INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN FLUIDS
Int. J. Numer. Meth. Fluids 2016; **80**:666–683
Published online 24 September 2015 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/fld.4170

Evaluation of the deflated preconditioned CG method to solve bubbly and porous media flow problems on GPU and CPU

R. Gupta[1,*,†], D. Lukarski[2], M. B. van Gijzen[1] and C. Vuik[1]

[1]*Delft Institute of Applied Mathematics, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, Delft, The Netherlands*
[2]*Department of Information Technology, Division of Scientific Computing, Uppsala University, Uppsala, Sweden*

SUMMARY

In both bubbly and porous media flow, the jumps in coefficients may yield an ill-conditioned linear system. The solution of this system using an iterative technique like the conjugate gradient (CG) is delayed because of the presence of small eigenvalues in the spectrum of the coefficient matrix. To accelerate the convergence, we use two levels of preconditioning. For the first level, we choose between out-of-the-box incomplete LU decomposition, sparse approximate inverse, and truncated Neumann series-based preconditioner. For the second level, we use deflation. Through our experiments, we show that it is possible to achieve a computationally fast solver on a graphics processing unit. The preconditioners discussed in this work exhibit fine-grained parallelism. We show that the graphics processing unit version of the two-level preconditioned CG can be up to two times faster than a dual quad core CPU implementation. Copyright © 2015 John Wiley & Sons, Ltd.

Received 7 October 2014; Revised 16 July 2015; Accepted 26 August 2015

KEY WORDS: deflation; preconditioning; conjugate gradient; GPU; PARALUTION

1. INTRODUCTION

In this article, we investigate the linear system arising from porous media flow and bubbly flow. The linear systems that we are interested in are discretizations of

$$-\nabla.(\kappa(x)\nabla p(x)) = f(x),\ x \in \Omega, \tag{1}$$

together with suitable boundary conditions. For both the flow problems (bubbly flow and porous media flow), Ω and p denote the computational domain and pressure, respectively. In case of porous media flow, the quantity $\kappa(x) = \sigma(x)$ denotes the permeability of the medium through which the flow occurs. For bubbly flow, $\kappa = \rho(x)$ denotes the ratio of densities of the two mediums.

To model and solve the bubbly flow problem, we use the mass-conserving level-set method [1]. For the porous media flow problem, we follow the solution approach described in [2].

There is a sharp contrast at the interfaces for both problems. In bubbly flow, at the boundary of the bubble, there is a jump in density, and in the case of porous media flow, the ratio of the permeability between two layers of material is high. Discretization of (1) yields

$$Ax = b. \tag{2}$$

*Correspondence to: R. Gupta, Delft Institute of Applied Mathematics, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, Delft, The Netherlands.
†E-mail: itabhiyanta@gmail.com

In (2), A is the coefficient matrix, x is the unknown pressure, and b is the right-hand side. The coefficient matrix A is highly ill-conditioned, sparse, and symmetric positive (semi) definite (SPD). The ill-conditioning results from the sharp contrasts in material properties (density and permeability) that are inherent to such problems. Because the matrix A is sparse and large, using iterative methods is preferable over direct methods. The conjugate gradient (CG) method is most suited because of the nature of the coefficient matrix.

Through this work, we want to show that for bubbly and porous media flow, we can use deflation with preconditioning to accelerate convergence on the graphics processing units (GPUs). In addition, we also want to provide a comparison with an optimized CPU implementation and show that GPUs can be advantageous.

We consider a variety of first-level preconditioning techniques [3] along with deflation [4] for a two-level preconditioned conjugate gradient (PCG) implementation.

On the one hand, we apply two-level preconditioning to accelerate convergence. On the other hand, we focus on implementing PCG method on the GPU. The GPU devices provide higher performance both in terms of memory bandwidth and in terms of computational power in comparison with traditional CPUs. For efficient utilization of the GPUs, the algorithms need to possess fine-grained parallelism. This is a critical aspect, and for this task, it is important to rethink, tailor, or even devise new preconditioning techniques that suit the GPU architecture and can be computationally efficient on the GPU.

In this paper, we use the deflated preconditioned conjugate gradient (DPCG) implemented within the PARALUTION library [5]. PARALUTION provides various sparse iterative solvers and preconditioners (including DPCG). It includes several preconditioners based on incomplete LU-type decomposition, which we have used in this paper.

Deflated PCG method for bubbly flow has been previously studied in [6–8]. In these papers, an incomplete Cholesky preconditioner is suggested for the first-level preconditioning and deflation for the second-level preconditioning. The authors point out the methods for effective deflation by studying a variety of deflation vectors. However, in this approach, incomplete Cholesky preconditioning is a bottleneck in a GPU implementation of the DPCG method. An implementation of the DPCG method on the GPU with a preconditioner exhibiting fine-grained parallelism was first reported in [4], and better variants of deflation vectors were studied in [9]. Both these works implemented DPCG on the GPU. For composite materials, DPCG has been previously studied in [10]. Preconditioners that are suitable for GPU implementation of the PCG method have also been discussed in [3, 11, 12]. The matrix storage formats that we mention in this work are discussed in detail in [13] along with their GPU implementations.

In the next two sections, we describe the preconditioning techniques that we use on both levels. In Section 4, we explain the main building blocks of the preconditioning schemes used on a GPU. In Section 5, we present our numerical experiments and analyze the performance for both the problems that we consider. In the final section, we present our conclusions.

2. FIRST-LEVEL PRECONDITIONING TECHNIQUES

In this section, we describe the first-level preconditioning techniques that we use to accelerate convergence of CG. They exhibit fine-grained parallelism and hence can be executed efficiently on a GPU. We begin with preconditioners based on classical incomplete LU-based factorizations followed by approximate inverse-based preconditioners, and finally, we explain the truncated Neumann series (TNS)-based preconditioners.

We note that PARALUTION uses ILU factorizations even for the symmetric problems because of the following:

- The memory footprint is the same as we need to store (in all cases) the U part (in Cholesky $U = L^T$). This is because calculating transposes for different storage formats is computationally expensive. This is a design choice in PARALUTION and is true for all situations where the transpose is needed.

DOI: 10.1002/fld

- The incomplete Cholesky factorization requires positive diagonal elements. The classical incomplete Cholesky factorization does not always exist for all sparse SPD matrices [14]. To solve that problem, there are non-symmetric permutations that are incompatible with the multi-colored techniques.

In the case when incomplete Cholesky factorization is not SPD, then it must be combined with Krylov methods for non-symmetric matrices (like the generalized minimal residual method).

2.1. Black-box ILU-type preconditioners

Incomplete LU factorization generates sparse matrices L and U such that $A \approx LU$, where L is lower triangular and U is upper triangular matrix. The sparsity pattern after factorization depends on factorization procedure – the L and U matrices can have the same non-zero structure as the original matrix A (without fill-in) or additional entries can be added based on level or threshold techniques [15]. The classical way for solving the forward and the backward substitution in LU-type preconditioners is to perform the sweeps sequentially. In the following subsections, we describe various techniques for parallelizing LU-type preconditioners. In the following methods, we make both the L and U components. We store both of them because the transpose operation is not very fast when considering multiple storage formats like ELL and DIA.

2.1.1. Level-scheduling method. Because of the sparsity structure of L and U, the forward and backward sweeps can be performed partly in parallel after analyzing the matrix graphs. This technique is called level scheduling [15]. To determine the levels, this technique requires an additional pre-processing step, which calculates the element dependence.

2.1.2. ILU(p, q). Typically, the inter-connectivity of the matrix graphs of L and U is high, and thus, we cannot extract much parallelism in the LU sweeps. To provide additional parallelism, we can permute the original matrix A with a permutation P based on the matrix structure of $|A|^q$, where $|A| = |a_{i,j}|$. This is called the power(q)-pattern method, and the factorization ILU(p, q) describes the level of fill-in p and the matrix power q. Using a multi-colored decomposition, we can define sub-blocks of the factorization. Specifically, for $q = p + 1$, it has been shown [3] that the diagonal sub-blocks have only diagonal entries – the fill-in elements do not appear in the diagonal blocks. Thus, the performance of the forward and backward substitutions can be improved in the block form. Details can be found in [3].

For finite element methods (also for finite difference and volumes), the underlying matrix graph of the discretized problem depends on the basis (linear, quadratic, etc.) of the elements and on the mesh topology. The degree of parallelism (the number of independent parallel tasks) for solving the forward and backward substitutions does not depend on the discretization size (the number of unknowns in the system). Thus, also for larger problem sizes (small discretization steps), this solution technique provides higher parallelism.

2.1.3. Multi-elimination ILU. As an alternative to the usual LU decomposition, we can factorize the system matrix A in the following block form:

$$A = \begin{bmatrix} T & F \\ E & C \end{bmatrix} = \begin{bmatrix} I & 0 \\ ET^{-1} & I \end{bmatrix} \times \begin{bmatrix} T & F \\ 0 & \hat{A} \end{bmatrix}, \tag{3}$$

where $\hat{A} = C - ET^{-1}F$. This is an exact factorization, and if we solve the sub-problem T and $\hat{A}$, we can obtain the solution of the problem in one iteration. To make the inversion of the T matrix easier, we perform a symmetric permutation of the original matrix PAP^{-1}, which is based on maximal independent set. After the permutation, the new block structure will contain a new block T with only diagonal elements. Thus, the forward substitution can be performed block-wise in parallel. For the backward step, first, we need to build the $\hat{A}$, which involves a sparse matrix–matrix multiplication and an addition. The matrix $\hat{A}$ is denser in comparison with the matrix C because of the additional fill-ins coming from the matrix–matrix multiplication. We can recursively apply this decomposition

DOI: 10.1002/fld

till we come to the last step when we have to provide a solution. The last block of the preconditioner can be solved with Jacobi, symmetric Gauss–Seidel (SGS), or ILU(p, q) preconditioner. Therefore, we have three flavors of multi-elimination LU-based method, which in our results are abbreviated as ME-ILU-J, ME-ILU-SGS, and ME-ILU-ILU(0, 1).

2.2. Multi-colored symmetric Gauss–Seidel

We parallelize the SGS preconditioner using a multi-coloring decomposition of the original matrix A. The multi-coloring exposes additional parallelism in the forward and backward substitutions. This gives us a block structure in the preconditioner $M := (D + L)D^{-1}(D + L^T)$, where L is the strict lower triangular part of A, the coefficient matrix, and D is the diagonal of A. For details, see [3]. In our results, this preconditioner is abbreviated as MCSGS.

2.3. Truncated Neumann series-based preconditioning

We can define another factorization-based preconditioner continuing on the idea of the decomposition used for the SGS-type preconditioner. In this case, we also approximate the inverse of the factors. The preconditioner can be written as

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

where I is the identity matrix and L and D follow from the SGS preconditioner.

We approximate the inverse of the preconditioner in Equation (4) using Neumann series. Thus, to calculate $(I + LD^{-1})^{-1}$, we use its Neumann series expansion

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

Similarly for the other term, $K = (I + L^T D^{-1})$. In order to make sure that the preconditioner is not very expensive, we truncate the series after three terms (including I).

2.4. Factorized sparse approximate inverse-based preconditioners

An efficient algorithm for construction of a sparse approximate inverse [16] with respect to the complexity of the building phase is the factorized sparse approximate inverse (FSAI) method [17]. This algorithm preserves the symmetry of the preconditioner if the initial matrix is SPD. FSAI builds an approximation to the Cholesky factorization of A, $A = L_A L_A^T$. Thus, we need to find an approximation $G_L \approx L_A^{-1}$. Reformulating, the approximate inverse preconditioned equation GA with $G := G_L^T G_L$ reads as follows:

$$G_L A G_L^T \approx I.$$

The approximation G_L is a lower triangular matrix. After some transformations, this leads to the following system:

$$\begin{aligned}(G_L A)_{i,j} &= 0 \quad \text{for } i \neq j, \\ (G_L A)_{i,j} &= (L_A)_{i,j} \quad \text{for } i = j.\end{aligned}$$

The FSAI method does not need to use the diagonal entries of the Cholesky decomposition; see [17]. Thus, to construct the approximate inverse matrix, we need to solve only small SPD systems, which correspond to each row of the original matrix. The FSAI method is different from the AINV method proposed in [18]. AINV approximates the matrix A^{-1} by a bi-conjugation process applied to a linearly independent set of vectors.

DOI: 10.1002/fld

3. SECOND-LEVEL PRECONDITIONING

The first-level preconditioning schemes can be augmented with deflation to further accelerate convergence. Hence, deflation is the second-level preconditioning. The spectrum of A, for the problems considered, has some very small eigenvalues related to the contrast in material properties at the interfaces in the flow problem under consideration. This contrast can be of the order of 10^3 or more. For treating these eigenvalues, we can employ deflation. We define a deflation matrix as follows:

$$\pi := I - AQ, \qquad Q := ZE^{-1}Z^T, \qquad E := Z^T AZ, \tag{6}$$

where I is the identity matrix, $Z \in \mathbb{R}^{N\times k}$ is called the deflation subspace matrix, and A is the original coefficient matrix.

The system to be solved now looks like

$$M^{-1}\pi A\hat{x} = M^{-1}\pi b, \tag{7}$$

where M is the preconditioner that we use on the first level. The algorithm for DPCG is listed in Algorithm 1 and was reported in [6]. The effectiveness of the deflation operation hinges on the choice of the matrix Z. It should be an approximation of the eigenvectors of the small eigenvalues in the spectrum of the matrix $M^{-1}A$. Depending on how close this approximation is to the actual eigenvectors for a given problem, the effectiveness of the deflation method varies.

Algorithm 1 Deflated preconditioned conjugate gradient algorithm

1: Select x_0. Compute $r_0 := b - Ax_0$
2: $r_0 = \pi r_0$
3: **for** i:=0,..., until convergence **do**
4: Solve $M w_{i-1} := r_{i-1}$
5: $\rho_{i-1} = {r_{i-1}}^T w_{i-1}$
6: **if** $i = 1$ **then**
7: $p_i = w_{i-1}$
8: **else**
9: $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
10: $p_i = w_{i-1} + \beta_{i-1} p_{i-1}$
11: **end if**
12: $q_i = \pi A p_i$
13: $\alpha_i = \rho_{i-1}/{p_i}^T q_i$
14: $x_i = x_{i-1} + \alpha_i p_i$
15: $r_i = r_{i-1} - \alpha_i q_i$
16: **if** $\| r_i \|_2 \leqslant \epsilon \| b \|_2$ **then**
17: exit
18: **else**
19: go to step 4
20: **end if**
21: **end for**
22: $x_{it} := Qb + \pi^T x_i$

There can be many choices for deflation vectors. They can be categorized into vectors based on geometry of the problem and vectors based on the physics of the underlying problem. They can also be combined to achieve another class of deflation vectors.

3.1. Geometry-based deflation vectors

Sub-domain deflation vectors. To begin with, we divide the computational domain into equal-sized domains. These deflation vectors related to the domains can be seen as piecewise constant vectors,

DOI: 10.1002/fld

which when stacked side by side in columns make up Z. Within a domain, there can be regions of different material contrasts. The boundary of two mediums is called an interface. Sub-domain deflation works best when a deflation vector captures at most one interface. If this is not true for any of the deflation vectors, then the small eigenvalues persist in the spectrum of deflated A and continue to delay convergence. Thus, the effectiveness of this method relies on the position of the sub-domains with respect to the interfaces.

3.2. Physics-based deflation vectors

3.2.1. Bubbly flow problem.

Level-set deflation. Level-set deflation [7] utilizes the information from the level-set function that is implicit in the mass-conserving level-set method. The level set is a signed distance function that has positive values inside the bubble and negative values outside the bubble or vice versa. In order to construct Z based on level-set information, the simplest technique can be to use the grid cells through which the interface passes as ones and zeros everywhere else. Moreover, for each interface, we make one such vector.

3.2.2. Porous media flow problem.

Sub-domain vectors with weighted overlapping. Deflation vectors that we use for porous media flow are similar in construction to the sub-domain deflation vectors used for bubbly flow. In addition, they use the knowledge of the permeability constants in the different layers of the domain and use a weighted average of these constants in the region where there is an interface. For the cells that lie on the interface, they have a weighted value, and for cells in one permeability, they have ones, and for the other medium, they have all zeros. The details of this construction are given in [2]. We make these vectors according to the structure of the layers. This way, they capture the physics of the problem well. This has been explained in [2].

3.3. Vectors based on the combination of geometry and physics of the problem

3.3.1. Bubbly flow problem.

Level-set sub-domain deflation. One can combine the sub-domain and the level-set technique to make a better approximation of the eigenvectors corresponding to the bad eigenvalues. A simple combination that involves putting sub-domain and level-set deflation vectors side by side is not the only choice though. One way of approximating the required eigenvectors is to make up the Z matrix consisting of a collection of two sets of vectors: those that have ones for cells that are inside a sub-domain but outside a bubble and those that are inside the bubble and inside the sub-domain also. While building this matrix, care must be taken to make sure that Z remains a full rank matrix.

4. IMPLEMENTATION DETAILS

All solvers and preconditioners are implemented in the PARALUTION library (version 0.7.0) [5]. This is a library that provides various preconditioners and solvers based on iterative methods. The methods can be performed on different back-ends such as OpenMP, CUDA, or OpenCL.

4.1. Sparse matrix storage

In Table I, we outline the storage formats (refer [13] for information on formats) that we use on the GPU. We keep the matrix A in the DIA format on the GPU. Keeping it in this format is instrumental in achieving high throughput for SpMVs using A. In Table I, we outline the formats that we have used for storing various matrices that we make for the DPCG method. Our choices for storage of matrices in the formats chosen (as described and also mentioned in later sections) give up to 20% improvements in wall-clock times over using the CSR format on the GPU. On the CPU,

DOI: 10.1002/fld

Table I. Sparse storage formats on GPU.

A	Preconditioners			
DIA	TNS DIA	FSAI	MCSGS HYB	Others CSR

we keep the CSR format for all matrices as there is no performance improvement by changing the storage formats.

On the CPU, we use the CSR format for all the matrices and preconditioners mentioned in Table I. Using the DIA format for the storage A, L, and L^T (for TNS-based preconditioner) for this problem size does not give a significant improvement in performance for this problem size on the CPU.

4.2. Speedup and stopping criteria

We measure it as the total time (build and solve) on the CPU divided by the total time on the GPU, which includes the transfer time as well.

We define our stopping criteria for a given tolerance ϵ as

$$\| r_i \|_2 \leqslant \| b \|_2 \epsilon \tag{8}$$

where r_i is the residual at the ith iteration and b is the right-hand side.

4.3. LU-type preconditioners

The LU traversing for the level-scheduling technique is performed via the CUSPARSE library in CUDA 5.5. In the building phase, the graph structure is analyzed and then passed for each traversing step.

For higher degree of parallelism in the ILU(p, q) and SGS preconditioners, a multi-coloring analysis is performed. This algorithm is inherently sequential and is therefore performed on the CPU – this requires an additional copy of the matrix to and from the GPU. The same procedure follows in the multi-elimination ILU preconditioner where the maximal independent set algorithm is performed sequentially on the CPU.

For the ILU(p, q), SGS, and multi-elimination ILU, all sub-blocks obtained after the permutation are extracted as single matrices. This is to say that the L, U, and D are stored in one matrix even though they can be interpreted as separate block matrices. In this way, the forward and backward substitutions (for solving $Mz = r$) are performed in the following block way:

$$\begin{aligned} x_i &:= D^{-1}\left(r_i - \sum_{j=1}^{i-1} L_{i,j}x_j\right) \\ z_i &:= I\left(x_i - \sum_{j=1}^{B-i} U_{i,j}z_{i+j}\right) \end{aligned}$$

for the preconditioner $LUz = r$, where matrices are divided into B sub-blocks and the indexes describe the corresponding blocks. In Equation (9), D is the main diagonal, and I is the identity matrix. The first equation is solved forward $i = 1 \ldots B$, and the second is solved backward $i = B \ldots 1$. The inversion of D_{Li} and D_{Ui} matrices is trivial because they contain only diagonal elements (by construction). Note that the usage of an extra vector x is not actually necessary and it is not used in the code.

In the multi-elimination ILU preconditioner, the computation of $\hat{A}$ (3) is carried out on the GPU via the CUSPARSE library. After the solution of this matrix, we apply a preconditioner – Jacobi, multi-colored SGS, or ILU(0,1).

DOI: 10.1002/fld

The permutation of the preconditioner is not applied to the original matrix A. Thus, during the solution of the preconditioning equation $Mz = r$, the input vector r is permuted, and after the computation of the solution, the vector z is permuted back to correspond to the original matrix.

4.4. Factorized sparse approximate inverse-based preconditioners

The construction of the FSAI preconditioner is performed entirely on the CPU. The building phase consists of three steps: extracting the lower triangular pattern of A, computing the inverse elements by a LU decomposition, and constructing the new matrix. All operations in the setup phase are performed with the CSR format. Further, it is stored in the HYB format when used in the solution phase.

4.5. Truncated Neumann series-based preconditioning

For the TNS-based preconditioning, we store LD^{-1} and $D^{-1}L^T$ in the DIA format. This is because their structure closely resembles that of the matrix A and hence, SpMVs involving them are computed faster. We store the transpose of LD^{-1} because doing transpose and then performing the SpMV operation are more time consuming. We use three terms of the Neumann series for approximating $(I + LD^{-1})^{-1}$. Four matrix–vector products (two with LD^{-1} and two with $D^{-1}L^T$) are required in the variant that we use. Furthermore, some vector updates and point-wise-vector multiplications are also used in this preconditioning scheme. PARALUTION provides routines for all these matrix/vector operations.

4.6. Deflation

For the deflation algorithm listed in Section 3, we can categorize the operations into sparse matrix–dense vector multiplications (for Ax and AZv), sparse matrix–matrix products (for calculating AZ), and dense matrix–vector products (for calculating $E^{-1}v$), where x is an $N \times 1$ column vector and v is a $d \times 1$ vector. N is the problem size, and d is the number of deflation vectors in matrix Z. Specifically, we store AZ in the ELL format and Z in the CSR format. For most of these operations, PARALUTION already contains the requisite functions in the linear solver class. However, we have made a special class named DPCG, which is derived from the linear solver class and is used to launch an object for running the DPCG solver class. This class also has special functions for setting up the deflation subspace matrix Z, which is computed on the CPU.

4.6.1. Solving the inner system. We implement deflation in four steps:

$$a_1 = Z^T r, \tag{9a}$$

$$a_2 = E^{-1} a_1, \tag{9b}$$

$$a_3 = AZ a_2, \tag{9c}$$

$$s = r - a_3. \tag{9d}$$

The inner system solve (9b) can be implemented in three different ways:

1. Calculating explicit inverse of E and multiplying with E^{-1} the vector a_1.
2. Calculating a_2 using triangular solve techniques, that is, first calculating the Cholesky factorization and then solving the system with this factorized matrix.
3. Using CG to solve the inner system.

We choose the first method of doing a dense matrix–vector product because it is very well optimized for GPUs and is more efficient than triangular solve on this device. However, the second and the third options can be beneficial in cases when the number of deflation vectors is too high and calculating inverse of E becomes computationally expensive.

DOI: 10.1002/fld

4.6.2. Building and solving phase for DPCG. After obtaining the matrix A, right-hand side b, and x, the building phase begins. A is stored in the CSR format initially after which it is to be converted to DIA format on the GPU. The matrices Z and Z^T are first constructed on the CPU in the CSR format. Storing Z^T follows the same reasoning as was mentioned in Section 4.5 for storing $D^{-1}L^T$. The matrices AZ and E are then constructed using matrix multiplication routines in PARALUTION. Because of the very small size of E to minimize the time, we perform the calculation of E^{-1} on the CPU. At this point, conversion of the matrices into ELL is carried out. The preconditioner is made at this step, and the resulting matrix or matrices is/are stored in an efficient format. In case of the CPU solver, the CG algorithm begins after this step. For GPU, there is an additional step where all the required vectors and matrices are moved to the GPU.

The solving phase can be performed entirely on the CPU or GPU depending on the selected platform.

5. NUMERICAL EXPERIMENTS

The experiments are performed on a NVIDIA K20 GPU with 6-GB memory and ECC on. The host system is Intel Xeon (E5620) dual quad-core CPU running at 2.4 GHz with 24-GB memory. We use CUDA 5.5 and gcc 4.7.4 compilers. The machines run CentOS Linux.

In the results that follow, we present a comparison of two-level preconditioned schemes using different preconditioners for the first level and deflation for the second. In this paper, we present results for level-set sub-domain deflation vectors discussed in Section 3.3.1. The setup time is defined as the time needed to build the solver. This includes memory allocation and transfers to the device, constructing the preconditioner and matrix conversions. The solution time only includes the solving phase of the preconditioned CG method. The 3D grids that we consider in both the problems are numbered lexicographically with the most varying index being the x-direction followed by y and then z.

5.1. Bubbly flow problem

For the computational domain, we use a 3D cube with dimensions $128 \times 128 \times 128$. This cube has nine bubbles whose density is different from the rest of the cube. The density contrast between outside and inside of the bubble is 10^3. The domain is discretized using the finite difference discretization. This results in a septa-diagonal matrix for the 3D problem that we have considered. For the iterative method, we use a relative stopping criteria of 10^{-6}.

For the DPCG solver, we have kept four blocks/sub-domains in each direction of the 3D domain. So in total, there are 64 sub-domains. To this, more vectors are added because some sub-domains intersect the bubbles (Section 3.3.1). All these vectors make the matrix Z, which is then used for level-set sub-domain deflation.

In interest of the space constraints, in Tables II–IX, we use some abbreviations for the preconditioning schemes. We provide their complete names before presenting the results.

1. MCSGS stands for multi-colored SGS preconditioner.
2. FSAI stands for factorized sparse approximate inverse preconditioner.
3. ILU(0) stands for ILU preconditioner without fill-in.
4. ME-ILU-J stands for multi-elimination ILU where Jacobi preconditioner is used for last block.
5. ME-ILU-SGS stands for multi-elimination ILU where SGS preconditioner is used for last block.
6. ME-ILU-ILU(0,1) stands for multi-elimination ILU where ILU preconditioner with matrix power 1 and level of fill-in zero is used for last block.
7. ILU(0,1) stands for ILU preconditioner with matrix power 1 and level of fill-in zero.
8. DIAGONAL refers to Jacobi or diagonal preconditioning. The preconditioner is the inverse of the main diagonal of the coefficient matrix.
9. TNS stands for truncated Neumann series-based preconditioner.

DOI: 10.1002/fld

Table II. Comparison of PCG schemes on the CPU.

Preconditioner	Iterations	Setup time	Solve time	Total time
MCSGS	660	0.8	28	28.8
FSAI	563	4.79	64.44	69.23
ILU(0)	465	0.37	52.1	52.47
ME-ILU-J	682	1.47	23.63	25.1
ME-ILU-SGS	442	2.81	22.89	25.7
ME-ILU-ILU(0,1)	**401**	3.64	**20.1**	**23.74**
ILU(0,1)	682	0.94	22.93	23.87
DIAGONAL	1318	**0.13**	26.14	26.27
TNS	585	0.54	26.7	27.24

Bubbly flow problem.

Table III. Comparison of PCG schemes on the GPU.

Preconditioner	Iterations	Setup time	Solve time	Total time
MCSGS	660	0.45	4.7	5.15
FSAI	563	4.54	5.25	9.79
ILU(0)	465	0.44	12.97	13.41
ME-ILU-J	682	1.59	4.16	5.75
ME-ILU-SGS	442	1.11	6.14	7.25
ME-ILU-ILU(0,1)	**401**	1.26	5.42	6.68
ILU(0,1)	682	0.54	4.16	4.7
DIAGONAL	1318	**0.116**	**4.2**	**4.316**
TNS	585	1.29	3.93	5.22

Bubbly flow problem.

Table IV. Comparing deflated PCG for different preconditioners on CPU.

Preconditioner	Iterations	Setup time	Solve time	Total time
MCSGS	158	2.21	8.13	10.34
FSAI	127	6.34	6.95	13.29
ILU(0)	108	1.97	13.28	15.25
ME-ILU-J	163	2.79	7.04	9.83
ME-ILU-SGS	158	2.26	7.33	9.59
ME-ILU-ILU(0,1)	**87**	5.19	**5.32**	10.51
ILU(0,1)	163	3.68	7.18	10.86
DIAGONAL	316	**1.72**	9.05	10.77
TNS	136	1.91	7.37	**9.28**

Bubbly flow problem.

We first present the results for the PCG method (Tables II and III) and then for the DPCG method (Tables IV and V). We combine different preconditioners available in the PARALUTION library with deflation and compare their performance. In Tables II and III, we see that the PCG method can be up to five times faster on the GPU compared with the CPU. Diagonal preconditioning emerges as the fastest preconditioning technique.

For the DPCG method, the best speedup (2.5×) is close to half as compared with best speedup (5×) of the PCG schemes as observed in Tables II and III. The ME-ILU-ILU(0,1) preconditioner achieves fastest convergence (in the number of iterations) followed by ILU(0) (for the DPCG method), but in terms of wall-clock times, ME-ILU-SGS preconditioner-based DPCG is fastest.

Comparing the results in Tables II and III with those in Tables IV and V, we see that deflation helps accelerate convergence in terms of the number of iterations and wall-clock times. ME-ILU-ILU(0, 1) is a superior preconditioning scheme in itself as it performs better in comparison with

DOI: 10.1002/fld

Table V. Comparing deflated PCG for different preconditioners on GPU.

Preconditioner	Iterations	Setup time	Solve time	Total time
MCSGS	158	3.124	1.48	4.604
FSAI	127	6.37	1.7	8.07
ILU(0)	108	2.27	3.27	5.54
ME-ILU-J	163	2.3	1.39	3.69
ME-ILU-SGS	158	2.2	1.45	**3.65**
ME-ILU-ILU(0,1)	**87**	3.04	1.39	4.43
ILU(0,1)	163	3.5	1.39	4.89
DIAGONAL	316	**1.89**	1.79	3.68
TNS	136	2.87	**1.26**	4.13

Bubbly flow problem.

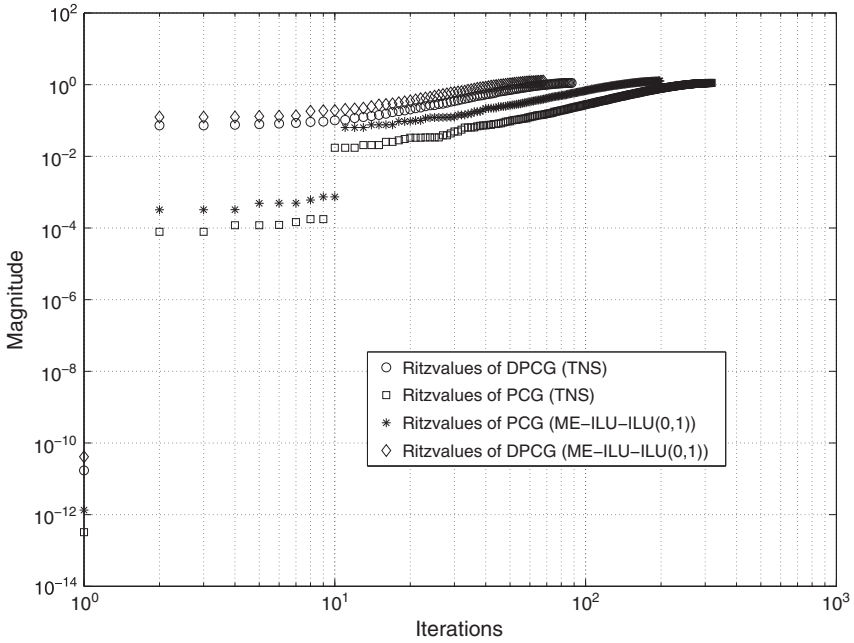


Figure 1. Comparison of Ritz values for DPCG and PCG. Preconditioners used TNS based and ME-ILU-ILU(0, 1).

other preconditioning techniques. With the addition of deflation, it is further accelerated in terms of convergence. It still lacks in wall-clock time compared with DPCG with ME-ILU-SGS/J or diagonal preconditioner. Diagonal preconditioning is a very simple and highly parallel scheme, and that is why it performs very well on the GPU. At the same time, it must be noted that the number of iterations for DPCG with ME-ILU-SGS or diagonal preconditioners is two to four times more than DPCG with ME-ILU-ILU(0, 1).

To understand the effect of preconditioning and deflation on the spectrum of the matrix A, we approximated the spectrum of A using Ritz values [15, 19]. For this experiment, we reduced the grid size to 32^3 while keeping the number of bubbles the same. In Figure 1, we compare two different DPCG implementations with their PCG counterparts.

The spectrum of A has eight small eigenvalues that correspond to the nine bubbles in the system. We see in Figure 1 that ME-ILU-ILU(0, 1) preconditioner does a slightly better job than TNS. With DPCG using either of these two schemes, all small eigenvalues are removed. Although, the number of iterations required for TNS is more than ME-ILU-ILU(0, 1), which can be verified in Tables II–V.

DOI: 10.1002/fld

5.2. *Porous media flows*

For porous media flow, we consider two problems: one that is defined on a regular domain and one with an irregular geometry. For both problems, we use finite-element discretization using parallelepiped elements. It must be noted that for a regular problem defined on a cuboid domain, these elements are cuboids.

The deflation vectors that we use for this problem have been briefly described in Section 3.2.2. They are piecewise constant but with additional factors added near the interface.

5.2.1. Problem defined on a regular geometry. The first problem that we consider is defined on a regular geometry. This problem has 15 layers with contrast distribution as given in Figure 2. These 15 layers are arranged in slabs with dimensions $63 \times 64 \times 64$. Therefore, the total number of unknowns is $63 \times 64 \times 64 \times 15$. The top most layer has a permeability of 10^4 followed by alternating layers of permeabilities 10^{-6} and 1. The problem is defined on a unit cube. The stopping criteria are as mentioned in Section 4.2, and the tolerance is kept at 10^{-6}.

In Tables VI and VII, we see that the GPU implementations of PCG method for this problem can be between three and four times as fast as compared with their CPU implementations.

In Tables VIII and IX, we notice that the speedup for the DPCG method is only 1.5 times or lower. The setup times for DPCG are much larger on the GPU as compared with the CPU. This is why the speedup is small. The setup times are larger because sparse matrix–matrix multiplication required to make AZ from A and Z and E from Z^T and AZ is 50% of the total time (95% of the setup time). It is optimized because it uses CSR storage format for all the matrices involved and uses latest versions of the CUSPARSE library.

The DPCG solver with TNS-based preconditioner emerges as the fastest DPCG method, whereas for MCSGS based, PCG is the best choice on the GPU in terms of wall-clock time. It must be noted

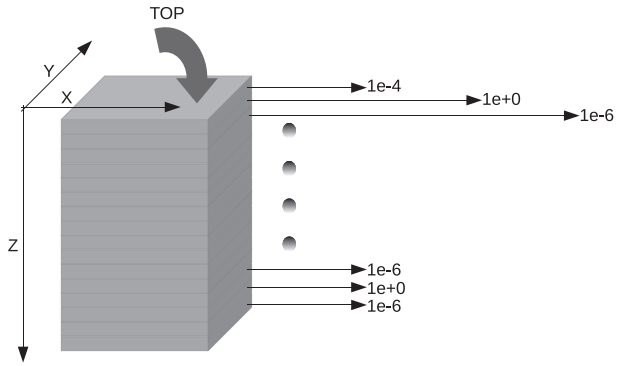


Figure 2. Layered problem. Fifteen layers with variable contrasts.

Table VI. Comparison of PCG schemes on the CPU.

Preconditioner	Iterations	Setup time	Solve time	Total time
MCSGS	2184	1.27	147.38	148.65
FSAI	1823	8.5	224.54	233.04
ILU(0)	1461	0.48	315.49	315.97
ME-ILU-J	2222	2.47	135.04	137.51
ME-ILU-SGS	1251	4.88	116.69	121.57
ME-ILU-ILU(0,1)	**1199**	6.21	**109.38**	**115.59**
ILU(0,1)	2196	1.43	130.31	131.74
DIAGONAL	4288	**0.05**	144.94	144.99
TNS	1902	0.6	153.75	154.35

Layered problem. The number of unknowns is $63 \times 64 \times 64 \times 15$.

DOI: 10.1002/fld

Table VII. Comparison of PCG schemes on the GPU.

Preconditioner	Iterations	Setup time	Solve time	Total time
MCSGS	2146	0.8	30.74	31.54
FSAI	1799	7.72	32.54	40.26
ILU(0)	1427	0.78	84.25	85.03
ME-ILU-J	2194	1.99	**27.99**	29.98
ME-ILU-SGS	1225	1.92	31.42	62.84
ME-ILU-ILU(0,1)	**1177**	2.15	29.49	31.64
ILU(0,1)	2177	0.92	28.66	**29.58**
DIAGONAL	4372	**0.14**	32.22	32.36
TNS	1868	1.35	34.84	36.19

Layered problem. The number of unknowns is $63 \times 64 \times 64 \times 15$.

Table VIII. Comparing deflated PCG for different preconditioners on CPU.

Preconditioner	Iterations	Setup time	Solve time	Total time
MCSGS	251	1.67	23.04	24.71
FSAI	220	8.77	20.21	28.98
ILU(0)	178	0.86	38.11	38.97
ME-ILU-J	254	2.37	17.77	**20.14**
ME-ILU-SGS	251	1.42	19.09	20.51
ME-ILU-ILU(0,1)	**139**	6.5	**14.14**	20.64
ILU(0,1)	271	1.77	19.16	20.93
DIAGONAL	497	**0.36**	22.28	22.64
TNS	214	0.96	20.19	21.15

Layered problem. The number of unknowns is $63 \times 64 \times 64 \times 15$.

Table IX. Comparing deflated PCG for different preconditioners on GPU.

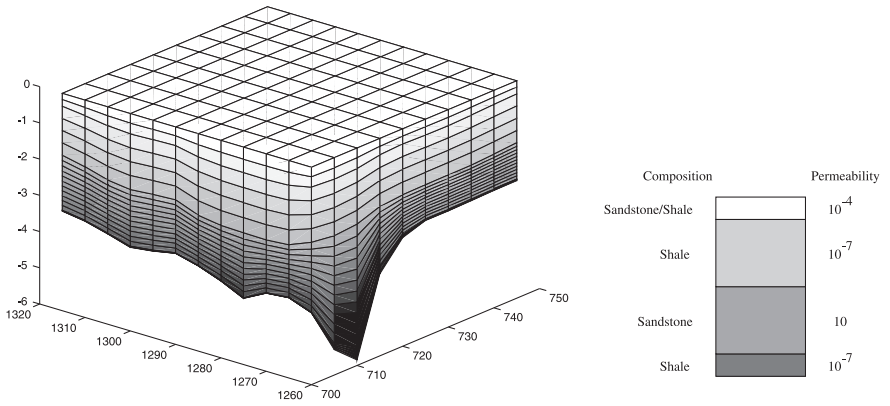
Preconditioner	Iterations	Setup time	Solve time	Total time
MCSGS	265	7.41	6.69	14.1
FSAI	221	14.32	6.41	20.73
ILU(0)	178	7.24	12.49	19.73
ME-ILU-J	254	7.37	6.06	13.43
ME-ILU-SGS	252	7.22	6.28	13.5
ME-ILU-ILU(0,1)	**139**	8.67	**5.03**	13.7
ILU(0,1)	256	7.41	6.12	13.53
DIAGONAL	528	**6.61**	9.75	16.36
TNS	213	6.8	6.3	**13.1**

Layered problem. The number of unknowns is $63 \times 64 \times 64 \times 15$.

here that the setup times on the GPU for the DPCG implementation are very high. This is because of the time it takes for doing two sparse matrix–matrix multiplications ($A \times Z$ and $Z^T \times AZ$), which are most time consuming. However, even though on first sight this might seem to limit the usability of this method, one must pay attention to the iteration times, which are 65% of the CPU times. In the problems where setup is carried out only once, for example, problems coming from the study of seismics of the earths' interior that the coefficient matrix and the other associated matrices are made only once, the GPU implementations can give a significant saving in execution time.

5.2.2. Problem from the oil industry with irregular geometry. In this section, we present results for a problem of porous media flows that has its origins in the oil industry (see Figure 3 and refer [2] for details). The total number of unknowns in this problem is 146,520. It contains nine layers with varying contrasts. The topmost layer has permeability 10^{-4} followed by layers with permeabilities alternating between 10^{-7} and 1. It is defined on an irregular geometry. The stopping criteria are as

DOI: 10.1002/fld



(a) Geometry (b) Legend

Figure 3. Unstructured problem from oil industry.

Table X. PCG (with ILU(0) preconditioner) on CPU and GPU.

Platform	Iterations	Setup time	Solve time	Total time
CPU	231	**0.04**	2.76	**2.8**
GPU	232	0.1	**2.29**	**2.39**

Problem from the oil industry.

Table XI. Deflated PCG (with ILU(0) preconditioner) on CPU and GPU.

Platform	Iterations	Setup time	Solve time	Total time
CPU	104	**0.06**	1.29	**1.35**
GPU	100	0.55	**1.08**	**1.63**

Problem from the oil industry.

mentioned in Section 4.2, and the tolerance is kept at 10^{-6}. We only present results for ILU(0)-based preconditioner in this section as for all other preconditioners, the DPCG and the PCG method do not converge.

In Tables X and XI, we see that for this problem, deflation provides an advantage in total times and also in the number of iterations. The improvement is almost twice. The setup times on the GPU for DPCG implementation are quite high (because of sparse matrix–matrix multiplications) in comparison with the CPU version and also compared with both versions of PCG implementation in Table X.

Ordering of elements in matrices and their favorability for ILU(0) preconditioner. For the problem from the oil industry, only (D)PCG with ILU(0) preconditioner converges. The matrix for this problem uses a numbering scheme that favors ILU(0). Specifically, the elements are stored in the matrix layer by layer. The application of the ILU(0) preconditioner does not involve any pre-processing (coloring and renumbering) to extract parallelism. However, for the other first-level preconditioners, a reordering is carried out, which leads to a worse performance and even stagnation of the convergence.

To validate this information, we conducted an experiment. We changed the ordering of the coefficient matrix. We used maximal independent set ordering (available in PARALUTION) to the coefficient matrix and then used CG with ILU(0) preconditioning to solve the system. The result was that the convergence of the PCG method was as delayed as it was for other preconditioners

DOI: 10.1002/fld

with the same coefficient matrix. This led us to conclude that the ordering of the layers based on the physical description of the problem is of much importance for the application of PCG (or DPCG) method with ILU(0) and the fact that other preconditioners use sophisticated techniques to extract parallelization also comes in the way of convergence.

6. CONCLUSIONS

We have presented a survey about various preconditioning techniques that one can use in the DPCG method to solve ill-conditioned linear systems arising from two different flow problems. Our special focus is to compare different methods for constructing and applying the preconditioner on GPU devices. We have considered several LU-type, sparse approximate inverse-type, and TNS-based preconditioning-type preconditioners in conjunction with deflation for this particular problem.

Through our results, we can conclude that the combination of a simple preconditioner with deflation can prove to be a computationally efficient choice in order to accelerate the convergence of an ill-conditioned problem.

Moreover, the reduced iteration times on the GPU could be very useful in situations where multiple right-hand sides must be solved with a given matrix or a stationary problem (e.g., from the domain of seismics) where setup that is essentially carried out once can benefit from our implementations.

APPENDIX A: EXPERIMENTS WITH VARYING GRID SIZES AND DENSITY RATIOS

In this appendix, we examine how the number of iterations for the different methods depends on the mesh size and on jumps in the coefficients. To this end, we present numerical experiments for the layered problem presented in Section 5.2.1 (Figure 2). We also compare the performance of the ILU-based preconditioners with algebraic multigrid for this problem (for varying contrasts and grid sizes).

In Figures A.1 and A.2, we consider two different scenarios. In Figure A.1, we observe how changing the contrasts in the layers of the problem affects the iteration count. In Figure A.2, we show how the iterations required for convergence change with increasing number of unknowns.

The individual bars in Figures A.1 and A.2 show the number of iterations it takes for the DPCG method (implemented within PARALUTION) with a certain kind of first-level preconditioner (as available in the legend). These experiments have been performed on the CPU.

As the contrast for the layers changes from 10^{-3} to 10^{-6}, the iteration count reduces by about 10% for most versions of DPCG method. The grid dimensions that we have used (in Figure A.1) are $32 \times 33 \times 33 \times 15$. The first layer has a contrast of 10^{-4} followed by six alternating sets of layers with contrasts 1 and 10^{-k} where k varies between 3 and 6. We conclude that DPCG is insensitive to the contrasts in layers.

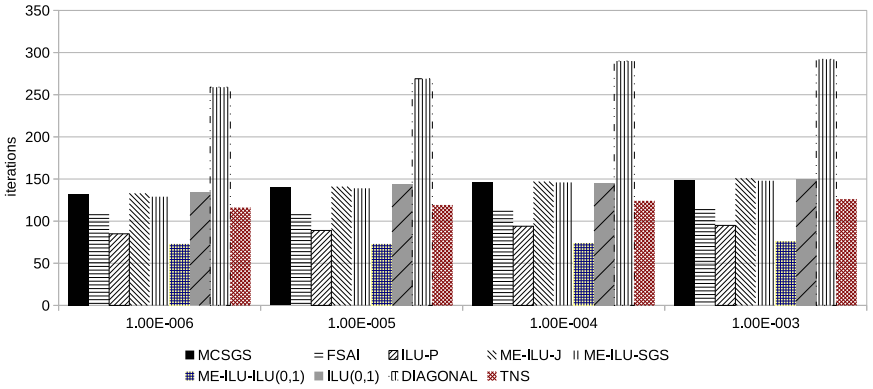


Figure A.1. Layered problem. Fifteen layers. The contrasts vary as 10^{-4}, $[1, 10^{-k}]$ [] repeats six more times where $k = 3, 4, 5, 6$.

DOI: 10.1002/fld

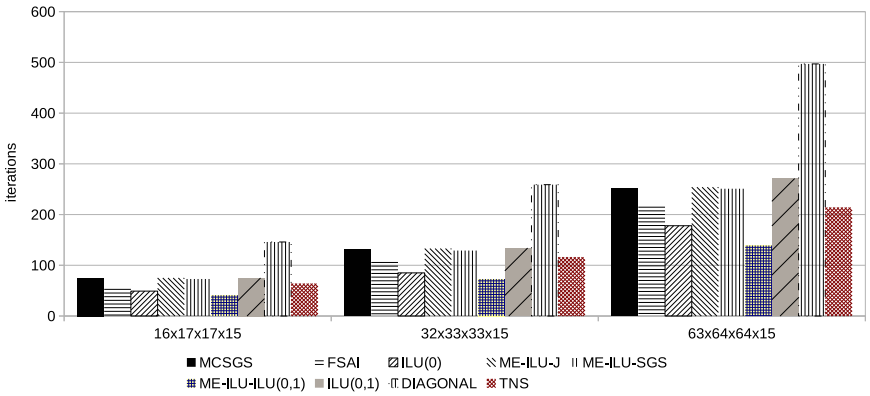


Figure A.2. Layered problem. Fifteen layers. The contrasts vary as $10^{-4}, [1, 10^{-6}]$ [] repeats six more times. Three grid sizes are considered.

Table A.1. CG-AMG results.

	Contrasts			
	10^{-6}	10^{-5}	10^{-4}	10^{-3}
Iterations	23	23	23	28

Layered problem. Fifteen layers. The contrasts vary as $10^{-4}, [1, 10^{-k}]$ [] repeats six more times where $k = 3, 4, 5, 6$.

Table A.2. CG-AMG results.

	Grid sizes		
	$16 \times 17 \times 17 \times 15$	$32 \times 33 \times 33 \times 15$	$63 \times 64 \times 64 \times 15$
Iterations	10	23	21
		CPU	
Setup time	0.171	0.86	7.27
Solve time	1.06	1.63	9.09
Total time	1.231	2.49	16.37
		GPU	
Setup time	0.171	1.07	9.64
Solve time	8.43	0.87	10.57
Total time	8.61	1.94	20.21

Layered problem. Fifteen layers. The contrasts vary as $10^{-4}, [1, 10^{-6}]$ [] repeats six more times. Three grid sizes are considered.

In Figure A.2, we observe that ILU-type preconditioners require twice the number of iterations if grid sizes are dubbed.

Corresponding to Figures A.1 and A.2, we also have results (Tables A.1 and A.2) for the implementation of conjugate gradient method within PARALUTION along with algebraic multigrid (AMG) as a preconditioner.

As can be seen in Tables A.1 and A.2, the number of iterations for AMG is almost constant when contrasts are varied and for increasing grid sizes, the iterations become constant for larger grid sizes.

The CG-AMG method provides 20% improvement in time for the layered problem with grid size $63 \times 64 \times 64 \times 15$ when compared (Table A.2) with the fastest DPCG method on the CPU for the same problem (Table VIII). On the other hand, on the GPU (Table A.2), the advantage of the DPCG method can be clearly observed. For the grid size $63 \times 64 \times 64 \times 15$, the fastest DPCG method (Table IX) is 1.5× better as compared with CG with AMG preconditioner.

DOI: 10.1002/fld

APPENDIX B: EXPERIMENTS WITH CG-AMG FOR PROBLEM FROM OIL INDUSTRY

In this appendix, we present the experiments with the CG-AMG variant introduced for the problem from the oil industry (Section 5.2.2; Figure 3). The CG-AMG method converges for this problem. However, it takes many iterations and takes much longer to converge. It provides no advantage over the DPCG method with ILU(0) preconditioning (Table B.1). So, for this problem, the DPCG method with ILU(0) is the best option.

Table B.1. CG with AMG preconditioner on CPU and GPU.

Platform	Iterations	Setup time	Solve time	Total time
CPU	1994	0.349	43.2	**43.549**
GPU	1994	0.347	60.05	60.397

Problem from the oil industry.

ACKNOWLEDGEMENTS

This work has been supported by the Linnaeus center of excellence UPMARC, Uppsala Programming for Multicore Architectures Research Center. The authors would also like to thank Guus Segal at the Delft Institute of Applied Mathematics for providing the matrices for porous media flow problems and Kees Verstoep at DAS-4 cluster Vrije University site for valuable help and suggestions for running the experiments on DAS-4. We would also like to extend our appreciation and thankfulness to Nico Trost from PARALUTION who helped us in double-checking the experiments added to this revision.

REFERENCES

1. van der Pijl SP, Segal A, Vuik C, Wesseling P. A mass-conserving level-set method for modelling of multi-phase flows. *International Journal for Numerical Methods in Fluids* 2005; **47**(4):339–361.
2. Vuik C, Segal A, Meijerink JA, Wijma GT. The construction of projection vectors for a deflated ICCG method applied to problems with extreme contrasts in the coefficients. *Journal of Computational Physics* 2001; **172**:426–450.
3. Lukarski D. Parallel sparse linear algebra for multi-core and many-core platforms – parallel solvers and preconditioners. *Ph.D. Thesis*, Karlsruhe Institute of Technology, Karlsruhe, Germany, 2012. (Available from: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000026568.) accessed on date -January 2012.
4. Gupta R, van Gijzen MB, Vuik C. Efficient two-level preconditioned conjugate gradient method on the GPU. *Proceedings of VECPAR 2012*, Springer LNCS, Berlin Heidelberg, 2013; 36–49.
5. Lukarski D. PARALUTION project. (Available from: http://www.paralution.com/.) January 2012.
6. Tang J, Vuik C. New variants of deflation techniques for pressure correction in bubbly flow problems. *Journal of Numerical Analysis, Industrial and Applied Mathematics* 2007; **2**:227–249.
7. Tang J, Vuik C. Efficient deflation methods applied to 3-D bubbly flow problems. *Electronic Transactions on Numerical Analysis* 2007; **26**:330–349.
8. Tang J. Two-level preconditioned conjugate gradient methods with applications to bubbly flow problems. *Ph.D. Thesis*, Delft University of Technology, The Netherlands, 2008.
9. Gupta R, van Gijzen MB, Vuik C. 3D bubbly flow simulation on the GPU – iterative solution of a linear system using sub-domain and level-set deflation. *Proceedings of PDP 2013*, IEEE CPS, Belfast, 2013; 359–366.
10. Jönsthövel T, van Gijzen MB, MacLachlan S, Vuik C, Scarpas A. Comparison of the deflated preconditioned conjugate gradient method and algebraic multigrid for composite materials. *Computational Mechanics* 2011; **50**: 1–13.
11. 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 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, PDP '10. IEEE Computer Society: Washington, DC, USA, 2010; 583–592.
12. Rudi H, Koko J. Parallel preconditioned conjugate gradient algorithm on GPU. *Journal of Computational and Applied Mathematics* 2012; **236**(15):3584 –3590.
13. Bell N, Garland M. Implementing sparse matrix-vector multiplication on throughput-oriented processors. *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, Supercomputing, ACM, New York, NY, USA, 2009; 18:1–18:11.
14. Wang X, Bramley R, Gallivan KA. A necessary and sufficient symbolic condition for the existence of incomplete Cholesky factorization. *Technical Report*, Indiana university: Bloomington, Indiana, 1995.
15. Saad Y. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2003.

DOI: 10.1002/fld

16. Grote MJ, Huckle T. Parallel preconditioning with sparse approximate inverses. *SIAM Journal of Scientific Computing* May 1997; **18**(3):838–853. DOI: 10.1137/S1064827594276552.
17. Kolotilina LY, Yeremin AY. Factorized sparse approximate inverse preconditionings I: theory. *SIAM Journal on Matrix Analysis and Applications* 1993; **14**:45–58.
18. Benzi M, Tůma M. A sparse approximate inverse preconditioner for nonsymmetric linear systems. *SIAM Journal of Scientific Computing* 1998; **19**(3):1135–1149.
19. van der Vorst H. *Iterative Krylov Methods for Large Linear Systems*. Cambridge University Press: Cambridge, 2003.