2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

3D bubbly flow simulation on the GPU - Iterative Solution of a linear system using sub-domain and level-set deflation

Rohit Gupta
DIAM, Faculty of EEMCS
Delft University of Technology
Delft, The Netherlands
rohit.gupta@tudelft.nl

Martin B. van Gijzen
DIAM, Faculty of EEMCS
Delft University of Technology
Delft, The Netherlands
m.b.vangijzen@tudelft.nl

Kees Vuik
DIAM, Faculty of EEMCS
Delft University of Technology
Delft, The Netherlands
c.vuik@tudelft.nl

***Abstract*—Solving an ill-conditioned linear system with a two level preconditioned Conjugate Gradient method on the GPU presents many options. The viability of these options is studied for different bubbly flow problems. On the basis of experiments conducted, we propose strategies that make our approach computationally suitable. We use the Truncated Neumann series based preconditioning scheme in combination with Deflation for implementing the two-level preconditioned Conjugate Gradient method and test different configurations on a unit cube with varying number of bubbles. Our results exhibit up to an order of magnitude speedup on the GPU. Our preconditioning scheme combined with deflation proves competitive (in terms of computation time and convergence) when compared to deflation with Incomplete Cholesky preconditioning.**

I. Introduction

In our research we model two-phase flow using the Navier Stokes equation which we solve using the Mass conserving Level-Set method. At every time-step of the solution of these equations we are confronted with the pressure-correction equation which is discretized and the resulting linear system (1) is solved. It is given by,

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

where $N = n^3$ is the number of degrees of freedom and n is the number of grid points in every coordinate direction. This linear system can be very large, sparse and symmetric positive semi-definite (SPSD). Therefore we use the method of conjugate gradients (CG) to solve it. System (1) is highly ill-conditioned due to jumps in the density of the different mediums. Consequently CG can take the bulk of computing time. Hence, preconditioning is required to accelerate convergence. We precondition CG at two levels.

In our research we focus on computationally 'speeding-up' the solution of this system using GPUs.

A. Focus of this work

We have been investigating, [1], the idea of Two Level Preconditioned Conjugate Gradient method on GPUs starting with a simple problem and testing our preconditioning schemes and deflation on it. We have found that it is possible to efficiently map the deflation operation onto the GPU so that most of the fine-grain parallelism can be exploited on it. In addition we propose a preconditioning scheme (Section III-A) which when coupled with deflation gives good results and speedups for large problems.

In continuation of our research, [2], we worked with simple deflation vectors called stripes on a 2D and a 3D problem and found out that there is a limitation to their effectiveness.

In the present paper we solve problems with bubbles in the domain. In particular we use sub-domain based, level-set and a combination of these vectors for deflation. Through our experimental results we confirm that the choices of various deflation vectors can be effective in accelerating convergence for different problems (with varying number/position of the bubbles).

We provide comparisons to established preconditioning schemes and existing software for these problems.

B. Related Work

Preconditioning of sparse linear systems on the GPU has now entered the mainstream GPU computing with the inclusion of preconditioners in libraries like CUSP[1] and CUSPARSE[2].

There is an entire class of preconditioners based on Multigrid [3], [4], [5] that are also viable options to solve the kind of problems we are dealing with. Deflation, however, can be advantageous, [6], for medium-sized problems that fit on a single GPU. The author in [6] compares Multigrid methods for solving similar problems and concludes that results from two-level preconditioning with deflation can be approximately similar to multigrid methods. In [7] a new preconditioning is presented based on ILU with multi-coloring but it requires re-organization of the input matrix. In our method we use the input matrix directly and derive our preconditioners (both levels) on the fly. A variation of the SSOR based technique of making a preconditioner (that

[1]http://code.google.com/p/cusp-library/
[2]http://developer.nvidia.com/cusparse

1066-6192/12 $26.00 © 2012 IEEE
DOI 10.1109/PDP.2013.58

also uses a Neumann Series based approach like we use for our preconditioner) is discussed in [8]. In our work we present the two-level approach that involves preconditioning first with a Truncated Neumann Series based preconditioner followed by deflation.

This article is organized as follows: in the next section we present the setup of our problem followed by the overview of our first-level preconditioning schemes in Section III. We also comment on the eigenvalue spectrum of the matrix after preconditioning and how its properties must be kept in mind while implementing the second level preconditioning. In Section IV we present the second level preconditioning (deflation) and the choices of implementation. We present our implementation approach and results for some test cases on the GPU in Section V. In conclusion (Section VI) we present a summary of our findings and an outlook for future research.

II. Problem Definition

We continue to use the problem (Figure 1(a)) defined in our previous work [2]. There can be two different configurations on the basis of the density of the medium and bubbles e.g. air bubbles rising in water or water droplets falling through air. For these two cases it is only necessary to consider the ratio of the density contrast which influences the conditioning of the problem and the number of small eigenvalues in the spectrum of the coefficient matrix. We change the position and number of these bubbles in our experiments, to cover different scenarios which can arise in physical problems (e.g. bubbles cutting sub-domains). The problem is solved on a single core of a dual core CPU (E8500 a @3.16GHz) and the solution (generated on the CPU) of the linear system is compared with those generated on NVIDIA C2070 GPU. As an extension to this experiment we also provide results for an OpenMP accelerated CPU implementation that is executed on a dual-quad core CPU from Intel. The CPU version of the code is highly optimized to reduce operations and computationally speedup the DPCG algorithm. All calculations are done in double precision. For CUSP implementations the GPU code runs and stays entirely on the GPU except for the synchronization points in CG. These occur for two reasons in the CG iteration.

1) For the calculation of the ratio of dot products; and
2) Comparing the norm of the current residual with the stopping criteria to decide whether to continue to the next iteration or not.

For CUSPARSE (in CUDA $5.0.7$) implementation this happens only once for the second point mentioned above. In this version of CUDA it is possible to use the scalar product/norm functions and store the result on the GPU instead of returning them to the CPU. However, the version of CUSP we use, 0.3, supports CUDA 4.1 only, so we have not used CUDA $5.0.7$ with CUSP 0.3.

A. Discretization

The pressure correction equation (in Figure 1(a) and mentioned in previous sections) is approximated using finite differences (with central discretization) and a uniform cubic mesh in three dimensions. The result is a 7-diagonal matrix, A, albeit with a jump in the coefficients at bubble/droplet interfaces. The number of unknowns is given by $N = n \times n \times n$, where n is the grid size in each dimension. The matrix is SPSD (symmetric positive semi-definite). This matrix is stored in diagonal (DIA) format where the 7 diagonals are stored in arrays of size N each. This is a very useful storage pattern, especially on the GPU [9].

Our solver is called by a CPU code written in Fortran. It is a Navier Stokes solver with additional routines for solving other variables (e.g. velocity) every time-step. It passes the coefficient matrix A, x and b and termination criteria as parameters, to the GPU based solver whose results are discussed in this work.

We define the termination criteria for our solvers (CPU and GPU) as

$$\| r_i \|_2 \leq \| b \|_2 \, \epsilon, \tag{2}$$

where r_i is the residual at the i^{th} step, b is the right-hand side and ϵ is the tolerance. The initial guess chosen (on CPU and GPU) is a random vector since we want to avoid the situation where (unrealistic) convergence could be aided by the choice of a zero or constant vector.

In our results we use the bubble geometry presented in Figure 1(b) and 1(c).

III. Preconditioning

In this section we give a brief overview of the preconditioning schemes that we consider. The preconditioning operation changes the linear system (1) into (3), which is given by

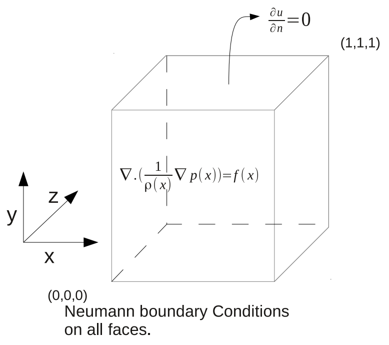
$$M^{-1}Ax = M^{-1}b. \tag{3}$$

The primary aim of the preconditioning operation is to accelerate convergence. The preconditioner must also be light on storage requirements and cheap to compute. The CPU code uses the Incomplete Cholesky (IC) preconditioner [10]. The combination of IC and CG with zero diagonals for fill in is called ICCG(0) method. However, this preconditioning technique has its limitations for GPU implementation since it is inherently sequential. On the CPU we use a sequential version of the code with IC preconditioner for the first level.

A. Truncated Neumann Series based Preconditioners

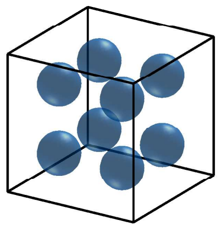
Factorization based preconditioners can be approximated by

$$M = KDK^T, \tag{4}$$

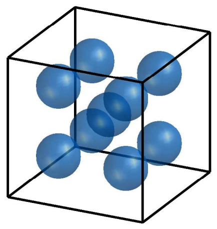


$\frac{\partial u}{\partial n}=0$
(1,1,1)
$\nabla.(\frac{1}{\rho(x)}\nabla p(x))=f(x)$
y
z
x
(0,0,0)
Neumann boundary Conditions on all faces.

(a) Problem Definition. Unit cube in 3-D.



(b) 8 bubble



(c) 9 bubble

Figure 1: $3D$ Test Problem.

where $K = (I + LD^{-1})$, L is strictly lower triangular, D is the diagonal of A and I is the identity matrix. Such approximations are valid for ICCG and successive over-relaxation (SSOR) based preconditioners. In order to calculate M^{-1} we have to approximate $(I + LD^{-1})^{-1}$. This can be achieved by using the Neumann series for $(I + LD^{-1})$:

$$\begin{aligned}(I+LD^{-1})^{-1} =& I - LD^{-1} + (LD^{-1})^2 - \\ &(LD^{-1})^3 + \cdots \text{ if } \parallel LD \parallel_\infty < 1.\end{aligned} \quad (5)$$

In order to limit the costs for this preconditioner we truncate the number of terms in the series (5) to 3 (up to $(LD^{-1})^2$). The approximation adds to the fine-grain parallelism of the preconditioner. It becomes as parallelizable as a diagonal scaling preconditioner but remains much more effective than diagonal scaling at reducing the condition number of $M^{-1}A$. In order to implement this preconditioner we only need to store LD^{-1}. $M^{-1}r$ can be calculated every time this preconditioning is applied. The cost of performing this preconditioning compared to the IC Preconditioning is almost twice (in terms of number of FLOPS) for the variant where we use two terms of the series (5). We use this variant in the GPU code and abbreviate it as (neu2).

B. *Properties of the Preconditioned Matrix*

In this section we give the motivation for second level preconditioning using deflation. In Figure 2 we show the plot for eigenvalues of a small problem with 8 bubbles defined similarly as the problem introduced in Section II, Figure 1(b). We use a loglog scale in Figure 2. $M^{-1}A$(ic) refers to ICCG(0).

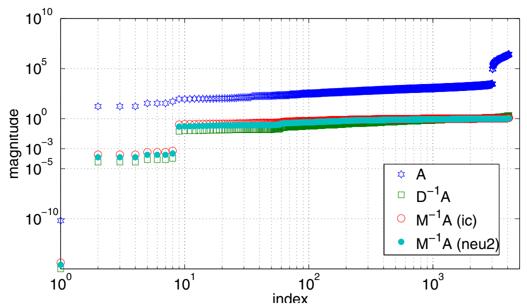


Figure 2: Spectrum of preconditioned matrices. 16^3 number of unknowns.

As noticed in Figure 2 there are 7 eigenvalues in the preconditioned matrix that are of the order of the density contrast i.e. 10^{-3}. In this case the outer medium (e.g., water) has density 10^3 times the inner medium (e.g., air). Therefore preconditioning must be complemented by another level of preconditioning that can remove these small eigenvalues from the spectrum of the preconditioned matrix.

There is also 1 small eigenvalue due to the choice of Neumann boundary conditions and preconditioning pushes it down further. We can invert the density contrast so that the outer medium's density is 10^{-3} times the inner one. In that case the problem is relatively less ill-conditioned and has no small eigenvalues. In our results we only show the data for the case when the outer medium's density is 10^3 times the density of the inner medium, because this problem is challenging (due to a bad condition number owing to small eigenvalues) and the deflation operation is therefore beneficial. In Figure 2 there are also a lot of large magnitude eigenvalues, which after scaling are condensed into the 7 small eigenvalues.

IV. Removing small eigenvalues using deflation: Choices and Reasons

In this section we briefly introduce the concept of deflation and point the reader to sources with elaborate information.

A. *Deflation*

Deflation is a technique specifically to bring the small eigenvalues to 0. It involves solving the system

$$M^{-1}PA\hat{x} = M^{-1}Pb, \tag{6}$$

where

$$P := I - AQ, \quad Q := ZE^{-1}Z^T, \qquad E := Z^TAZ. \tag{7}$$

in which M is the preconditioner we use, and $Z \in \mathbb{R}^{N\times k}$ can be interpreted as a deflation subspace matrix. The matrix Z should contain the approximation to the eigenvectors of $M^{-1}A$. The closer the approximation, the better the result from the deflation operation. The resulting method is called the Deflated Preconditioned Conjugate Gradient (DPCG) Method. If M is the ICCG(0) preconditioner it is called the DICCG(0) method. We implement the algorithm as given in [11].

All the deflation vectors we use are piece-wise constant. They take the value of 1 in the domain where they are defined and 0 elsewhere.

B. *Implementing deflation*

For the deflation operation we have to calculate the product Pr every iteration, where P is the deflation matrix introduced earlier and r is the residual of the i^{th} iteration. This operation can be broken down as follows,

$$a_1 = Z^Tr, \tag{8a}$$

$$a_2 = E^{-1}a_1, \tag{8b}$$

$$a_3 = AZa_2, \tag{8c}$$

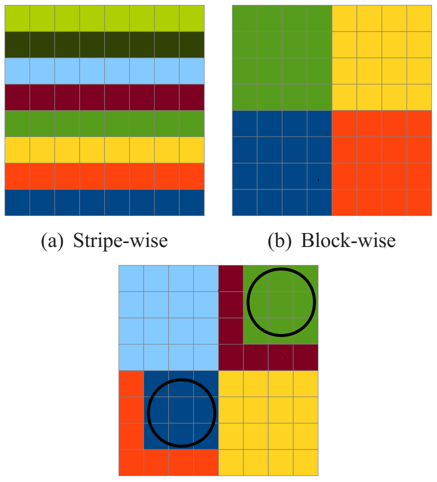
$$s = r - a_3. \tag{8d}$$

(8b) shows the solution of the inner system that results during the implementation of deflation. E is the so called Galerkin matrix. Solving this system presents us with the following options:

1) calculate E^{-1} on the CPU, copy it to the GPU and use it every iteration in *gemv* for (8b),
2) use (triangular) factorization to solve $Ea_2 = a_1$,
3) use Conjugate Gradient (without preconditioning) to solve $Ea_2 = a_1$, but with a higher tolerance than the outer iteration.

We have tested all these options on the GPU. However in our results we only use the option of calculating the explicit inverse of E since the number of vectors considered is small (dimensions of E = number of vectors). The explicit inverse approach consumes the least time when the matrix E^{-1} is small. The CPU code against which we test solves the inner system $Ea_2 = a_1$ with a CG iteration.

1) Stripe-wise Vectors: Stripe-wise vectors (Figure 3(a)) are easy to implement and use on the GPU. For an effective deflation operation many of these vectors must be used (as found out in [2]). However, this directly translates into a larger Galerkin matrix E and a consequent bottleneck in the calculation of E^{-1}.


(a) Stripe-wise (b) Block-wise

(c) Level-Set

Figure 3: 3 kinds of deflation vectors. Each color corresponds to a column in Z.

2) Block-wise Sub-domain (SD) Vectors: One can also make deflation vectors that are based on dividing the grid into sub-domains. They are shown in Figure 3(b). In 3 dimensions one can think of these as cubes made up of individual cells. These vectors are more effective than stripwise vectors when used for deflation.

Let Ω, the unit cube, be divided into open (equal, non-overlapping) sub-domains Ω_i, $i = 1, 2, \cdots, k$, such that $\bar{\Omega} = \bigcup_{i=1}^{k} \bar{\Omega}_i$ and $\Omega_i \bigcap \Omega_j = \emptyset$ for all $i \neq j$. The discretized sub-domains are denoted by Ω_{h_i}. For each Ω_{h_i}, we introduce a deflation vector, z_i, as follows:

$$(z_i)_j := \begin{cases} 0, & x_j \in \Omega_h \backslash \bar{\Omega}_{h_i}; \\ 1, & x_j \in \Omega_{h_i} \end{cases} \tag{9}$$

Then the Z matrix for sub-domain (block-wise) deflation vectors is defined by

$$Z_{SD} := [z_1 z_2 \cdots z_{k-1}], \tag{10}$$

On the GPU Z_{SD} (where SD stands for sub-domain) is built with $k-1$ vectors but on the CPU it is made with k vectors. The reason is that with k vectors the sum of all columns, z_i of Z_{SD} is given by,

$$\sum_{i=1}^{k} z_i = \mathbf{1_n}, \tag{11}$$

where 1_n is a vector with all $1's$. Since $\mathbf{1_n}$ is an eigenvector of A, corresponding to eigenvalue 0, the inner system E will be singular, therefore inverse calculation fails. On the CPU the inner system is solved using a CG iteration which can handle a singular system as well.

An important idea here is to make sure that each sub-domain contains a part of at most one bubble. Otherwise one or more small eigenvalues will remain in the spectrum of $M^{-1}PA$, the deflated preconditioned matrix.

3) Level-Set (LS) Vectors: The level-set vectors [12] utilize the knowledge of the physical problem underlying the simulation and can be very effective at accurately approximating the deflation subspace. The level-set function is defined as a signed distance function which takes e.g. a positive value within bubble(s) and a negative value if outside the bubble(s).

Hence if there are k bubbles in the system then we can define k vectors v_k which can be assembled to form Z. It must be noted that the number of columns in Z_{LS} is chosen to be one less than the number of bubbles as mentioned in [13].

4) Level-Set Sub-domain (LSSD) Vectors: It is possible to extend the level-set vectors with sub-domain vectors and improve their effectiveness. We can define

$$Z_{LSSD} := [Z_1, Z_2], \tag{12}$$

where,

$$Z_1 := Z_{SD} \bigcap (\mathbf{1_n} - \bigcup Z_{LS}), \tag{13a}$$

$$Z_2 := Z_{LS} \bigcap Z_{SD}. \tag{13b}$$

The operation $\bigcap$ in (13a) and (13b) creates a matrix (or a vector) whose columns are equal to all possible component-wise multiplications between the columns of the two arguments on either side of the operator $\bigcap$. Operation $\bigcup$ on the other hand creates a vector out of the matrix (on the right hand-side of the operator $\bigcup$) whose entries are the maximum entries of each row of the matrix. Hence, Z_1 consists of all sub-domain vectors of Z_{SD} where the entries corresponding to the medium outside the bubble are zero. Z_2 consists of columns whose entries correspond to the bubbles divided by the sub-domains of Z_{SD}. As one additional step we remove the last column of Z_{LSSD} (where $LSSD$ stands for Level-Set Sub-domain). The reasons for such a construction of Z_{LSSD} and the choice of the number of deflation vectors has been previously studied in [13]. The choice for removing one vectors in constructing the vectors in Section IV-B3 and IV-B4 has been made in accordance with the conjectures first presented in [13].

V. Experiments

In this section we present results of experiments we conducted and comment on the results obtained.

A. Notes on Implementation

We use MAGMABLAS[3] library for the *gemv* operation and for triangular solve (*dpotrf* and *dpotrs*), depending on how we solve the inner system.

We use CUSP and CUSPARSE in our implementations on the GPU. Particularly for CUSP we store

1) A in DIA,
2) AZ in HYB,
3) E^{-1} in dense,
4) Z in ELL,
5) LD^{-1} in DIA.

For CUSPARSE we store all matrices in CSR format. As mentioned earlier in Section IV-B there are three options to solve the inner system.

We will only present results for the 1^{st} option. This is because with the improved (block-wise sub-domain) vectors we do not have to choose a large number of vectors to achieve faster convergence and in that case the triangular solve and explicit inverse method have the same computational times. Similarly and because the GPU implementation of the 3^{rd} option takes a bit more time than the 1^{st} option we do not report the results of the last option. We note, however, that for larger number of vectors (O(n^2)) it might prove useful compared to the 1^{st} option.

B. Results

In this section we present the results of two-level preconditioned CG for 2 representative geometries mentioned in Figure 1. These geometries are instrumental in capturing the effect of different deflation vector choices.

1) Speedup and reported timing: The speedups we report are the ratios of the time it takes for the iterations of the DPCG algorithm on the CPU vis-a-vis the GPU. We report Total time which includes the time required to do iterations and setup time. Setup time refers to translating raw data (A, x and b) into the library data structures. It also includes the time to setup Z, AZ, E^{-1} (in explicit solve case) and LD^{-1} (for preconditioner) and the time it takes to do the operations before entering the CG iteration. Note that memory allocation times are not included in 'Total time', since they can be done once when this iterative linear system solver (after the prototyping phase) is integrated into the full software for the simulation of bubbly flow using the Level-Set approach. We do not include setup time as it varies across different choices of libraries. It is implicit and can be expressed from our results as Total Time minus the Iteration Time.

The CPU optimization saves on storage and uses much less operations for the deflation steps (more details in the Appendices of [13]).

In all our experiments we have a 3D grid with 128 gridpoints per dimension. The tolerance (ϵ) is set to 10^{-6}. The outer medium's density is 10^3 times the density of the inner medium. In the results that follow this jump in density is mentioned as the density contrast of 10^{-3}.

An explanation of some of the abbreviations in the tables that follow are given below.

1) DICCG(0) - runs exclusively on the CPU.
2) The following apply to the GPU code.

[3]http://icl.cs.utk.edu/magma/docs/

- DPCG(neu2) - refers to DPCG with the neu2 preconditioner and an explicit inverse based inner system solve.
- SD-j refers to Sub-domain and LSSD-j refers to Level-Set Sub-domain vectors for deflation. The j refers to the number of columns in Z. x been calculated according to the information given in Sections IV-B2 and IV-B4. In subsequent sections we also have LS-j which has been constructed on the basis of the discussion in Section IV-B3.

2) 8 Bubbles: We consider 8 bubbles placed symmetrically inside the 3D unit cube (Figure 1(b)). The arrangement of the bubbles is such that when (block-wise) sub-domain vectors are used, each of the block vectors contains a bubble. It is a favorable arrangement of bubbles as it helps in making a point about sub-domain deflation and the speedup we have achieved with this variety of deflation vectors.

	CPU
	DICCG(0)
	SD-8
Number of Iterations	197
Total Time	33.79
Iteration Time	33.49

Table I: 8 bubbles. CPU implementation.

Further we discuss the implementation on the GPU for the problem where the inner system, (8b), is solved with an explicit inverse of E using a *gemv* operation.

	CUSP			CUSPARSE		
	SD-7	LS-7	LSSD-15	SD-7	LS-7	LSSD-15
Number of Iterations	245	381	203	245	381	203
Total Time	7.4	9.5	6.6	8.65	9.56	8.8
Iteration Time	4.4	6.5	3.6	7.3	6.3	7.92
Speedup	7.6	5.1	9.3	4.58	5.31	3.80

Table II: 8 bubbles. Comparison of deflation vector choices on the GPU (CUSP & CUSPARSE based implementation).

From Table II we can see how Level-Set Sub-domain deflation can be effective at accelerating convergence. The speedup can be attributed to the fact that,

- There are more vectors in the Level-Set Sub-domain based Z, and;
- For this geometry all bubbles are inside the sub-domains so that Level-Set Sub-domain is the optimal choice to create Z.

For CUSPARSE results in Table II the speedup falls across all deflation variants. Setup time for CUSPARSE is less than CUSP (for some versions) but iteration times are larger in almost all cases. This continues to be the trend for all the experiments that follow so we do not present CUSPARSE results from this point on.

3) 9 bubbles: The 9 bubble case (from Figure 1(c)) is an extension of the scenario presented in the previous section (V-B2). It has the 8 bubbles as in the previous case but in addition it has a 9^{th} bubble. In this problem the number of sub-domains are kept fixed at 8 as in the previous problem. The sub-domains are now cutting (Figure 1(c)) the bubble in the middle and this delays the convergence of sub-domain deflation. This example highlights the advantage of using Level-Set and Level-Set Sub-domain deflation vectors.

In Table III the convergence seems to be considerably delayed compared to the neatly arranged 8 bubble case discussed in the previous section. To remedy this we have to consider better deflation vector choices.

	CPU	GPU-CUSP		
	DICCG(0)	DPCG(neu2)		
	SD-8	SD-7	LS-7	LSSD-23
Number of Iterations	508	632	381	206
Total Time	85.9	14.4	9.3	6.8
Iteration Time	85.6	11.3	6.5	3.8
Speedup	-	7.57	13.1	22.5

Table III: 9 bubbles. Comparison of deflation vector choices for deflation on the GPU (CUSP based implementation) vs. CPU.

For GPU results in Table III we can infer that the Level-Set vectors alone can accelerate convergence, but Level-Set sub-domain vectors are better.

4) More Vectors: In this section, we increase the number of deflation vectors (in all variants) for the 9 bubble problem and see the effect on speedup and convergence. We consider two new sizes. One of 64 sub-domains and another of 512 sub-domains.

The Level-Set only case is not presented since the results do not change as the number of Level-Set vectors stay the same.

The tolerance had to be increased when increasing the number of vectors from 8 to 64 and 512. This is due to the increase in rounding errors in the solution of the inner system which becomes increasingly ill-conditioned.

For the case of 8 sub-domain vectors, we refer to Table III. We observe how speedup for these implementations changes. Level-Set sub-domain GPU versions are the most promising.

Looking at Table IV we can say that for CUSP based implementations it is possible to obtain more than 30 times speedup. Due to the increase in the number of vectors for the deflation operation for the Level-Set Sub-domain case, the *gemv* operation (using explicit inverse of E) for (8b) has greater data parallelism to offer. Level-Set sub-domain based vectors better approximate the deflation subspace so the iteration counts also go down.

	CPU	GPU-CUSP	
	DICCG(0)	DPCG(neu2)	
	Inner Tolerance=10^{-9}	-	
	SD-64	SD-63	LSSD-135
Number of Iterations	472	603	136
Total Time	81.39	13.61	5.58
Iteration Time	81.1	10.61	2.48
Speedup	-	7.64	32.7

Table IV: 9 bubbles. Two deflation variants. GPU and CPU Execution Times and Speedup. 64 sub-domains.

In table V Level-Set Sub-domain deflation does not show any effect on convergence because the sub-domains have become so small that each sub-domain has at most one part of the bubble in the center. Hence the problem again is suited to sub-domain deflation more than to Level-Set sub-domain (like in Section V-B2). It is also interesting to note that the speedup in Table V solely reduces (when compared with 64 vectors case in Table IV) due to the decrease in number of iterations of the CPU version (owing to more accurate solution of the inner system) of the code by a drastic amount.

	CPU	GPU-CUSP	
	DICCG(0)	DPCG(neu2)	
	Inner Tolerance=10^{-10}	-	
	SD-512	SD-511	LSSD-583
Number of Iterations	67	81	81
Total Time	12.51	4.56	4.62
Iteration Time	12.18	1.56	1.62
Speedup	-	7.81	7.52

Table V: 9 bubbles. Two deflation variants. GPU and CPU Execution Times and Speedup. 512 sub-domains.

We have tested the CPU version with OpenMP parallelization but the CPU version gains are limited due to the fact that majority of the time is spent in the IC preconditioner which is inherently serial.

In Table VI we show the new execution times for the CPU results presented in Tables III, IV and V but with the CPU code accelerated with OpenMP. The CPU used is a dual-quad-core system running at 2.4GHz and the number of OpenMP threads is set to be 8.

In Table VII we show the results for the execution of our CPU implementation on a single core of the same quad-core machine used to generate results in Table VI.

In Figure 4 we compare the speedups that we achieve for a sequential CPU code (from Table VII) (running on one core of the dual quad core CPU) versus the OpenMP version(from Table VI) as compared to the GPU implementation presented in Tables III, IV and V. On the Y-axis in Figure 4 the CPU version / GPU version of the code are mentioned.

	CPU-OpenMP(8 threads)		
	SD-8	SD-64	SD-512
Inner Tolerance	10^{-8}	10^{-9}	10^{-10}
Number of Iterations	508	472	67
Total Time	72	68.35	10
Iteration Time	71.76	68.1	9.7

Table VI: 9 bubbles. CPU versions of DPCG with 8, 64 and 512 vectors with OpenMP acceleration.

	CPU(single thread)		
	SD-8	SD-64	SD-512
Inner Tolerance	10^{-8}	10^{-9}	10^{-10}
Number of Iterations	508	472	67
Total Time	82.1	77.56	13.26
Iteration Time	81.83	77.28	12.96

Table VII: 9 bubbles. CPU versions of DPCG with 8, 64 and 512 vectors without OpenMP acceleration (on single core of a dual quad core).

The speedup figures change by at most 25% by the use

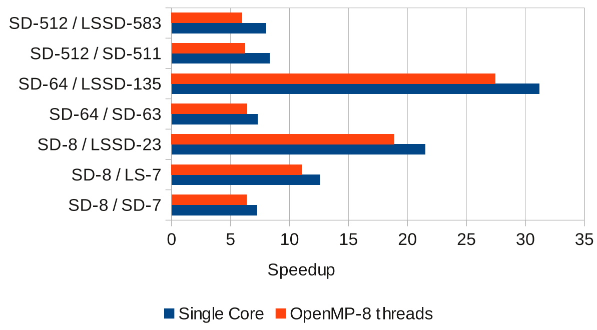


Figure 4: Comparison of Speedup with openMP parallelization of CPU code

of OpenMP (8 threads) primarily because in the CPU version the Incomplete Cholesky(IC) Preconditioning is a bottleneck.

VI. Conclusion and Outlook

In our results we observed that neu2 Preconditioning performs similar to ICCG(0) based preconditioning when coupled with Deflation. This is very useful for solving such ill-conditioned systems since preconditioning is required in order to solve them in a realistic time-frame. ICCG(0) is very difficult to parallelize and is even more difficult for general stencils. In comparison, the neu2 preconditioners we present, exhibit fine-grain parallelism as desired for the best performance on GPUs. It is simple to construct and one can use this preconditioner for general stencils, which makes it applicable to a wider variety of problems and discretization choices. We store the matrices (A in DIA, AZ in HYB, Z in ELL) in generic storage formats. This makes the performance of the operations involved in the DPCG algorithm less dependent on the choice of stencils and structure of the meshes. Our results show that with suitable deflation vectors and neu2 preconditioning it is possible to achieve between 5-30 times speedup for certain problems. We note that increasing the number of vectors must be balanced with the approach to reduce setup times.

As a continuation of our research we want to further optimize the code so that setup times can be reduced. We plan to implement our algorithm for multiple GPUs and multiple multi-core CPUs connected through a fast interconnect to test the scalability of our two-level preconditioning approach.

References

[1] R. Gupta, M. B. van Gijzen, and C. Vuik, "Efficient two-level preconditioned conjugate gradient method on the GPU," Delft University of Technology, Delft, The Netherlands, Tech. Rep., 2011, DIAM Report 11-15.

[2] ——, "Efficient two-level preconditioned conjugate gradient method on the gpu." *Springer Lecture Notes in Computer Science.*, submitted for publication.

[3] D. Jacobsen and I. Senocak, "A full-depth amalgamated parallel 3d geometric multigrid solver for GPU clusters," in *49th AIAA Aerospace Sciences Meeting*. American Institute of Aeronautics and Astronautics (AIAA), 2011.

[4] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, "Sparse matrix solvers on the GPU: conjugate gradients and multigrid," *ACM Trans. Graph.*, vol. 22, no. 3, Jul. 2003.

[5] H. Knibbe, C. W. Oosterlee, and C. Vuik, "GPU implementation of a helmholtz krylov solver preconditioned by a shifted laplace multigrid method," *J. Comput. Appl. Math.*, vol. 236, no. 3, Sep. 2011.

[6] J. M. Tang, S. P. MacLachlan, R. Nabben, and C. Vuik, "A comparison of two-level preconditioners based on multigrid and deflation," *SIAM J. Matrix Anal. Appl.*, vol. 31, no. 4, pp. 1715–1739, Mar. 2010. [Online]. Available: http://dx.doi.org/10.1137/08072084X

[7] V. Heuveline, D. Lukarski, N. Trost, and J.-P. Weiss, "Parallel smoothers for matrix-based geometric multigrid methods on locally refined meshes using multicore CPUs and GPUs," in *Facing the Multicore - Challenge II*, ser. Lecture Notes in Computer Science, 2012.

[8] H. Rudi and J. Koko, "Parallel preconditioned conjugate gradient algorithm on GPU," *Journal of Computational and Applied Mathematics*, vol. 236, no. 15, pp. 3584 – 3590, 2012.

[9] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. Supercomputing, 2009.

[10] J. A. Meijerink and H. A. van der Vorst, "An iterative solution method for linear systems of which the coefficient matrix is a symmetric m-matrix," *Mathematics of Computation*, vol. 31, no. 137, pp. 148–162, jan 197.

[11] J. Tang and C. Vuik, "New variants of deflation techniques for pressure correction in bubbly flow problems," *Journal of Numerical Analysis, Industrial and Applied Mathematics*, vol. 2, pp. 227–249, 2007.

[12] ——, "Efficient deflation methods applied to 3-D bubbly flow problems," *Electronic Transactions on Numerical Analysis*, vol. 26, pp. 330–349, 2007.

[13] J. Tang, "Two-level preconditioned conjugate gradient methods with applications to bubbly flow problems," Ph.D. dissertation, Delft University of Technology, Delft, The Netherlands, 2008.