DELFT UNIVERSITY OF TECHNOLOGY

REPORT 11-15

Efficient Two-Level Preconditionined Conjugate Gradient Method on the GPU.

Rohit Gupta, Martin B. van Gijzen and Kees Vuik

ISSN 1389-6520

Reports of the Department of Applied Mathematical Analysis

Delft 2011

Abstract

We present an implementation of Two-Level Preconditioned Conjugate Gradient Method for the GPU. We investigate a Truncated Neumann Series based preconditioner in combination with deflation and compare it with Block Incomplete Cholesky schemes. This combination exhibits fine-grain parallelism and hence we gain considerably in execution time. It's numerical performance is also comparable to the Block Incomplete Cholesky approach. Our method provides a speedup of up to 16 times for a system of one million unknowns when compared to an optimized implementation on the CPU.

1 Introduction

Multi-phase flows occur when different fluids interact. These fluids have different properties, e.g., in densities. To simulate two-phase flow we use the Incompressible Navier Stokes Equation. A representative example of such a flow could be thought of as an air bubble rising in water. Such phenomena frequently arise in physical processes like oil refineries and nuclear reactors and understanding them can affect the design and efficiency of such processes.

1.1 Motivation

Our work is motivated by the Mass-Conserving Level Set approach (8) to solve the Navier Stokes equations for multi-phase flow. The most time consuming step in this approach is the solution of the (discretized) pressure-correction equation, which is a poisson equation with discontinuous coefficients.

For our research we choose a model problem with an interface layer that divides a square domain. More details are provided in Section 2. The discretized pressure-correction equation, takes the form of a linear system

$$Ax = b,\ A \in \mathbb{R}^{N\times N},\ N \in \mathbb{N} \tag{1}$$

where N is the number of degrees of freedom. A is symmetric positive definite (SPD). The entries of this matrix depend on the densities of the fluids involved. So, e.g., if we consider a gas and liquid mixture then their densities have a ratio of the order of 10^3. This leads to a large condition number $\kappa^\dagger$ for the matrix A, and slow convergence.

$^\dagger\kappa = \frac{\lambda_N}{\lambda_1}$, where $0 < \lambda_1 \leq \lambda_2 ... \leq \lambda_N$ are eigenvalues of the matrix A arranged in ascending order

1.2 Focus of this research

The convergence for the system $Ax = b$ mentioned in the previous section is slow when an iterative method like the Conjugate Gradient (CG) is applied. To achieve faster convergence, the linear system is preconditioned:

$$M^{-1}Ax = M^{-1}b, \tag{2}$$

where the matrix M is symmetric and positive. The choice of M is such that the operation $M^{-1}y$, for some vector y, is computationally cheap and M can also be stored efficiently. We first consider Block Incomplete Cholesky Precondtioner which is an established method and provides some level of parallelism. However, due to the block structure of the preconditioner and also because of small eigenvalues (in the matrix A) the preconditioned matrix $M^{-1}A$ still has a high condition number. In order to further reduce it we have to apply a second level of preconditioning called deflation (10). Deflation removes the smaller eigenvalues so that the condition number of the matrix $M^{-1}A$ is reduced. For more details we refer the interested reader to (23).

For the GPU, however, Block Incomplete Cholesky preconditioning is not the optimal choice. It is inherently sequential in every block. The amount of data-parallelism is limited by the number of blocks. However, increasing the number of blocks degrades the effectiveness of the preconditioner. Through this research we aim to find preconditioning schemes that offer fine-grain parallelism. Hence they would be better suited to the GPU and at the same time should prove effective in bringing down the condition number of $M^{-1}A$. We compare the schemes we have developed with Block-Incomplete Cholesky (Block-IC) Preconditioners, as a benchmark to check the quality of our preconditioning schemes. The numerical performance of the preconditioners we introduce in this paper comes close to it's Block-IC counterparts for our model problem and they also offer more parallelism for the GPU.

1.3 Related work

With the advent of CUDA in 2007 scientific computing, it became easier to develop for the NVIDIA GPU platform. Some of the earliest advances into many core computing had to do otherwise (6).

Developing with CUDA had its caveats and it required an understanding of the way applications are executed on the GPU hardware (22) to get the maximum performance and parallelism promised by a GPU. This challenge was duly taken up by the scientific community. In particular for iterative methods the GPUs were proven to be extremely useful as suggested in a number of earlier works ((15), (18), (19), (20) and (14)).

There have been some previous works that have explored preconditioners with similar properties and we provide a brief overview in this section. In (1),

a highly parallelizable preconditioner was introduced that was specifically designed for a Poisson type problem. Our experience with this preconditioner tells us that it is not an effective preconditioner for ill-conditioned matrices, though it works very well for smooth Poisson problems. The preconditioning technique we introduce in this paper is able to overcome this limitation. Comparisons with existing schemes (implemented on both CPU and GPU) establish its relevance to achieve convergence quickly and accurately. We discuss this preconditioning in more detail in Section 3.3.

The preconditioning technique mentioned in (11), utilizes the same idea as presented in (1) albeit with a relaxation factor. In (11) the authors use the Neumann Series approximation to devise a new preconditioner with the same focus i.e. to adapt the preconditioning techniques for the many-core platform.

In (12), the authors used an LU decomposition based preconditioner with fill-in, reordered using multi-coloring. They decide in advance on the sparsity pattern of the incomplete factorization based on the matrix power $|A|^{p+1}$ and it's multi-coloring permutation. In contrast, our approach is based on the strictly lower triangular part of the scaled version of the coefficient matrix A. The scaling is applied once and can be done in parallel so the cost is minimal.

This paper is organized as follows: in the next section we present a brief discussion of the discretization approach used to construct the matrix A. A brief overview of the preconditioning schemes and their features can be found in Section 3. We discuss the approach of second level preconditioning in Section 4. In Section 5 we list the Conjugate Gradient Algorithm with Preconditioning and Deflation. Furthermore we comment on two different implementation methods for this method in Section 6. In Section 7 we present our results and we end with a discussion in Section 8.

2 Problem Definition

In this section we present the test problem used to evaluate the different algorithms.

2.1 Test Problem

The discretized pressure-correction equation for a 2-D square grid ($n \times n$) takes the form of 5-point Poisson type matrix for our method (as mentioned in Section 1.1). One set of diagonals have offsets ± 1 and the other two have offsets of $\pm n$ with respect to the main diagonal. For a system with only one phase, the matrix would have the stencil, $[-1,\ -1,\ 4,\ -1,\ -1]$ for an inner cell.

However, with the introduction of multiple phases we see a jump in the coefficient values at the interface. This jump is also visible in the eigenval-

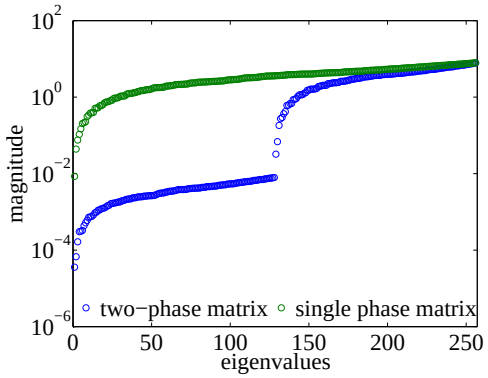


Figure 1: 2D grid (16×16) with 256 unknowns. Jump at the Interface due to density contrast.

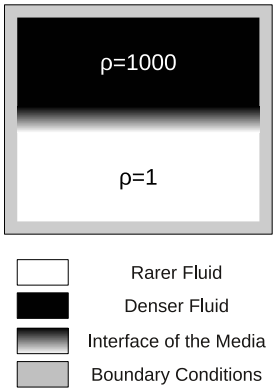


Figure 2: Two phase Flow Computational Model

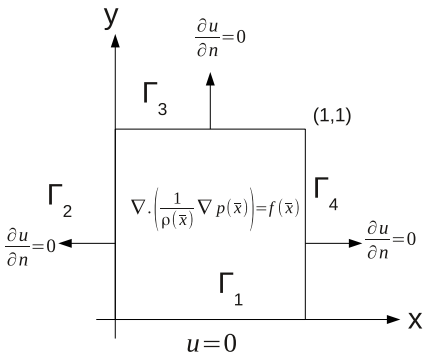


Figure 3: Unit Square with boundary conditions.

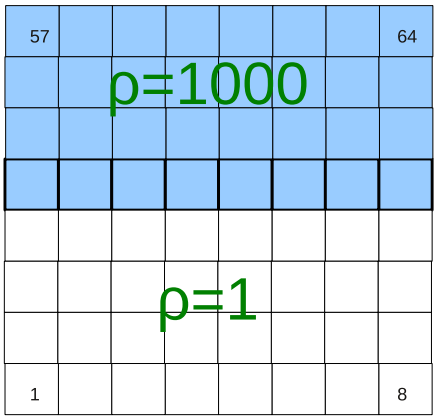


Figure 4: Discretized Unit Square with Interface

ues. To show this we consider the simple case of a 2-D grid and plot the eigenvalues in Figure 1. The jump results from the density contrast of the two fluids. This jump in eigenvalues leads to a large condition number, κ.

The computational domain can be pictured as in Figure 2. It has two fluids with a high density contrast and appropriate boundary conditions. We define a unit square as our domain (Figure 3) and an interface at the middle of this square (Figure 4).

2.2 Solution of the Discrete System

After making the coefficient matrix using the stencils, we solve the resulting system with an iterative method. We define a desired accuracy of the solution we wish to achieve. The iterative method refines the solution vector x

at every step until it reaches a desired norm of the residual

$$\| r_k \|_2 \leq \| b \|_2 \epsilon, \tag{3}$$

where r_k is the residual at the k-th step, b[‡] is the right-hand side and ϵ is the tolerance. In our experiments we choose $\epsilon = 10^{-6}$. We measure the accuracy of our results using the relative error norm of the solution. It is defined as the difference of the computed solution and the exact[§] solution,

$$\frac{\| x_{exact} - x_k \|_2}{\| x_{exact} \|_2}, \tag{4}$$

where x_k is the solution computed by the iterative method at the k^{th} step of the iteration and x_{exact} is the exact solution. The initial guess (x_0) is a random vector to avoid artificially fast convergence due to a smooth solution. The ill-conditioned two-phase matrix requires that we augment the Preconditioned Conjugate Gradient method with a second level preconditioner. This is essential to achieve reasonable performance of the iterative solver. This technique of second level preconditioning, called Deflation, was investigated in detail in (5).

It is essential to mention here that our simple model can be easily extended to a 3D model since then the only change is to the coefficient matrix where the number of non-zeros in each row of A increases. More diagonals are added with larger offsets. For example for a 7-point stencil in 3D we can have two more diagonals at $n_x \times n_y$ offsets if the grid dimensions are $n_x \times n_y \times n_z$.

For the realistic problem of bubbly flow we consider the domain being composed of air bubbles rising in water. The coefficient matrix for this problem has similar properties as defined in Section 1.1. Only now the interfaces are not as clear as in the model problem shown in Figure 2. Instead there would be spheres/circles cutting a cell partially. To formulate the matrix A for such a multi-bubble/multi-phase case, suitable approaches (cut cell, weighted averaged) can be used as suggested in (8).

3 Preconditioning Schemes

In this section we first discuss standard preconditioning schemes for matrix A when using an iterative method like Conjugate Gradient. Further, we provide details of the preconditioning schemes that we have developed.

[‡] $b = Ax_{exact}$

[§] The exact solution is generated using the cosine function $x_{exact}(i) = cos(i-1)$, where $i = 1...N$. N is the number of unknowns.

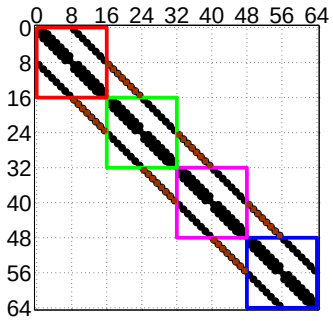


Figure 5: Block-IC block structure. Elements excluded from a 8×8 grid for a block size of $2n = 16$.

3.1 Block Incomplete Cholesky preconditioning

We consider an adapted incomplete Cholesky decomposition: $A = LD^{-1}L^T - R$ where the elements of the lower triangular matrix L and diagonal matrix D satisfy the following rules:

1. $l_{ij} = 0 \ \forall \ (i,j)$ where $a_{ij} = 0 \ i > j$,
2. $l_{ii} = d_{ii}$,
3. $(LD^{-1}L^T)_{ij} = a_{ij} \ \forall \ (i,j)$ where $a_{ij} \neq 0 \ i \geq j$.

In order to make blocks for the Block-Incomplete Cholesky approach we first apply the block structure on the matrix A. In Figure 5 we show how some of the elements belonging to the 5-point Poisson type matrix are dropped when a block incomplete scheme is applied to make the preconditioner for A. The red colored dots are the elements that are dropped since they lie outside the blocks. With this A we make the L as suggested in (13).

In each iteration we have to compute y from

$$y = M_{BIC}^{-1} r, \text{ where } M_{BIC} = LD^{-1}L^T. \tag{5}$$

This is done by doing forward substitution followed by diagonal scaling and then backward substitution. The number of blocks is N/g, where g is the size of the blocks. Within a block all calculations are sequential. However, each block forms a system that can be solved independently of other blocks. Hence they can be solved in parallel.

The parallelism offered by Block-IC is limited to the number of blocks. In order to increase the number of parallel operations we must decrease the block size, g. However, doing this would lead to more loss of information, and consequently, delayed convergence. So, we must think of alternatives. In this paper the Block-IC Preconditioners are characterized by their block size. We denote it by a number in brackets e.g. $blkic(8n)$ or $M_{Blk-IC(8n)}^{-1}$.

Here $8n$ is the block-size so for a matrix A with $N = n \times n$ unknowns, hence the number of blocks is $\frac{n}{8}$.

3.2 Incomplete Poisson preconditioning

In (1), a new kind of incomplete preconditioning is presented. The preconditioner is based on a splitting of the coefficient matrix A using it's lower triangular part L and the diagonal D,

$$A = L + D + L^T. \tag{6}$$

Specifically the preconditioner is defined as,

$$M_{IP}^{-1} = (I - LD^{-1})(I - D^{-1}L^T), \tag{7}$$

where L is the strictly lower triangular part of A and D is the diagonal matrix containing diagonal elements of A. The arrangement of the terms in (7) seems unusual (transpose terms after non-transpose terms) but the authors use this expression in their work. After calculation of the entries of M^{-1}, the values that are below a certain threshold value are dropped, which results in an incomplete decomposition that is applied as the preconditioner. As this preconditioner was used for a Poisson type problem, the authors call it the Incomplete Poisson (IP) preconditioner. A detailed heuristic analysis about the effectiveness of this preconditioning technique can be found in (1). The main advantage of this technique is that it (Equation (5)) is reduced to sparse matrix vector products[¶] which have been heavily optimized for many-core platforms (7). However, this algorithm if used in the form as presented in the original publication gives slow convergence for a two-phase matrix. It must be used with the scaled versions of A, x and b.

$$\tilde{A} = D^{-\frac{1}{2}} A D^{-\frac{1}{2}}. \tag{8}$$

$$\tilde{x} = D^{\frac{1}{2}} x. \tag{9}$$

$$\tilde{b} = D^{-\frac{1}{2}} b. \tag{10}$$

In such a setting the results are much better. We call this improvement Incomplete Poisson preconditioning with scaling. This technique can be further improved as we show in the next section.

3.3 Neumann Series based preconditioning

Building up on the improvements we found in the previous section for Incomplete Poisson Preconditioning we now define the preconditioning matrix, M as

$$M = (I + \tilde{L})(I + \tilde{L}^T), \tag{11}$$

[¶] M_{IP}^{-1} is stored just like A in DIA format

where $\tilde{L}$ is the strictly lower triangular part of $\tilde{A}$ as mentioned in (8). In order to calculate M^{-1} we use the Neumann Series of $(I+\tilde{L})$ and $(I+\tilde{L}^T)$. This is defined as

$$(I+\tilde{L})^{-1} \approxeq I - \tilde{L} + \tilde{L}^2 - \tilde{L}^3 + \cdots . \tag{12}$$

The series converges if

$$\| \tilde{L} \|_\infty < 1. \tag{13}$$

This is true for our problem and hence the Neumann Series is a valid choice for approximating the inverse of $(I+\tilde{L})$. So we can redefine M^{-1} as

$$M^{-1} = (I - \tilde{L}^T + \cdots)(I - \tilde{L} + \cdots) \tag{14}$$

For making our preconditioners we truncate the series (12) after 1 or 2 terms. We refer to these as the Neu1 and Neu2 Preconditioners. Note that

$$M_{Neu1}^{-1} = (I - \tilde{L}^T)(I - \tilde{L}) \tag{15}$$

and

$$M_{Neu2}^{-1} = (I - \tilde{L}^T + (\tilde{L}^T)^2)(I - \tilde{L} + \tilde{L}^2). \tag{16}$$

We define $K = (I - \tilde{L})$ for (15) and $K = (I - \tilde{L} + \tilde{L}^2)$ for (16). Note that the order of terms (transposed and non-transposed) is as expected. It appears that M_{Neu1}^{-1} and M_{IPScal}^{-1} have approximately the same convergence behavior. M_{Neu1}^{-1} has the same number of operations per application when compared to Incomplete Poisson and Block-IC variants. M_{Neu2}^{-1}, which has another higher order term in K has 2 times as many.

For Incomplete Poisson, we store M_{IP}^{-1} one time (similar to A or $\tilde{A}$) and use it over and over again in the iteration for the operation $M^{-1}r$. Since M^{-1} has the same sparsity structure as A (5 diagonals). Total computation cost of $M^{-1}r$ is $10N$ operations which is the same as for Ax.

For the preconditioner as given by (15) we calculate K^TKx by term$^{\|}$ in every iteration. Every term in the expansion of $M^{-1} = K^TK$ can be (roughly) computed at the cost of one Lx operation. This is around $2N$ multiplications and N additions.

3.4 Eigenvalue Spectrum Comparison of different preconditioning schemes

In this section we show how the eigenvalue spectrum changes for different preconditioning schemes. The plots are for a 2-D grid with $N = 4096$ unknowns arranged in an $n \times n$ grid where $n = 64$. The contrast in densities is $1000 : 1$. Condition numbers for the different preconditioners applied to this matrix are listed in Table 1.

Preconditioning Scheme	Condition Number for ($M^{-1}A$ or $M^{-1}\tilde{A}$)
Block-IC (blocksize=$2n$)	2.50e+03
Block-IC (blocksize=$4n$)	1.93e+03
Block-IC (blocksize=$8n$)	1.60e+03
IP (without scaling)	2.97e+06
IP (with scaling)	2.68e+03
Truncated Neumann(Neu1)	2.66e+03
Truncated Neumann(Neu1)	1.74e+03

Table 1: Condition Numbers after Preconditioning. Condition number of $A = 9.68e + 03$.

In each of the plots of Figure 6 the eigenvalues of A and the eigenvalues of the preconditioned versions $M^{-1}A$ are plotted. The eigenvalues of A show a jump due to the contrast in densities. All preconditioners are efficient except for IP without scaling (M_{IP}^{-1}). IP with scaling (M_{IPScal}^{-1}) performs comparable to M_{Neu1}^{-1}. The Block Incomplete Cholesky preconditioners are successful in bringing most eigenvalues to 1 which also brings down the condition number, κ. Note that κ decreases as the block size increases. The Neumann type preconditioning schemes perform similar to Block-IC in these respects, with the second type (with $K = I - L + L^2$) performing as good as Block-IC($8n$). For the Truncated Neumann Series preconditioners we always use the diagonally scaled version of A. For Block-IC and Truncated Neumann Series based preconditioners we provide a separate zoom to show the effect of preconditioner on smaller eigenvalues.

4 Deflation

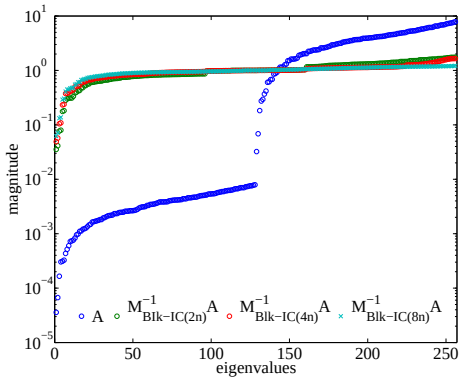
To improve the convergence of our method we also use a second level of preconditioning. Deflation aims to remove the remaining bad eigenvalues from the preconditioned matrix, $M^{-1}A$ or $M^{-1}\tilde{A}$. This operation increases the convergence rate of the Preconditioned Conjugate Gradient (PCG) method.

If we assume

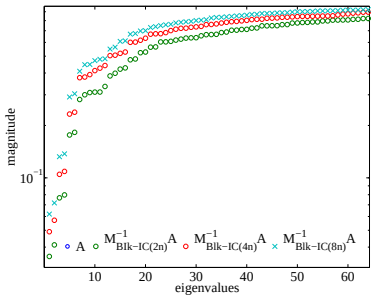
$$P = I - AQ, Q = ZE^{-1}Z^T, E = Z^TAZ, \tag{17}$$

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 operator. Z is the so-called 'deflation-subspace matrix' whose k columns are called 'deflation' vectors

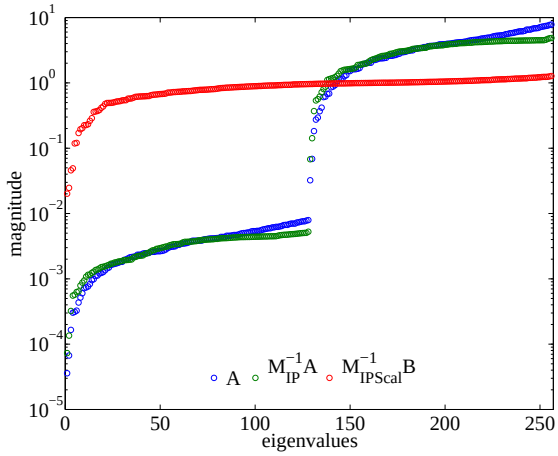
$\|K^TKx = (I - L^T)y$, where $y = (I - L)x$
Number of operations = $(2N{+}2N)$[multiplications] + $(4N)$[additions]



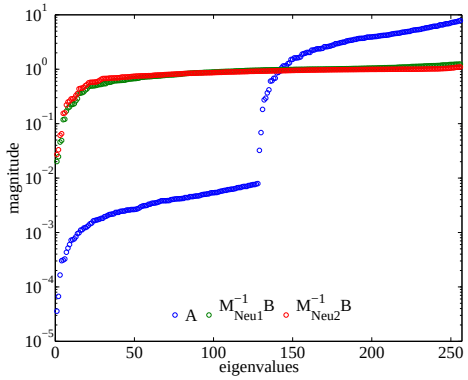
(a) Blk-IC



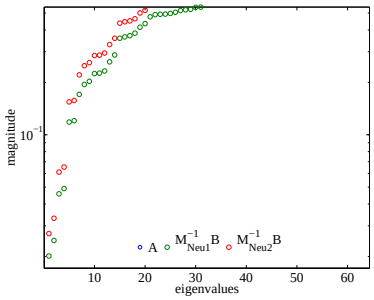
(b) Blk-IC : small eigenvalues



(c) IP



(d) Truncated Neumann



(e) Truncated Neumann : small eigenvalues

Figure 6: Spectrum of Preconditioned Matrix for a 2D two-phase problem. $B = \tilde{A} = D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$

or 'projection' vectors. The linear system $Ax = b$ can then be solved by employing the splitting

$$x = (I - P^T)x + P^T x \Leftrightarrow x = Qb + P^T x \tag{18}$$

$$\Leftrightarrow Ax = AQb + AP^T x \tag{19}$$

$$\Leftrightarrow b = AQb + PAx \tag{20}$$

$$\Leftrightarrow Pb = PAx. \tag{21}$$

The x at the end of the expression is not necessarily a solution of the original linear system, since it might contain 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

$$PA\hat{x} = Pb. \tag{22}$$

4.1 Deflated Preconditioned Conjugate Gradient Method

The deflated preconditioned version of the Conjugate Gradient Method can now be presented. The deflated system (22) can be solved using a symmetric positive definite (SPD) preconditioner, M^{-1}. We therefore seek a solution of

$$M^{-1}PA\hat{x} = M^{-1}Pb. \tag{23}$$

The resulting method is called the Deflated Preconditioned Conjugate Gradient (DPCG) method (details in (23)).

We choose Sub-domain Deflation and use piecewise constant deflation vectors. We make stripe-wise deflation vectors (see Figure 9) unlike the block deflation vectors suggested in (5). These vectors lead to a regular structure for AZ and, therefore, an efficient storage of AZ.

5 Two Level Preconditioned Conjugate Gradient -Implementation

In Algorithm 1 we list out the steps involved in a typical implementation of the Deflated Preconditioned Conjugate Gradient method. We follow this implementation in writing out the code for GPU and CPU.

The deflation operation $\hat{w_j} := PAp_j$ is broken into the following steps:

1. Set $a_1 = Z^T p_j$.
2. Solve $Ea_2 = a_1$.
3. Set $a_3 = AZa_2$.
4. Set $w_j = p_j - a_3$.

Algorithm 1 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: $x_{it} := Qb + P^T x_{j+1}$

Step 2 can be solved in two different ways as we will see in Section 7. The kernel for step 1 is a sum operation. Steps 3 and 4 can be done by one kernel. Given the storage format we choose for AZ these operations reduce to similar number of operations, memory access pattern and performance as the sparse matrix vector product Ax.

5.1 Storage of the matrix AZ

The structure of the matrix AZ if stored as an $N \times d$ matrix, where d is the number of domains/deflation vectors, can be seen in Figure 7. In Figure 7 to 9 it must be noted that $d = 2 \times n$ here and $N = n \times n = 64$, $n = 8$. The AZ matrix is formed by multiplying the Z matrix (a part of which is shown in the adjoining figure of matrix AZ in Figure 7) with the coefficient matrix, A. The colored boxes indicate non-zero elements in AZ. They have been color coded to provide reference for how they are stored in the compact form. The red elements are in the same space as the deflation vector. The green elements result from the horizontal fill-in and the blue elements result from the vertical fill-in. The arrangement of the deflation vectors (on the grid) is shown in Figure 9. Each ellipse corresponds to the non-zero part of the corresponding deflation vector in matrix Z. The trick to store AZ in an efficient way (for the GPU) is to make sure that memory accesses are ordered. For this we need to have a look at how the operation $a_3 = AZa_2$ works, where a_2 is a $d \times 1$ vector. For each element of the resulting vector a_3 we need an element from at most 5 different columns of the AZ matrix. Now it must be recalled that in case of A multiplied with x we have 5 elements of A in a single row multiplied with 5 elements of x as detailed in (7). So we start looking at the different colored elements and group them so that the access pattern to calculate each element of a_3 is similar to the Sparse-

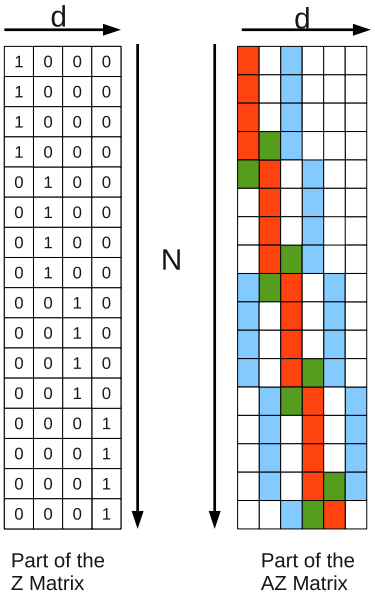


Figure 7: Parts of Z and AZ matrix. Number of Deflation Vectors $=2n$.

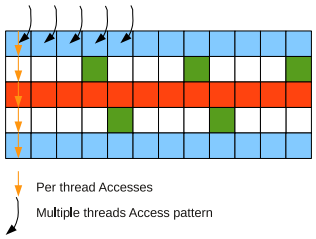


Figure 8: AZ matrix after compression.

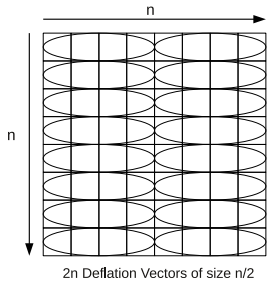


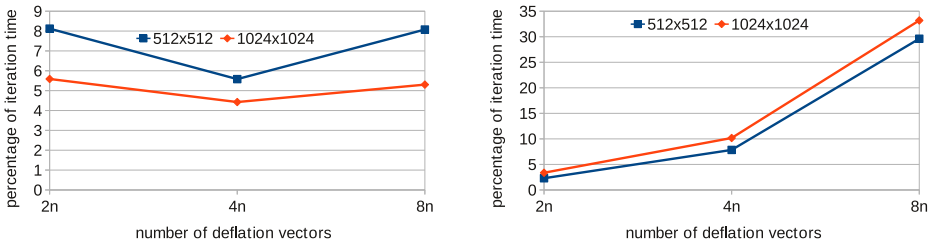
Figure 9: Deflation Vectors for the 8×8 grid.

Matrix Vector Product operation. Wherever there is no element in AZ we can store a zero. So writing out all the red elements is trivial as they are N in number. Similarly the blue elements can also be written in a row. Only they would have an offset at the beginning or the end that must be padded with zeros in order to make them N entries long.

For the green elements we look at each of the columns in which they are present. We store one set of green non-zeros which correspond to the left of the domain in the second row of the data structure shown in Figure 8 and the other set is stored in the $4th$ row. Thus in the compacted form the $N \times d$ matrix AZ can be stored in $5N$ elements as illustrated in Figure 8.

Stored in such a way the AZ matrix can be used to do faster calculation of matrix vector products within the iteration. The golden arrows in Figure 8 show how each thread on the GPU can compute one element when the operation AZa_2 is performed where a_2 is a $d \times 1$ vector. The black arrows show the accesses done by multiple threads. This is similar to the DIA format of storage and calculating Sparse Matrix Vector Product as suggested in (7).

Other than deflation and preconditioning the algorithm involves Sparse Matrix Vector Products (SpMVs), Dot Products, Saxpy's etc. These operations form the building blocks that must be optimized for a parallel/many-core version of the code. For standard operations like dot products, norms, daxpys we use the CUBLAS library on the GPU and ATLAS library on the



(a) CPU (b) GPU

Figure 10: Setup+Initialization Time as a percentage of the total time for triangular solve approach across different sizes of deflation vectors for DPCG.

CPU.

6 GPU Implementation of Deflation

Since we are working with a 5-point stencil we have a regular sparse matrix. We store the matrix in the Diagonal (DIA) format and we follow the implementation as detailed in (7). We also managed to store the matrix AZ in a similar format. The details can be found in (9).

Other operations include solving the system $E^{-1}x = b$ which can be solved in two ways. Here E is the Galerkin Matrix given by $E = Z^T AZ$.

1. Calculating E^{-1} explicitly so that the $E^{-1}b$ becomes a dense matrix vector product which can be calculated using the MAGMA BLAS library for the GPU.

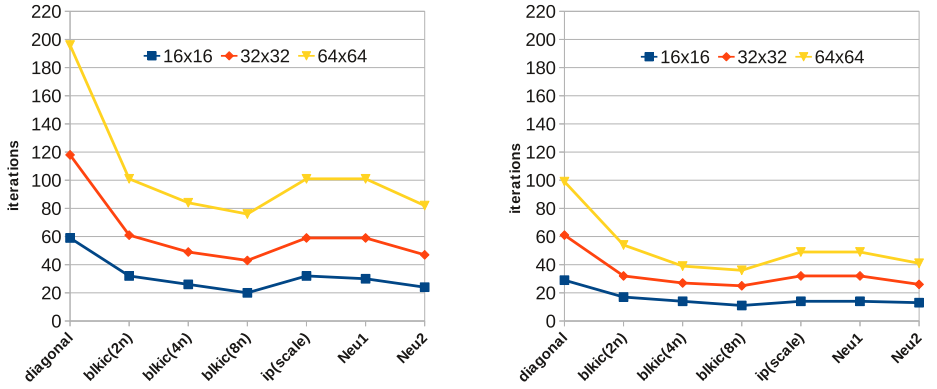
2. Using the $dpotrs$** and $dpotrf$†† functions to solve the system $Ex = b$ on the CPU using an optimized BLAS library.

In the second method we have to do a host-to-device transfer and back in *every* iteration. This adversely affects the run-time of the complete algorithm when we consider the GPU implementation.

In the first method calculation of E^{-1} (which is only done once in the setup phase) becomes prohibitively expensive (almost 100%) as the number of unknowns increases. In Figure 10 we see how the time for setup (for the triangular solve approach using *dpotrs* and *dpotrf*) stays below 10% for the CPU while for the GPU it scales with the number of deflation vectors.

** *dpotrs* solves a system of linear equations $Ax = B$ with a symmetric positive definite matrix A using the Cholesky factorization $A = U^T U$ or $A = LL^T$ computed by *dpotrf*

†† *dpotrf* computes the Cholesky factorization of a real symmetric positive definite matrix A.



(a) with preconditioning only

(b) with deflation and preconditioning

Figure 11: convergence rate variation with two levels of preconditioning.

7 Numerical Results

In this section we present the results for the preconditioning methods we propose compared with some well known preconditioners. We comment on the numerical performance of the preconditioner and further present results on its ability to exploit the parallel computation power of the GPU.

7.1 Comparing Preconditioning Schemes

We first demonstrate with MATLAB implementations of our DPCG algorithm, how the two levels of preconditioning incrementally work at reducing the number of iterations it takes for the Conjugate Gradient Method to converge. The experiments are done for three different 2 dimensional grids with sizes 16×16, 32×32 and 64×64. The ratio of densities for the two fluids modeled is 10^3. Figure 11 shows how the convergence rate (measured in terms of number of iterations) varies with different preconditioners for the CG method. We note that the Neumann type Preconditioners denoted by $Neu1$ and $Neu2$ are comparable to the Block-IC approaches with block sizes $2n$ ($blkic(2n)$) and $8n$ ($blkic(8n)$). We have not shown the results for plain Incomplete Poisson preconditioning (without scaling) here since they are at least 3 times higher than the diagonal preconditioning results. In Figure 11 we also notice how deflation effectively halves the number of iterations it requires to converge.

7.2 Experiments on the GPU

We performed our results on the hardware available with the Delft Institute of Applied Mathematics.

- For the CPU version of the code we used a single core of Q9550 @ 2.83 Ghz with 6MB L1 cache and 8 GB main memory.
- For the GPU version we used a NVIDIA Tesla(Fermi) C2070 with 6GB memory.

The time we report for our implementations is the total time it takes to complete k iterations (excluding the setup time) required for convergence. A single iteration involves steps 2 to 10 in Algorithm 1. In our results, speedup is measured as a ratio of the time taken to complete k iterations (of the DPCG method) on the two different architectures,

$$Speedup = \frac{k_{CPU}}{k_{GPU}} \tag{24}$$

The setup phase includes the following operations

1. Assigning space to variables required for temporary storage during the iterations.
2. Making matrix AZ.
3. Making matrix E.
4. Populating x, b.
5. Doing the operations as specified in the first line of Algorithm 1 in Section 5.

We kept the setup time out of our results since we had two alternate approaches for handling the operation $Ea_2 = a_1$ as mentioned in Section 6. This way it is easier to see that explicit inverse calculation could be beneficial for speedup whereas the triangular solve is not.

The problem description is as described in Section 2. The stopping criteria is defined at $\epsilon = 10^{-6}$. For deflation we have used $2n$ deflation vectors. The number of unknowns is N, where $N = n \times n$. The CPU version of the code uses optimized BLAS library ATLAS and is compiled with $-O3$ optimization flags. The GPU version uses MAGAMABLAS for some of the operations like *daxpy*, *dcopy*, *ddot* etc. For the triangular solve every step we use a MAGMABLAS *dpotrs* function every iteration after having used *dpotrf* **once** before entering the iteration loop. This function is executed in co-operation with the CPU.

In Figure 12 we see that the speedup values for all schemes stay more or less constant. This is because in these cases the largest part of the time for the iteration is spent in the highly sequential Block Incomplete Cholesky Preconditioning on the GPU. The speedup increases with increasing problem size since more and more parallelism (fine-grain) is exposed.

In Figure 13 and 14 we note that:

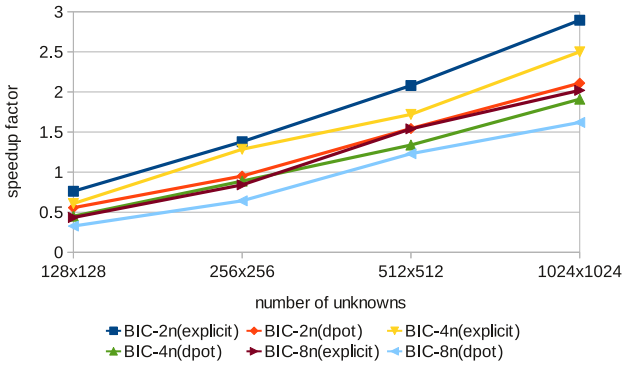


Figure 12: Comparison of Explicit versus triangular solve strategy for DPCG. Block-IC Preconditioning with $2n$, $4n$ and $8n$ block sizes.

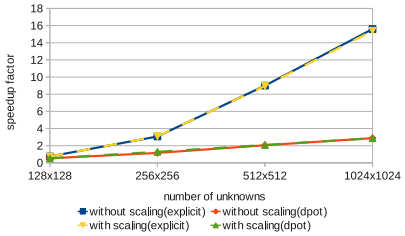


Figure 13: Comparison of Explicit versus triangular solve strategy for DPCG. Incomplete Poisson Preconditioning (with and without scaling).

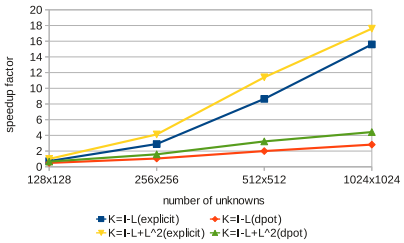


Figure 14: Comparison of Explicit versus triangular solve strategy for DPCG. Neumann Series based Preconditioners $M^{-1} = K^T K$.

1. For Incomplete Poisson Schemes with *dpotrs* and *dpotrf* based calculation of $E^{-1}a_1$ the speedup is comparable to the results for the choice where E^{-1} is explicitly calculated.

2. Using explicit E^{-1} calculation combined with Incomplete Poisson(IP), and both the variants of Truncated Neumann Series based preconditioners ($M^{-1}_{Neu1}/M^{-1}_{Neu2}$), we observe a factor 4 increase in speed with respect to the triangular solve solve approach using *dpotrs* and *dpotrf*.

A comparison of how the wall-clock times for the different preconditioning algorithms vary for the Deflated Preconditioned Conjugate Gradient Method is presented in Figure 15. Finally in Figure 16 we present the number of iterations required for convergence for the different preconditioning schemes considered in this paper. It can be noticed that the second type of Neumann Series based Preconditioner (with $K = (I - L + L^2)$ lies between

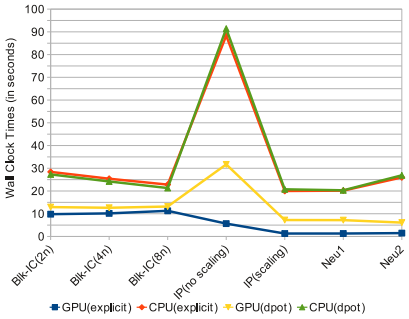


Figure 15: Deflated Preconditioned Conjugate Gradient. Wall-Clock Times for a 1024×1024 grid. $2n$ deflation vectors. Stopping Criteria $10e-06$.

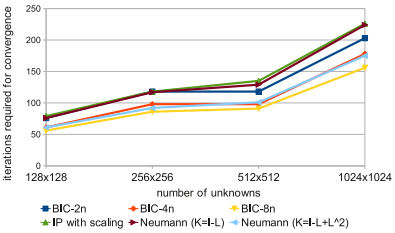


Figure 16: Deflated Preconditioned Conjugate Gradient. Iterations required for convergence.

the Block-IC scheme with block sizes $4n$ and $8n$.

8 Conclusions and Future work

We have shown how two level preconditioning can be adapted to the GPU for computational efficiency. In order to achieve this we have investigated preconditioners that are suited to the GPU. At the same time we have made new data structures in order to optimize deflation operations.

Through our results we demonstrate that the combination of Truncated Neumann based preconditioning and deflation proves to be computationally efficient on the GPU. At the same time it's numerical performance is also comparable to the established method of Block-Incomplete Cholesky Preconditioning.

We have also evaluated two different approaches of implementing the deflation step. From the model problem we have learned that the choice of implementing deflation method could be crucial in the overall run-time of the method.

Using this knowledge we are now working on model problems with bubbles instead of simple interfaces. With these geometries we can use the Level-Set Sub-domain based deflation vectors to capture and eliminate small eigenvalues with considerably less deflation vectors. This way we can use the explicit inverse calculation for E and have double-digit speedups for larger problems.

References

[1] Ament, M. and Knittel, G. and Weiskopf, D. and Strasser, W.: A Parallel Preconditioned Conjugate Gradient Solver for the Poisson Problem on a Multi-GPU Platform, pp. 583-592, 2010, 18th Euromicro Conference on Parallel, Distributed and Network-based Processing.

[2] Stanley O. and Sethian, James A.: Fronts Propagating with Curvature Dependent Speed: Algorithms Based on Hamilton-Jacobi Formulations, Journal of Computational Physics, 1988.

[3] Demmel, J. and Hoemmen, M.F. and Mohiyuddin, M. and Yelick, K. A., Avoiding Communication in Computing Krylov Subspaces, EECS Department, University of California, Berkeley, 2007, Oct, http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-123.html, UCB/EECS-2007-123.

[4] Saad, Y. :Iterative Methods for Sparse Linear Systems. Society for Industrial and Applied Mathematics; 2^{nd} edition, Philadelphia, (2003).

[5] Tang J.M.: Two-Level Preconditioned Conjugate Gradient Methods with Applications to Bubbly Flow Problems, PhD Thesis, 2008. Delft University of Technology, Delft, The Netherlands.

[6] Bolz, J. and Farmer, I. and Grinspun, E. and Schröoder, P.: Sparse matrix solvers on the GPU: conjugate gradients and multigrid, ACM Trans. Graph. 22,**3** , 2003, 917-924.

[7] Bell, N. and Garland, M.: Efficient Sparse Matrix-Vector Multiplication on CUDA,2008. NVIDIA Corporation, NVR-2008-04.

[8] Pijl, S.P. Van der and Segal, A. and Vuik, C. and Wesseling, P.: A mass conserving Level-Set Method for modeling of multi-phase flows, International Journal for Numerical Methods in Fluids,**47**, 2005, 339-361.

[9] Gupta, R.: Implementation of the Deflated Preconditioned Conjugate Gradient Method for Bubbly Flow on the Graphical Processing Unit(GPU), Master Thesis, 2010. Delft University of Technology, Delft, The Netherlands.

[10] Tang, J.M. ,Vuik, C.: Efficient Deflation Methods applied to 3-D Bubbly Flow Problems. Electronic Transactions on Numerical Analysis **26** (2007) 330-349.

[11] Helfenstein R. and Koko J., Parallel preconditioned conjugate gradient algorithm on GPU. Journal of Computational and Applied Mathematics.In Press, Corrected Proof,2011, http://www.sciencedirect.com/science/article/pii/S0377042711002196.

[12] V. Heuveline, D. Lukarski, C. Subramanian, J.-P. Weiss, Parallel Preconditioning and Modular Finite Element Solvers on Hybrid CPU-GPU Systems, in P. Ivnyi, B.H.V. Topping, (Editors), Proceedings of the Second International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering, Civil-Comp Press, Stirlingshire, UK, Paper 36, 2011.

[13] Meijerink, J. A. and Vorst, H. A. van der., An Iterative Solution Method for Linear Systems of Which the Coefficient Matrix is a Symmetric M-Matrix. Mathematics of Computation, 31, 137, Jan., 1977, pp. 148-162, American Mathematical Society.

[14] Asgasri, A. and Tate J. E., Implementing the Chebyshev Polynomial Preconditioner for the Iterative Solution of Linear Systems on Massively Parallel Graphics Processors, CIGRE Canada Conference on Power Systems, 2009, Toronto, Canada.

[15] Griebel M. and Zaspel P., A multi-GPU accelerated solver for the three-dimensional two-phase incompressible Navier-Stokes equations, Computer Science - Research and Development, 2010, 25, Springer Berlin / Heidelberg.

[16] Baskaran, M. and Bordawekar, R., Opimizing Sparse Matrix-Vector Multiplication on GPUs, IBM Research Division, 2008, NY, USA.

[17] M. Baboulin and Buttari, A. and Dongarra, J. and Kurzak, J. and Langou, J. and Langou, J. and Luszczek, P. and Tomov, S., Accelerating Scientific Computations with Mixed Precision Algorithms, CoRR, abs/0808.2794, 2008.

[18] Buatois, L. and Caumon, G. and Levy, B., Concurrent number cruncher: a GPU implementation of a general sparse linear solver, Int. J. Parallel Emerg. Distrib. Syst., 24, 3, 2009, Taylor & Francis, Inc., Bristol, PA, USA.

[19] Monakov, A. and Avetisyan, A., Implementing Blocked Sparse Matrix-Vector Multiplication on NVIDIA GPUs, SAMOS '09: Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation, 2009, 289–297, Samos, Greece, Springer-Verlag.

[20] Wang, M. and Klie, H. and Parashar, M. and Sudan, H., Solving Sparse Linear Systems on NVIDIA Tesla GPUs, ICCS '09: Proceedings of the 9th International Conference on Computational Science, 2009, 864–873, Baton Rouge, LA, Springer-Verlag.

[21] NVIDIA CUDA Programming Guide v4.0, NVIDIA Corporattion,Santa Clara, 2011.

[22] NVIDIA CUDA C Programming Best Practices Guide CUDA Toolkit v4.0, NVIDIA Corporattion, Santa Clara, 2011.

[23] C. Vuik and A. Segal and J.A. Meijerink, An efficient preconditioned CG method for the solution of a class of layered problems with extreme contrasts in the coefficients, J. Comp. Phys., 152, 1999,385–403.