Efficient Two-Level Preconditioned Conjugate Gradient Method on the GPU

Rohit Gupta, Martin B. van Gijzen, and Cornelis Kees Vuik

Delft University of Technology, Mekelweg 4, 2628CD, Delft, The Netherlands
{rohit.gupta,m.b.vangijzen,c.vuik}@tudelft.nl

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

1 Introduction

Our work is motivated by the Mass-Conserving Level Set approach [6] 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. The discretized 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). Due to the large contrast in the densities of the fluids involved, the matrix A has a large condition number κ, which results in slow convergence when the system (1) is solved using the iterative Conjugate Gradients (CG) method.

1.1 Focus of This Research

To overcome the slow convergence it is imperative to use preconditioning. The resulting system then looks like, $M^{-1}Ax = M^{-1}b$, where the matrix M is symmetric and positive definite. 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. This research aims to find preconditioning schemes that can exploit the computing power of the GPU. To this end the preconditioning schemes should offer fine-grain parallelism. At the same time they should prove effective in bringing down the condition number of $M^{-1}A$. We use a two-level preconditioner. The

M. Daydé, O. Marques, and K. Nakajima (Eds.): VECPAR 2012, LNCS 7851, pp. 36–49, 2013.

first level preconditioner is based on the Truncated Neumann series of the triangular factors of the coefficient matrix A. After this, we apply Deflation to treat the remaining small eigenvalues in the spectrum of the preconditioned matrix.We compare our schemes with Block-Incomplete Cholesky (Block-IC) Preconditioners, as a benchmark to check their quality. The numerical performance of the preconditioners we introduce in this paper comes close to their Block-IC counterparts for our model problem and they also offer fine-grain parallelism making them very suitable for the GPU.

1.2 Related Work

Preconditioning has been studied previously for GPU implementations of the Conjugate Gradient method. The preconditioner in [2] offers as much parallelism as the number of degrees of freedom, N (or the number of unknowns). However, our experiments [1] show its use is limited for two-phase (high condition number, (κ)) flow problems. An extension to [2] is provided in [8] wherein a relaxation factor is utilized. In [9] an incomplete LU decomposition based preconditioner with fill-in is used combined with reordering using multi-coloring.

One of the first works [4] using GPU computing used Multigrid with CG. More recently in [3] also multigrid has been investigated for solving Poisson type problems. In [11] a comparative study is presented between deflation and multigrid. It shows that the former is a competitive technique in comparison with the latter.

This paper is organized as follows: in the next section we present the test problem. A brief overview of the preconditioning schemes and their features can be found in Section 3. We discuss the approach of two level preconditioning in Section 4. In Section 5 we introduce the Conjugate Gradient Algorithm with Preconditioning and Deflation. Furthermore we comment on two different implementations for this method in Section 5.1. In Section 6 we present our results and we end with a discussion in Section 7.

2 Problem Definition

We define a test problem in order to test our preconditioning schemes. We define a unit square as our computational domain in 2D (Figure 2). It has two fluids with a density contrast ($\rho_1 = 1000$, $\rho_2 = 1$). It has an interface layer (at $y = 0.5$), where there is a jump in coefficient values due to contrast in densities of the two fluids. This jump is also visible in the eigenvalue spectrum as shown in Figure 1. Boundary conditions are applied to this domain as indicated in Figure 2. The resulting discretization matrix A is sparse and SPD. It has a pentadiagonal structure due to the 5-point stencil discretization. For a grid of dimensions $(n + 1) \times n$ the matrix A is of size $N = n \times n$. Stopping criteria are defined for convergence as $\| r_i \|_2 \leq \| b \|_2 \epsilon$, where r_i is the residual at the i-th step, b is the right-hand side and ϵ is the tolerance. For our experiments we have kept ϵ at 10^{-6}. The initial guess (x_0) is a random vector to avoid artificially fast convergence due to a smooth initial error.

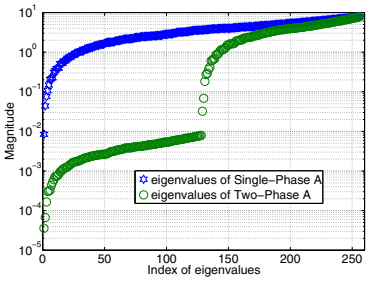


Fig. 1. 2D grid (16×16) with 256 unknowns. Jump at the interface due to density contrast.

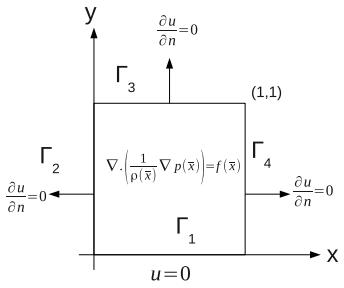


Fig. 2. unit square with boundary conditions

Through this test case we can ascertain the effectiveness of deflation for such problems on the GPU. The final goal however remains to be able to make a solver capable of handling the linear systems arising in bubbly flow problems.

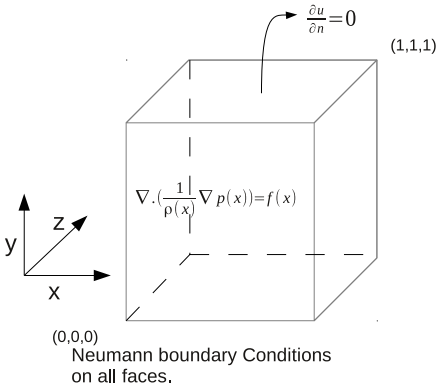
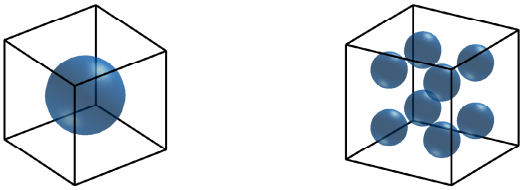


Fig. 3. Problem Definition. Unit cube in 3-D.

To this end we also define a test case with a unit cube with bubbles. This 3D formulation poses additional challenges and is a harder problem to solve due to many more small eigenvalues corresponding to the number of bubbles in the system. In Figure 4 we present two cases where there is a single bubble and 8 bubbles in the domain presented in Figure 3. The contrast between the densities of the bubble and the surrounding medium is of the same order as in the 2D problem.

In the 3D case we apply Neumann boundary conditions on all faces. The matrix is SPD and has a septadiagonal structure. The problem size is $N = n \times n \times n$. We maintain the same stopping criteria, tolerance and initial conditions as the 2D problem. The bubbles are placed symmetrically in the test cases (depicted in Figure 4) whose results we present in Section 6.2.



(a) 1 bubble (b) 8 bubble

Fig. 4. 3 Geometries for Realistic Problems

3 Preconditioning Schemes

Preconditioning operation $y_i = M^{-1}r_i$ involves the preconditioning matrix M and the residual vector, r_i at the i-th iteration. The preconditioner matrix M, for our problem, is sparse.

We compare our results to the standard Block Incomplete Cholesky preconditioner (for which $M = LL^T$). We apply the block structure to A and generate L as suggested in [10]. The Block Incomplete Cholesky preconditioners in our results are suffixed with a number like $2n$, $4n$ etc. which denotes the block-size. So for example in a Block-IC preconditioner with blocksize $8n$ where the matrix A has $N = n \times n$ unknowns the preconditioner will be named like $M^{-1}_{Blk-IC(8n)}$. Since the data parallelism in Block Incomplete schemes is limited by the block-size (refer [1] for details) we turn our attention to preconditioners that have more inherent parallelism.

3.1 Neumann Series Based Preconditioning

We define the preconditioning matrix, $M = (I + LD^{-1})D(I + (LD^{-1})^T)$, where L is the strictly lower triangular part and D is the diagonal of A, the coefficient matrix. We apply the truncated Neumann Series for approximation of M^{-1}. Specifically for $(I + LD^{-1})$ (and similarly for $(I + (LD^{-1})^T)$) the series can be defined as

$$(I + LD^{-1})^{-1} = I - LD^{-1} + (LD^{-1})^2 - (LD^{-1})^3 + \cdots \text{ if } \| LD^{-1} \|_\infty < 1. \tag{2}$$

In our problem $\| LD^{-1} \|_\infty < 1$, hence the Neumann Series is a valid choice for approximating the inverse of $(I + LD^{-1})$. So we can redefine M^{-1} as

$$M^{-1} = (I - D^{-1}L^T + \cdots)D^{-1}(I - LD^{-1} + \cdots). \tag{3}$$

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

$$M^{-1}{}_{Neu1} = (I - D^{-1}L^T)D^{-1}(I - LD^{-1}) \tag{4}$$

$$M^{-1}{}_{Neu2} = (I - D^{-1}L^T + (D^{-1}L^T)^2)D^{-1}(I - LD^{-1} + (LD^{-1})^2). \quad (5)$$

We define $K = (I - LD^{-1})$ for M^{-1}_{Neu1} and $K = (I - LD^{-1} + (LD^{-1})^2)$ for $M^{-1}{}_{Neu2}$. For the preconditioners as given by (4) and (5) we only store LD^{-1} and calculate $K^T D^{-1} K x$ term-by-term every time required. Every term in the expansion of $M^{-1}x = K^T D^{-1} K x$ can be (roughly) computed at the cost of one $LD^{-1}x$ operation. This is around $2N$ multiplications and N additions. This is only true for the stencil we discuss in this paper.

4 Deflation

To improve the convergence of our method further we also use a second level of preconditioning. Deflation aims to remove the remaining bad eigenvalues from the preconditioned matrix, $M^{-1}A$. This operation increases the convergence rate of the Preconditioned Conjugate Gradient (PCG) method. We define the matrices $P = I - AQ, Q = ZE^{-1}Z^T, E = Z^T AZ$, where $E \in \mathbb{R}^{d \times d}$ 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. $Z \in \mathbb{R}^{N \times d}$ is the so-called 'deflation-subspace matrix' whose d columns are called 'deflation' or 'projection' vectors. The deflated system is now

$$PA\hat{x} = Pb. \quad (6)$$

The vector $\hat{x}$ is not necessarily a solution of the original linear system, since x might contain components in the null space of PA, $\mathcal{N}(PA)$. Therefore this 'deflated' solution is denoted as $\hat{x}$ rather than x. The final solution has to be calculated using the expression $x = Qb + P^T\hat{x}$. The deflated system (6) can be solved using a symmetric positive definite (SPD) preconditioner, M^{-1}. We therefore seek a solution of $M^{-1}PA\hat{x} = M^{-1}Pb$. The resulting method is called the Deflated Preconditioned Conjugate Gradient (DPCG) method as listed in

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** i:=0,..., until convergence **do**
3: $\hat{w}_i := PAp_i$
4: $\alpha_i := \frac{(\hat{r}_i, y_i)}{(p_i, \hat{w}_i)}$
5: $\hat{x}_{i+1} := \hat{x}_i + \alpha_i p_i$
6: $\hat{r}_{i+1} := \hat{r}_i - \alpha_i \hat{w}_i$
7: Solve $My_{i+1} = \hat{r}_{i+1}$
8: $\beta_i := \frac{(\hat{r}_{i+1}, y_{i+1})}{(\hat{r}_i, y_i)}$
9: $p_{i+1} := y_{i+1} + \beta_i p_i$
10: **end for**
11: $x_{it} := Qb + P^T x_{i+1}$

Algorithm 1. We choose Sub-domain Deflation and use piecewise constant deflation vectors.We make stripe-wise deflation vectors (see Figure 7) unlike the block deflation vectors suggested in [7]. These deflation vectors lead to a regular structure for AZ and, therefore, an efficient storage of AZ.

In order to implement deflation on the GPU we have to break it down into a series of operations,

$$a_1 = Z^T r, \tag{7a}$$
$$a_2 = E^{-1} a_1, \tag{7b}$$
$$a_3 = AZ a_2, \tag{7c}$$
$$s = r - a_3. \tag{7d}$$

(7b) shows the solution of the inner system that results during the implementation of deflation.

5 Two Level Preconditioned Conjugate Gradient Implementation

The implementation of the Deflated Preconditioned Conjugate Gradient(DPCG) method follows Algorithm 1. The deflation operation requires solving the system $Ea_2 = a_1$ in every iteration. Also a matrix-vector product, AZa_2 is required in every iteration. The first operation can be performed in two different ways as we will see in Section 5.1. To optimize the second operation we store AZ in such a format such that we get the same number of operations, memory access pattern and (approximately) performance as the sparse matrix vector product Ax.

5.1 GPU Implementation of Deflation

We store the matrix A in the Diagonal (DIA) format and follow the implementation as detailed in [5]. For deflation, every iteration we have to solve the system $Ea_2 = a_1$. This can be done in two ways.

1. Calculating E^{-1} explicitly so that the $E^{-1}a_1$ becomes a dense matrix-vector product which can be calculated using the *gemv* routine from MAGMA BLAS library for the GPU.
2. Using triangular solve routines from the MAGMA BLAS library. Specifically we use the *dpotrs* and *dpotrf* functions ([12]).

The parallelism available in the second method drops for larger systems compared to the first method which is embarrassingly parallel on the GPU. On the other hand, in the first method calculation of E^{-1} (which is only done once in the setup phase) becomes expensive as the number of deflation vectors increases. In case of our test problem the setup times for the second method are one-third when compared to the first method (details in [1]). However, this one

time calculation can make the operation $a_2 = E^{-1}a_1$ very quick on the GPU. So a selection of high-quality deflation vectors (such that $d << N$), which lead to a smaller E matrix and hence computationally cheaper inversion provides an advantage for a GPU implementation.

5.2 Storage of the Matrix AZ

The structure of the matrix AZ stored as an $N \times d$ matrix, where d is the number of domains/deflation vectors, can be seen in Figure 5. In Figures 5 to 7 it must be noted that $d = 2n$ 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 5) 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 7. 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 coalesced. 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 times x we have 5 elements of A in a single row multiplied with 5 elements of x as detailed in [5]. 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-Matrix

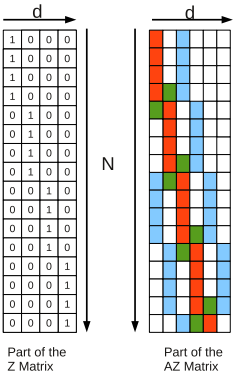


Fig. 5. Parts of Z and AZ matrix. number of deflation vectors $=2n$.

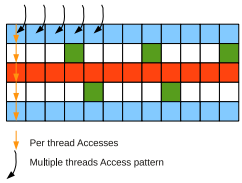


Fig. 6. AZ matrix after compression

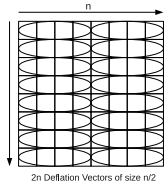


Fig. 7. Deflation vectors for the 8×8 grid

Vector Product operation. Wherever there is no element in AZ we can store a zero. Thus in the compacted form the $N \times d$ matrix AZ can be stored in $5N$ elements as illustrated in Figure 6. The golden arrows in Figure 6 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 [5].

5.3 Extension to Real (Bubble) Problems and 3D

This storage format can be extended to include bubbles in the domain. In this case, only the values of coefficients change but the structure of the matrix remains the same. For a 3D problem, deflation vectors that correspond to planes or stripes can lead to an AZ matrix that is similar in structure compared to the matrix A and hence can be stored using the ideas presented in the previous section.

In Figure 8 we provide an example for a 3D scenario in order to explain what planar and stripe-wise vectors look like. One can notice that stripe wise vectors are piecewise constant vectors. We briefly talk about stripe-wise vectors. Every vector has length N. Each vector has ones for the row on which it is defined and zeroes for the rest of the column. Planar vectors are an extension of stripe-wise vectors and are defined on n^2 cells (have n^2 ones and rest of the column has zeroes). It must be noted that for a 3D problem the number of unknowns or problem size is $N = n^3$ where n is the size of the grid in any one dimension.

For our experiments in Section 6.2 we use n^2 stripe-wise and n planar vectors.

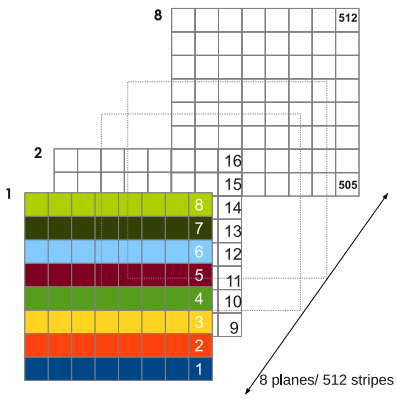


Fig. 8. Planes and stripes for a 8^3 uniform cubic mesh

6 Numerical Results

We performed our experiments 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 12MB L2 cache and 8 GB main memory.

– For the GPU version we used a NVIDIA Tesla(Fermi) C2070 with 6GB memory.

We use optimized BLAS libraries (MAGMA and ATLAS) on both GPU and CPU for daxpys, dot products and calculation of norms.

All times reported in this section are measured in seconds. The time we report for our implementations is the time taken (this excludes the setup time, specifically the steps 2 to 10 in Algorithm 1) for iterations required for convergence. In our results, speedup is measured as a ratio of this iteration time on the CPU versus the GPU. The effect of setup time vis-a-vis the iteration time is reported in detail in [1] (Figure 11 and 12 in Appendix A for quick reference). The setup phase includes the assigning and initializing the memory and the operations required to be done before entering the iteration loop, namely,

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 4.

It also involves the setup for the operation $Ea_2 = a_1$ using either of the two approaches mentioned in Section 5.1.

6.1 Stripe-Wise Deflation Vectors - Experiments with 2D Test Problem

For the 2D problem we have used $2n$ deflation vectors unless otherwise mentioned. For the DPCG implementation which uses Block Incomplete Cholesky as the first-level preconditioner, the difference in speedup between the two different implementations to compute coarse grid solution ($Ea_1 = a_2$) as mentioned in the previous section is negligible (Figure 9 in Appendix A). This is due to the fact that in this case the majority of the time is spent in the preconditioning step and it dominates the iteration time, so the effect of the deflation operation is overshadowed. However, for the Truncated Neumann Series based preconditioners the difference between GPU and CPU execution times is significant (Figure 10 in Appendix A) since preconditioner is highly parallelizable. Consequently the choice of inner solve in the deflation step becomes decisive in the length of execution time. The speedup attainable for the complete solver with explicit inverse (E^{-1}) based calculation of a_2 is four times that of the triangular solve strategy (Figure 10 in Appendix A). A comparison of how the wall-clock times for the different preconditioning algorithms vary for the DPCG method is presented in Table 1. Grid Size is $N = 1024 \times 1024$, $n = 1024$ and $2n$ deflation vectors have been used. These times and number of iterations shown in Table 2 are presented for the deflation implementation with explicit E^{-1} calculation.

In Table 2 we present the number of iterations required for convergence of different preconditioning schemes. The number of iterations is not affected by

Table 1. Wall Clock Times for DPCG on a 2D problem with $N = 1024 \times 1024$

Preconditioning Variant	CPU	GPU
${M^{-1}}_{Blk-IC(2n)}$	28.4	9.8
${M^{-1}}_{Blk-IC(4n)}$	25.48	10.15
${M^{-1}}_{Blk-IC(8n)}$	22.8	11.28
${M^{-1}}_{Neu1}$	20.15	1.29
${M^{-1}}_{Neu2}$	25.99	1.47

Table 2. Iterations required for Convergence of 2D problem using DPCG with $2n$ deflation vectors

	Grid Sizes			
Preconditioning Variant	128^2	256^2	512^2	1024^2
${M^{-1}}_{Blk-IC(2n)}$	76	118	118	203
${M^{-1}}_{Blk-IC(4n)}$	61	98	98	178
${M^{-1}}_{Blk-IC(8n)}$	56	86	91	156
${M^{-1}}_{Neu1}$	76	117	129	224
${M^{-1}}_{Neu2}$	61	92	101	175

the choice of implementations for the Deflation Method discussed in Section 5.1. It can be noticed that the results for the second type (Neu2) of Neumann Series based Preconditioner (with $K = (I - LD^{-1} + (LD^{-1})^2)$) lie between the Block-IC scheme with block sizes $4n$ and $8n$.

6.2 Stripe and Plane-Wise Deflation Vectors - Experiments with 3D Problems

It is possible to use stripes for 3D problems and problems involving bubbles as well. However, stripe-wise deflation vectors are not the best choice one can make for the deflation subspace. For 3D experiments we measure our results against an optimized CPU implementation that utilizes Sub-domain deflation vectors (block-shaped vectors). Block vectors do not suit the storage pattern that we have utilised for this study but they can also give good results. In Table 3 and 4 we see the results for a case when we have 3D geometries. For the first set of results presented in Table 3 the geometry is that of slabs of different material. It must be noted now that $N = n^3$ and not n^2. The computational domain is now a unit cube. We present the results with n plane and n^2 stripe-wise deflation vectors. There are three slabs in the unit cube. The middle slab is 0.5 units thick. Its density is 10^{-3} times the density of the surrounding slabs.

As we can see in the results of Table 3 the speedup drops. This is a consequence of the fact that the inner system takes a lot of time to solve now and the data structure and the associated kernels for the operation AZa_2 do not perform very well for very large number of deflation vectors. Moreover, if more (n^2) vectors are used the setup times become prohibitive and there is no speedup

Table 3. $3D$ Problem (128^3 points in the grid) with 3 layers. Middle layer 0.5 units thick. Tolerance set at 10^{-6}. Density contrast 10^{-3}. Comparison of CPU and GPU implementations.

	CPU[1]	GPU[2]	
	8 block vectors	128 plane vectors	16384 stripe vectors
	DICCG(0)	DPCG(neu2)	
Number of Iterations	206	324	259
Setup Time	0.3	0.36	148.5
Iteration Time	35.18	7.66	112
Speedup	-	4.59	–

at all. The iteration times are high since we use the triangular solve method for inner system. In Table 4 we continue to have a unit cube but instead of slabs of different material we now consider bubbles in the system. In particular, we have a single bubble with its center coinciding with the center of the cube and another case when we have eight bubbles, 2 in each dimension and equally spaced (refer Figure 4). It can be noticed from the results that the speedup becomes worse for the problem with more bubbles and that can be explained by the fact that stripe-wise vectors cut the bubbles and are poor approximations of the eigenvectors of the preconditioned matrix.

Table 4. $3D$ Problem (128^3 points in the grid) with 1 and 8 bubbles. Tolerance set at 10^{-6}. Density contrast 10^{-3}. Comparison of CPU and GPU implementations.

1 bubble		
	CPU[1]	GPU[2]
	8 block vectors	128 plane vectors
	DICCG(0)	DPCG(neu2)
Number of Iterations	237	287
Setup Time	0.31	0.64
Iteration Time	40.44	6.79
Speedup	-	5.95
8 bubble		
Number of Iterations	142	402
Setup Time	0.3	0.36
Iteration Time	24.4	9.51
Speedup	-	2.56

In Tables 3 and 4 the GPU version uses triangular solves for the inner system since with explicit solve and stripe-wise vectors the round-off errors in the solution of the inner system (due to explicit inverse calculation) grow very quickly

[1] CPU version uses CG for inner system solve.
[2] GPU version uses triangular factorization based inner solve.

and convergence is never achieved. We only show the results with n vectors in Table 4 since with n^2 vectors there is no speedup.

7 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 its numerical performance is also comparable to the established method of Block-Incomplete Cholesky Preconditioning.

The approach of using stripe-wise vectors is applicable to 3D problems and problems with bubbles in the domain. However, these deflation vectors, though simple to implement are not the most effective choice for the deflation of more ill-conditioned problems.

Through this study we have learnt that the choice made in the implementation of deflation method is crucial for the overall run-time of the method. We are now continuing to extend our work on 3D problems with bubbles. We believe that the approach of calculating inverse of the matrix E explicitly can be very effective for the GPU. In order to overcome the possibly large setup time of this scheme and to avoid delayed convergence we are now working on better deflation vectors based on Level-Set Sub-domain deflation. A small number of these vectors can capture the small eigenvalues and result in an effective deflation step (this is discussed in [7]). This directly translates to a low setup time and overall gain in this approach of implementing deflation.

A Detailed Results

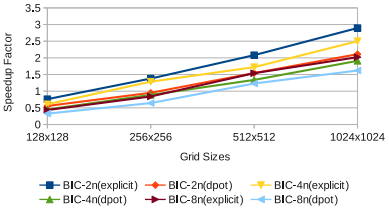


Fig. 9. Comparison of Explicit versus triangular solve strategy for DPCG. Block-IC Preconditioning with $2n$, $4n$ and $8n$ block sizes.

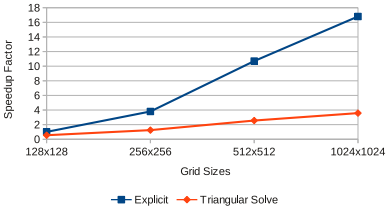


Fig. 10. Comparison of Explicit versus triangular solve strategy for DPCG. Neumann Series based Preconditioners $M^{-1} = K^T D^{-1} K$, where $K = (I - LD^{-1} + (LD^{-1})^2)$

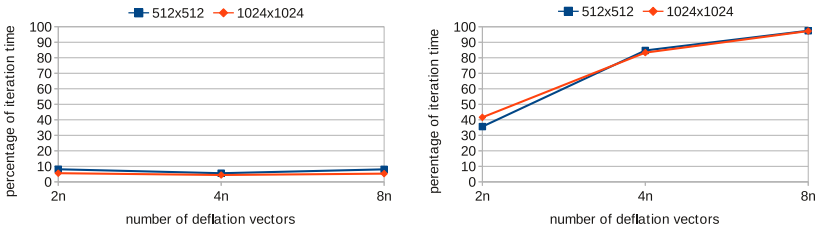


Fig. 11. Setup Time as percentage of the total (iteration+setup) time for triangular solve approach across different sizes of deflation vectors for DPCG

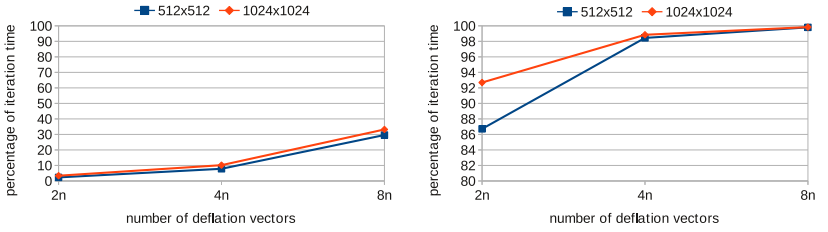


Fig. 12. Setup Time as percentage of the total (iteration+setup) time for explicit E^{-1} approach across different sizes of deflation vectors for DPCG

References

1. Gupta, R., van Gijzen, M.B., Vuik, C.: Efficient Two-Level Preconditioned Conjugate Gradient Method on the GPU, Reports of the Department of Applied Mathematical Analysis, Delft University of Technology, Delft, The Netherlands. Report 11–15 (2011)
2. Ament, M., Knittel, G., Weiskopf, D., Strasser, W.: A Parallel Preconditioned Conjugate Gradient Solver for the Poisson Problem on a Multi-GPU Platform. In: Proceedings of the 18th Euromicro Conference on Parallel, Distributed, and Network-based Processing, pp. 583–592 (2010)
3. Jacobsen, D.A., Senocak, I.: A Full-Depth Amalgamated Parallel 3D Geometric Multigrid Solver for GPU Clusters. In: Proceedings of 49th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition, AIAA 2011-946, pp. 1–17 (January 2011)
4. Bolz, J., Farmer, I., Grinspun, E., Schröder, P.: Sparse matrix solvers on the GPU: conjugate gradients and multigrid. ACM Trans.Graph. 22, 917–924 (2003)
5. Bell, N., Garland, M.: Efficient Sparse Matrix-Vector Multiplication on CUDA, NVR-2008-04, NVIDIA Corporation (2008)

6. Van der Pijl, S.P., Segal, A., Vuik, C., Wesseling, P.: A mass conserving Level-Set Method for modeling of multi-phase flows. International Journal for Numerical Methods in Fluids 47, 339–361 (2005)
7. Tang, J.M., Vuik, C.: New Variants of Deflation Techniques for Pressure Correction in Bubbly Flow Problems. Journal of Numerical Analysis, Industrial and Applied Mathematics 2, 227–249 (2007)
8. Helfenstein, R., Koko, J.: Parallel preconditioned conjugate gradient algorithm on GPU. Journal of Computational and Applied Mathematics 236, 3584–3590 (2011)
9. Heuveline, V., Lukarski, D., Subramanian, C., Weiss, J.P.: Parallel Preconditioning and Modular Finite Element Solvers on Hybrid CPU-GPU Systems. In: Proceedings of the Second International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering, paper 36 (2011)
10. Meijerink, J.A., van der Vorst, H.A.: An Iterative Solution Method for Linear Systems of Which the Coefficient Matrix is a Symmetric M-Matrix. Mathematics of Computation 31, 148–162 (1977)
11. Jönsthövel, T.B., van Gijzen, M., MacLachlan, S., Vuik, C., Scarpas, A.: Comparison of the deflated preconditioned conjugate gradient method and algebraic multigrid for composite materials. Computational Mechanics 50, 321–333 (2012)
12. MAGMABLAS documentation, `http://icl.cs.utk.edu/magma/docs/`