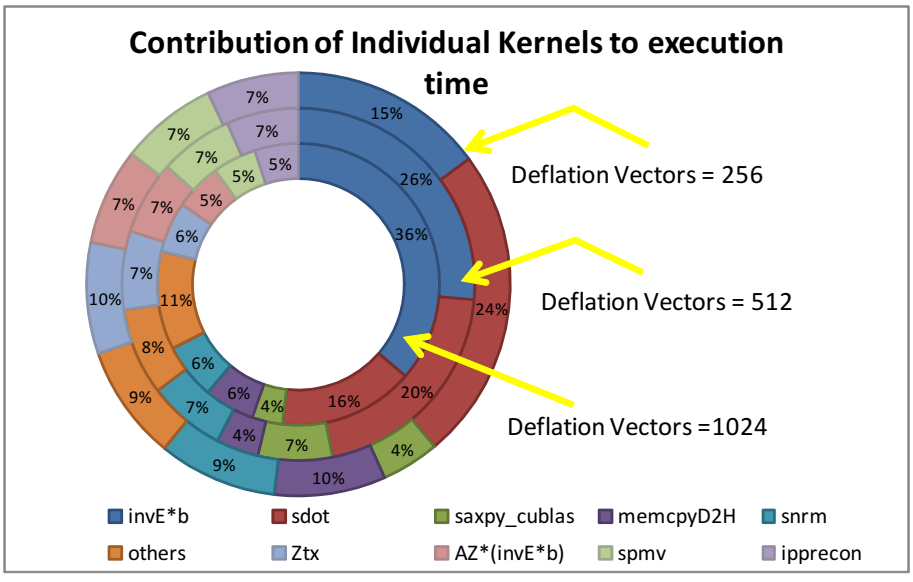

Figure 50: Deflated (IP) Preconditioned Conjugate Gradient. (AZ storage optimized and Magma BLAS based gemv) Grid Size (128×128)

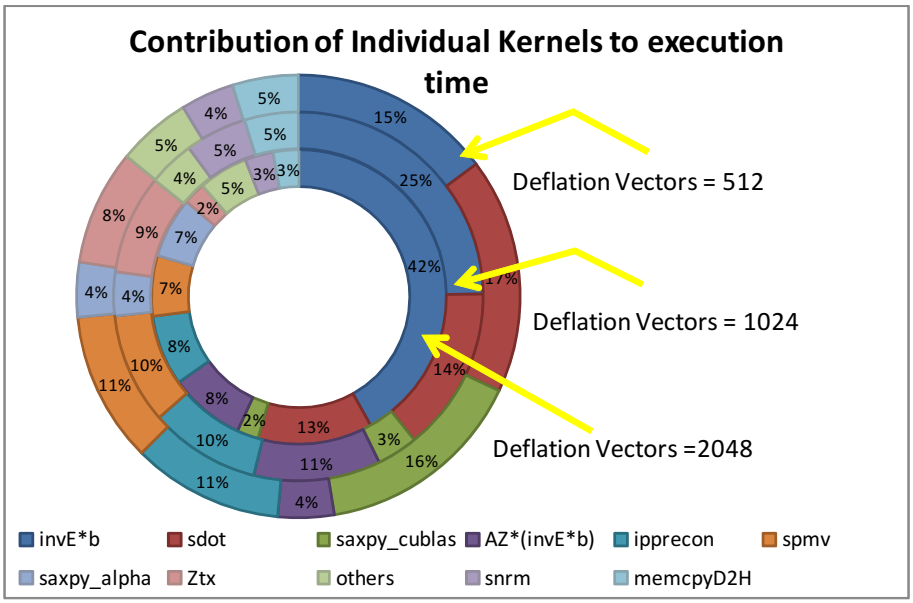

Figure 51: Deflated (IP) Preconditioned Conjugate Gradient. (AZ storage optimized and Magma BLAS based gemv) Grid Size (256×256)

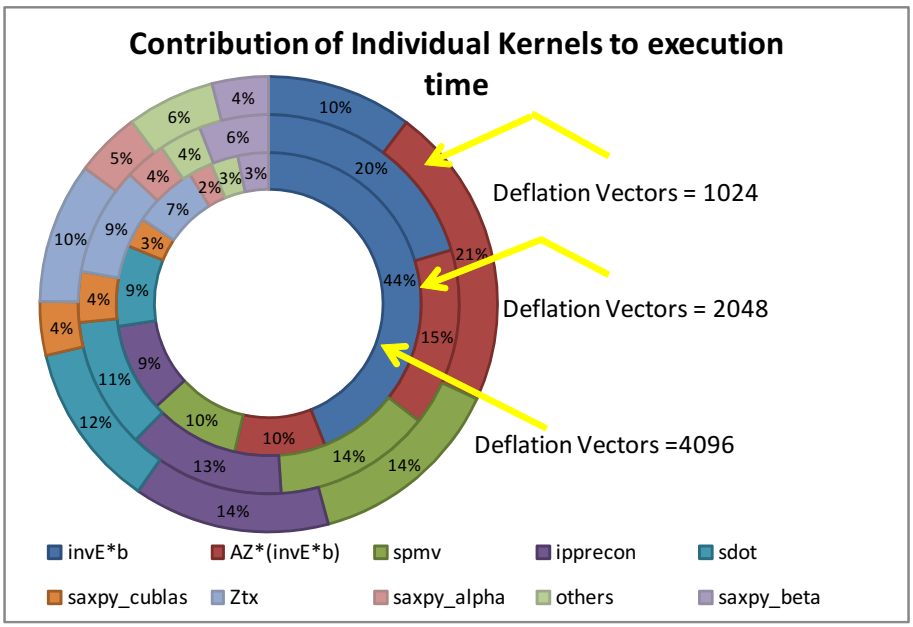

Figure 52: Deflated Preconditioned Conjugate Gradient. (AZ storage optimized and Magma BLAS based gemv) Grid Size (512×512)

error norm of the solution and the residual norm over multiple iterations (even after the convergence). What we notice is that the residual norm keeps on falling. This is not correct at all since, if the norm starts from 10^2 and falls below 10^{-6} it is exceeding machine precision for single-precison floating point operations and that is not possible. In the graphs that follow from Figure 53 to 57 for Conjugate Gradient with Diagonal Preconditioning and from Figure 58 to Figure 62 for Conjugate Gradient with Block-IC Preconditioning we see this phenomenon clearly. The thin vertical blue line in all the graphs is where convergence is achieved. The solid blue line traces the norm of the error in the solution and the red line is the norm of the residual. The precision criteria for these tests was set to 10^{-5}. They have have been run on smaller grids in order to show the effect in lesser number of iterations. With larger grid sizes (those above 16384 unknowns) the behavior is similar. The green vertical line shows where machine precision is hit.

These two results prove that even though convergence happens and relative residual norm reaches a value lower than the expected precision (10^{-5} in these calculations) it is a sort of false stop. The residual rises and falls some more times before it becomes meaningless (seems to become better than machine precision) and the error norm of the solution which is at a relatively higher value at the time of convergence also comes down after the observed convergence. This is in line with the work previously done [Tang, 2008].

8.1.1 Comparisons with GPU versions

We first ran the Conjugate Gradient algorithm on the two phase matrix, followed by two experiments with Diagonal and then Block Incomplete Cholesky Preconditioning. We get comparable speedups (Figure 63) to Section 7.1.2. However the norm (Figure 64) of the relative error in the solution remains high due to the ill-conditioning of the original problem.

For Block Incomplete Cholesky Version we have three different *block sizes* which result in three different *number* of blocks. These are denoted by $\frac{X\_DIREC}{2}$, $\frac{X\_DIREC}{4}$ and $\frac{X\_DIREC}{8}$. Here X_DIREC denotes the dimension of the grid in x-direction. For our experiments we have a square grid hence dimensions are $n \times n$ and number of unknowns $N = n \times n$. Hence $n = X\_DIREC$. For e.g. if $n = 256$ then $X\_DIREC = 256$ and subsequently we have Number of Blocks as $128, 64$ and 32.

8.2 Conjugate Gradient with Deflation and Block-IC Preconditioning

In this section and the next one we summarize the speedups for Conjugate Gradient with Deflation and Preconditioning. Deflation takes toll on the precision condition we fix as 10^{-6} for the previous tests (without deflation only preconditioning in section 8.1). Result being that the precision condition has to be scaled down to 10^{-2} and then the method seems to converge. For deflation we use the most optimized version that we have had from the previous results in Section 7.7.

We present the results for calculated norms on three grid sizes 128×128, 256×256 and 512×512 across three different number of deflation vectors and preconditioning block sizes in Figure 65. In these results the ratio of the densities of the two mediums is 10, unlike the previous experiments (Section 8.1) where it was 1000. As can be seen except for Grid Size 128×128 at number of deflation vectors = 1024 we have a peculiar case when the error norm rises and the solution is more erroneous than the initial guess. Otherwise the error norm is a between 0 and 1.

More detailed results are available in the Appendix C.2.1. The reason for this behavior is the deteriorating condition number of A due to the two phase matrix. It is worsened for the Projection matrix P that has to be constructed for deflation.

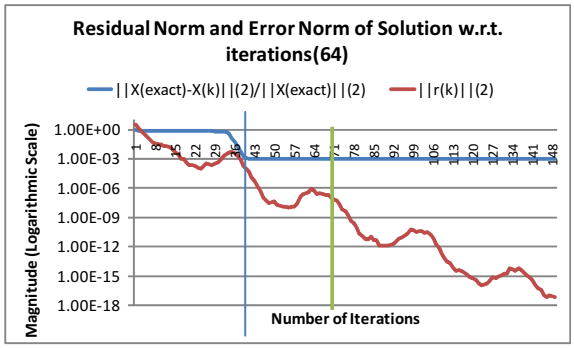

Figure 53: Conjugate Gradient with Diagonal Preconditioning for Two-Phase Matrix (Grid Size 8×8)

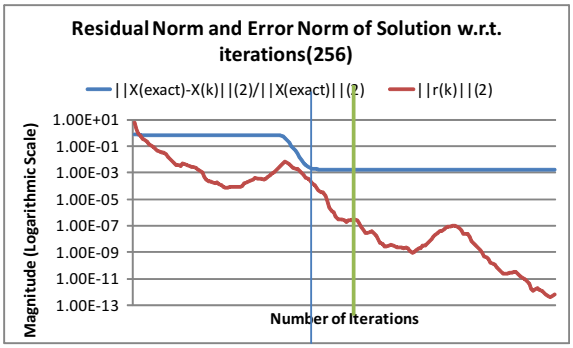

Figure 54: Conjugate Gradient with Diagonal Preconditioning for Two-Phase Matrix (Grid Size 16×16)

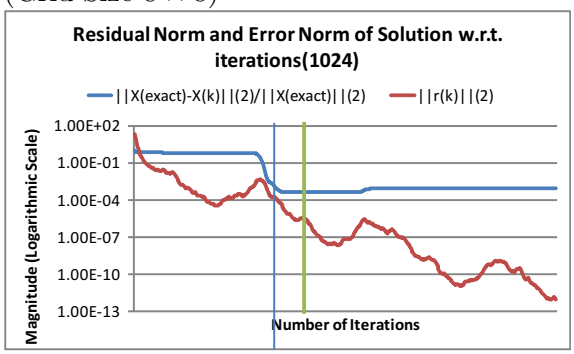

Figure 55: Conjugate Gradient with Diagonal Preconditioning for Two-Phase Matrix (Grid Size 32×32)

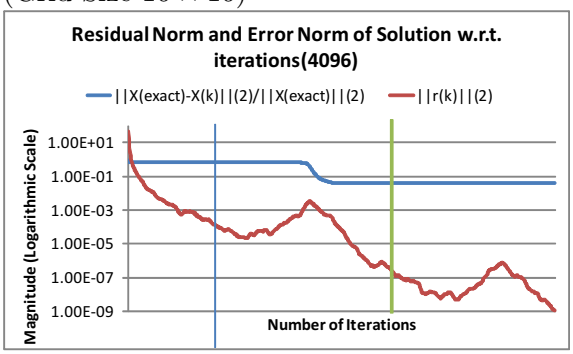

Figure 56: Conjugate Gradient with Diagonal Preconditioning for Two-Phase Matrix (Grid Size 64×64)

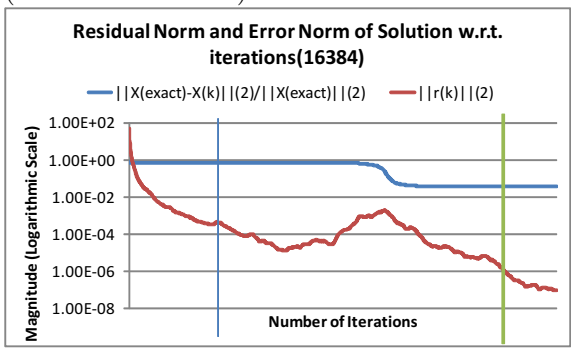

Figure 57: Conjugate Gradient with Diagonal Preconditioning for Two-Phase Matrix (Grid Size 128×128)

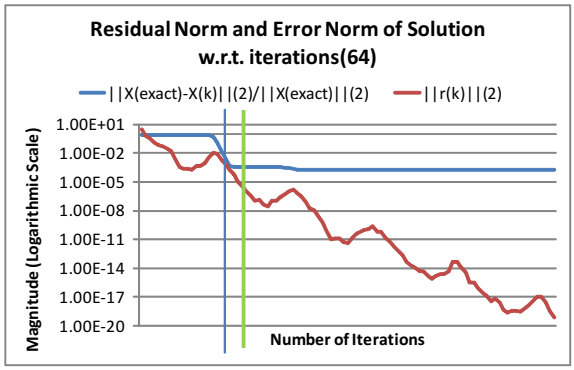

Figure 58: Conjugate Gradient with Block-IC Preconditioning for Two-Phase Matrix (Grid Size 8×8)

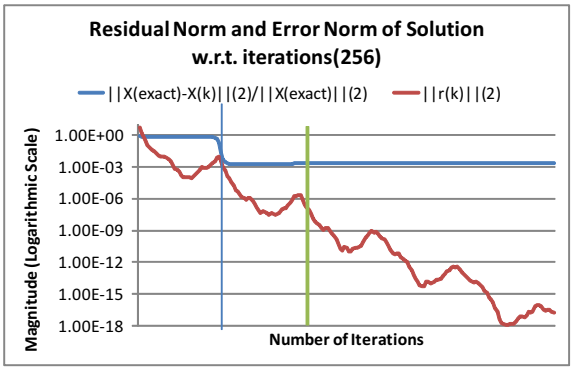

Figure 59: Conjugate Gradient with Block-IC Preconditioning for Two-Phase Matrix (Grid Size 16×16)

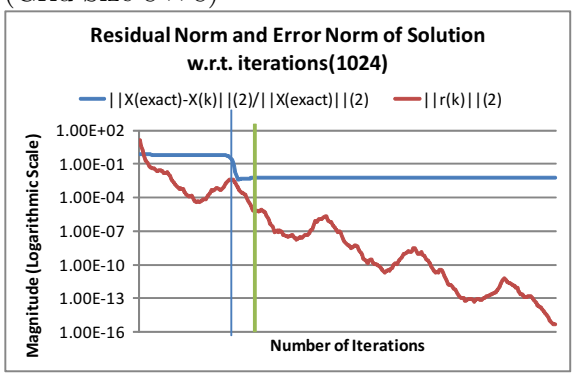

Figure 60: Conjugate Gradient with Block-IC Preconditioning for Two-Phase Matrix (Grid Size 32×32)

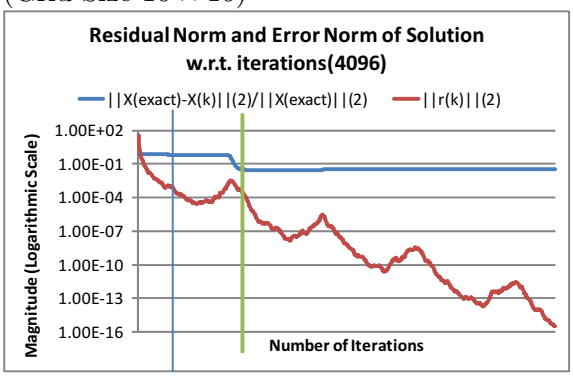

Figure 61: Conjugate Gradient with Block-IC Preconditioning for Two-Phase Matrix (Grid Size 64×64)

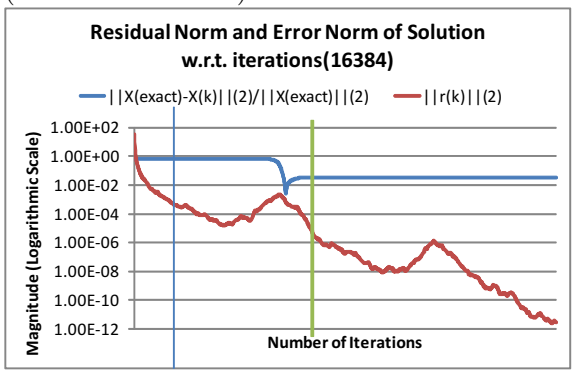

Figure 62: Conjugate Gradient with Block-IC Preconditioning for Two-Phase Matrix (Grid Size 128×128)

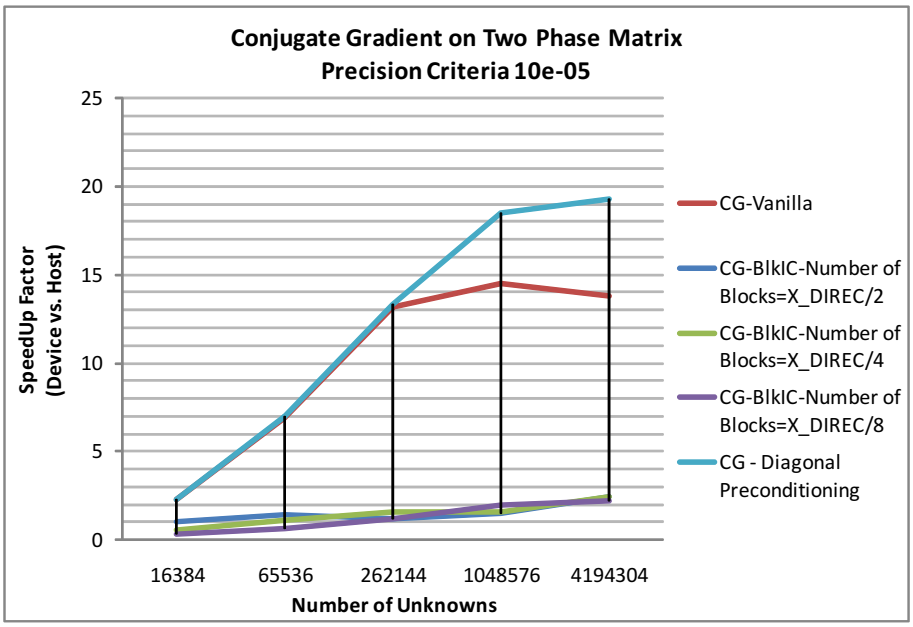

Figure 63: Speedup across various sizes for different variants

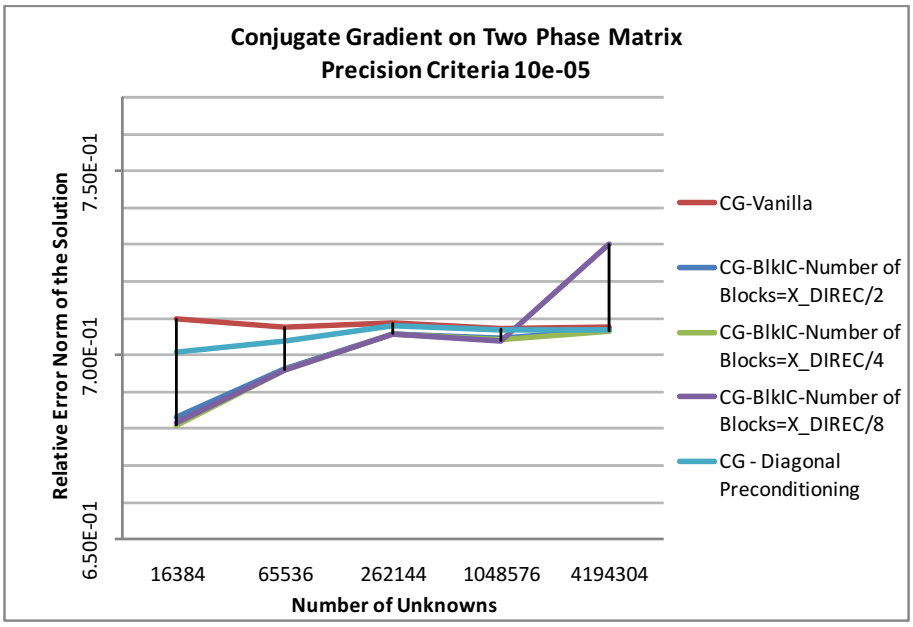

Figure 64: Relative Error Norm of the Solution $\frac{\|X_{exact}-X_k\|_2}{\|X_{exact}\|_2}$ across various sizes for different variants

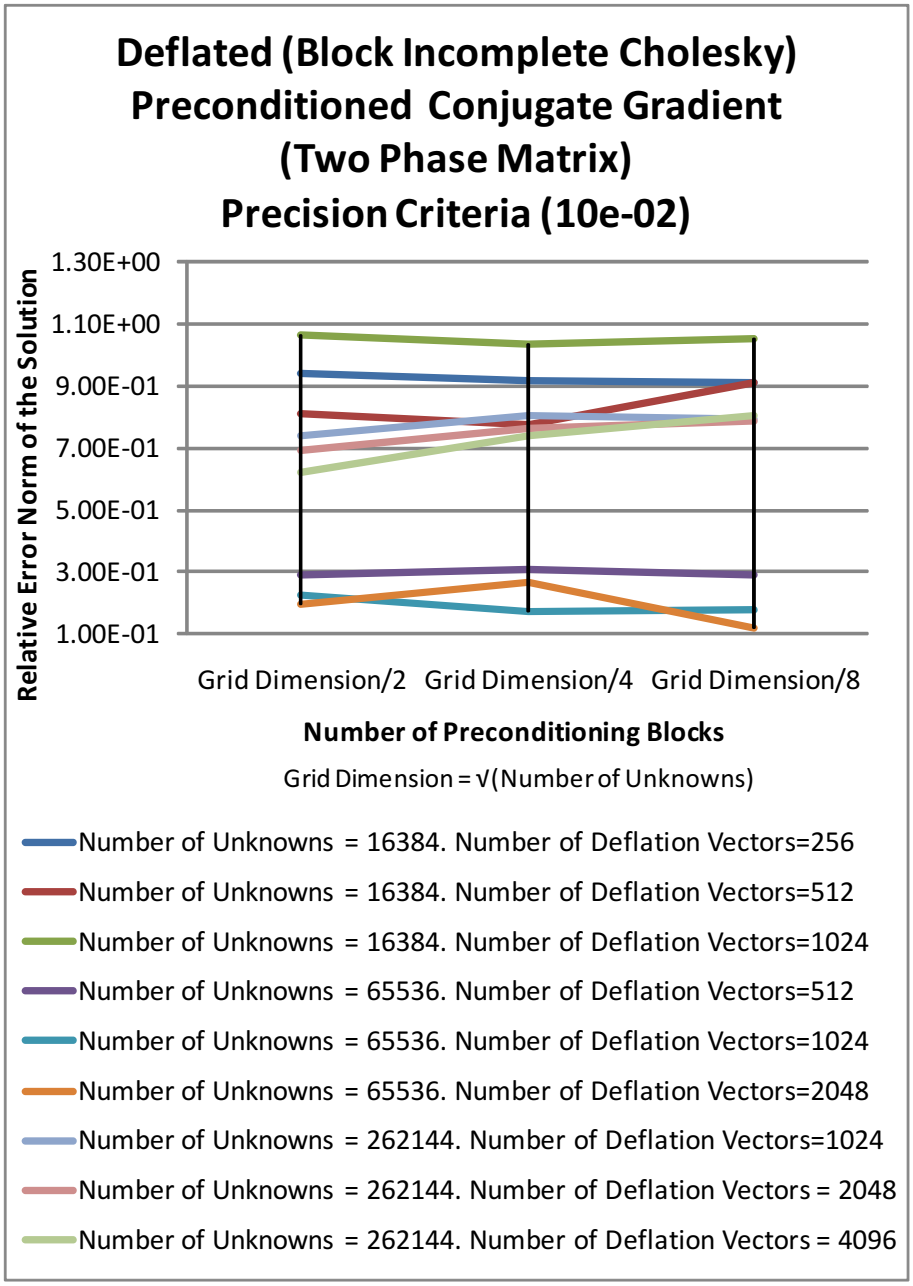

Figure 65: Two Phase Matrix with Density Contrast $(10:1)$. Relative Error Norm of the Solution, $\frac{\|X_{exact}-X_k\|_2}{\|X_k\|_2}$

Looking at the Speedup picture (Figure 66) we find that speedups closely resemble the results in a previous section(7.5) with a simple 5-point Laplacean Matrix.

8.3 Conjugate Gradient with Deflation and IP Preconditioning

Following the setup of Section 8.2 we conduct the test runs for the IP Preconditioner case and see slightly better speedups attributed to the parallel nature of the preconditioner (Figure 68). However the relative error norm of the solution at convergence is very high (comparable to values in Section 8.2).

We present the results for three grid sizes 128×128, 256×256 and 512×512 across three different number of deflation vectors per grid size in Figure 67. In these results the ratio of the densities of the two mediums is 100, unlike the previous experimentSection 8.2) where it was 10 and 1000 in the first experiments Section 8.1.

The speedups (Figure 68) closely resemble the speedups from the Section 7.6.

The profiler pictures don't change much for sections 8.2 and 8.3 since only the matrix changes and not the methods of computation. The kernel execution times follow similar pattern as that of the Poisson type Matrix (similar to Sections 7.5 and 7.6).

9 Analysis

In this section we look at the different aspects of our implementations. We try to find out how much parallelism we exploit and how much bandwidth we are able to utilize on the GPU. We end this section with a discussion on what might be possibly limiting the achievable speedup and how far we are from that point. Throughout this section we analyze the results with a grid size of 512×512 and 4096 deflation vectors and Incomplete Poisson Preconditioning. Specifically the results from Section 7.7 are used unless otherwise mentioned.

9.1 Static Analysis

In this section we calculate how many Floating Point Operations (FLOPs) each kernel does in each run. Also we list out how many memory accesses happen both during load and stores.

We list them for all the Kernels. In general a few variables can be defined.

- N, Number of Unknowns
- d, Number of Deflation Vectors
- m, Number of Iterations
- n, Grid Dimension

From Table 1 we can find out the number of Operations being performed in one complete run of the methods we have implemented.

We now elaborate some of the Kernel names:

$Z^T x$, $E^{-1}b$ and $AZ \times E^{-1}b$ form the steps of the deflation operation. Forward Substitution, Diagonal Scaling and Back Substitution form the steps of Block Incomplete Cholesky Preconditioning. Sdot is the Dot product function as named in BLAS libraries. We use *cublasSdot*. Saxpy is the Saxpy Kernel as available in BLAS libraries. We use *cublasSaxpy* and also write custom kernels to club saxpy with scaling operations to minimize memory transfers. Sscal is the BLAS scaling operation and Snrm is the 2-Norm operation available in the BLAS libraries.

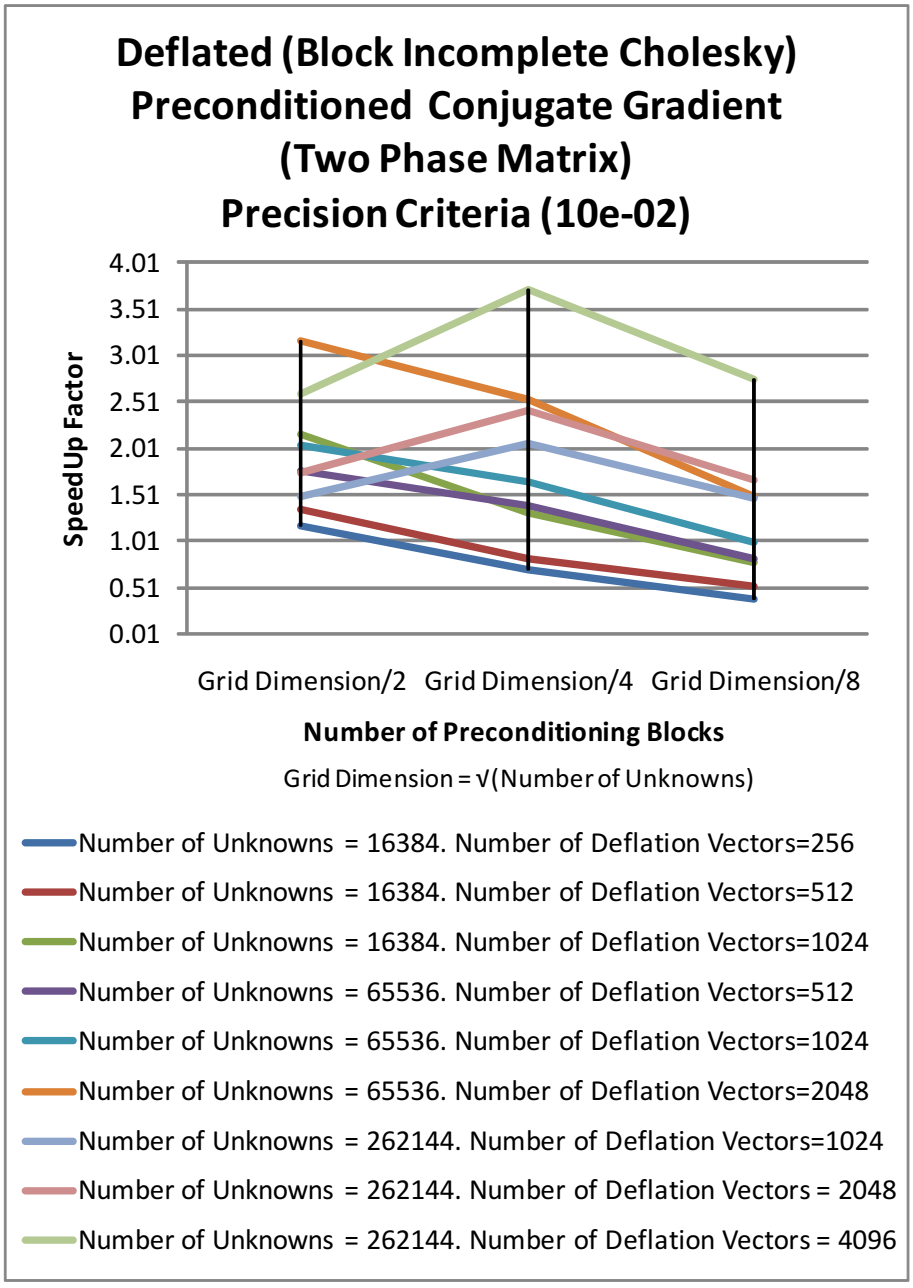

Figure 66: SpeedUp Graph. Density Contrast(10:1)

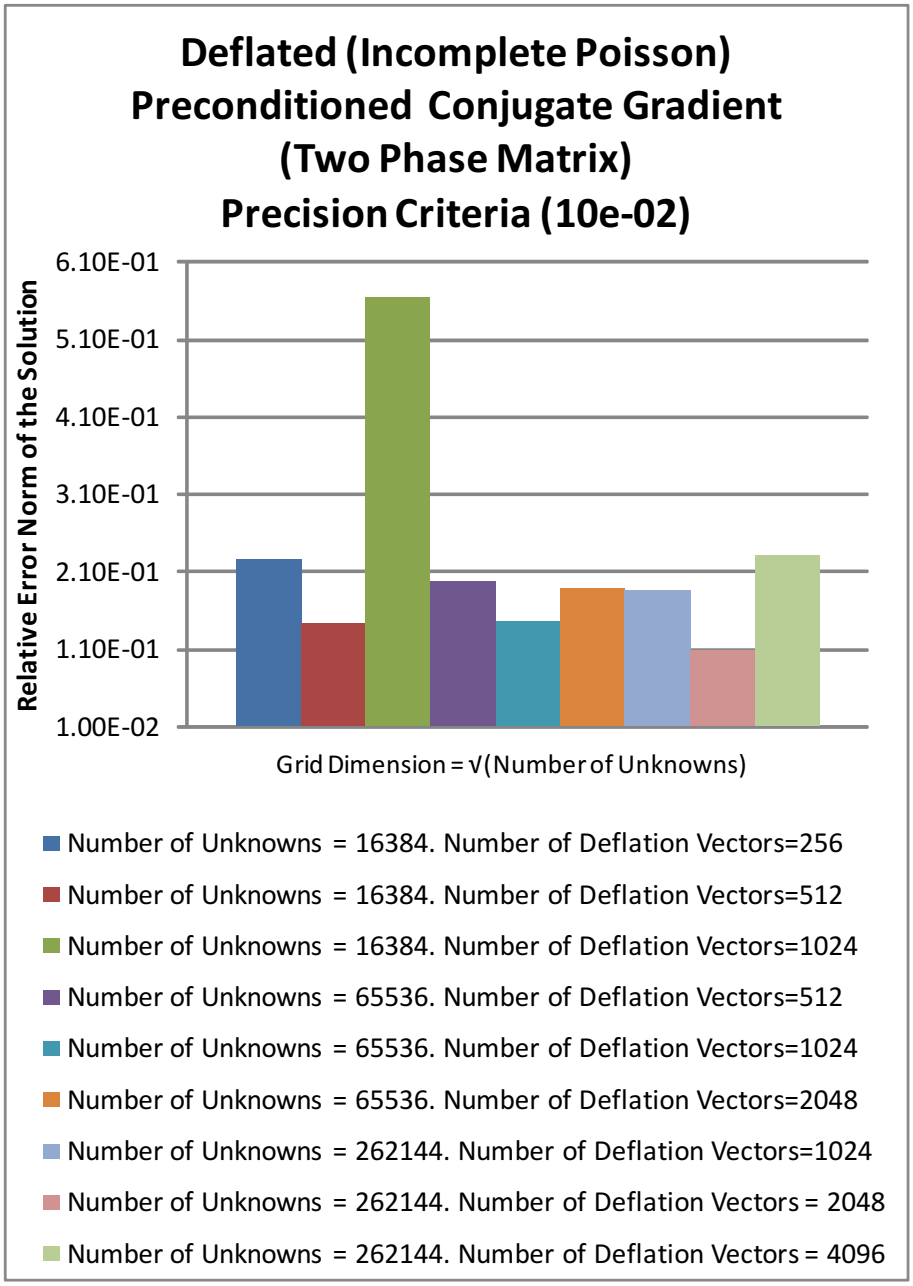

Figure 67: Two Phase Matrix with Density Contrast (100 : 1). Relative Error Norm of the Solution $\frac{\|X_{exact}-X_k\|}{\|X_k\|}$

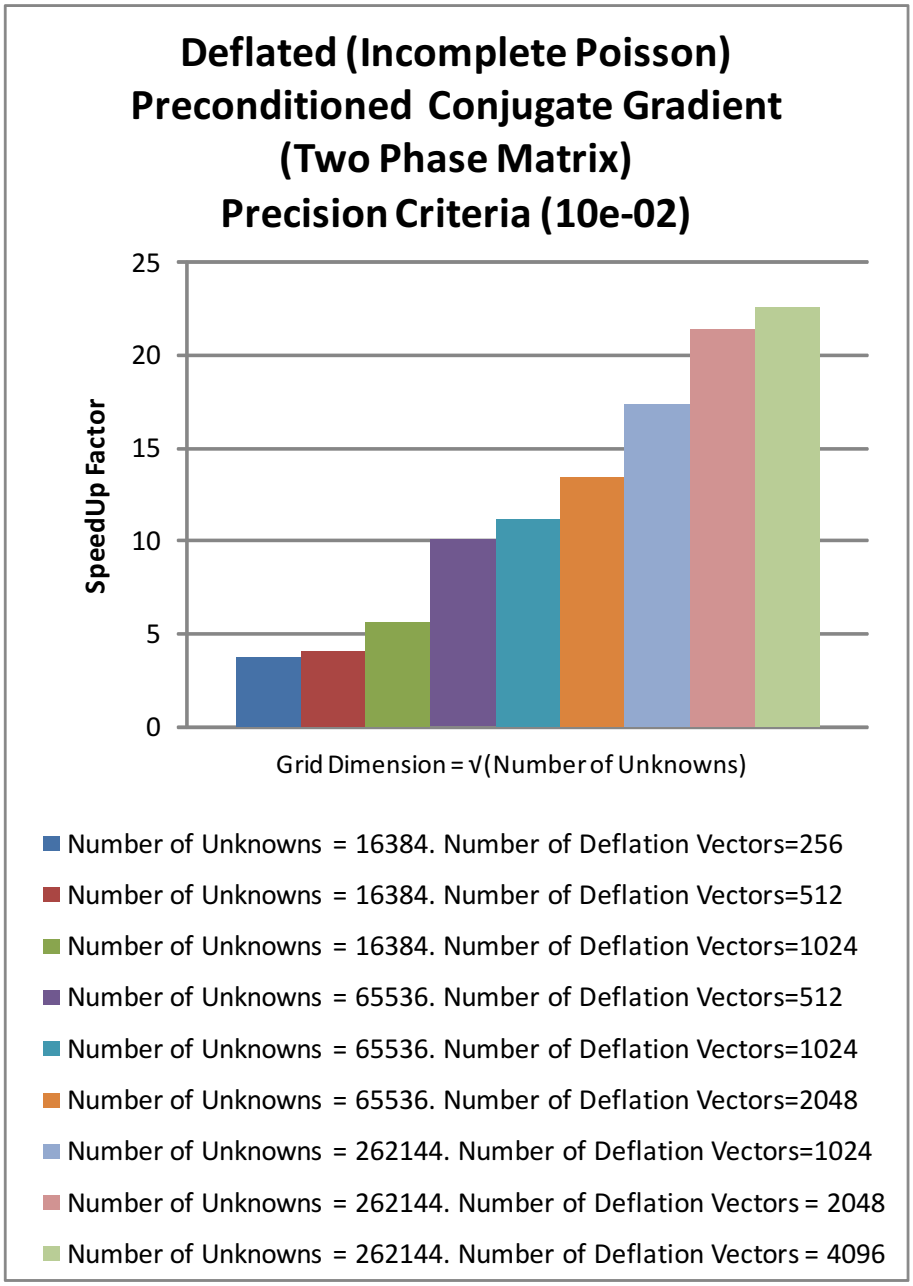

Figure 68: SpeedUp Graph. Density Contrast(100:1)

Kernel	Data Read In	Computations Done	Writes Performed	Maximum Degree Of Parallelism	Degree of Parallelism in use	Number of Calls
Sparse-Matrix Vector Product	$6N$	$9N$	N	N	N	$m+1$
$Z^T x$	N	N	d	d	d	$m+2$
$E^{-1}b$ (gemv)	$d(d+1)$	$d \times d$	d	d	d	$m+3$
$AZ \times E^{-1}b$	$5N+d$	$9N$	N	N	N	$m+1$
Incomplete Poisson Preconditioning	$6N$	$9N$	N	N	N	$m+1$
Forward Substitution	$4N$	$3N$	N	$\frac{\sqrt{N}}{2}$	$\frac{\sqrt{N}}{2}$	$m+1$
Diagonal Scaling	$2N$	N	N	N	N	$m+1$
Back Substitution	$4N$	$3N$	N	$\frac{\sqrt{N}}{2}$	$\frac{\sqrt{N}}{2}$	$m+1$
$AZ^T x$	$6N$	$5N$	d	d	d	1
Sdot	N	$2N$	N	N	–	$4m$
Saxpy	$2N$	$2N$	N	N	–	$3m$
Sscal	N	N	N	N	–	m
Snrm	N	$2N$	N	N	–	m

Table 1: Kernels - Computation and Parallelism

Let us take the case of the method implemented in Section 7.7. It is the Deflated Preconditioned (Incomplete Poisson) Conjugate Gradient method that uses optimized AZ storage and the *gemv* routine from $MAGMA$ Blas library. It also has some optimizations that club certain operations like scaling and saxpy for calculation of β as given in the step 9 of Algorithm 8.

The kernels involved in this variant then are:

- Sparse-Matrix Vector Product
- $Z^T x$
- $E^{-1}b$(gemv)
- $AZ \times E^{-1}b$
- Sdot
- Saxpy
- Incomplete Poisson Preconditioning
- Snrm
- $AZ^T x$

Summing up the FLOPs for m iterations we have

$$9N(m+1)+N(m+2)+d^2(m+3)+9N(m+1)+9N(m+1)+8Nm+6Nm+Nm+2Nm. \tag{104}$$

or

$$45Nm + d^2m + 29N + 3d^2 \tag{105}$$

So the computational intensity is governed by the first three factors of the expression in 105. Now let us take a specific case of $N = 262144$, $d = 4096$ and $m = 49$. These correspond to the experiment discussed in 7.7 with grid size as 512×512 and the Number of Deflation Vectors = 4096. It takes the 49 iterations to converge both on the host and the device. The time on the device is 0.0987 seconds and on the host is 2.237 seconds. The speedup is 22.7 times.

Now the GPU theoretically(peak throughput) can deliver 933 GFlops/s. The CPU on the other hand, when talking about one core (which we use in our experiments), can deliver a peak throughput of 12 GFlops/s. The numbers for NVIDIA are available from the website which talks about the Tesla C1060 specifications [NVIDIA, 2010] . For Intel Processors also the number are provided on the website [Intel, 2010].

The computational load as calculated in 105 comes out to be 1.46 GFlops. Dividing this by the time taken we get 0.65GFlops/s for the CPU and 14.79 GFlops/s for the GPU.

These numbers can be further divided by the peak throughput to understand the Platform Utilization on the GPU as 1.585% and on the CPU as 5.41%.

9.2 Kernels- Performance

We refer to some of the works that outline how to effectively characterize a kernels' performance and its ability to scale across new generations of hardware that will have more processors to facilitate parallel execution. [Nickolls, Buck, Garland, and Skadron, 2008] and [Ryoo, Rodrigues, Baghsorkhi, Stone, Kirk, and Hwu, 2008] and [Komatitsch, Michéa, and Erlebacher, 2009] bring about certain methods by which we can find

1. How to find if a kernel is compute bound or bandwidth bound?

2. Expected Speedup from an application.

3. Examination of PTX(CUDA assembly) code for finding percentage of code that is memory or compute intensive.

Also these documents detail important things to keep in mind when designing a kernel or optimizing it. These documents put to use, in their respective contexts, the Best Practices guide provided by NVIDIA [NVIDIA best prac, 2009].

The most important factor in a kernels' effectiveness is its ability to do memory accesses in the best possible way. To this end a couple of important techniques are instrumental. This step comes obviously after the point of minimizing memory transfers as much as possible between the CPU and GPU.

1. coalesced memory access

2. caching

3. minimize divergence

In Table 2 below we list which techniques are being used by the (except $CUBLAS$) kernels in the variant of our code in Section 7.7. We also list if there are shared memory conflicts.

Kernel Characteristics					
Method	Coalescing	Caching (Shared Memory)	Divergence	Shared-Memory Bank Conflicts	Warp Serialization
$MagmaSgemv$	Yes	Yes	No	No	No
$IPPreconditioning$	Yes	Minimal	Yes	Yes	No
$SpMV$	Yes	Minimal	Yes	Yes	No
$AZE^{-1}b$	Yes	Minimal	Yes	Yes	No
Z^Tx	Yes	Yes	Yes	Yes	Yes
$saxpy\_alpha$	Yes	Yes	No	No	No
$saxpy\_beta$	Yes	No	No	No	No

Table 2: Grid of 512×512 points. Number of Deflation Vectors = 4096. With optimizations applied to AZ storage and calculation, $E^{-1}b$ with MagmaSgemv and other optimizations.

9.3 Bandwidth Utilization

First let us take a look at the bandwidth utilization of the kernels in the most optimized version (Section 7.7) of the code that we have. This is the Deflated Incomplete Poisson Preconditioned Conjugate Gradient Method with optimizations for AZ storage and calculation and also with the *gemv* operation from the $MAGMA$ library.

In this version we consider the Grid Size 512 × 512 with the Number of Deflation Vectors = 4096. In the Figure 69 we list the Memory Throughput of Individual Kernels and the percentage of time they take of the total execution on the device.

The $CUBLAS$ Kernels are prefixed with $Cublas$ and other kernels have been hand-coded with exception of the $Magma\_Sgemv$ which is from the $MAGMA$ blas library. In this picture (Figure 69) we show kernels that form more than 98% of the total execution time. The last 2% or so is taken up by transfers from Device to Host and a few calls to kernels used for correcting x at the end of the iteration by doing $x = Qb + P^Tx$ as the last step of Algorithm 8.

The Tesla system on which we have run all of our tests offers a memory bandwidth of 101Gb/s. As can be seen the *Gemv* is utilizing a majority of the available bandwidth

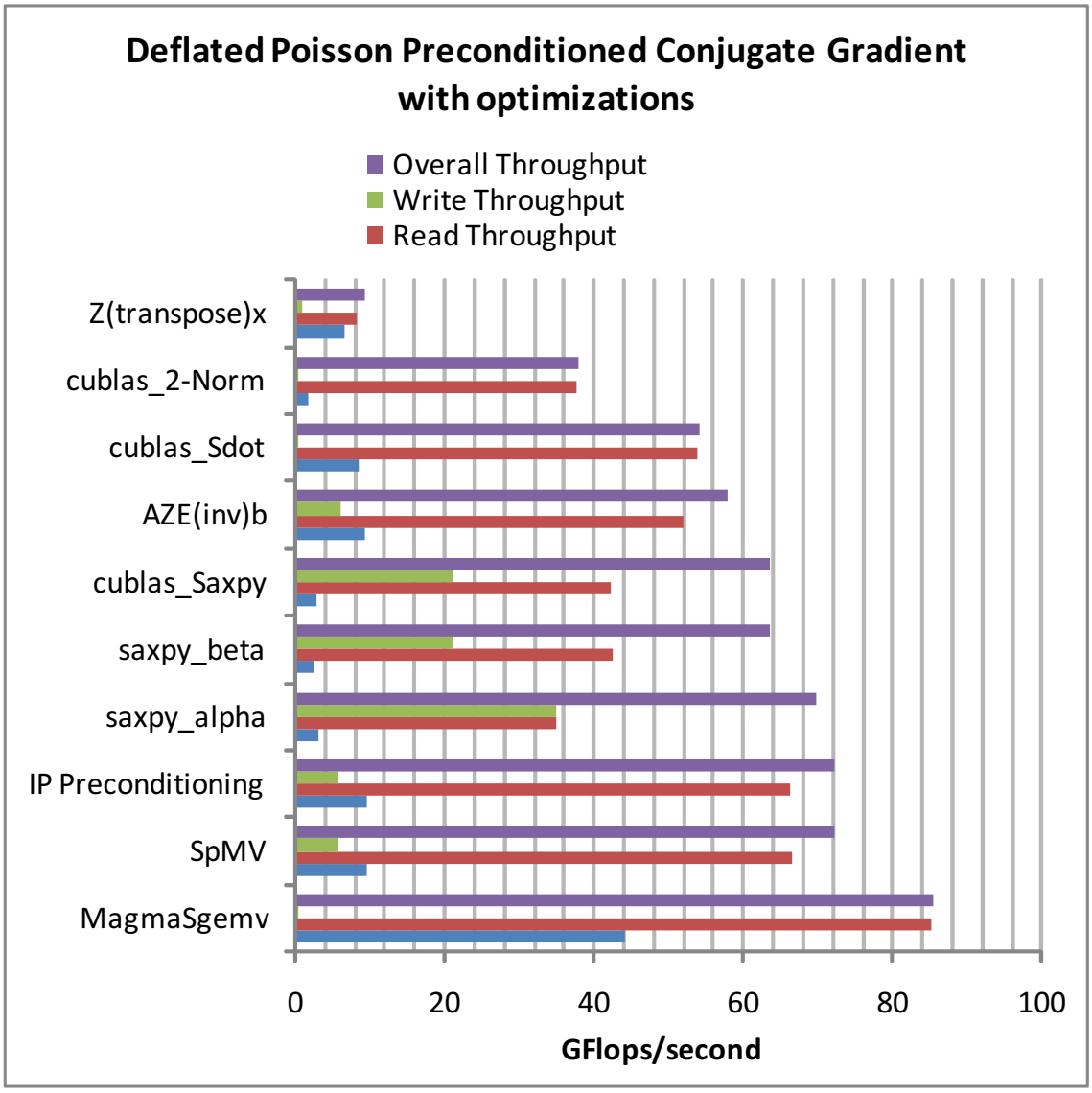

Figure 69: Bandwidth Break-Up. Grid Size (512×512). Number of Deflation Vectors $= 4096$. Optimizations applied to AZ storage and calculation. $E^{-1}b$ calculation optimized with $MAGMA$ Library. Grouping optimizations in Saxpy operations.

(85Gb/s). Followed closely by the IP Preconditioning and SpMV Kernels at 72 Gb/s. These three kernels form 60% of the total execution time. Except for the $CUBLAS$ call for calculating the 2-Norm of the updated residual (stopping criterion - required to be checked every iteration) and the call to calculate $Z^T x$ all the kernels utilise more than half of the available bandwidth. The average Memory throughput of this execution is 68 Gb/s.

Table 3 states the numbers shown in the Figure 69 along with the Occupancy of each of the Kernels.

Kernel Statistics					
Method	%GPUTime	Read Throughput	Write Throughput	Overall Throughput	Occupancy
$MagmaSgemv$	44	85.2	0.02	85.4	50%
$IPPreconditioning$	9.6	66.39	5.75	72.15	100%
$SpMV$	9.6	66.44	5.76	72.2	100%
$AZE^{-1}b$	9.4	51.93	6.08	58.02	100%
$Z^T x$	6.6	8.175	1.02	9.19	50%
$saxpy\_alpha$	3.2	34.85	34.85	69.7	100%
$saxpy\_beta$	2.5	42.64	21.3	63.67	100%
$cublas\_Sdot$	8.6	53.88	0.197	54.08	100%
$cublas\_Saxpy$	2.9	42.34	21.17	63.52	100%
$cublas\_2-Norm$	1.81	37.69	0.223	37.92	100%

Table 3: Grid of 512×512 points. Number of Deflation Vectors = 4096. With optimizations applied to AZ storage and calculation, $E^{-1}b$ with Magma_Sgemv and other optimizations.

9.4 Discussion on Possible Speedup Limits

Given that two of the kernels seem to be operating at 50% occupany we try to find out if they can deliver more performance and hence, a possibility of a higher speedup.

The current kernel for $Z^T x$ is trying to utilize both shared memory and parallel reduction in order to achieve its current bandwidth utilization. We have kept as many threads in the block as are the elements whose sum is required to make one element of the new vector y resulting from $y = Z^T x$. Since in this kernel N/d elements have to be summed in chunks to produce d elements where

$$N = \text{ Number of Unknowns}, \qquad d = \text{ Number of Deflation Vectors.} \tag{106}$$

$$y = d \times 1 \text{ vector}, \qquad x = n \times 1 \text{ vector.} \tag{107}$$

The occupancy varies according to the ratio of N/d but the bandwidth never crosses that indicated in table 3. The kernel's occupancy varies with the factor N/d. For N/d above and equal to 128 (we have values only in multiples of 16) the occupancy is 100%. For the case under consideration the occupancy is 50% but for a lower number of deflation vectors (for e.g. 2048) it is 100% (since N/d becomes 128). Even then the bandwidth does not change. This means that the kernel cannot perform better than this. Trying to comment out the summing operations to confirm this shows that the maximum gain could be that of 4% in the speedup since this kernel only takes 7% of the total time and when only made bandwidth bound it delivers 28Gb/s and takes only 2% of the total time so the speedup can only increase by 5% for this case(particular grid size and deflation vectors).

Though it will increase the shared memory bank conflicts by a factor of 2 (conflicts are already happening when 64 threads reduce to one value). So the effects might be

mitigated. Shared Memory Bank conflicts can also be overcome by changing the storage structure of the vector x however this is not useful since this would require changing many other kernels (which are already performing at 100% occupancy and are bandwidth limited) and also because this kernel is not the most time consuming kernel in the whoel operation.

Other than this kernel ($y = Z^T x$) the other place where there is a possibility of improvement is the $MagmaSgemv$ kernel. Although it is utilizing most of the memory bandwidth it is still having an occupancy of 50%. A closer look at the occupancy for this kernel shows that it has an execution configuration of

$$\text{Grid Size } 64 \times 1 \times 1 \tag{108}$$

$$\text{Block Size } 64 \times 1 \times 1. \tag{109}$$

We used the code for double precision *gemv* posted on the the $MAGMA$ forums which we change to single precision and verify that it is exactly similar (in execution time and execution configuration to the library call).

By modifying the number of blocks in the code form 64 to 128 we get an occupancy of 100%. However the bandwidth stays at 80Gb/s. This shows that the kernel is bandwidth-bound. Since at maximum occupancy we see no change in the bandwidth.

All the other kernels are at 100% occupancy and have simple mathematical operations (Snrm, Sdot, Saxpy, $AZE^{-1}b$) so we can safely say that they are bandwidth bound.

More elaborate analysis of Kernels and the cost of Inter-Warp Parallelism based on Memory Accesses and Computational overlap is possible. [Hong and Kim, 2009] discuss an analytical model for such analysis. However they do not address the issues with Shared Memory Bank Conflicts.

9.4.1 Summary

Considering the maximum speedups that we have had over the entire cycle of development we see a sequence of Ups and downs. In the Figure 70 we try to summarize the time-line of our development process.

As can be seen the speedup did not increase linearly and there were always algorithmic changes that suited the Graphical Hardware sometimes and sometimes they did not. The green line traces how speedup gradually falls. The red line traces the time-line of iterative improvements that we applied to the method.

[Lee, Kim, Chhugani, Deisher, Kim, Nguyen, Satish, Smelyanskiy, Chennupaty, Hammarlund, Singhal, and Dubey, 2010] have recently conducted a study of various kernels across state-of-the-art CPUs and GPUs and have suggested that a single GPU and a single CPU (with 8 cores) have minor speedups on an average case. This kind of analysis must be possible in the further work that can be done in distributing this problem across multiple GPUs and CPUs.

10 Future Work and Conclusions

There are many open questions that still need to be answered for accurate simulation of two phase flows on many-core architectures. These include two important approaches.

1. Try to mathematically optimize or bring close the current model to a more realistic one.

2. The other approach is that of looking at different architectural and software paradigms that can aide in a better comparison and/or implementation.

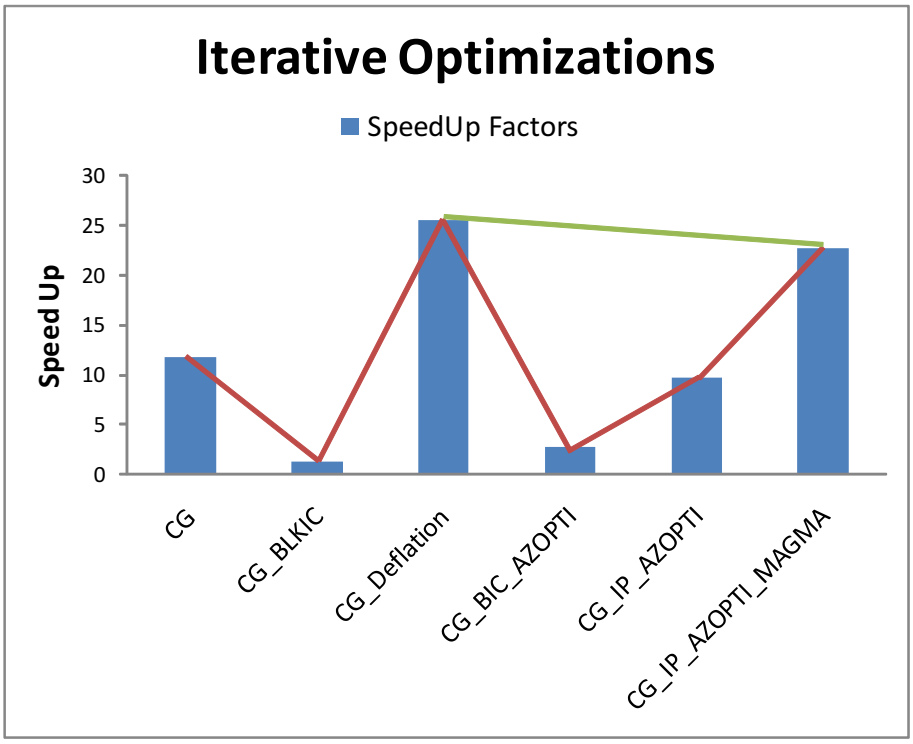

Figure 70: All Variants of the Two Phase Flow Problem

We list some of the tasks that still need to be done in the first place followed by a short discussion and then a similar treatment to the second space.

- Multi-GPU versus Multi-CPU implementation,

[Cevahir, Nukada, and Matsuoka, 2009] suggest a domain decomposition approach for running Conjugate Gradient on Multiple GPUs however they pay a heavy price for CPU-GPU communication. In an improved approach [Griebel and Zaspel, 2010] recently announced a Preconditioned Conjugate Gradient method implemented on Multi-GPU for a similar problem as ours and they utilize the idea of having ghost cells on every GPU. These ghost cells lie on boundaries of parallel blocks which each GPU computes and are packed and unpacked on the GPU/CPU during transfers. Also these memory transfers are overlapped with computation of other kernels on inner data (for e.g. Sparse Matrix Vector Products).

Also on the CPU side it would be interesting to see how the GPU algorithm fares when all the cores on a single CPU die are utilized using openMP. A scale-up study where multiple-GPUs could be compared with multiple CPUs in an MPI based setting could also be interesting future work. Since openMP is based on the idea of cache-coherence its scaling to more than 8 cores will also bring about effects of the overhead involved in keeping the cache up to the mark. Comparing multi-GPU performance to an MPI based cluster implementation has been conducted in the past [Batenburg, 2010] and that kind of comparison could be useful in adapting iterative solution techniques for such hardware.

There is also a software engineering paradigm of further study that can be done on the GPU platform. At present our implementation has used a proprietary language in order to harness the power of the many core paradigm. However OpenCL is an alternative that can make this code future proof to some extent and all also scalable to other architectures that might emerge/evolve from the many-core regime.

In our implementation we have taken some approaches in order for rapidly developing a working solver that can be extended to accommodate more real-world features. Some of

the steps that still need to be incorporated into the model to make it more comprehensive are:

1. Block domains instead of stripes,
2. More phases (More interfaces),
3. 3D Models; and
4. Increased Accuracy through Mixed Precision.

Block domains could be efficiently implemented by re-ordering the coefficient matrix A for the deflation operation. This would consume 5% more space on the GPU, however it would increase the coalescing possibilities manyfold. Block Domains for Deflation have been used in the background work [Tang, 2008] that forms the basis of this implementation.

In Section 8 we have discussed the effects of adding two very different mediums in our simulation. Due to two mediums we have an interface layer where discontinuities amongst the coefficients in the maximum. A next step would be to increase the number of such interfaces. This can also be extended to irregular shapes of such phases for e.g. in the shape of bubbles.

At present we have taken a 2-D view of our problem. However a 3D model would be more interesting and useful for visualizing the results generated.

The GPUs are suited for Single-Precision Arithmetic however future generations (for e.g. Fermi) seem to be incorporating more double precision crunching power into them. This is important for having the precision scale up to the number the CPU can do today. However even with current GPUs it is possible to club two or more 32-bit words to provide a single 64-bit floating point number. Also it is possible to have some parts of the iteration performing calculations in double precision and some in single. Such techniques called the Mixed Precision methods could be used to generate higher precision results on the GPU. Our work can benefit from such implementation and this could be yet another direction in which this study can move forward.

Implemented in Present Version		
Software Engineering	Algorithmic	Architectural
Cublas Magma Meschach	Block Incomplete Cholesky Preconditioning Deflation Incomplete Poisson Preconditioning	Shared Memory Memory Coalescing Lesser Divergence Lesser Warp Serialization
For Future Work		
Software Engineering	Algorithmic	Architectural
OpenCL OpenMP/MPI	Block Domains More Phases 3-D Models	Multi-GPU/Multi-CPU Mixed Precision

Table 4: Present and Future

Some important conclusions and results that have come out of this study must be summarized at this point.

1. Deflation becomes attractive for many core systems due to the inherent parallelism
2. Single (FP) Precision based Deflated Preconditioned Conjugate Gradient for Multi-Phase flows does not have a reliable convergence. It also suffers from a large error.
3. Preconditioning by far remains one of the important bottlenecks while trying to achieve speedup.
4. Incomplete Poisson Preconditioning coupled with Deflation can give real benefits when implemented on the GPU for Conjugate Gradient Method.

References

M. Ament, G. Knittel, D. Weiskopf, and W. Straβer. A parallel preconditioned conjugate gradient solver for the poisson problem on a multi-GPU platform. http://www.vis.uni-stuttgart.de/ amentmo/docs/ament-pcgip-PDP-2010.pdf, 2010.

A. Asgasri and J. E. Tate. Implementing the Chebyshev Polynomial Preconditioner for the iterative solution of linear systems on massively parallel graphics processors. http://www.ele.utoronto.ca/ zeb/publications/, 2009.

M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov. Accelerating scientific computations with mixed precision algorithms. *CoRR*, abs/0808.2794, 2008. URL `http://dblp.uni-trier.de/db/journals/corr/corr0808.html`. informal publication.

J. Batenburg. Fastra. website, 2010. `http://fastra2.ua.ac.be/`.

N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. Technical Report NVR-2008-04, NVIDIA Corporation, December 2008.

Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröoder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, 2003. ISSN 0730-0301.

L. Buatois, G. Caumon, and B. Levy. Concurrent number cruncher: a GPU implementation of a general sparse linear solver. *Int. J. Parallel Emerg. Distrib. Syst.*, 24(3): 205–223, 2009.

A. Cevahir, A. Nukada, and S. Matsuoka. Fast conjugate gradients with multiple GPUs. In *ICCS '09: Proceedings of the 9th International Conference on Computational Science*, pages 893–903, Berlin, 2009. Springer-Verlag.

S. Georgescu and H. Okuda. Conjugate gradients on graphic hardware. Under review in *Lecture Notes in Computer Science*, 2007.

M. Griebel and P. Zaspel. A multi-gpu accelerated solver for the three-dimensional two-phase incompressible navier-stokes equations. *Computer Science - Research and Development*, 25(1-2):65–73, 2010.

M. Harris, S. Sengupta, and J. D. Owens. *Parallel Prefix Sum (Scan) with CUDA*, 2007.

M. Harris, S. Sengupta, J. D. Owens, S. Tseng, Y. Zhang, and A. Davidson. Cudpp. `http://gpgpu.org/developer/cudpp`, 2009.

S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. *SIGARCH Comput. Archit. News*, 37(3):152–163, 2009. ISSN 0163-5964. doi: http://doi.acm.org/10.1145/1555815.1555775.

Intel. Processor specifications - by family. Website, 2010. `http://www.intel.com/support/processors/sb/cs-023143.htm`.

D. Komatitsch, D. Michéa, and G. Erlebacher. Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA. *J. Parallel Distrib. Comput.*, 69(5):451–460, 2009. ISSN 0743-7315. doi: http://dx.doi.org/10.1016/j.jpdc.2009.01.006.

Victor W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, Anthony D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *SIGARCH Comput. Archit. News*, 38(3):451–460, 2010. ISSN 0163-5964. doi: http://doi.acm.org/10.1145/1816038.1816021.

M. M. Baskaran and R. Bordawekar. Optimizing sparse matrix-vector multiplication on GPUs. Technical report, IBM Research Division, NY, USA, December 2008. `http://gpgpu.org/2009/04/13/optimizing-sparse-matrix-vector-multiplication-on-gpus`.

A. Monakov and A. Avetisyan. Implementing blocked sparse matrix-vector multiplication on NVIDIA GPUs. In *SAMOS '09: Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 289–297, Berlin, 2009. Springer-Verlag.

J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008. ISSN 1542-7730.

NVIDIA. Tesla processor specifications. Website, 2010. `http://www.nvidia.com/object/product_tesla_c1060_us.html`.

NVIDIA best prac. *NVIDIA CUDA C Programming Best Practices Guide CUDA Toolkit v*2.3. NVIDIA Corporattion, Santa Clara, 2009.

NVIDIA prog. *NVIDIA CUDA Programming Guide v*2.2. NVIDIA Corporattion, Santa Clara, 2009.

S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu. Optimization principles and application performance evaluation of a multithreaded *gpu* using CUDA. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-795-7.

Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics; 2 edition, Philadelphia, 2003.

Y. Saad, M. Yeung, J. Erhel, and F. Guyomarc'h. A deflated version of the conjugate gradient algorithm. *SIAM J. Sci. Comput.*, 21(5):1909–1926, 2000.

S. SenGupta, M. Harris, Y. Zhang, and J.D. Owens. Scan primitives for GPU computing. *Graphics Hardware*, 2007.

Y.M. Tang. *Two-Level Preconditioned Conjugate Gradient Methods with Applications to Bubbly Flow Problems*. PhD thesis, Delft Unniveristy Of Technology, 2008.

57	58	59	60	61	62	63	64
49	50	51	52	53	54	55	56
41	42	43	44	45	46	47	48
33	34	35	36	37	38	39	40
25	26	27	28	29	30	31	32
17	18	19	20	21	22	23	24
9	10	11	12	13	14	15	16
1	2	3	4	5	6	7	8

Figure 71: 8×8 grid of unknowns

M. Wang, H. Klie, M. Parashar, and H. Sudan. Solving sparse linear systems on NVIDIA Tesla GPUs. In *ICCS '09: Proceedings of the 9th International Conference on Computational Science*, pages 864–873, Berlin, 2009. Springer-Verlag.

A How the Appendix is organized

The Appendix has three main parts.

- Important Concepts used in Implementation.
- Detailed Results.
- Actual Implementation in C and CUDA(available *only* in extended version of this document).

We try to explain what some of the abstractions like domains, blocks and grids (in the algorithmic sense not related to CUDA) mean and how we arrange the matrix we work with to solve $Ax = b$. Section B deals with this introduction.

In the Following section we present the detailed results for Deflated Preconditioned Conjugate Gradient for the Poisson Type and Two Phase Matrix.

B Grid, Matrix, Blocks, Domains, Matrices

We pictorially show how the grid is made followed by the matrix and the abstractions of blocks and domains. These give an idea of how the computation can be divided on the GPU.

B.1 The Grid

The Grid comprises of $n \times n$ elements arranged in the lexicographic order.

We use a 5-point Stencil on it which results in an ordered matrix shown in the next section.

B.2 The Matrix

The matrix (Figure 72)consists of N points where

$$N = n \times n. \tag{110}$$

As can be seen we have 5 diagonals and the matrix is symmetric. The array *offsets* in the $CUDA$ code refers to these diagonals by offsets -2, -1, 0, 1 and 2. The offset -2 for example begins at row 9 column 1. For the reasons mentioned earlier in the $CUDA$ code the diagonals are padded with zeros in the beginning (n zeros for diagonal with offset -2 and 1 zero for diagonal with offset -1) or in the end(n zeros for diagonal with offset 2 and 1 zero for the diagonal with offset 1). This results in 5 arrays of length N. It becomes easier then to multiply the array with a vector of length N.

B.3 Blocks for Incomplete Cholesky

The blocks for IC factorization are always chosen at least larger than n. What it effectively does it is chop out some elements of the off-diagonals (Figure 72) thereby making them (blocks) independent of each other in calculation. This is done so that parallelism can be introduced and further exploited in solving the system.

$$KK^T y = x \tag{111}$$

where x is known and y is sought.

Like in the example previously shown n is 8 and the block size is 16.

B.4 Domains for Deflation

The domains have been chosen stripe-wise for now on the Grid. We vary the number of deflation vectors in multiples of 16. This means that in the 8×8 matrix we will work with 4 partitions of 16 rows each. On the device this means that we will have four separate threads. These threads can work independently. It may be noted that all of these domains have at least as many unknowns as there are in one row of the grid as shown in Figure 71.

So if we take Figure 71 as an example the maximum number of domains possible is 8. Since anymore will lead to a domain having less than 8 unknowns and hence less than one row.

B.5 Coefficients in different types of Matrices

B.5.1 Poisson Type

This is the matrix we use in all the experiments of Section 7. It is based on the 5-point Laplacean in Two Dimensions. In the Table 5 we show a part of an 8×8 grids' resulting matrix. It has five diagonals.

B.5.2 Two-Phase Matrix

This matrix results when we add two mediums to the grid. Pictured in the Table 6 is a part of the two phase matrix for an 8×8 grid. In this case one phase is having three rows of the 8×8 grid and the other one is having 4. The remaining row is that of the interface layer. It can be seen that due to the density contrast between the two mediums $(1000 : 1)$ there are discontinuities on the diagonals.

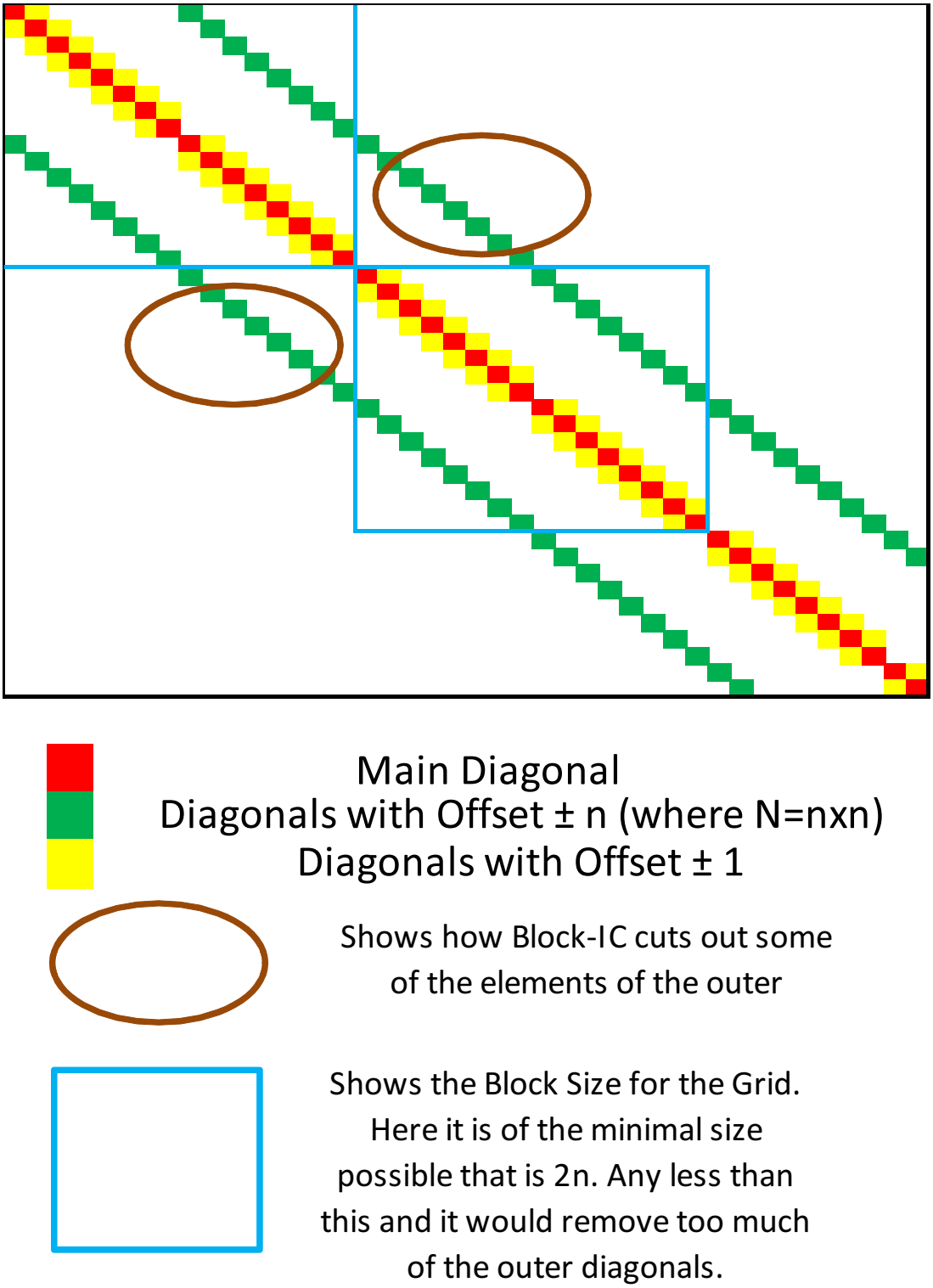

Figure 72: Part of the 64×64 matrix for 8×8 grid.$N = 64$, $n = 8$, Block-Size=16.

4	-1	0	0	0	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	0
-1	4	0	0	0	0	0	0	0	-1	0	0	0	0	0	0	0	0	0	0
0	0	4	-1	0	0	0	0	0	0	-1	0	0	0	0	0	0	0	0	0
0	0	-1	4	-1	0	0	0	0	0	0	-1	0	0	0	0	0	0	0	0
0	0	0	-1	4	-1	0	0	0	0	0	0	-1	0	0	0	0	0	0	0
0	0	0	0	-1	4	-1	0	0	0	0	0	0	-1	0	0	0	0	0	0
0	0	0	0	0	-1	4	-1	0	0	0	0	0	0	-1	0	0	0	0	0
0	0	0	0	0	0	-1	4	-1	0	0	0	0	0	0	-1	0	0	0	0
-1	0	0	0	0	0	0	-1	4	-1	0	0	0	0	0	0	-1	0	0	0
0	-1	0	0	0	0	0	0	-1	4	0	0	0	0	0	0	0	-1	0	0
0	0	-1	0	0	0	0	0	0	0	4	-1	0	0	0	0	0	0	-1	0
0	0	0	-1	0	0	0	0	0	0	-1	4	-1	0	0	0	0	0	0	-1
0	0	0	0	-1	0	0	0	0	0	0	-1	4	-1	0	0	0	0	0	0
0	0	0	0	0	-1	0	0	0	0	0	0	-1	4	-1	0	0	0	0	0
0	0	0	0	0	0	-1	0	0	0	0	0	0	-1	4	-1	0	0	0	0
0	0	0	0	0	0	0	-1	0	0	0	0	0	0	-1	4	-1	0	0	0
0	0	0	0	0	0	0	0	-1	0	0	0	0	0	0	-1	4	-1	0	0
0	0	0	0	0	0	0	0	0	-1	0	0	0	0	0	0	-1	4	0	0
0	0	0	0	0	0	0	0	0	0	-1	0	0	0	0	0	0	0	4	-1
0	0	0	0	0	0	0	0	0	0	0	-1	0	0	0	0	0	0	-1	4

Table 5: Grid Size (8×8). Poisson Type Matrix

4	-1	0	0	0	0	0	0	-1	0	0	0	0	0	0	0	0	0	0
-1	3	0	0	0	0	0	0	0	-1	0	0	0	0	0	0	0	0	0
0	0	1.5015	-0.5005	0	0	0	0	0	0	-0.001	0	0	0	0	0	0	0	0
0	0	-0.5005	2.002	-0.5005	0	0	0	0	0	0	-0.001	0	0	0	0	0	0	0
0	0	0	-0.5005	2.002	-0.5005	0	0	0	0	0	0	-0.001	0	0	0	0	0	0
0	0	0	0	-0.5005	2.002	-0.5005	0	0	0	0	0	0	-0.001	0	0	0	0	0
0	0	0	0	0	-0.5005	2.002	-0.5005	0	0	0	0	0	0	-0.001	0	0	0	0
0	0	0	0	0	0	-0.5005	2.002	-0.5005	0	0	0	0	0	0	-0.001	0	0	0
-1	0	0	0	0	0	0	-0.5005	2.002	-0.5005	0	0	0	0	0	0	-0.001	0	0
0	-1	0	0	0	0	0	0	-0.5005	1.5015	0	0	0	0	0	0	0	-0.001	0
0	0	-0.001	0	0	0	0	0	0	0	0.003	-0.001	0	0	0	0	0	0	-0.001
0	0	0	-0.001	0	0	0	0	0	0	-0.001	0.004	-0.001	0	0	0	0	0	0
0	0	0	0	-0.001	0	0	0	0	0	0	-0.001	0.004	-0.001	0	0	0	0	0
0	0	0	0	0	-0.001	0	0	0	0	0	0	-0.001	0.004	-0.001	0	0	0	0
0	0	0	0	0	0	-0.001	0	0	0	0	0	0	-0.001	0.004	-0.001	0	0	0
0	0	0	0	0	0	0	-0.001	0	0	0	0	0	0	-0.001	0.004	-0.001	0	0
0	0	0	0	0	0	0	0	-0.001	0	0	0	0	0	0	-0.001	0.004	-0.001	0
0	0	0	0	0	0	0	0	0	-0.001	0	0	0	0	0	0	-0.001	0.003	0
0	0	0	0	0	0	0	0	0	0	-0.001	0	0	0	0	0	0	0.003	-0.001
0	0	0	0	0	0	0	0	0	0	0	-0.001	0	0	0	0	0	-0.001	0.004

Table 6: Grid Size (8×8). Two Phase Matrix. With Density Contrast ($1000 : 1$)

57	58	59	60	61	62	63	64
49	50	51	52	53	54	55	56
41	42	43	44	45	46	47	48
33	34	35	36	37	38	39	40
25	26	27	28	29	30	31	32
17	18	19	20	21	22	23	24
9	10	11	12	13	14	15	16
1	2	3	4	5	6	7	8

Figure 73: 4 deflation vectors on a 8×8 grid

C Detailed Results

C.1 Poisson Type

C.1.1 Deflated CG-with Block Incomplete Cholesky Preconditioning

Tables 74 to 76 show the results for this variant across three different grid sizes.

C.2 Two Phase

C.2.1 Deflated CG-with Block Incomplete Cholesky Preconditioning

Tables 77 to 79 show the results for this variant across three different grid sizes.

	Deflated Preconditioned Conjugate Gradient						
	Number of Preconditioning Blocks						
	64		32		16		
Number Of Deflation Vectors							
	Host	Device	Host	Device	Host	Device	
256	68	68	58	58	54	54	Number of Iterations
	0.98436	0.1258	0.8407	0.17238	0.7837	0.2729	Time Taken
	5.64E-05	5.70E-05	5.57E-05	5.60E-05	4.70E-05	4.83E-05	Relative Error Norm of Solution
	7.824		4.879		2.871		SpeedUp
512	53	53	46	46	42	42	Number of Iterations
	1.487	0.1157	1.2921	0.1521	1.1808	0.2265	Time Taken
	7.49E-05	7.30E-05	6.25E-05	5.96E-05	6.51E-05	6.40E-05	Relative Error Norm of Solution
	12.8469		8.4908		5.1213		SpeedUp
1024	34	35	30	30	27	28	Number of Iterations
	1.918	0.1016	1.6424	0.1208	1.5294	0.1713	Time Taken
	2.54E-05	1.76E-05	3.52E-05	2.33E-05	3.14E-05	2.05E-05	Relative Error Norm of Solution
	18.873		13.594		8.9253		SpeedUp

Figure 74: Grid Size 128×128. Different Block Sizes and Deflation Vectors.

	Deflated Precondioned Conjugate Gradient						
	Number of Preconditioning Blocks						
Number Of Deflation Vectors	128		64		32		
	Host	Device	Host	Device	Host	Device	
512	49	49	44	44	40	40	Number of Iterations
	5.482	0.32781	4.9267	0.3512	4.4827	0.48348	Time Taken
	1.37E-03	1.37E-03	1.19E-03	1.19E-03	1.47E-03	1.47E-03	Relative Error Norm of Solution
	16.723		14.0283		9.271		SpeedUp
1024	44	44	43	43	36	36	Number of Iterations
	9.7168	0.39116	9.50174	0.43833	7.9667	0.51661	Time Taken
	1.36E-03	1.36E-03	8.40E-04	8.40E-04	1.04E-03	1.04E-03	Relative Error Norm of Solution
	24.84		21.676		15.421		SpeedUp
2048	38	38	35	35	31	31	Number of Iterations
	16.7373	0.524	15.431	0.52294	13.6899	0.5987	Time Taken
	5.38E-04	5.38E-04	4.80E-04	4.78E-04	4.91E+00	4.91E-04	Relative Error Norm of Solution
	31.935		29.508		22.865		SpeedUp

Figure 75: Grid Size 256×256. Different Block Sizes and Deflation Vectors.

	Deflated Precondioned Conjugate Gradient						
	Number of Preconditioning Blocks						
Number Of Deflation Vectors	256		128		64		
	Host	Device	Host	Device	Host	Device	
1024	43	43	38	38	34	34	Number of Iterations
	37.849	2.0226	33.496	1.5849	29.9782	1.6505	Time Taken
	3.76E-03	3.75E-03	3.57E-03	3.57E-03	3.84E-03	3.85E-03	Relative Error Norm of Solution
	18.693		21.133		18.162		SpeedUp
2048	43	43	38	38	34	34	Number of Iterations
	75.013	3.15	66.428	2.586	59.412	2.5636	Time Taken
	4.40E-03	4.39E-03	4.05E-03	4.05E-03	4.14E-03	4.14E-03	Relative Error Norm of Solution
	23.8		25.686		23.175		SpeedUp

Figure 76: Grid Size 512×512. Different Block Sizes and Deflation Vectors.

Deflated (Block - IC) Preconditioned Conjugate Gradient							
	Two Phase Matrix with Precision Criteria (10e-02)						
	Number of Preconditioning Blocks						
Number Of Deflation Vectors	64		32		16		
	Host	Device	Host	Device	Host	Device	
256	3	3	3	3	3	3	Number of Iterations
	0.00572	0.00495	0.0057	0.0083	0.00573	0.01456	Time Taken
	1.18E+00	1.21E+00	1.18E+00	1.25E+00	1.14E+00	1.16E+00	Norm of Solution
	1.155		0.6866		0.393		SpeedUp
512	3	3	3	3	3	3	Number of Iterations
	0.00676	0.00542	0.00679	0.00869	0.00684	0.0149	Time Taken
	1.00E+00	1.06E+00	1.05E+00	1.02E+00	9.58E-01	1.02E+00	Norm of Solution
	1.2471		0.7808		0.4575		SpeedUp
1024	4	4	3	3	3	3	Number of Iterations
	0.0142	0.00813	0.0113	0.0965	0.0113	0.158	Time Taken
	9.13E-01	8.83E-01	1.02E+00	1.03E+00	1.01E+00	1.07E+00	Norm of Solution
	1.757		1.172		0.717		SpeedUp

Figure 77: Grid Size 128×128. Different Block Sizes and Deflation Vectors.

Deflated (Block - IC) Preconditioned Conjugate Gradient							
	Two Phase Matrix with Precision Criteria (10e-02)						
	Number of Preconditioning Blocks						
Number Of Deflation Vectors	128		64		32		
	Host	Device	Host	Device	Host	Device	
512	4	4	4	4	4	4	Number of Iterations
	0.0318	0.018	0.032	0.0232	0.032	0.0395	Time Taken
	4.13E-01	4.15E-01	4.36E-01	4.58E-01	4.60E-01	3.67E-01	Norm of Solution
	1.739		1.376		0.808		SpeedUp
1024	1000NC	1000NC	4	4	4	4	Number of Iterations
	8.858	4.503	0.039	0.024	0.0401	0.0401	Time Taken
	1.53E+02	3.24E+03	2.59E-01	1.64E-01	3.15E-01	2.73E-01	Norm of Solution
	1.967		1.64		0.99		SpeedUp
2048	1000NC	1000NC	1000NC	1000NC	1000NC	1000NC	Number of Iterations
	12.49	5.228	12.714	6.523	12.96	10.642	Time Taken
	2.00E+06	2.45E+05	9.80E+03	1.62E+06	1.43E+05	5.07E+05	Norm of Solution
	2.38		1.949		1.217		SpeedUp

Figure 78: Grid Size 256×256. Different Block Sizes and Deflation Vectors.

Deflated (Block - IC) Preconditioned Conjugate Gradient							
	Two Phase Matrix with Precision Criteria (10e-02)						
	Number of Preconditioning Blocks						
Number Of Deflation Vectors	256		128		64		
	Host	Device	Host	Device	Host	Device	
1024	4	4	3	3	3	3	Number of Iterations
	0.13028	0.0861	0.1	0.0486	0.1006	0.068	Time Taken
	3.56E-01	4.71E-01	4.39E-01	5.09E-01	4.03E-01	5.36E-01	Norm of Solution
	1.511		2.07		1.478		SpeedUp
2048	1000NC	1000NC	1000NC	1000NC	1000NC	1000NC	Number of Iterations
	34.56	21.8	34.63	16.253	34.93	22.8	Time Taken
	2.67E+03	1.37E+04	2.92E+04	2.18E+03	1.97E+04	3.26E+04	Norm of Solution
	1.583		2.13		1.532		SpeedUp
4096	1000NC	1000NC	1000NC	1000NC	1000NC	1000NC	Number of Iterations
	49.3	24.356	49.603	19.02	49.739	24.589	Time Taken
	3.12E+07	4.81E+06	2.69E+05	9.24E+04	3.56E+05	3.88E+05	Norm of Solution
	2.026		2.606		2.022		SpeedUp

Figure 79: Grid Size 512×512. Different Block Sizes and Deflation Vectors.