GPU acceleration of preconditioned solvers for ill-conditioned linear systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties, in het openbaar te verdedigen op
maandag 9 november 2015 om 12:30 uur

door

ROHIT GUPTA
Master of Science in Computer Engineering, Technische Universitiet Delft

geboren te Kanpur, U.P., India

Dit proefschrift is goedgekeurd door de
promotor : Prof.dr.ir. C. Vuik
copromotor : Dr.ir. Martin B. van Gijzen

Samenstelling promotiecommissie:

Rector Magnificus,	voorzitter
Prof.dr.ir. Kees Vuik	Technische Universiteit Delft, promotor
Dr.ir. Martin B. van Gijzen,	Technische Universiteit Delft, copromotor

Onafhankelijke leden

Prof.dr.ir. A.W. Heemink,	Technische Universiteit Delft, The Netherlands
Prof. B.J. Boersma,	Technische Universiteit Delft, The Netherlands
Dr. D. Lukarski	PARALUTION Labs UG (haftungsbeschrnkt) & Co. KG, Germany
Dr. Edmond Chow,	Georgia Institute of Technology, Atlanta, United States
Prof.dr.ir. K.J. Batenburg,	University of Antwerp, Belgium

GPU acceleration of preconditioned solvers for ill-conditioned linear systems.
Dissertation at Delft University of Technology.

TU Delft

The work described in this thesis was financially supported by the Delft Institute of Applied Mathematics.

Cover page and invitation design:
'Bubbly Flow' as imagined by Alyona Vyacheslavovna Mezentseva (age 11)
ISBN 978-94-6186-526-7

Summary

GPU acceleration of preconditioned linear solvers for ill-conditioned linear systems

Rohit Gupta

In this work we study the implementations of deflation and preconditioning techniques for solving ill-conditioned linear systems using iterative methods. Solving such systems can be a time-consuming process because of the jumps in the coefficients due to large difference in material properties. We have developed implementations of the iterative methods with these preconditioning techniques on the GPU and multi-core CPUs in order to significantly reduce the computing time. The problems we have chosen have a symmetric and positive definite coefficient matrix. We have further extended these implementations for scalability on clusters of GPUs and multi-core CPUs.

We outline the challenges involved in making a robust preconditioned solver that is suitable for scaling in a parallel environment. To start with, we experimented with simple techniques to establish the feasibility of implementing specialized preconditioning schemes (deflation) for specific problems (bubbly flow). We provide an analysis for the choices we make for implementing certain parts (e.g. solution of inner system in deflation) of these operations and tune the data structures keeping in mind the hardware capabilities. We improve our solvers by refining the choices we make for the application of these techniques (Neumann Preconditioning and better deflation vectors). For different options available we compare the effect when varying problem parameters (e.g. number of bubbles and deflation vectors). After testing our methods on standalone machines with multi-core CPUs and a single GPU we make a parallel

implementation using MPI. We explore different data divisions in order to establish the effect of communication and choose the more scalable approach. In order to make our results more comprehensive we also test the implementations on two different clusters. We show the results for two versions of our code: one for multi-core CPUs and another one for multiple GPUs per node. Our methods show strong scaling behavior.

To further evaluate the use of deflation combined with a simple preconditioning technique we test our implementation of the iterative method for solving linear systems from porous media flow problems. We use a library with many different kinds of preconditioners on parallel platforms. We test implementations with and without deflation. Our results show good performance of the iterative methods we use in combination with deflation and preconditioning for a number of problems.

Through our experiments we bring about the effectiveness of deflation for implementation on parallel platforms and extend its applicability to problems from different domains.

Samenvatting

GPU versnelling van gepreconditioneerde solvers voor slecht geconditioneerde lineaire stelsels

Rohit Gupta

In dit werk bestuderen we implementaties van deflatie- en preconditioneringstechnieken voor het oplossen van slecht geconditioneerde lineaire stelsels met behulp van iteratieve methoden. Het oplossen van dergelijke stelsels kan een tijdrovend proces zijn vanwege de sprongen in de cofficinten als gevolg van grote verschillen in materiaaleigenschappen. We hebben implementaties van iteratieve methoden met deze preconditioneringstechnieken voor de GPU en multi-core CPUs ontwikkeld om de rekentijd aanzienlijk te verminderen. De problemen die we gekozen hebben, hebben een symmetrische en positief definiete cofficintmatrix. Verder hebben we deze implementaties uitgebreid voor schaalbaarheid op clusters van GPU's en multi-core CPU's.

We beschrijven de uitdagingen van het maken van een robuuste gepreconditioneerde solver die schaalbaar is in een parallelle omgeving. Om te beginnen hebben we gexperimenteerd met eenvoudige technieken om de haalbaarheid van de implementatie van gespecialiseerde preconditioneringsmethoden (deflatie) voor specifieke problemen (bubbly flow) te bepalen. We bieden een analyse van de keuzes die we maken voor de implementatie van bepaalde onderdelen (bijv. de oplossing van de binneniteratie in deflatie) van deze operaties en passen de datastructuren aan, rekening houdend met de hardwaremogelijkheden. We verbeteren onze solvers door het verfijnen van de keuzes die we maken voor de toepassing van deze technieken (Neumann preconditionering en betere deflatievectors). Voor verschillende opties die beschikbaar zijn vergeli-

jken we het effect door probleemparameters te variren (bijv. het aantal bellen en deflatievectors). Na het testen van onze methoden op autonome machines met multi-core CPU's en n GPU maken we een parallelle implementatie met behulp van MPI. We onderzoeken verschillende dataopdelingen om het effect van de communicatie te onderzoeken en kiezen de meer schaalbare aanpak. Om onze resultaten omvattender te maken testen we de implementaties ook op twee verschillende clusters. We tonen de resultaten voor twee versies van onze code: een voor multi-core CPU's en een andere voor meerdere GPU's per node. Onze verbeterde implementaties tonen een sterke schaalbaarheid.

Om het gebruik van deflatie gecombineerd met een eenvoudige preconditioneringstechniek verder te evalueren, testen we onze implementatie van de iteratieve methode voor het oplossen van lineaire stelsels van poreuze media stromingsproblemen. We maken gebruik van veel verschillende soorten preconditioners op parallelle platformen. We testen implementaties met en zonder deflatie. Onze resultaten tonen goede prestaties van de iteratieve methoden die we gebruiken in combinatie met deflatie en preconditionering voor een aantal problemen.

Door middel van onze experimenten verhogen we de effectiviteit van deflatie voor de uitvoering op parallelle platformen en breiden we de toepasbaarheid uit naar problemen in andere domeinen.

Acknowledgments

Throughout my time in Delft, first during my masters and then while working on this dissertation I came across so many kind people, without acknowledging whom this work will not be complete. First and foremost I want to thank Kees Vuik for trusting me when I only had fluffy excitement about, parallel computing in general and wanted to do something with it. The masters thesis that I completed under his guidance was a big challenge for me as I chartered into the unknown waters of parallel computing and Applied Mathematics. I came across, collaborated and befriended many people just by working with you, Kees. The funny thing is that these were the fringe benefits of working with you. The primary return is this book. The wealth of knowledge, humility and patience which you practice, and which I have benefited from on countless occasions, leave me thankful for having worked with you. You are the most important reason that Delft feels like home every time I come back there.

I want to express my gratitude to my co-promotor, Martin van Gijzen, for being critical about my work and for carefully going through my excitement and helping me understand tough concepts. It was you who brought me back on course many a times during this research.

This research might not have been possible if Jok Tang would not have done a remarkable research before me on the Deflated Preconditioned Conjugate Gradient Method (DPCG). Not only was your thesis book my reference for extending this method on parallel platforms, but your advice on research and patience with my incessant questions was something I will stay thankful for.

The people I met here at the Numerical Analysis department during my study and research have also taught me many things. I would like to thank Fred for being a strict but friendly teacher. The Finite Elements course that you taught helped me figure out all the ditches and gorges that were present

in my understanding of the subject and in being precise with derivations in general. I want to thank Duncan van der Heul for the most constructive and complete critique of the first presentation I gave about my work. I kept the suggestions you gave to me safe inside my mind and recall them when I present to any audience till date. I also had the chance to take interesting and challenging courses with Domenico, Jennifer and Henk which I want to thank them for.

The most interesting and relieving activity that I indulged all these years in research was conversations with my colleagues. Dennis has to be on the top of this list. I must say that our ways of thinking are different (and certainly the taste in music). I seemed to have had so many 'aha' moments just explaining the issue at hand in my work to you on the whiteboard. Your rigor (and apparent ease) at solving those ugly equations (step by step not missing even one minor detail) in finite elements and volumes is something I have always hoped I could cultivate.

I wish to thank my colleagues Daniel, Martijn and Serguey with whom I had the chance to share the office and have great conversations during my time at the department.

I had the chance to be with wonderful colleagues like Fahim, Abdul, Joost, Guido, Yue, Elwin, Domenico, Fei, Reinaldo and Manuel. Our conversations on various topics whether on the whiteboard, over lunches in the EWI cafe or just in the corridors of our department have been always refreshing. A special thanks to Fahim and Abdul for organizing the multi-cultural dinner which I enjoyed a lot.

I must acknowledge Kees Lemmens for being an excellent coach and a very interesting colleague who has shared many a philosophies with me and also his passion for flying airplanes. Kees, I hope you forgive me for never coming with you to the airstrip again.

Thank you Thea for helping me with the part of this thesis that required struggling with my limited knowledge of dutch. I would also like to thank Xiaozhou for helping me with ideas on printing this book. I also wish to thank Shiming Xu for sharing his knowledge on performance analysis of GPU implementations.

At a professional level I was also lucky to get support from some very good system administrators at the clusters I misused and abused during my research. Kees Verstoep(DAS-4) and Thomas Geenen (SURFSARA) are some of the most patient people who helped me complete my work and on many instances helped me make the best use of the resources available at their discretion.

I want to thank Deborah and Cindy for being an excellent and helpful secretaries who helped get administrative issues resolved efficiently.

During my research I had a chance to collaborate with people within the

numerical analysis group, outside of it, within Netherlands and also outside of it. These opportunities have left my work enriched and added value to my experience as a researcher and an engineer.

I was lucky to collaborate with Paulien van Slingerland, Guus Segal, Duncan and Tom Jönsthövel on various occasions during the course of my research and I want to thank all of you for giving me a chance to use your work and for the fruitful collaborations.

I was lucky to have worked with Jan thorbecke from CRAY and Jonathan Cohen, Joe Eaton and Maxim Naumov from NVIDIA. I want to thank you for the collaboration in writing the software that you were developing.

Johan van de Koppel shares my appreciation amongst the list of people who trusted my fluffy understanding of parallel computing and applied mathematics. Thank you for giving me the chance to work with you at the Netherlands Institute of Ecology. The work I was able to complete with the help of excellent researchers like you at the institute has given me a breadth of experience which I couldn't have built otherwise.

My time in Delft and in the Netherlands would have been lifeless and uneventful, had I not gotten the chance to befriend some of the best people I know today. I would start with Tabish who just like me arrived in Delft for Masters and we shared student housing. There are many things which I have learned from you and that includes cooking and having a good taste in movies. I want to thank you for being around during all this time as a helpful friend whether it was in studies, finances or advice. Not to mention that you also are an excellent person to talk to about all the things that crossed my mind about life, worries of the future and other vagaries. I also want to thank Pranjal for his jovial companionship during our time in Delft and for making many of us smile through during the hard times.

It seems during my time here somebody high above the clouds was choosing roommates for me: those who cook excellent food and those who are kind-hearted and extremely good at making interesting conversations. Tabish was the first and then Aditya Thallam Thattai. Thanks Aditya for being a great friend all this time.

I met people who really make me believe that language and nationality is secondary when it comes to friendship. Behrouz Raftarnagabi, Jing Zhao and Alexandro Mancusi I thank you for trusting me and making me feel comfortable in your presence. Thank you for sharing your thoughts, fears and ideas with me.

My list of friends will not be complete if I do not mention the three lovely ladies that taught me how to make friends and keep them steadfast by my side. I was lucky to get to know Alessia Moneta, Juliette Ly and Yayu La Nafie whom I met during my first stint at the NIOO [now NIOZ] in Yerseke. You are few of the most beautiful women I have come to know. Thank you

for the numerous gifts, cakes, chocolates that have added to my inches, not to mention the hugs, kisses and wishes that are very dear to me.

A host of people made Delft more homely and comfortable than it initially was when I arrived here a good 7 years back. Rajat, Akhil, Purvil, Harsha, Bhaskar, Ramavatar, Rahul, Tiggy, Krishna, and Choka, thank you very much for the most amazing dinners and times together when we celebrated birthdays and festivals that made me feel at home in the Netherlands.

When I left my family for my dreams of higher education, they stood by my decision and supported me with love and encouragement. I cannot be who I am without the blessings of my father, mother and my sister. I want to thank my mother for making me feel comfortable with my trifling struggles to get good grades in the initial few months of my masters. I have learnt so much at giving from you. I consider that as one of the most beautiful things I have come to practice in life.

No man can claim to be the man he is without recognizing and marveling at what a woman does for him. As a mother and as a wife. I want to take this opportunity to express my gratitude to my wife for believing in me and for giving me the best gift I have ever got in the form of our son, Fyodor.

Rohit Gupta
Breda, October 2015.

CONTENTS

CHAPTER 1

Introduction

1.1 Background

The solution of very large linear system of equations can take the bulk of computation time of a numerical simulation. The linear system is denoted by

$$Ax = b, \tag{1.1}$$

where $A \in \mathbb{R}^{N \times N}$ is the coefficient matrix, $b \in \mathbb{R}^N$ is the right-hand side and $x \in \mathbb{R}^N$ is the vector of unknowns.

The coefficient matrix A is often large and sparse. It could also have some special characteristics like positive-definiteness and symmetry. For linear systems having such coefficient matrix, the iterative method of conjugate gradients (CG) is most suitable. If the coefficient matrix A is very ill-conditioned then the convergence of CG method is slow. Ill-conditioning refers to the large condition number[1], which is the ratio of the largest to the smallest eigenvalue of the matrix A, when A is Symmetric Positive Definite (SPD).

In order to accelerate convergence we have to combine the CG method with preconditioning. This changes the system (1.1) to

$$M^{-1}Ax = M^{-1}b, \tag{1.2}$$

where matrix M is the preconditioner. M is SPD and (1.3) holds for M.

$$\kappa(M^{-1}A) << \kappa(A), \tag{1.3}$$

[1]For a general matrix A the condition number is defined as $\kappa(A) = \| A \| \| A^{-1} \|$

where κ is the condition number. The preconditioner must be computationally cheap to store and to apply.

In this work we precondition A on two levels. At first we precondition A, transforming the linear system in (1.1) to (1.2). The spectrum of the preconditioned matrix $M^{-1}A$ can still have small eigenvalues and in order to remove them from the spectrum we apply another level of preconditioning which is called deflation. We implement this deflated preconditioned CG (DPCG) method on a Graphical Processing Unit (GPU). We minimize computation time by exposing the fine-grain parallelism in CG, preconditioning and deflation steps. We also explore options to improve the deflation method and test our method for different applications. In order to test the scalability of our method we perform tests with multiple GPUs and CPUs. We report the challenges we face in such an implementation with respect to the balance between communication and computation.

1.2 Graphical Processing Unit (GPU) computing

1.2.1 GPU architecture

GPUs were originally designed to deliver very high-fidelity video/gaming experience on desktop machines. They have a large array of simple processors that can be divided to *paint* different parts of the screen and can do that in parallel. This is different from the CPU which has few cores and a lot of special control logic which takes care of features like branch prediction, out-of-order execution, score-boarding and such advanced techniques in order for the CPU to deliver a reliable desktop experience. In Figure 1.1 we see a general

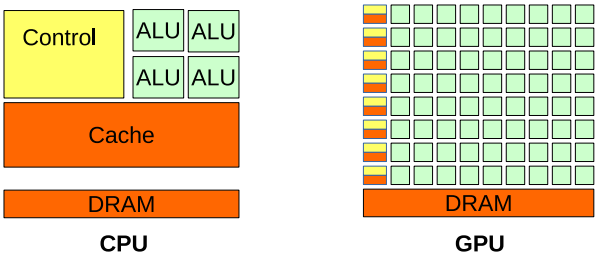


Figure 1.1: Comparing CPU and GPU architectures. (photos courtesy NVIDIA CUDA programming guide [17])

example of the CPU and GPU architecture. The CPU has fewer (Arithmetic and Logical Units) ALUs and a large chunk of the silicon is devoted to control. The ALUs share a fast cache memory and a main memory which is one level higher than the cache. In comparison, the GPU has a large number of ALUs.

They are grouped together and each group shares a relatively small control and cache and fixed number of registers. These units or simple processing elements can work independently on different data using the same set of instructions. In other words, they can process large amounts of data in parallel. Large transfers of data back and forth from the DRAM are supported by the wide bandwidth of the GPU.

Over the past decade the scientific computing community has seen a number of different architectures emerge which could be used to accelerate applications. Some of the prominent names are the CELL processor [2], the GPUs from NVIDIA [3], GPUs from ATI [4], Intel Xeon Phi processors [5] and ClearSpeed processors [6]. Amongst these the GPUs from NVIDIA have been used extensively and have evolved (Figure 1.2) (and are being improved[7]) rapidly to cater to the demands of scientific applications. In Figure 1.2 we see how the normalized SGEMM (single precision general matrix-matrix product) per watt capability has improved and is predicted to improve for GPUs from NVIDIA.

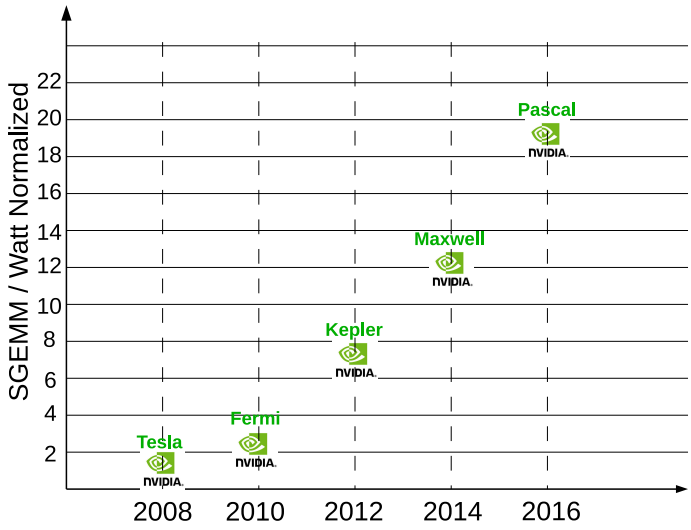


Figure 1.2: NVIDIA roadmap

For these reasons and because the GPUs from NVIDIA have an active community of developers, support and libraries we have chosen GPUs from

[2] http://researcher.watson.ibm.com/researcher/view_group.php?id=2649

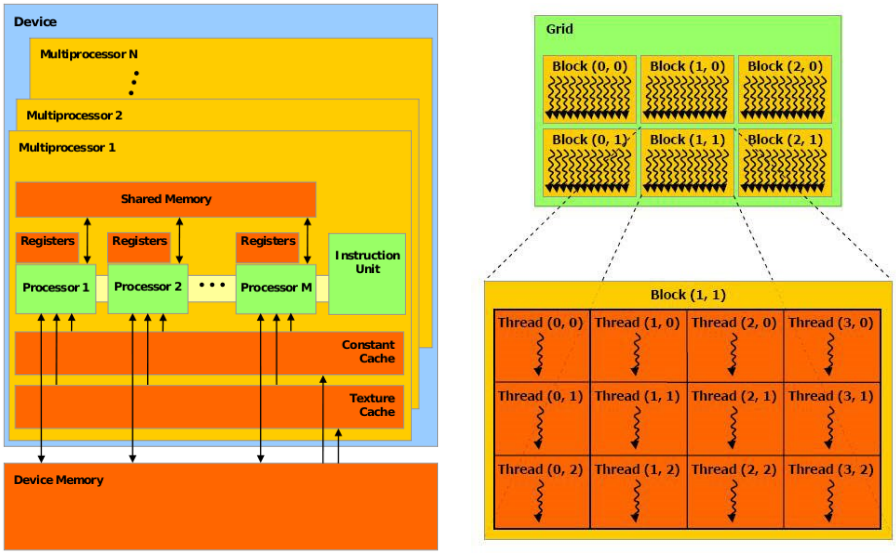
[3] http://www.nvidia.com/object/what-is-gpu-computing.html

[4] http://www.amd.com/en-us/products/graphics

[5] http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html

[6] http://www.clearspeed.com/products/csx700.php

[7] http://www.anandtech.com/show/7900/nvidia-updates-gpu-\roadmap-unveils-pascal-architecture-for-2016



(a) CUDA abstraction of a GPU (b) CUDA threads, blocks and grids

Figure 1.3: CUDA programming model. (photos courtesy NVIDIA CUDA programming guide [17])

NVIDIA for our implementations. NVIDIA GPUs have evolved from being a small co-processor inside a desktop to being able to deliver desktop supercomputing at affordable price and energy budget.

1.2.2 Compute Unified Device Architecture (CUDA)

NVIDIA provides an extension to the C/Fortran programming language in order to enable its GPUs to execute scientific computing codes. The Compute Unified Device Architecture or CUDA programming platform allows for using GPU much like an SIMD[33] (Single Instruction Multiple Data) processor (or Single Instruction multiple threads, SIMT as per NVIDIA documentation). In each GPU (refer Figure 1.3(a)) there are a number of so called Streaming Multiprocessors or SMs. Each SM has a set of processor cores which have a fixed amount of registers, shared memory and caches. These are simple processors capable of doing floating point calculations. The responsibility is handed over to the programmer to write their application in such a way that all cores on all SMs can be engaged in useful computation and the device is used to its full potential. Each core on the SM works on a *kernel*. A kernel is a sequence of commands that execute on a set of data. Each kernel can have its own data or can share it amongst the same SM. A kernel can be thought of as a simple for loop that executes a matrix addition code. On GPU devices

it is possible to launch thousands of these kernels so that the CUDA cores can compute with all the data in parallel using the kernel.

The kernels themselves are called by threads and the threads on the GPU are arranged in grids (refer Figure 1.3(b)). The grids are sub-divided into blocks. Each block contains a set of threads which are launched on the CUDA cores in each SM. The grid/block structure is defined by the programmer when launching a kernel.

To utilize the *device* or the GPU to its maximum capacity is often not straightforward as there are only a limited number of registers that a kernel can use. It could also be the case that there is not enough work for the GPU due to dependencies in data or control flow. It is also possible that there are unavoidable branches in the code or the data access pattern does not allow maximal bandwidth usage. The programmers must be able to maximize performance of their codes within these constraints. In newer architectures (e.g. from NVIDIA) some of these constraints have been relaxed but still care must be taken when writing kernels to maximize the device utilization.

1.2.3 Performance pointers for Scientific Computing with GPUs

Prior to GPUs, parallel machines with Single Instruction Multiple Data (SIMD), vector architectures[8] and extended instruction sets[9] were used to accelerate computing the solution of large systems. GPUs present us with a new paradigm of computation. Notably, there are two things which must be taken into account when designing an algorithm to extract the best performance out of a parallel platform and specifically for the GPU:

1. Fine-grain Parallelism;

2. High Flop/Byte Ratio.

For an algorithm to exhibit fine-grain parallelism it must be possible to re-write it in such a way that there are a lot of individual units of work. These must be completely independent from each other. For example, consider matrix addition. The core *kernel* of this operation looks like

$$C[i][j] = A[i][j] + B[i][j] \tag{1.4}$$

where A, B are matrices whose sum is calculated and stored in C. The pair i, j denote the specific element of each matrix. In this operation each sum can be calculated separately without any relation to any other sum. No information is being shared and there are no overlaps between any two individual sum

[8] http://www.freescale.com/webapp/sps/site/homepage.jsp?code=PCPPCP
[9] http://cache.freescale.com/files/32bit/doc/ref_manual/ALTIVECPEM.pdf

operations. This means that if we can launch all these individual units of computation in parallel then we are only limited by the hardware required for such a parallel execution.

On the GPU one can launch thousands of threads, so such operations are very well suited for the GPU. On modern GPUs[10] the peak number of calculations per second exceeds a teraFLOP (=10^{12} FLOPs), where a FLOP means a floating point operation.

The second important characteristic that must be exposed in an algorithm is a high ratio of computation done compared to the memory transfers. We discuss the example of Sparse Matrix Vector Multiplication (SpMV). A sparse matrix has a significantly smaller number of non-zeros per row than the number of columns in the matrix. Let us assume that there are only 5 elements per row. Now in order to calculate the product of such a matrix with the vector one has to read 5 elements per row of the matrix and 5 elements of the vector. Followed by this read from memory one has to compute 5 products and 4 sums and one write back to the (at the corresponding row index) memory of the final product. Summing the individual operations we get 11 memory operations and 9 floating point operations. These numbers assume the addition and multiplication operation to have the same cost (in terms of clock cycles) and that the matrix and vector are stored in single/double precision format.

This leads to a ratio of $\frac{9}{11}$ for computation versus communication, which is less than one. Such a behavior is common to algorithms that use a lot of bandwidth and do comparatively less computation. They are also called bandwidth-bound algorithms. Such algorithms are most challenging for GPU implementation. However, by arranging the data in such a way that bandwidth usage can be maximized (coalescing) it is possible to improve performance.

A very intuitive way of looking at such a relation between communication and computation is the FLOP/Byte ratio. It is useful to know this number to understand/predict the speedup or performance available on a particular machine. The Roofline Model [80] (Figure 1.4) shows how for different architectures one can have differing performance in number of computations per second when compared to their FLOP/Byte ratio.

The model predicts that if you have a FLOP/Byte ratio less than one it is hard to reach the peak FLOP performance that the device can offer. Although the authors in [80] do not say much about the current GPUs (or the ones used in this thesis), it is equally applicable to them.

We can compare the example of SpMV to one that has a FLOP/Byte ratio higher than one. Dense matrix-matrix multiplication is one such example. A dense matrix has non-zeros equal to the product of number of rows and

[10] http://www.nvidia.com/content/tesla/pdf/NVIDIA-Tesla-Kepler-Family-Datasheet.pdf

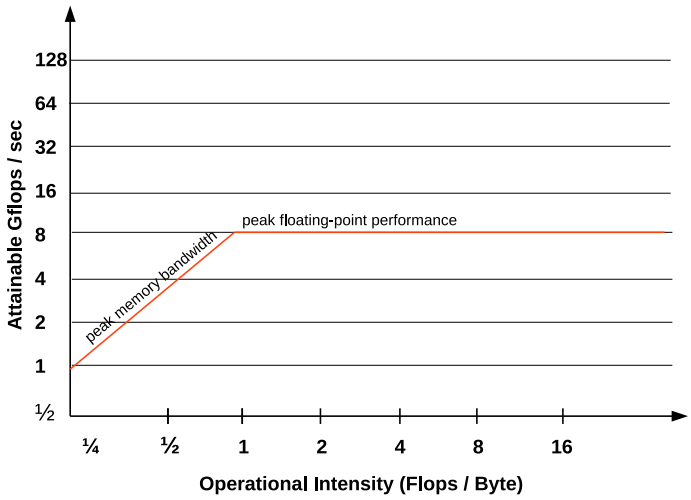


Figure 1.4: Roofline model for AMD Opteron X2 from [80].

columns of the matrix. For two matrices of dimensions $N \times N$. We have $O(N^2)$ memory transfers ($2N^2$ read and N^2 write) and $O(N^3)$ FLOPs. This gives a FLOP/Byte ratio of $O(N)$ which directly translates to a better performance according to the roofline model.

It must be noted in this example that the denominator (memory transfers) are taken to be the order of the problem size and it is considered as the maximum memory bandwidth available. For dense matrix operations that can be the case for sufficiently large problems.

1.3 Applications

The accelerated solution of a linear system can be exploited for problems arising from very different applications. Our research is mainly concerned with the solution of problems emanating from elliptic partial differential equations with large contrasts in material properties. In this section we enumerate some of the applications which can benefit from our implementations.

1.3.1 Bubbly flow

Flow phenomena occur in different physical processes. Some of these processes are crucial to understand for a wide variety of industries. A clear understanding of these processes allow for a better design and can be instrumental in improving the efficiency, robustness and even safety of certain equipment.

An example is bubbly flow, where the bubbles of one medium travel within another medium. One can picture these to be air bubbles in water or water

bubbles in oil. This sort of composition is common in extraction mechanisms for crude oil production.The densities of these two mediums vary significantly with ratio of the densities in excess of 10^3.

Figure 1.5: Bubbly flow

To model two-phase flow, the incompressible Navier Stokes equations are used. These equations define the continuous change in velocity, pressure, momentum and mass of the fluid whose flow is to be monitored. However, in a computer simulation we must work with discrete quantities so the equations are discretized using a suitable scheme.

In our research we use the Mass-Conserving Level-Set (MCLS) method (refer [68]) for solving the two-phase Navier Stokes equations. While solving the discretized system of equations over time using the MCLS method one encounters a linear system (1.1). It arises from the discretization of the pressure correction equation,

$$-\nabla.(\frac{1}{\rho(x)}\nabla p(x)) = f(x),\ x \in \Omega \tag{1.5a}$$

$$\frac{\partial}{\partial n}p(x) = g(x),\ x \in \partial\Omega \tag{1.5b}$$

where Ω, p, ρ, x and n denote the computational domain, pressure, density, spatial coordinates, and the unit normal vector to the boundary, $\partial\Omega$, respectively.

Due to the contrasts in densities the matrix A is ill-conditioned. The DPCG method applied to this problem has been previously studied in [62, 63, 64, 65, 61]

1.3.2 Porous media flow

To describe porous media flows one uses the Darcys' Law and the continuity equation. As a part of the simulation of such flows one needs to solve

$$-\nabla.(\sigma(x)\nabla p(x)) = f(x),\ x \in \Omega \tag{1.6}$$

where Ω, p, $\sigma(x)$ denote the computational domain, pressure, and the permeability of the medium through which the flow occurs respectively. Suitable boundary conditions are imposed on Ω. Discretizing (1.6) gives a linear system of equations. These problems can arise while modeling extraction of oil from deep within the earths' surface. Different layers within the earths' crust can have varying densities and flow of the oil through these layers is dependent on their permeability. Modeling this phenomena gives insight that is valuable in deciding where to drill, how much to drill and it can be used to predict the potential yield.

The discretized system from (1.6) has a large contrast owing to different permeabilities for the rigid and soft layers of material in the subsurface. This leads to some very small eigenvalues in the spectrum of A (which is also SPD) and therefore we can use the DPCG method to solve this system. The choice of deflation vectors can be problem dependent and has been studied previously in [76].

1.3.3 Mechanical problems

The authors of [38] consider mechanical problems that are characterized by high contrast in the coefficients. Specifically they consider a problem where two (or more) different materials are involved with varying stiffness.

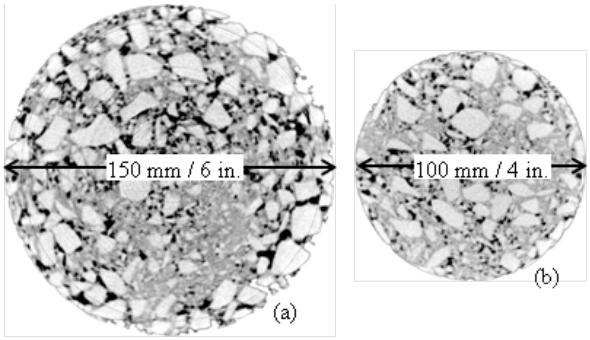


Figure 1.6: Tomography scans of asphalt. Photo courtesy CAPA-3D http://capa-3d.org/CAPA3D/Home.html.

In Figure 1.6 we see the slice of asphalt which is composed of different materials. These scans are used to generate accurate discretizations of the material to be studied. The matrices that arise from the discretization of the equations that describe the mechanical response of these materials are SPD and therefore suited to the CG method. However, they are ill-conditioned and may have small eigenvalues corresponding to the different stiffness (rigid body) modes corresponding to each material. The convergence of CG method can be accelerated in this case by using deflation. The authors of [38] use the rigid body modes deflation or the mathematically equivalent kernel deflation. Using these variants of deflation they manage faster convergence of the iterative method.

1.4 Scope of the thesis

This thesis work deals with acceleration of the DPCG method on the GPU. In the prior work [62], DPCG was proven to be an effective technique for solving linear systems arising from bubbly flow. The linear system solved and studied in [62] uses the Mass Conserving Level-Set Method which appeared as result of the work in [69]. Even though the DPCG method discussed in [62] is better than just one level preconditioning it was not optimized for parallel platform like GPUs or multiple CPUs. In this research we extend [62] by dealing with the challenges of parallel implementation of DPCG on parallel machines (GPUs, multicore CPUs and multi-GPU/CPU clusters). We also explore the application of the DPCG method on other applications. Specifically our research brings forth implementation challenges for the DPCG method on single and multiple GPUs and CPUs. We have developed preconditioning schemes that exhibit fine-grain parallelism and are competitive to parallel variants of the preconditioning schemes discussed in [62]. In our work we use techniques of optimized storage to maximize bandwidth usage on the GPU, overlap communication with computation for multiple GPUs and CPUs and use optimized libraries along with our own computation kernels.

In order to compare our method with different choices of first level preconditioning we use the PARALUTION[11] library. These preconditioning schemes are implemented on the GPU and multi-core CPUs. They are a direct result of the work presented in [49] and later extended into the PARALUTION library. Our work differs from the work presented in [49] since they utilize re-organization of the matrix using multi-coloring to make preconditioners (with fine-grain parallelism) whereas we use a preconditioner that has structural similarity to the coefficient matrix. They focus mainly on one level of preconditioning whereas we couple first level preconditioning with deflation.

[11] http://www.paralution.com

We began our investigations with a combination of a preconditioner with fine-grain parallelism combined with deflation [31] followed by experimenting with deflation vectors [29]. We explore different techniques of deflation [65] and report improvements for their efficient execution on the GPU. We take our implementations a step further with multi-GPU/CPU implementations that were reported in [30].

1.5 Outline of the thesis

The outline of the thesis is as follows:

2. Iterative methods and GPU computing - In this section we review Krylov subspace methods, preconditioning techniques and the use of GPUs for preconditioned iterative methods.

3. Neumann preconditioning based DPCG - We introduce our approach to combine Truncated Neumann Series Preconditioning with deflation in Chapter 3. Experiments with simple problems are presented with analysis to support further research.

4. Improving deflation vectors - In Chapter 4 we discuss improvements made to deflation vectors and their implementation on the GPU.

5. Extending DPCG to multiple GPUs and CPUs - In this Chapter we discuss how our implementation of the DPCG method has been extended to multiple GPUs and CPUs.

6. Comparing DPCG on GPUs and CPUs for different problems -We compare our preconditioning scheme with other preconditioning schemes for two different problems in Chapter (6).

7. Conclusions - We present our conclusions and summarize the key findings of our work.

CHAPTER 2

Iterative methods and GPU computing

Iterative methods are used to solve systems of linear equations by generating approximate solution vectors x_j in every iteration j. The aim of these methods is to come as close as desired to the exact solution x. A stopping criteria with tolerance, ϵ, defines how much reduction in error is desired. When the j^{th} iterate has achieved the desired precision the method is said to have converged.

2.1 Basic iterative methods

A basic iterative method can be built by considering a splitting of the coefficient matrix.

$$A = M - N,\ M, N \in \mathbb{R}^{N \times N} \tag{2.1}$$

where M is assumed to be invertible. If we substitute this splitting into the original linear system given by

$$Ax = b \tag{2.2}$$

we get

$$Mx = Nx + b \tag{2.3}$$

From (2.3) a basic iterative method can be devised as

$$Mx_{j+1} = b + Nx_j \tag{2.4}$$

where the unknown at iteration $j+1$ can be determined using the unknown at the previous iteration, x_j. If we define the residual as $r_j = b - Ax_j$ then we can re-write (2.4) as

$$x_{j+1} = x_j + M^{-1}r_j, \tag{2.5}$$

where M is called the preconditioner. Now we have a recurrence relation (2.5) for the update to the unknown we wish to calculate. Further, M can be chosen so that iterates can be generated easily and a desired tolerance can be achieved quickly.

The basic iterative methods using the recurrence in (2.5) can take a lot of iterations and may prove to be impractical in achieving convergence for large and ill-conditioned systems. Therefore, we explore the more general Krylov subspace methods.

2.2 Krylov subspace methods

Krylov subspace methods search for the solution in a space spanned by vectors of the type $A^j v$, where $j \in 0, 1, \ldots$. This subspace is given by

$$\mathcal{K}_j(A, r_0) = \text{span } \{r_0, Ar_0, A^2r_0, ...A^{j-1}r_0\}, \tag{2.6}$$

where $r_0 = b - Ax_0$. We focus our attention on the CG method which is also the method we use in our implementations.

2.2.1 Conjugate Gradient method

The Conjugate Gradient (CG) method utilizes the fact that A is symmetric and positive definite. It was discovered by Hestenes and Stiefel and reported in [34]. In the CG method we update the residual every iteration by looking in a particular search direction. These search directions p_j are conjugate with respect to A which means that

$$(Ap_i, p_j) = 0, \ i \neq j, \tag{2.7}$$

where we denote the dot product (`dot`) operator by (,). It can be shown that (2.7) is equivalent to having orthogonal residuals,

$$(r_i, r_j) \neq 0, \ i \neq j. \tag{2.8}$$

In order to generate new iterates, recurrence (2.9) is used in CG, where j denotes the iteration number.

$$x_{j+1} = x_j + \alpha_j p_j, \text{ where } \alpha_j \in \mathbb{R}. \tag{2.9}$$

Multiplying (2.9) by A yields a recurrence for r.

$$r_{j+1} = r_j - \alpha_j A p_j. \tag{2.10}$$

α_j is calculated using,

$$\alpha_j = \frac{(r_j, r_j)}{(Ap_j, p_j)}. \tag{2.11}$$

It can be shown that (2.11) minimizes $\| x_j - x \|_A$ for all choices of α_j and also that (2.8) is satisfied. The search directions are updated using the residuals:

$$p_{j+1} = r_{j+1} + \beta_j p_j, \tag{2.12}$$

where $\beta_j \in \mathbb{R}$ is calculated using

$$\beta_j = \frac{(r_{j+1}, r_{j+1})}{(r_j, r_j)}. \tag{2.13}$$

Calculating β using (2.13) ensures that (2.7) is satisfied. More details about the method and its derivation can be found in [34] and also in the text of [58].

Preconditioned Conjugate Gradient

Convergence to a desired accuracy of the solution vector in CG can be slow if there are small eigenvalues (λ_i) in the spectrum of the coefficient matrix. In other words if the condition number of A is large then the convergence of the CG method could be prohibitively slow. In order to accelerate convergence we instead convert the linear system into an equivalent one that preserves the properties of the matrix (e.g. Symmetric Positive Definite (SPD)). The transformed system reads

$$\tilde{A}\tilde{x} = \tilde{b} \tag{2.14}$$

where,

$$\tilde{A} = M^{\frac{-1}{2}} A M^{\frac{-1}{2}},\ \tilde{x} = M^{\frac{1}{2}} x,\ \tilde{b} = M^{\frac{-1}{2}} b. \tag{2.15}$$

The matrix M is called the preconditioner and it is assumed that M is SPD. In general it is never required to calculate the inverse square root of the matrix M. Instead the preconditioned CG algorithm is formulated using M^{-1}.

2.3 First Level preconditioning

In order to improve the convergence of the linear system, a wide variety of preconditioners exist in the literature [9]. In this section we briefly summarize the preconditioning schemes that are well-known including the preconditioning schemes we have used in our implementations.

2.3.1 Diagonal scaling

Diagonal Scaling is a highly parallel preconditioner. The preconditioner is defined as

$$M = D, \tag{2.16}$$

D=diag(A), the main diagonal of A. It is also called Jacobi preconditioner. Its application to a vector is also completely parallelizable. However, it is not a very effective preconditioner for a matrix which is ill-conditioned.

2.3.2 Incomplete LU (ILU) preconditioning

A sparse matrix A can be decomposed into a lower, L and and upper triangular, U matrix as in

$$A = LU. \tag{2.17}$$

Such a decomposition is called an LU decomposition of A. An incomplete LU decomposition can be generated by deciding on a sparsity pattern for L and U. This could be the sparsity pattern of A. For the decomposition (2.17) a fill-in strategy or a threshold can be decided, that is to say, which elements to keep or drop. The generated 'incomplete' LU decomposition can then be used as a preconditioner in the CG method (if LU is Symmetric Positive Definite (SPD)).

Many versions of ILU preconditioners are discussed in literature. For a detailed introduction to these we refer the reader to [58]. We briefly mention some of the variants here.

ILU with fill-in is denoted as ILU(p) where p denotes the level of fill-in. A common fill-in pattern used is from the fill-in pattern of the powers of A. So ILU(2) refers to ILU decomposition with the fill-in pattern resembling that of A^2. Threshold techniques are denoted as ILU-T(t) where t is a value below which all non-zeros are discarded. Combinations of fill-in and threshold techniques are also possible.

2.3.3 Incomplete Cholesky

For an SPD matrix one can decompose the coefficient matrix into a product of a lower triangular matrix and its transpose.

$$A = LL^T, \text{ where } A \in \mathbb{R}^{N \times N}. \tag{2.18}$$

In order to calculate the elements of L which is a lower triangular matrix we use Algorithm 1. a_{ij} denotes the element of the original coefficient matrix A. In this algorithm a_{ij} is overwritten by l_{ij} $\forall$ $i \geq j$.

Algorithm 1 Cholesky decomposition (Column version)

1: **for** $k := 1, 2, \ldots, N$ do **do**
2: $\quad a_{ii} = (a_{ii} - \sum_{k=1}^{i-1} a_{ik}^2)^{\frac{1}{2}}$
3: $\quad$ **for** $i := k+1, \ldots, N$ do **do**
4: $\quad\quad a_{ji} = \frac{1}{a_{ii}}(a_{ij} - \sum_{k=1}^{i-1} a_{ik}a_{jk})$
5: $\quad$ **end for**
6: **end for**

The most common way of making this factorization incomplete is to use the sparsity pattern of the coefficient matrix A and impose it on L. This is to say that L has non-zeros only where A has non-zeros. The CG method with incomplete Cholesky factorization without any fill-in (following only sparsity pattern of A) is abbreviated as ICCG(0). More details can be found in the text of [24].

2.3.4 Block incomplete Cholesky

The incomplete Cholesky preconditioner discussed in Section 2.3.3 can be modified for use on a parallel platform as suggested by the authors in [51]. We consider an adapted incomplete Cholesky decomposition: $A = LD^{-1}L^T - R$ where the elements of the lower triangular matrix L and diagonal matrix D satisfy the following rules:

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

R defines the set of non-zeros that have to be dropped. In order to make blocks for the block incomplete Cholesky (Block-IC) approach we first apply the block structure on the matrix A. As an example in Figure 2.1 we show how

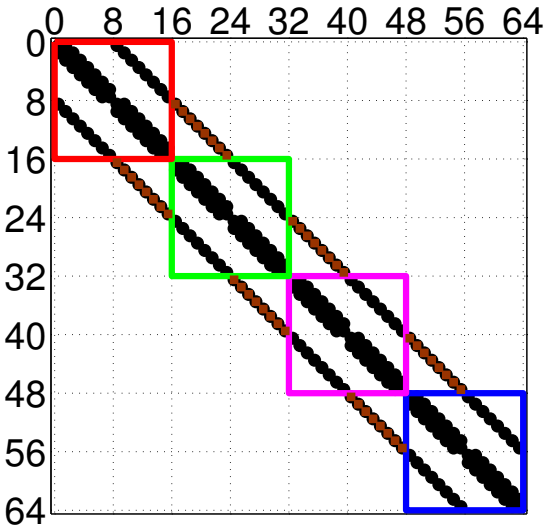


Figure 2.1: Block-IC block structure. Elements excluded from a 8×8 grid for a block size of $2n = 16$.

some of the elements belonging to the 5-point Poisson type matrix are dropped when a block incomplete scheme is applied to make the preconditioner for A. The red colored dots are the elements that are dropped since they lie outside the blocks. We make L as suggested in Section 2.3.4. In each iteration we have to compute y from

$$y = {M_{BIC}}^{-1} r, \text{ where } M_{BIC} = LD^{-1}L^T. \tag{2.19}$$

This is done by doing forward substitution followed by diagonal scaling and then backward substitution. The blocks shown in Figure 2.1 can be solved in parallel. The number of blocks is $\frac{N}{g}$, where g is the size of the blocks. Within a block all calculations are sequential. However, each block forms a system that can be solved independently of other blocks. The parallelism offered by Block-IC is limited to the number of blocks. In order to increase the number of parallel operations we must decrease the block size, g. However, doing this would lead to more loss of information in the preconditioner (the elements outside the blocks are dropped in Figure 2.1), and consequently, convergence is delayed. The diagonal preconditioner discussed in Section 2.3.1 is an extreme case of the Block-IC preconditioner when the number of blocks is equal to the number of rows in the matrix.

2.3.5 Multi-elimination ILU

In this section we closely follow the description for multi-elimination ILU as described in [58]. The idea of multi-elimination ILU has to be explained by first looking at Gaussian elimination. Within Gaussian elimination one has to find independent sets of unknowns. Independent unknowns are those that do not depend on each other according to a binary relation defined by the graph of the matrix. This puts the original matrix into the form given by,

$$\begin{pmatrix} D & E \\ F & C \end{pmatrix} \tag{2.20}$$

where D is a diagonal and C can be arbitrary. If we define the binary relation as the color of each node in the graph of the matrix then no two adjacent nodes must have the same color. In this method of multi-coloring the set of vertices from a adjacency graph are grouped such that the unknowns in these equations do not have unknowns from any other set. The rows belonging to an independent set can be used as pivots rows simultaneously. When such rows are eliminated we are left with a part of the original matrix in which the process of finding independent sets is repeated. For details on this process we refer to [58].

Now we present an exact reduction step. Let A_j be the matrix obtained after j^{th} step of reduction, $j = 0, \ldots, nlev$ with $A_0 = A$ and $nlev$ are the number of decided levels. After an independent set ordering is applied to A_j and the matrix is permuted as follows:

$$P_j A_j P_j^T = \begin{pmatrix} D_j & F_j \\ E_j & C_j \end{pmatrix} \tag{2.21}$$

where D_j is a diagonal matrix. Now we eliminate the unknowns of the independent set to get the next reduced matrix,

$$A_{j+1} = C_j - E_j D_j^{-1} F_j \tag{2.22}$$

This results in an implicit block-LU factorization

$$P_j A_j P_j^T = \begin{pmatrix} D_j & F_j \\ E_j & C_j \end{pmatrix} = \begin{pmatrix} I & 0 \\ E_j D_j^{-1} & I \end{pmatrix} \times \begin{pmatrix} D_j & F_j \\ 0 & A_{j+1} \end{pmatrix} \tag{2.23}$$

with A_{j+1} as defined previously in (2.22). In order to solve a system with matrix A_j both forward and backward substitution need to be performed with block matrices on the right-hand side of the system in (2.23). The backward solves involve solution with A_{j+1}. The block factorization can be used until a system results that can be solved by the standard method.

In order to formulate the ILU preconditioner with Multi-elimination we first need to make the decomposition incomplete. This is achieved by redefining A_{j+1}

$$A_{j+1} = C_j - E_j D_j^{-1} F_j - R_j \tag{2.24}$$

where R_j is the matrix of elements dropped in the j^{th} reduction step. Thus in order to generate the entire factorization we have to do a sequence of ILU factorizations of the form

$$P_j A_j P_j^T = \begin{pmatrix} D_j & F_j \\ E_j & C_j \end{pmatrix} = \begin{pmatrix} I & 0 \\ E_j D_j^{-1} & I \end{pmatrix} \times \begin{pmatrix} D_j & F_j \\ 0 & A_{j+1} \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 0 & R_j \end{pmatrix} \tag{2.25}$$

with A_{j+1} defined as (2.24).

2.3.6 Sparse approximate inverse (SPAI) preconditioner

The main idea of this approach is that a sparse matrix M^{-1} can be explicitly computed and used as a preconditioner. However the inverse of A could be dense. Some of the techniques discussed in [11] are worth mentioning:

SPAI based on Frobenius norm minimization

We use the description of this method as presented in [8]. The main idea is to construct a sparse matrix $H \approx A^{-1}$ as the solution of the following constrained minimization problem

$$\arg\min_{H \in \mathcal{S}} \parallel I - AH \parallel_F \tag{2.26}$$

where $\mathcal{S}$ is a set of sparse matrices and $\parallel . \parallel_F$ denotes the Frobenius norm of the matrix. Since

$$\parallel I - AH \parallel_F^2 = \sum_{j=1}^{N} \parallel e_j - Ah_j \parallel_2^2, \tag{2.27}$$

where e_j denotes the j^{th} column of the identity matrix. The computation of H involves solving N independent least squares problems with sparsity constraints. This makes it an ideal candidate for parallelization and distributed implementation.

Factorized sparse approximate inverse

If the matrix A admits an incomplete factorization of the kind $A = LDU$, where $U = L^T$ then $A^{-1} = U^{-1}D^{-1}L^{-1}$. Further one can make approximations to U^{-1} and L^{-1}. One of the first works with this approach was studied in [43].

Inverse ILU techniques

Using the idea of incomplete factorization in the previous technique once can also say $A \approx \hat{L}\hat{U}$. In this method, one approximates the inverses for $\hat{L}$ and $\hat{U}$. Approximate inverses can be calculated by solving the $2N$ triangular systems

$$\hat{L}x_i = e_i, \hat{U}y_i = e_i, \tag{2.28}$$

where $1 \leq i \leq N$. In principle these linear systems can be solved in parallel so ample parallelism can be exploited in this preconditioning technique. Some of the approaches are given in [1, 71].

2.3.7 Multigrid based preconditioners

A representative discussion on the use of multigrid as a preconditioner appears in [54]. The authors measure the effectiveness of the multigrid method as a solver and a preconditioner. Multigrid methods accelerate convergence by global correction while solving the coarse problem. Multigrid method in general has three steps:

1. Smoothing - Reduction of the errors with high frequency with a basic iterative method.
2. Restriction - Projecting the reduced errors on a coarse grid.
3. Prolongation - A correction calculated on the coarse grid is interpolated to a fine grid.

Multigrid Method

The multigrid method can be defined as a sequence of the operations of smoothing, restriction and prolongation applied to a set of unknowns. When using multigrid as a preconditioner, the preconditioning step, $My = r$ in the standard PCG algorithm [58] involves solving a system of equations. This can be solved using the two-level algorithm in Algorithm 2.

Algorithm 2 Two level Multigrid algorithm

1: **for** i:=0,..., until convergence **do**
2: $y^- = S(y_i, M, r, m)$
3: $r^c = R(r - My^-)$
4: $e^c = (M^c)^{-1}r^c$
5: $y^+ = y^- + Pe^c$
6: $y_{i+1} = S(y^+, M, r, m)$
7: **end for**

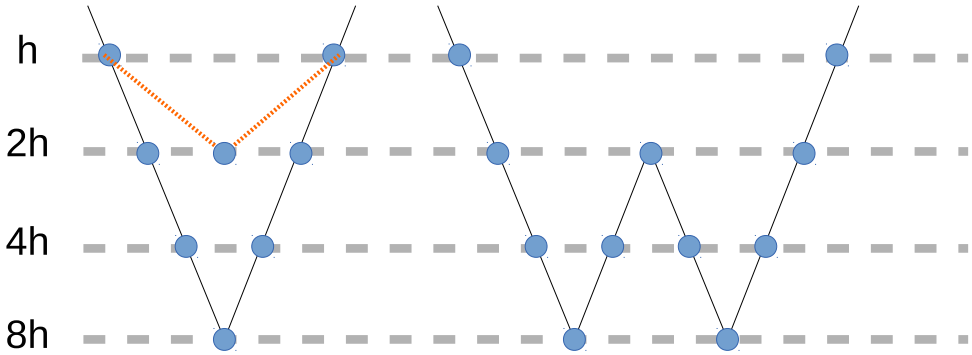


Figure 2.2: V and W cycles in multigrid

In Algorithm 2 we denote the prolongation and restriction operators by P and R, the coarse grid operator by M^c and the smoother by S. In the first step of Algorithm 2, m smoothing steps are performed on the system of equations followed by restricting the residual in the next step. Subsequently, coarse grid equations are solved to get a coarse grid approximation to the fine grid error. This coarse grid error is then prolonged to the fine grid and added to the current fine grid solution approximation calculated in the previous step. Finally some more smoothing steps are applied on to the finer system. This is defined as a a two-grid V-cycle. It is highlighted in orange in Figure 2.2. One can make W cycles which stay longer on the coarse grids[1]. In Figure 2.3 we can see a coarse and a fine grid. In geometric multigrid the unknowns on these two grids take part for example in the 2-grid V-cycle.

Multigrid methods can also be constructed directly from the system matrix (instead of refining/coarsening the grid/ discretized domain). Such methods are classified as the Algebraic multigrid (AMG) methods. The levels created in the AMG method are oblivious to any geometric interpretation.

In [3] the authors merge the benefits of both these multigrid approaches to precondition a linear system arising from a groundwater flow problem. We would like to mention that this is one of the techniques and many other variants of multigrid method for geometric and algebraic methods are known in literature [44, 55, 56, 61].

In order to ensure that the preconditioner is symmetric and positive definite the multigrid preconditioner must have equal number of smoothing steps before and after each coarse grid correction.

[1]From MIT opencourseware http://math.mit.edu/classes/18.086/2006/am63.pdf

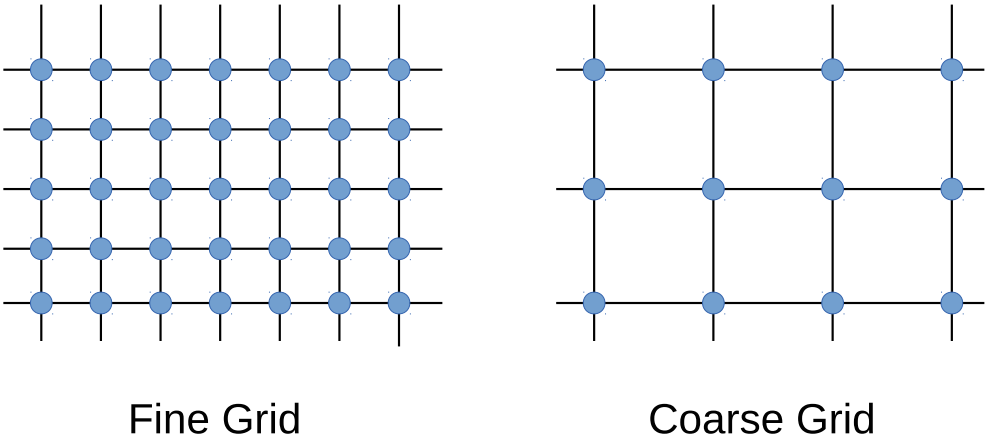


Figure 2.3: Multigrid method

2.3.8 IP preconditioning

In [2], a new kind of incomplete preconditioning is presented. The preconditioner is based on a splitting of the coefficient matrix A using its strictly lower triangular part L and its diagonal D,

$$A = L + D + L^T. \tag{2.29}$$

Specifically the preconditioner is defined as,

$${M_{IP}}^{-1} = (I - LD)(I - DL^T). \tag{2.30}$$

After calculation of the entries of M^{-1}, the values that are below a certain threshold value are dropped, which results in an incomplete decomposition that is applied as the preconditioner. As this preconditioner was used for a Poisson type problem, the authors call it the Incomplete Poisson (IP) preconditioner. A detailed heuristic analysis about the effectiveness of this preconditioning technique can be found in [2]. The main advantage of this technique is that the preconditioning operation $M^{-1}r$ is reduced to sparse matrix-vector products which have been heavily optimized for many-core platforms [5, 6, 47, 79].

2.4 Second level preconditioning

For ill-conditioned problems that may have small eigenvalues that retard the convergence of the PCG method, one can use an additional level of preconditioning. Deflation [19, 53] is one such technique that 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

$$P = I - AQ, Q = ZE^{-1}Z^T, E = Z^TAZ, \tag{2.31}$$

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 consist of 'deflation' or 'projection' vectors. Z is full rank with $d < N - k$, where k is the number of zero eigenvalues if any. The deflated system is now

$$PA\hat{x} = Pb. \tag{2.32}$$

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 $x = Qb + P^T\hat{x}$. It must be noted that PA is a singular matrix with $PA = AP^T$. The matrix P is a projector and $P^2 = P$. More of the properties of P and PA are discussed in [62, 66]

The deflated system in (2.32) can be solved using a symmetric positive definite (SPD) preconditioner, M^{-1}. We therefore seek a solution of

$$M^{-1}PA\hat{x} = M^{-1}Pb. \tag{2.33}$$

The resulting method is called the Deflated Preconditioned Conjugate Gradient (DPCG) method [62, 66] as listed in Algorithm 3.

Algorithm 3 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: $x_{it} := Qb + P^T x_{i+1}$

In order to implement deflation one can break the computation of Pr_j down into a series of operations,

$$a_1 = Z^T r_j, \tag{2.34a}$$
$$a_2 = E^{-1} a_1, \tag{2.34b}$$
$$a_3 = AZ a_2, \tag{2.34c}$$
$$s = r_j - a_3. \tag{2.34d}$$

(2.34b) shows the solution of the coarse system that results during the implementation of deflation.

2.4.1 Motivation to use deflation for problems with strongly varying coefficients

Deflation is useful in solving a linear system when the spectrum of the coefficient matrix has some very small eigenvalues. By making a suitable deflation subspace these small eigenvalues can be removed from the spectrum of PA. In the case of bubbly flow, due to the heavy contrast between the densities of the two media, small eigenvalues appear corresponding to the interfaces. Advantages of using deflation as opposed to only using one level of preconditioning (for the CG method) like Incomplete Cholesky with zero fill-in, ICCG(0) has been studied in [62, 63].

2.4.2 Choices of deflation vectors

Deflation vectors form the columns of the matrix Z. In order to remove the small eigenvalues in the spectrum of $M^{-1}A$ or A that delay convergence eigenvectors corresponding to these eigenvalues must be sored in the columns of Z. To calculate the exact eigenvectors would be ideal but it is also computationally expensive. This is the reason why approximations to these eigenvectors are calculated in practice and they constitute the matrix Z.

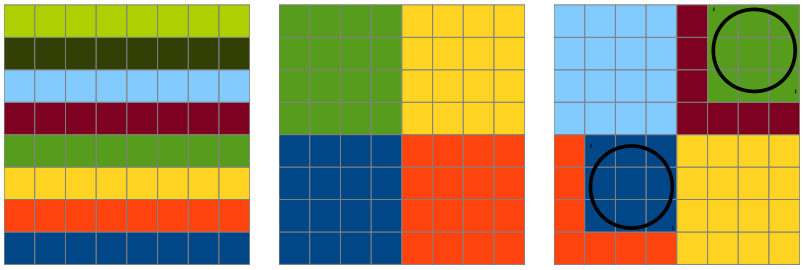
A relatively cheap method for making reasonably good approximations to the eigenvectors is to use piece-wise constant vectors.

Piece-wise constant deflation vectors

Piece-wise constant deflation vectors can be constructed in 2 broad varieties

1. Depending on the geometry of the problem.
2. Depending on the underlying physics of the problem.

The first class of vectors divides the domain into sub-domains. Each sub-domain corresponds to a vector. The vector has non-zero values only for cells over which its sub-domain is defined. For the second type additional information from the problem is used to generate deflation vectors.



(a) Stripe-wise sub-domain (b) Block-wise sub-domain (c) Level-set

Figure 2.4: Three kinds of deflation vectors. Each color corresponds to a column in Z.

Sub-domain deflation vectors Let Ω, be the domain on which the problem is defined, be divided into sub-domains Ω_i, $i = 1, 2, \cdots, k$, such that $\Omega_i \bigcap \Omega_j = \emptyset$ 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:

$$(z_i)_j := \begin{cases} 0, & x_j \in \Omega_h \\ 1, & x_j \in \Omega_{h_i} \end{cases} \tag{2.35}$$

Then the Z matrix for sub-domain deflation vectors is defined by

$$Z_{SD} := [z_1 z_2 \cdots z_k], \tag{2.36}$$

It must be mentioned that sub-domain vectors can also be made in another way using the stripe-wise division. As shown in Figure 2.4(a) it is slightly different from block-wise in the places where ones are placed in each column. In the example of Figure 2.4(a) and 2.4(b) one can consider a 2D grid with 8 stripe-wise vectors in Figure 2.4(a) and 4 block-wise vectors in Figure 2.4(b) corresponding to different colors. The Z_{SD} matrix composed of z_i vectors is further used to create the AZ matrix and the $E = Z^T AZ$ matrix.

Level-set deflation vectors The level-set deflation vectors [64] utilize the knowledge of the physical problem underlying the simulation and can be very effective at accurately approximating the deflation subspace. Level-set function is defined as a signed distance function which takes values of opposite signs on either side of the interface between two materials with different physical properties. One can imagine the Level-Set vectors also to be a kind of a piece-wise constant vectors. Since the only difference from sub-domain based piece-wise constant vectors is that the non-zero values are now prescribed by a function (level-set) as opposed to a domain.

Level-set sub-domain deflation vectors It is possible to extend the level-set deflation vectors with sub-domain vectors and improve their effectiveness. In particular such a combination must aim at being able to capture the physics underlying the problem and combine it with the sub-domain approach effectively. These vectors will be explained in detail for the individual problems for which we use deflation. These problems are discussed in subsequent chapters.

2.4.3 Cost and benefits of deflation

The deflation operation when broken down into the steps mentioned in (2.34) involves one or two matrix vector multiplications, solution of a small linear system and a vector update. The operation $Z^T r$ may not always be implemented as a matrix-vector operation. It depends on what kind deflation vectors are chosen to make Z. These could be stripe-wise / block-wise sub-domain, level-set vectors or the combination of the two. The sparsity pattern of Z depends on the choice of the constituent vectors of Z. For simpler Z (e.g. from stripe-wise sub-domain vectors) the operation (2.34a) can be as simple as a set of reduction operations.

To solve the coarse system in (2.34b) one can use different approaches. The size of the matrix E depends on the size of the deflation subspace d. So if a lot of vectors are chosen, d is large. This can be true if there are a lot of undesirable eigenvalues and/or the information from the physics of the problem is used in conjunction with e.g. sub-domain/ level-set sub-domain deflation vectors. When d is large, E is also large ($E \in \mathbb{R}^{d \times d}$). In this case CG can be used to solve this coarse system making a nested method. For comparatively smaller sizes one can use a factorization followed by a solve or for considerably smaller sizes a good approach is to invert E. The dense E^{-1} can be then subjected to a general matrix vector (`gemv`) operation to obtain a_2.

2.5 Matrix storage formats and SpMV

In this research we work with coefficient matrices that are sparse. Sparse matrices can have non-zeros distributed arbitrarily across the matrix. Choosing a right storage format is instrumental in getting best locality of data and utilization of the memory bandwidth for the GPU or the CPU. Sparse matrices can be stored in a variety of formats depending on the positions of non-zeros in the matrix. On the GPU the benefits of having a storage format that can exploit its bandwidth could be very promising. This is partly the reason why a lot of research [5, 13, 14, 15, 52, 74] has been devoted to devising smarter storage formats for sparse matrices.

2.5.1 CSR - Compressed Sparse Row

This data format is the most commonly used and optimized sparse matrix storage formats. Prominent libraries on CPU[2] and GPU[3] provide optimized implementations for SpMV product routines that use this matrix format. To store a matrix in this format one requires 3 arrays. One array is that of non-zero values. Another one is for the column indices of each of these non-zeros. The last array stores the row offsets and has the length equal to the number of rows in the matrix plus one. An example is shown in Figure (2.5).

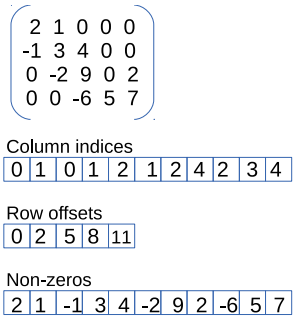


Figure 2.5: CSR matrix storage format

More details of this storage format, its performance for different matrices and the SpMV routines are given in [6].

BCSR - Block compressed sparse row

The BCSR format is an extension on the CSR format. A certain block size is decided and if the matrix is $N \times M$ and the block size chosen is m then the matrix is divided into block matrices of size $m \times m$. Each of these matrices can be given a row and column index between 0 and $\frac{N}{m}/\frac{M}{m}$ assuming N and M are multiples of m. It requires three arrays for its storage. The first array stores all the non-zero elements. The second array stores the column indices of the block matrices and the third array stores row offsets for these block matrices per block row. A description of this storage format with examples is given in the CUSPARSE documentation[4].

2.5.2 DIA - Diagonal

For problems having a regular stencil like a 5-point or 7-point stencil, where there is a main diagonal and an equal number of off-diagonals on either side,

[2] IntelMathKernelLibrary(MKL)https://software.intel.com/en-us/intel-mkl
[3] http://docs.nvidia.com/cuda/cusparse/index.html
[4] http://docs.nvidia.com/cuda/cusparse/#block-compressed-sparse-row-format-bsr

DIA format is most suitable. As the name suggests, we store each diagonal in a separate array. So if we assume a penta-diagonal matrix arising out of the discretization of a poisson type (2D) problem with $N = n \times n$ unknowns. We have diagonals with offsets ± 1 and $\pm n$. Now the main diagonal has N non-zeros but the diagonals with offsets ± 1 and $\pm n$ have less number of non-zeros. In DIA format of storage all the diagonals are stored in separate arrays of length N and the empty places are filled with zeros. This format is most useful for GPU implementations of SpMV for structured matrices as it maximizes memory bandwidth utilization. For details we refer to [6].

2.5.3 COO - Co-ordinate

This is the most intuitive sparse matrix storage format with three arrays, each the size of the number of non-zero elements. The first two arrays store the row and column index of the non-zero element and at the same position in the third array the non-zero element is stored. This storage scheme is most general and takes the maximum space amongst those discussed in this text. We often use it for debugging as it is the easiest to understand.

2.5.4 ELL

In the ELL storage format a fixed number of non-zeros per row are decided for each row of the matrix. This is usually the largest number of non-zeros any row has. Then a storage space of number of rows times maximum number of non-zeros in any row is created. In order to store the rows and columns indices another matrix of the same size is created that has column indices in place of non-zeros for each of the non-zeros. It can be inferred that the ELL format is most useful in the case when the rows have equal number of non-zeros and average number of non-zeros is less. Otherwise there is either too much wastage or little saving from compression of a full matrix into this format. The SpMV routine for this method can be referred to in the article from NVIDIA [6].

2.5.5 HYB - Hybrid

HYB format combines the benefits of the ELL and COO format. It calculates a typical number of non-zeros per row. This might as well be the average number of non-zeros in the matrix per row. It then uses ELL format for storing non-zeros up to this average number in the ELL format. For all other non-zeros which lie outside of the average number of non-zeros range the COO format is utilized. Details, benchmarks and SpMV routines for this method can be found in the paper from Bell and Garland [6].

2.6 A brief overview of GPU computing for preconditioned Conjugate Gradient (PCG)

2.6.1 Linear algebra and GPU computing

In 2003 a new approach [12] was suggested where GPUs could be utilized for the Conjugate Gradient Method. The authors used GPUs from NVIDIA and made abstractions to use the video card as a general purpose computing device for solving a linear system.

This was followed by the advent of CUDA in 2006. CUDA allowed extensions to be added to C or Fortran code which could hide the architectural details of NVIDIA GPUs in order to make parallel code run on the GPU. This launched a series of efforts to write and optimize codes for building blocks of linear algebra operations for the GPU. One of the most useful of these is the work done on SpMV products. Researchers at NVIDIA presented a report [7] which showed the comparison of different storage schemes for a set of matrices and how the SpMV operation can be optimized for each of the storage formats. Researchers at IBM also [50] came up with optimized SpMV implementations at the same time. Block-based SpMV approach was used in [52]. Following the block-based approach they divide their matrix into strips which are then divided into blocks. They further economize on the memory footprint by storing block indices in a single word. Their implementations perform up to 2 times better on average on a set of matrices. MAGMABLAS, which is a library for improving dense matrix operations on the GPU [36] was introduced in 2009. It includes various decompositions and triangular solvers for linear systems amongst its collection of optimized BLAS operations on dense matrices.

2.6.2 OpenCL and OpenACC

OpenCL[5] and OpenACC[6] are alternative methods of parallel programming with accelerators.

OpenCL is a language which has its own API and has a very generic outlook at being able to exploit any platform that allows parallel computing. This can include multi-core CPUs, GPUs, APUs and even application specific hardware. It accesses and uses the resources of an accelerator as specified by the programmer in order to run a parallel program. In [60], we come across a good introduction to OpenCL as a paradigm for accelerator programming. A detailed comparison of OpenCL and CUDA appears in [20, 21, 40]. The authors in these works compare CUDA with OpenCL for different applications and arrive at varying conclusions. In particular they measure the portability ad-

[5] https://www.khronos.org/OpenCL/
[6] http://www.openacc-standard.org/

vantages of OpenCL versus the specialized capabilities of CUDA at extracting performance out of a parallelizable application. Overall the OpenCL platform shows promise, however, CUDA based acceleration gives tough competition to OpenCL in all studies. The authors in [20] also comment on the discrepancies in both OpenCL and CUDA and suggest auto-tuning as an option to automate parallelization.

OpenACC on the other hand is much more like CUDA. Although it targets a variety of accelerators. Some people even compare OpenACC as a counterpart for OpenMP, which indeed is a set of compiler directives for utilizing multi-core CPUs. The programmer has to provide the compiler with supplementary information like specifying which parts or loops of the code must be placed on the accelerator platform and other such pointers for performance improvement. In [78] the authors note the usefulness of OpenACC as opposed to using low level accelerator programming and weigh it against the possibility of sub-optimal performance. The potential of OpenACC can be estimated from the works in [32, 46] where the authors port complex codes and try to tune them for single GPU, multi-core CPUs and even exascale machines. They report the ease of programming OpenACC provides for simple (single GPU, multi-core CPU) and complex (multi-core CPU+GPU over MPI) programming platforms.

2.6.3 PCG with GPUs

Preconditioning has been a subject of active research on the GPU since it is often a highly sequential operation in the PCG method. Some of the most effective schemes like ILU or Incomplete Cholesky are sequential methods because of the forward and backward substitution steps. On the GPU such schemes are not efficient as they do not utilize the device to its full capacity. In [2] one of the first preconditioners that exhibited fine-grain parallelism was reported. Another preconditioner based on Truncated Neumann Series approximation was reported in [57]. They make this approximation to the SSOR preconditioner [58] and suggest an optimum value of ω for their problem. We also use a similar approach for our preconditioners but do not use the relaxation parameter ω. Chebyshev polynomial based preconditioners were also optimized for the GPUs as reported in [39]. The authors mention the design challenges and also show computational savings with respect to a Block Incomplete LU preconditioner. Multigrid based preconditioners have also been explored for Krylov subspace based iterative linear solvers. In [22] the authors use multigrid based preconditioners for solving linear systems arising from power networks. They report being able to simulate larger test cases to accurately model their problem and up to an order of magnitude computational speedup. In [35] an incomplete LU decomposition based preconditioner

with fill-in is used combined with reordering using multi-coloring. By using these techniques the preconditioners exhibit greater degree of fine-parallelism which make them attractive for GPU implementation. In [13] the authors do a comparison between a CPU and a GPU PCG solver and point out the different basic linear algebra operations (`axpy`, `dot`) that must be tuned for best performance of the PCG algorithm.

2.6.4 Multi-GPU implementations

Multi-GPU implementations have now become more common even for a bandwidth bound method like CG. The research focus has been at developing preconditioners and optimizing the matrix-vector product routines to be able to utilize the parallelism available in compute nodes (multi-core CPUs, GPUs).

In [14] the authors use 4 GPUs on a single board and a specific data storage format called JDS (Jagged diagonal storage format) in order to achieve double the number of FLOPs using 4 GPUs instead of one. They cite cache misses as the primary reason for the low scaling they observe.

In [81] the authors present the result of a Jacobi preconditioned CG algorithm for a fluid dynamics solver and show the results for up to 8 GPUs using MPI. They notice that communication becomes central when an algorithm is ported to multiple GPUs across MPI ranks.

GMRES, which is another Krylov subspace method, was used by the research presented in [4], where they used 12 GPUs (2 on each node) and compared against 24 CPU cores for up to 1 million unknowns. They point out that using GPUs in a distributed way is only feasible for large sparse matrices.

In [41] the authors use a shifted Laplace multigrid based preconditioner for a Helholtz solve and report an improvement in speedup when a split algorithm based implementation is used. They also compare it to multi-core CPUs and outline the challenges of communication when using a multi-GPU/CPU approach. An interesting study and a theoretical model to predict gains from multi-GPU implementations appeared in [75]. The authors propose using a new implementation of the BCSR (Section 2.5.1) sparse matrix storage format and show that it is superior for their test cases. They also predict that sparse matrices with sizes larger than 2.5 million can achieve good scalability on GPUs. Discussion on bandwidth, matrix storage and parameters of SpMV are identified as key areas of their contribution and focus for future research.

Using Pthreads and MPI the authors in [67] manage to get impressive speedups on an Intel Duo core CPU. They show results for both single and double precision versions of their code and quantify the speedup for adding multiple GPUs to their implementation.

Exploring parallelism at all levels and using MPI, OpenMP and CUDA in order to accelerate their solvers the authors in [37] present a large scale

study with 17.2 billion unknowns on 256 GPUs. They use 1-D partitioning for their data to be shared amongst processors. They conclude that improvements in network speeds and throughput could be crucial to such implementations. They also note that three level parallel (OpenMP + MPI +CUDA) model holds very little advantage as compared to two level (MPI + CUDA) model using only MPI and CUDA.

A shifted Laplace based multigrid preconditioner for the Bi-CGSTAB method is studied in [42] and implementations on multi-GPU, multi-core CPUs and single GPU are compared. The authors report better performance on the GPU and face a challenge at recovering scaling when they move to multiple GPUs.

In order to address the problems that arise with communication in algorithms like CG, the authors in [23] suggest delaying inner products, accumulating a certain error in calculating, say, the residual and correcting it later on when the communication is complete. By this way they save on synchronization time in multi-GPU implementations (of Krylov subspace methods) for global operations like dot-products.

CHAPTER 3

Neumann preconditioning based DPCG

3.1 Introduction

In this chapter we present the results of our preliminary experiments with deflation. We show how our choices for the first level preconditioner were made and what effects they have on our implementations. Through simple test problems we show the advantages of deflation with preconditioning and the need for exposing fine-grain parallelism. We report on the data structures used to expose parallelism and limitations of certain kind of deflation vectors. In this chapter we focus on the bubbly flow problem.

The work presented in this chapter also appears in:

Rohit Gupta, Martin B. van Gijzen, and Kees Vuik. Efficient two-level preconditioned Conjugate Gradient method on the GPU. In Michel Dayd, Osni Marques, and Kengo Nakajima, editors, *High Performance Computing for Computational Science - VECPAR 2012*, volume 7851 of *Lecture Notes in Computer Science*, pages 36–49. Springer Berlin Heidelberg, 2013.

3.2 Preconditioning

3.2.1 IP preconditioning with scaling

The incomplete Poisson preconditioner introduced in section 2.3.8 cannot accelerate convergence for even a 2D problem with contrasts (refer [26]). To improve the performance we used scaling of the linear system. Specifically in the results that follow we use IP preconditioner with scaling, abbreviated as $ip(scale)$. We modify the linear system (1.1) into

$$\tilde{A} = D^{-\frac{1}{2}} A D^{-\frac{1}{2}}. \tag{3.1}$$

$$\tilde{x} = D^{\frac{1}{2}} x. \tag{3.2}$$

$$\tilde{b} = D^{-\frac{1}{2}} b. \tag{3.3}$$

3.2.2 Truncated Neumann series based preconditioning

Another preconditioner that can be derived from the factorization of A is the Truncated Neumann Series (TNS) based preconditioner. The IP preconditioner mentioned in section 2.3.8 and 3.2.1 is not necessarily a good choice for highly ill-conditioned problems [28]. Moreover, in [2] the choice of the preconditioner is not very well motivated. So we start with a standard Symmetric Gauss Siedel preconditioner. 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. This is also called the Symmetric Successive Over Relaxation (SSOR) preconditioner (for details we refer to [58]). We apply the truncated Neumann series for approximating M^{-1}. Specifically, for $(I + LD^{-1})$ (and similarly for $(I + (LD^{-1})^T)$) the series can be defined as

$$I - LD^{-1} + (LD^{-1})^2 - (LD^{-1})^3 + \cdots \tag{3.4}$$

which converges to $(I + LD^{-1})^{-1}$ if $\| LD^{-1} \|_\infty < 1$. So we can redefine M^{-1} as

$$M^{-1} \approx (I - D^{-1}L^T + \cdots)D^{-1}(I - LD^{-1} + \cdots). \tag{3.5}$$

For making this preconditioner (computationally) feasible, the series (3.4) must be truncated after 1 or 2 terms. We refer to these as the TNS1 (3.6) and TNS2 (3.7) preconditioners. Note that

$${M^{-1}}_{TNS1} = (I - D^{-1}L^T)D^{-1}(I - LD^{-1}) \tag{3.6}$$

$${M^{-1}}_{TNS2} = (I - D^{-1}L^T + (D^{-1}L^T)^2)D^{-1}(I - LD^{-1} + (LD^{-1})^2). \tag{3.7}$$

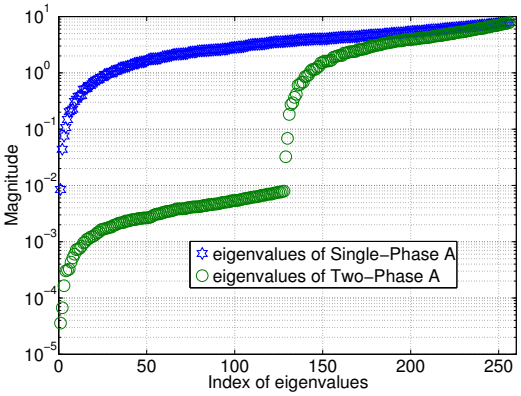


Figure 3.1: 2D grid (16×16) with 256 unknowns. Jump at the interface due to density contrast.

We define $K = (I - LD^{-1})$ for M^{-1}_{TNS1} and $K = (I - LD^{-1} + (LD^{-1})^2)$ for $M^{-1}{}_{TNS2}$. 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 close to $2N$ multiplications and N additions. It must be noted that only L and D^{-1} are stored when applying this preconditioner and K is not explicitly computed or stored.

3.3 Problem definition

We define a test problem (see Figure 3.2 and 3.3) for our preconditioning schemes. We define a unit square as our computational domain in 2D (Figure 3.2). 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 the contrast in densities of the two fluids. This jump is also visible in the eigenvalue spectrum as shown in Figure 3.1. Boundary conditions are applied to this domain as indicated in Figure 3.2. The resulting discretization matrix A is sparse and SPD. It has a penta-diagonal 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 is defined as

$$\| r_i \|_2 \leq \| b \|_2 \, \epsilon, \tag{3.8}$$

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 choose ϵ equal to 10^{-6}. The initial guess, x_0 is a random vector to avoid artificially fast convergence due to a smooth initial error.

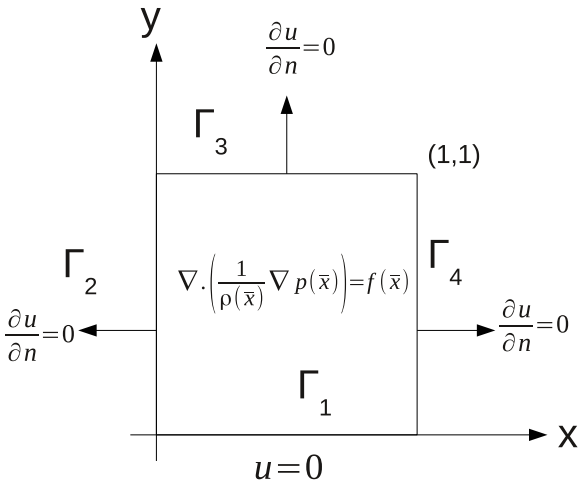


Figure 3.2: unit square with boundary conditions.

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. To this end we also define a test case with a unit cube containing

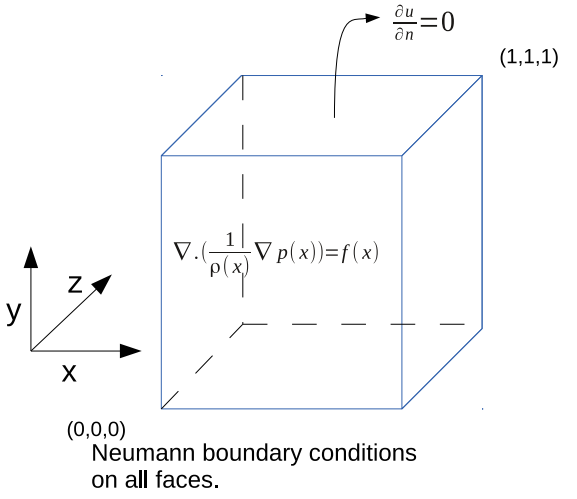
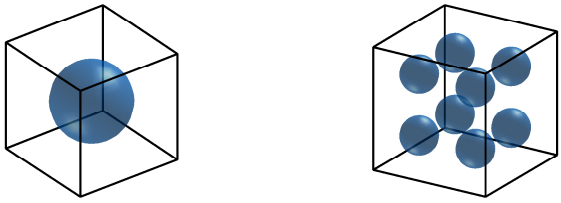


Figure 3.3: unit cube in 3D.

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 3.4 we present two cases where there is a single bubble and 8 bubbles inside the domain presented in Figure



(a) 1 bubble (b) 8 bubble

Figure 3.4: 3 geometries for realistic problems.

3.3. The contrast between the densities of the bubble and the surrounding medium is of the same order as in the 2D problem.

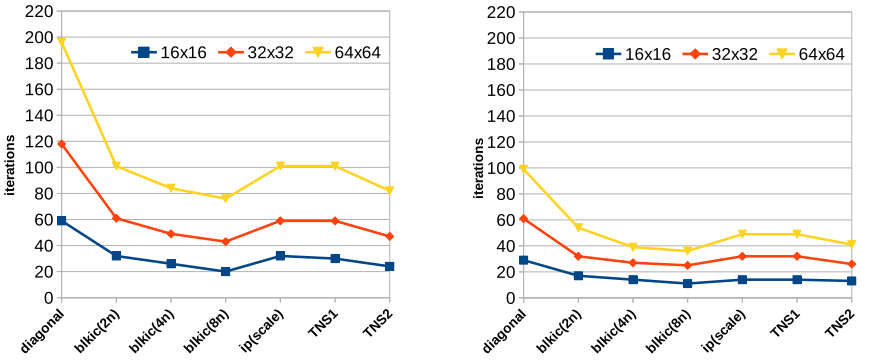
In the 3D case we apply Neumann boundary conditions on all faces. The matrix is SPSD (Symmetric Positive Semi-Definite) and has a septa-diagonal structure. The problem size is $N = n \times n \times n$. We maintain the same stopping criteria, tolerance and initial vector as the 2D problem. The bubbles are placed symmetrically in the test cases (depicted in Figure 3.4) whose results we present in section 3.6.2.

In the text that follows we refer to the Block Incomplete Cholesky Preconditioners (refer section 2.3.4) as $blkic(g)$, where g can be a multiple of n. n is the dimension of the domain in any one direction. So for a 2D grid where the number of unknowns are given by $N = n \times n$ we can have block sizes $n, 2n, \ldots$ etc.. The truncated Neumann series based preconditioning (refer section 3.2.2) is referred to as $TNS1$ and $TNS2$. Jacobi preconditioner is referred to as *diagonal* and we also use a variant of IP Preconditioning (refer section 3.2.1) called as $ip(scale)$.

3.4 Comparison of preconditioning schemes and a case for deflation

We first demonstrate with MATLAB implementations of our DPCG algorithm, how the two levels of preconditioning incrementally work at reducing the number of iterations it takes for the Conjugate Gradient Method to converge. The experiments are done for three different 2 dimensional grids (refer Figure 3.3) with sizes 16×16, 32×32 and 64×64. The ratio of densities for the two fluids modeled is 10^3. The deflation vectors used are stripe-wise (refer section 2.4.2).

Figure 3.5 shows how the number of iterations varies with different preconditioners for the CG method. We note that the Neumann type preconditioners



(a) with preconditioning only

(b) with deflation and preconditioning

Figure 3.5: iterations with two levels of preconditioning.

denoted by $TNS1$ and $TNS2$ are comparable to the Block-IC approaches with block size $4n$ ($blkic(4n)$). We have not shown the results for incomplete Poisson preconditioning (without scaling) here since they are at least 3 times higher than the diagonal preconditioning results. That is the reason why we do not give IP preconditioning any further attention in this thesis. In Figure 3.5 we also notice how deflation effectively halves the number of iterations required for convergence.

3.5 Implementation

We store the matrix A in the diagonal (DIA) format and follow the implementation as detailed in [6] for the sparse matrix-vector product (SpMV). For BLAS operations we use CUBLAS in CUDA 5.0. 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 (we use Meschach library[1] to calculate the inverse) so that the $E^{-1}a_1$ becomes a dense matrix-vector product which can be calculated using the general matrix vector ($gemv$) routine from MAGMABLAS (v1.2)[36] library for the GPU.

2. Using triangular solve routines from the MAGMABLAS (v1.2)[36] library. Specifically we use the $dpotrs$[2] and $dpotrf$[3] functions.

[1] http://homepage.math.uiowa.edu/~dstewart/meschach/

[2] $dpotrs$ - solves a system of linear equations with a factorization previously prepared by $dpotrf$

[3] $dpotrf$ - computes the Cholesky factorization of a real SPD matrix A

We define the degree of parallelism as the number of independent tasks that an algorithm can be broken down into. The first method has a higher degree of parallelism compared to the second. Dense matrix-vector multiplication is an embarrassingly parallel operation on the GPU. On the other hand, in the first method, the 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 (refer Figure 3.13). 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, proves to be advantageous for a GPU implementation.

3.5.1 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 3.6. In Figures 3.6 to 3.8 it must be noted that $d = 2n$ 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 3.6) 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 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 3.8. Each ellipse corresponds to the non-zero part of the corresponding deflation vector in matrix Z. Matrix AZ stored in this way (for the GPU) makes 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 [6]. Different colored elements are grouped together so that the access pattern to calculate each element of a_3 is similar to the Sparse-Matrix 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 3.7. The down pointing arrows in Figure 3.7 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 curved 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 [6].

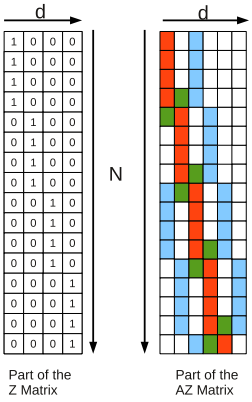


Figure 3.6: parts of Z and AZ matrix. number of deflation vectors $=2n$.

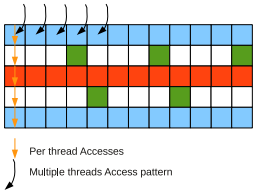


Figure 3.7: AZ matrix after compression.

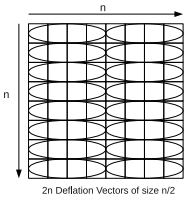


Figure 3.8: deflation vectors for the 8×8 grid.

3.5.2 Extension to real (bubble) problems and 3D

The storage format discussed for AZ in the previous section can accommodate 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 3.9 we provide an example for a 3D scenario in order to explain what planar and stripe-wise vectors look like. We use piecewise constant

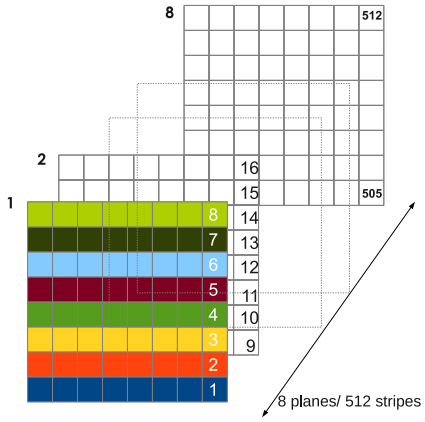


Figure 3.9: Planes and stripes for a 8^3 uniform cubic mesh.

stripe-wise deflation vectors. Every vector has length N. Each vector has ones for the row on which it is defined and zeros 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 zeros). 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 dimension. For our experiments in section 3.6.2 we use n^2 stripe-wise and n planar vectors.

3.6 Experiments and results

We performed our experiments on the hardware available within 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 3 from section 2.4) for completing iterations required for convergence. In our results, speedup is measured as a ratio of this iteration time on the CPU versus the GPU. The setup phase includes the initializing the memory and assigning values to the variables 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 3 in section 2.4.

It also involves the setup for the operation $Ea_2 = a_1$ using either of the two approaches mentioned in section 3.5.

3.6.1 Stripe-wise deflation vectors - Experiments with 2D test problem

For the 2D problem we have used $2n$ deflation vectors unless otherwise mentioned. In Table 3.1 we present the number of iterations required for convergence of different preconditioning schemes across four different grid sizes. The number of iterations is not affected by the choice of implementation for the deflation method discussed in section 3.5. It can be noticed that for nearly all grid sizes the number of DPCG iterations for the second type (TNS2) of Truncated Neumann Series (TNS) based Preconditioner (with $K = (I - LD^{-1} + (LD^{-1})^2)$) are comparable to the Block Incomplete Cholesky (Block-IC) scheme with block size $4n$. These results show that TNS-based preconditioners can be used to replace some variants of Block-IC preconditioner, as they are able to accelerate the convergence of the DPCG method equally well in comparison to the block-IC preconditioners.

	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
${M^{-1}}_{TNS1}$	76	117	129	224
${M^{-1}}_{TNS2}$	61	92	101	175

Table 3.1: Iterations required for convergence of 2D problem using DPCG with $2n$ deflation vectors.

Next in Figure 3.10 we compare the speedup for all flavors of block-IC preconditioning for two deflation implementations. We notice that the speedup for Block-IC variants (of DPCG) with a larger block size (8n) is similar to those with smaller block sizes (2n) for smaller grid sizes but for larger grid sizes the smaller block-size variants have a larger speedup. This observation holds for both cases of implementing DPCG with triangular and explicit solve. However, the overall speedup of the DPCG method for smaller block sizes with explicit inverse based deflation implementation is larger. This shows for the given data that

1. Using more blocks can be beneficial on the GPU.
2. Explicit inverse based solve of the inner system is always faster than triangular solve.

The reason for better performance of Block-IC variants with smaller block-size is the increased fine grain parallelism. However, smaller block-sizes have

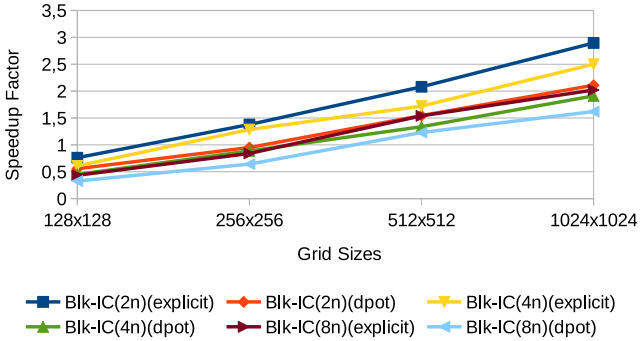


Figure 3.10: Comparison of explicit versus triangular solve strategy for DPCG. Block-IC preconditioning with $2n$, $4n$ and $8n$ block sizes.

the adverse effect of slower convergence. This was visible in Table 3.1. Increasing the number of blocks (and decreasing block sizes) delays convergence since the quality of the preconditioner is degraded due to loss of more information.

In Table 3.2 we present the execution times for DPCG implementations that use explicit inverse for inner system solve on the CPU and GPU for one of the grid sizes which has one million unknowns. We notice that the speedup

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
${M^{-1}}_{TNS1}$	20.15	1.29
${M^{-1}}_{TNS2}$	25.99	1.47

Table 3.2: Wall-clock times for DPCG on a 2D problem with $N = 1024 \times 1024$.

for DPCG implementations with TNS based preconditioners is substantial for explicit inverse based implementation. The reason is that TNS based preconditioners exhibit fine-grain parallelism and hence are very well suited to the GPU. In comparison, for the DPCG implementation which uses Block incomplete Cholesky as the first-level preconditioner, the difference in speedup between the two deflation implementations to compute coarse grid solution ($Ea_1 = a_2$) as mentioned in the beginning of section 3.5 is relatively low (Figure 3.10). 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.

In contrast, the choice of coarse system solve in the deflation step be-

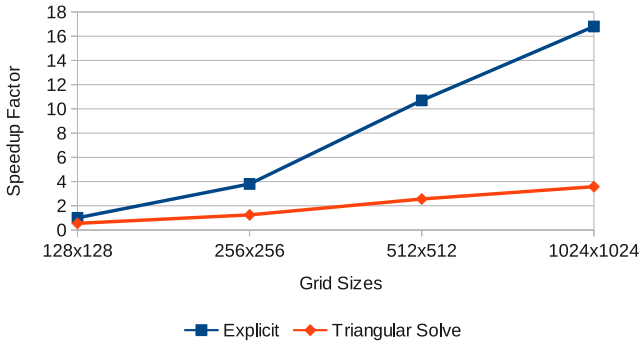
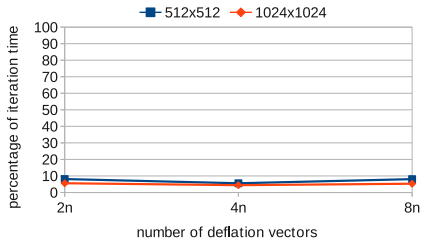
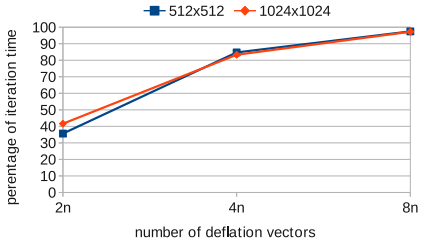


Figure 3.11: Comparison of explicit versus triangular solve strategy for DPCG. Neumann series based preconditioners $M^{-1} = K^T D^{-1} K$, where $K = (I - LD^{-1} + (LD^{-1})^2)$

comes decisive in the length of execution time for DPCG implementation with TNS-based first-level preconditioners (refer Figure 3.11). As the number of unknowns increase the solution of the inner system becomes increasingly time consuming with triangular solve. However, for the explicit inverse case there is an increase in parallelism with increase in size of the inner system and the resources of the GPU can be better utilized. 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.



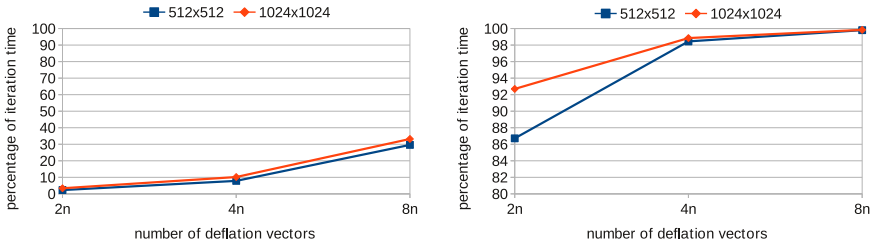
(a) CPU



(b) GPU

Figure 3.12: Setup time as percentage of the total (iteration+setup) time for triangular solve approach across different sizes of deflation vectors for DPCG.

In Figure 3.12 and 3.13 we compare the setup time as a percentage of the total time required to solve the linear system for three different deflation vector sizes across two different grid sizes. As one may expect, for a large number of deflation vectors the explicit inverse based scheme becomes very expensive on the GPU. This is also true for the triangular solve case since it is mostly sequential. For the CPU version of the code however in both cases



(a) CPU (b) GPU

Figure 3.13: Setup time as percentage of the total (iteration+setup) time for explicit E^{-1} approach across different sizes of deflation vectors for DPCG.

there is little variation in the percentages. This is because on the CPU the solution time is much larger than the GPU.

3.6.2 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 (refer section 2.4.2)). Block vectors do not suit the storage pattern that we have utilized for this study but they give good results. In Table 3.3 and 3.4 we see the results for a case when we have a 3D geometry. For the first set of results presented in Table 3.3 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 a unit cube. We present the results with n plane and n^2 stripe-wise deflation vectors for the GPU. 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.

The results of Table 3.3 show that the speedup is reduced and is absent for the case with 16384 deflation vectors. 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 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 at all. The iteration times are high since we use the triangular solve method for inner system. This is because, with the explicit inverse based inner solve we cannot solve up to

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

	CPU[1]	GPU[2]	
	8 block vectors	128 plane vectors	16384 stripe vectors
	DICCG(0)	DPCG(TNS2)	
Number of iterations	206	324	259
Setup time	0.3	0.36	148.5
Iteration time	35.18	7.66	112
Speedup	-	4.59	–

Table 3.3: $3D$ 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.

the tolerance of 10^{-6}. To find out the reason why the explicit inverse based solve doesn't converge for this tolerance we plotted the relative norm of the residual (Figure 3.14) for the solution of the layered problem with 3 layers (results shown in Table 3.4) we notice how the explicit inverse based solution cannot handle lower tolerances and begins to diverge.

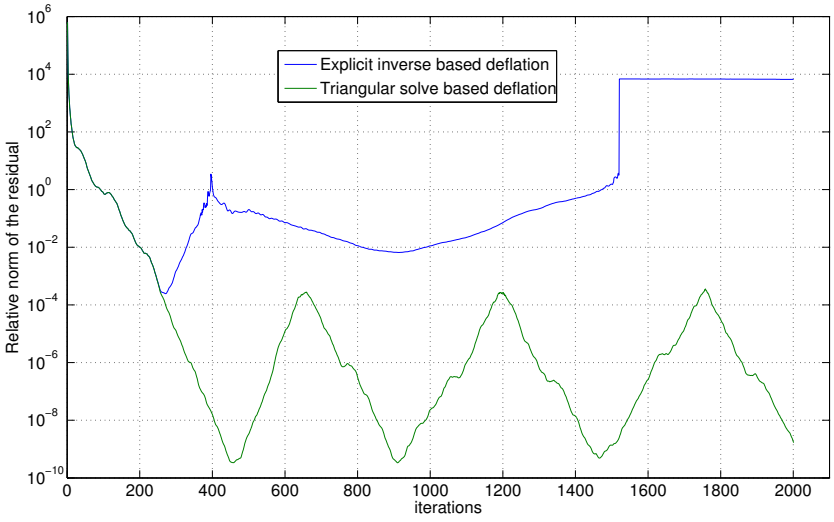


Figure 3.14: Relative norm of the residual across two deflation implementation techniques for a 128^3 3D grid with 3 layers with contrast in material densities.

We also plot the relative norm of the residual (Figure 3.15) for the inner (coarse) systems that are solved using the two different solution methods.

We can notice that the relative norm of the residual for the inner system can never go below 10^{-7} whereas for the triangular solve based inner system

solve the relative norm of the residual becomes machine precision (10^{-16}) after a little over 450 iterations and it also fails to converge for higher tolerances.

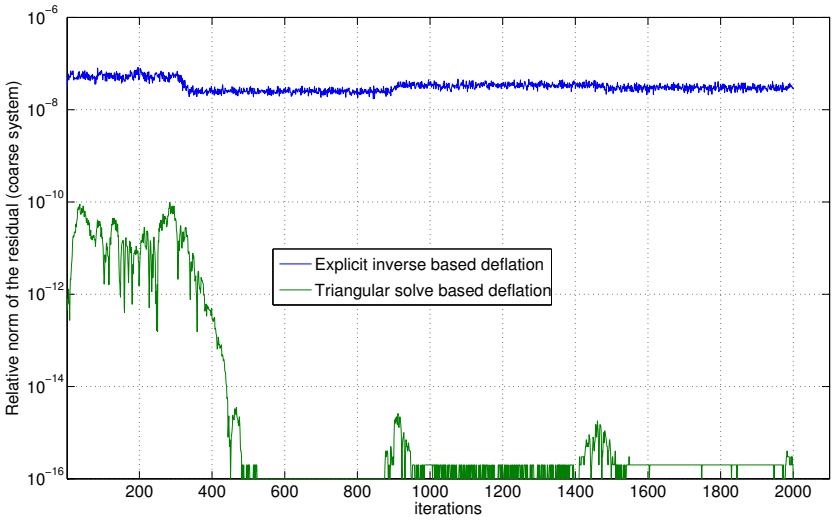


Figure 3.15: Relative norm of the residual (*coarse system*) across two different deflation implementation techniques for a 128^3 3D grid with 3 layers with contrast in material densities.

In results presented in Table 3.4 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 (Figure 3.4). 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.

We only show the results with n vectors in Table 3.4 since with n^2 vectors there is no speedup. In Tables 3.3 and 3.4 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 and convergence is never achieved. One explanation for this is that in the case of inverse calculation the Meschach library uses an LU decomposition followed by multiple forward and back substitution steps to form the individual columns of the inverse matrix. So there are many

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

1 bubble		
	CPU[1]	GPU[2]
	8 block vectors	128 plane vectors
	DICCG(0)	DPCG(TNS2)
Number of iterations	237	287
Setup time	0.31	0.64
Iteration time	40.44	6.79
Speedup	-	5.95
8 bubbles		
Number of iterations	142	402
Setup time	0.3	0.36
Iteration time	24.4	9.51
Speedup	-	2.56

Table 3.4: $3D$ 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.

more additions and multiplications where the chance of round-off error to add up, is increased. However, in the case of the MAGMABLAS routines which use triangular solve an LU decomposition is calculated of the symmetric positive define matrix E and only one solve is done every iteration to arrive at the solution for the coarse system.

3.7 Conclusions

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 suitable to the GPU. At the same time we have made new data structures in order to optimise deflation operations.

With our results we demonstrate that the combination of Truncated Neumann series based preconditioning and deflation proves to be computationally efficient on the GPU. At the same time the number of iterations it takes to converge are also comparable to the method of Block-incomplete Cholesky preconditioning which is not suitable for GPU implementation.

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. The main drawback we faced in using stripe-wise vectors was that a lot of them must be used in order the DPCG method converged in fewer iterations. This means a larger E matrix, whose inverse calculation takes most of the time. In the subsequent chapters we show how to extend our work to 3D

problems with bubbles. We continue using the approach of calculating the inverse of the matrix E explicitly. We also saw that using stripe-wise vectors for 3D problems required using the triangular solve based inner system solve since with explicit inverse based solve, convergence was not possible for the desired tolerance (10^{-6}).

In the next chapter we employ better deflation vectors based on the underlying physics of the problem in order to overcome the possibly large setup time and to avoid delayed convergence. For bubbly flow we use level-set subdomain deflation. Using these deflation vectors for bubbly flow we can better approximate the eigenvectors corresponding to the small eigenvalues in the spectrum of $M^{-1}A$ which we wish to project out using deflation. This overcomes the slow convergence we witnessed for 3D domains as mentioned in section 3.6.2. A small number of these vectors can capture the small eigenvalues and result in an effective deflation step (this is discussed in [29] and [65]). This directly translates into a low setup time and overall gain in this approach of implementing deflation.

CHAPTER 4

Improving deflation vectors

4.1 Introduction

In the previous chapter we have seen how deflation can be used to improve the performance of PCG. Moreover, using new data structures that expose the fine grain parallelism on the GPU can make deflation computationally efficient.

In this chapter we propose deflation vectors that are better than stripe-wise vectors at accelerating convergence for 3D problems (section 3.6.2). We propose using block-wise sub-domain deflation vectors and combine them with level-set information to make effective deflation vectors. Throughout this chapter we consider the problem of bubbly flow and use deflation vectors that suit this problem.

4.2 Problem definition

For our experiments we consider a problem defined on a unit cube like in the previous chapter (Figure 3.3). Neumann boundary conditions are applied on all sides of this cube. In addition to the 8 bubble problem we used for

The work presented in this chapter also appears in:

R. Gupta, M. B. van Gijzen, and C. Vuik. 3D bubbly flow simulation on the GPU - iterative solution of a linear system using sub-domain and level-set deflation. In *Proceedings of PDP 2013*, pages 359–366. IEEE CPS, 2013.

our experiments earlier (Figure 3.4(b)) we also consider a 9 bubble problem (Figure 4.1). For deflation vectors we use block-wise sub-domain, level-set and level-set sub-domain (piece-wise constant) deflation vectors.

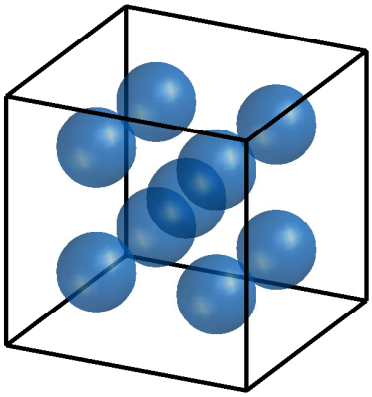


Figure 4.1: 9 bubbles in a unit cube

4.3 Block-wise sub-domains based deflation vectors

We have previously described block-wise sub-domain deflation vectors in section 2.4.2 in the background section of this thesis. The block-based sub-domain vectors are different from stripe-wise vectors in the way they are constructed. For a unit cube, block-based sub-domain deflation vectors are constructed by dividing the unit cube into smaller cubes.

In this section we enumerate two restrictions that must be adhered to in order to ensure effectiveness of block-based sub-domain deflation vectors. Firstly, it must be ensured, whenever possible, that at most one bubble or a part of it is captured by a single sub-domain. If this does not hold e.g. when two bubbles intersect with a sub-domain; then even after application of the deflation operation small eigenvalues will persist in the spectrum of A or $M^{-1}A$.

Another guideline to ensure effective sub-domain deflation vectors for bubbly flow is to ensure the matrix E is not singular. While making sub-domain vectors it is possible that we end up with a Z matrix that causes the coarse system matrix E to be singular. This is possible e.g. in a case when A has a Laplacian stencil and Neumann boundary conditions are imposed on the domain of discretization. A singular E will no longer admit the calculation

of E^{-1}. To remedy this problem we can make a Z and then remove one column from it. This renders E non-singular and an inverse can be calculated. Consequently, we can still solve the inner system using an explicit inverse.

4.3.1 Level-set deflation vectors

To construct level-set deflation vectors, we utilize the information in the level-set function [68]. The level-set function is a signed distance function. This means that the function takes, say for bubbly flow, a positive value inside a bubble and a negative value outside of it. Using this information we can mark the interfaces and this information can be used to make better deflation vectors.

These vectors are piece-wise constant (section 2.4.2). If there are k bubbles in the system then we can define k vectors z_k which can be assembled to form Z. The vectors have a constant (non-zero) value over the part of the domain on which the bubble is defined and zeros elsewhere. For details about this technique for assembling matrix Z we refer the interested reader to [62].

Level-Set Sub-Domain (LSSD) vectors

It is possible to extend the level-set vectors with sub-domain vectors and improve their effectiveness for solving the bubbly flow problem. We can define

$$Z_{LSSD} := [Z_1, Z_2], \tag{4.1}$$

Z_1 consists of all sub-domain vectors of Z_{SD} (Z matrix containing sub-domain deflation vectors) 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}. It utilizes the information from Z_{LS}, which is the matrix containing level-set deflation vectors. The Z_{LSSD} matrix may cause the inner system to be singular and therefore dropping a column of Z_{LSSD} could be used to ensure we can calculate the inverse of E. More details on this method of creating level-set sub-domain deflation vectors can be found in [62].

Stripe-wise vectors

Stripe-wise vectors (Figure 2.4(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 [31] and in the previous chapter). However, this directly translates into a larger Galerkin matrix E and a consequent bottleneck in the calculation of E^{-1}. In this chapter we do not use these vectors.

4.4 Using the explicit inverse for the solution of the coarse system

In order to solve the coarse system that appears while applying deflation we have to choose a solution strategy. In the previous chapter we saw that for stripe-wise vectors explicit inverse based solution of the coarse system imposed serious restrictions on the kind of tolerance to which we solve the linear system iteratively using the DPCG method. This is evident from the plot of the relative norm of the residual for the inner system shown in Figures 3.15. The relative norm of the residual for the explicit inverse based solve cannot be less than 10^{-7} when stripe-wise vectors are used.

In Figure 4.2 we present the results for the relative norm of the residual for the inner system when block-wise vectors are used. The problem chosen for the plots discussed is the 9 bubble problem mentioned in section 4.2. The deflation vectors used are block sub-domain type. It must be noted that for level-set sub-domain type vectors the plots are similar.

In Figure 4.2 we see a much lower relative norm for the residual for both explicit and triangular solve based deflation implementations.

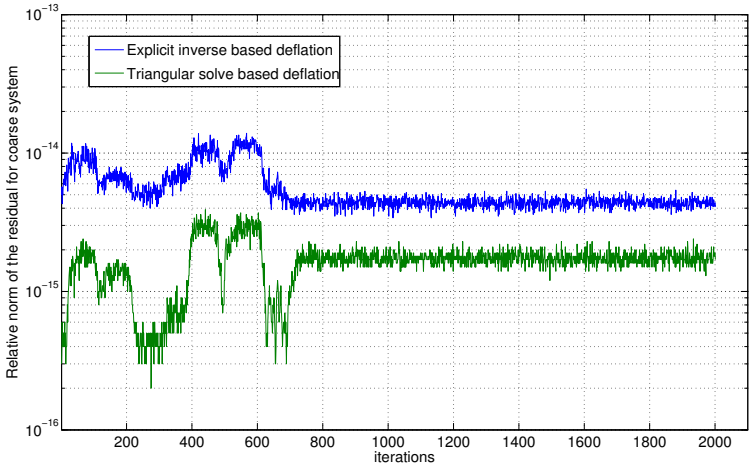


Figure 4.2: Comparison of the relative norm of the residual for inner system across two deflation implementations.

The performance of the explicit inverse is better with block-based sub-domain deflation vectors (or level-set sub-domain deflation vectors) as we have less vectors and hence a reduced setup time to calculate the explicit inverse. Further, the solution of the coarse system is quicker because of the highly parallelizable dense matrix vector product. Finally, the method converges

faster as the block sub-domain based deflation vectors augmented with level-set information are a better approximation of the deflation subspace. This is the reason why we use explicit inverse based solution for the coarse system for the experiments discussed in this chapter in combination with block sub-domain vectors.

4.5 Experiments and results

In this section we present results of experiments we conducted and comment on the results obtained. The hardware we use is 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 Xeon CPU from Intel. The CPU version of the code is highly optimized to reduce operations and computationally speedup the DPCG algorithm (more details in the appendices of [62]). All calculations are done in double precision.

4.5.1 Notes on implementation

We use MAGMABLAS (v1.2.1) [1] library for the *gemv* operation and for triangular solve (*dpotrf* and *dpotrs*), depending on how we solve the inner system.

We use CUSP(v0.3.0) and CUSPARSE (with CUDA v5.5) in our implementations on the GPU. This is in contrast to our software using which we presented results in the previous chapter. The motivation for this change is the structure of the Z matrix based on block sub-domain and/or level-set vectors. The level-set vector is used to extract the position of the interfaces, which in a time-dependent problem will change every time step when the linear system (1.1) is solved. This change is reflected in the storage of the AZ matrix which can now have non-zeros which do not have a regular pattern as was shown in Figure 3.6 with stripe-wise vectors. This means that the Sparse Matrix-Vector (SpMV) routine for multiplying AZ with a vector has to accommodate a generic sparse matrix storage format (e.g. CSR, HYB etc.). Generic storage formats are also required for Z and correspondingly sparse matrix-matrix multiplication routines are also required with the changes in Z. CUSP and CUSPARSE provide optimized routines for these operations.

We use different storage formats in order to extract the best performance for the SpMV routine corresponding to a specific matrix. Particularly for CUSP we store

1. A in DIA,

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

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.

We use the CUSP[2] library for GPU implementation of the DPCG method. CUSP provides BLAS routines that are based on the THRUST[3] library for GPUs. For CUSP implementations the GPU code runs and stays entirely on the GPU except for the following steps in the CG algorithm, namely,

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.

Since these operations only involve sending back one scalar value to the host (CPU) and potentially do not make a significant contribution to the complete execution time we have not tried to write our own kernels for dot products and/or residual norm calculation and have used standard routines in the CUSP library. As mentioned earlier in section 3.5 there are two options to solve the inner system. We will only present results for deflation implemented using explicit inverse of E. 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 this case the triangular solve and explicit inverse method have the same computational times.

4.5.2 Differences between CPU and GPU implementations

On the CPU we have a FORTRAN code that runs the DICCG(0) algorithm for solving the same problem that we solve on the GPU with DPCG and TNS2 preconditioner for the first level. On the CPU only sub-domain deflation vectors are used whereas on the GPU we show results with level-set and level-set sub-domain deflation vectors also. The reason for no level-set deflation on the CPU is that we have a FORTRAN code that was developed as a part of the work in [62] and only sub-domain vectors were dealt with in that software. Significant optimizations to reduce storage space and increase computational efficiency were implemented in this software (refer appendices of [62]).

[2] http://cusplibrary.github.io/
[3] http://thrust.github.io/

Our focus has been to develop a new DPCG implementation (especially for the GPU), and therefore we did not implement the level-set sub-domain deflation vectors in the FORTRAN code. All of the vectors on the CPU and GPU are piece-wise constant. On the GPU for the coarse system solve we use the explicit inverse of E whereas on the CPU the inner system is solved using the CG method, which is the standard method used when in the CPU code. The choice for using explicit solves stems from the fact that on the GPU we can exploit the parallelism in the solution of the inner system. On the GPU, solving the inner system iteratively using CG takes more memory and computational time than the explicit inverse based option so we do not use CG to solve the inner system.

We note, however, that for larger number of vectors ($O(n^2)$) it might prove useful compared to the explicit E^{-1} option. On the CPU using CG makes the solution of the inner system immune to the singularity in E. It must also be said that in the CPU version of the code (written in FORTRAN) the focus was not to parallelize but to research the properties of deflation.

The tolerance for the inner system is kept lower in comparison to the outer tolerance which is 10^{-6} unless otherwise mentioned.

4.5.3 Results

In this section we present the results of two-level preconditioned CG for two representative geometries mentioned in Figure 3.4(b) and 4.1. These geometries are instrumental in capturing the effect of different deflation vector choices.

Speedup and computing times

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), LD^{-1}, $D^{-1}L^T$ (for TNS2 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 is integrated into the full software for the simulation of bubbly flow using the Level-set approach.

In all our experiments we have a 3D grid with 128 grid points 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.
    - DPCG(TNS2) - refers to DPCG with the TNS2 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. j has been calculated according to the information given in sections 4.3 and 4.3.1. In subsequent sections we also have LS-j which has been constructed on the basis of the discussion in section 4.3.1.

Eight Bubbles

We consider 8 bubbles placed symmetrically inside the 3D unit cube (Figure 3.4(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 4.1: 8 bubbles. CPU implementation.

Further we discuss the implementation on the GPU for the problem where the inner system, (2.34b), is solved with an explicit inverse of E using a *gemv* operation.

From Table 4.2 we can see how level-set sub-domain deflation can be effective at accelerating convergence but takes more time per iteration. 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.

	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 4.2: 8 bubbles. Comparison of deflation vector choices on the GPU (CUSP & CUSPARSE based implementation).

For CUSPARSE results in Table 4.2 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.

Nine bubbles

The 9 bubble case (from Figure 4.1) is an extension of the scenario presented in the previous section (4.5.3). 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 4.1) 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 4.3 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(TNS2)		
	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 4.3: 9 bubbles. Comparison of deflation vector choices for deflation on the GPU (CUSP based implementation) vs. CPU.

For GPU results in Table 4.3 we can infer that the level-set vectors alone can accelerate convergence, but level-set sub-domain vectors are better.

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. This is because level-set vectors stay the same since the number and position of the bubbles do not change.

The tolerance had to be made bigger 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 4.3. We observe how speedup for these implementations changes. Level-set sub-domain GPU versions are the most effective.

Looking at Table 4.4 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 (2.34b) has greater data parallelism to exploit. level-set sub-domain based vectors better approximate the deflation subspace so the iteration counts also go down.

	CPU	GPU-CUSP	
	DICCG(0)	DPCG(TNS2)	
	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 4.4: 9 bubbles. Two deflation variants. GPU and CPU execution times and speedup. 64 sub-domains.

In table 4.5 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 4.5.3). It is also interesting to note that the speedup in Table 4.5 solely reduces (when compared with 64 vectors case in Table 4.4) 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(TNS2)	
	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 4.5: 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 4.6 we show the new execution times for the CPU results presented in Tables 4.3, 4.4 and 4.5 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.

	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 4.6: 9 bubbles. CPU versions of DPCG with 8, 64 and 512 vectors with OpenMP acceleration.

In Table 4.7 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 4.6.

In Figure 4.3 we compare the speedups that we achieve for a sequential CPU code (from Table 4.7) (running on one core of the dual quad core CPU) versus the OpenMP version(from Table 4.6) as compared to the GPU implementation presented in Tables 4.3, 4.4 and 4.5. On the Y-axis in Figure 4.3 the CPU version / GPU version of the code are mentioned. The speedup figures change by at most 25% by the use of OpenMP (8 threads) because in the CPU version the Incomplete Cholesky (IC) preconditioning is the bottleneck.

	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 4.7: 9 bubbles. CPU versions of DPCG with 8, 64 and 512 vectors without OpenMP acceleration (on single core of a dual quad core).

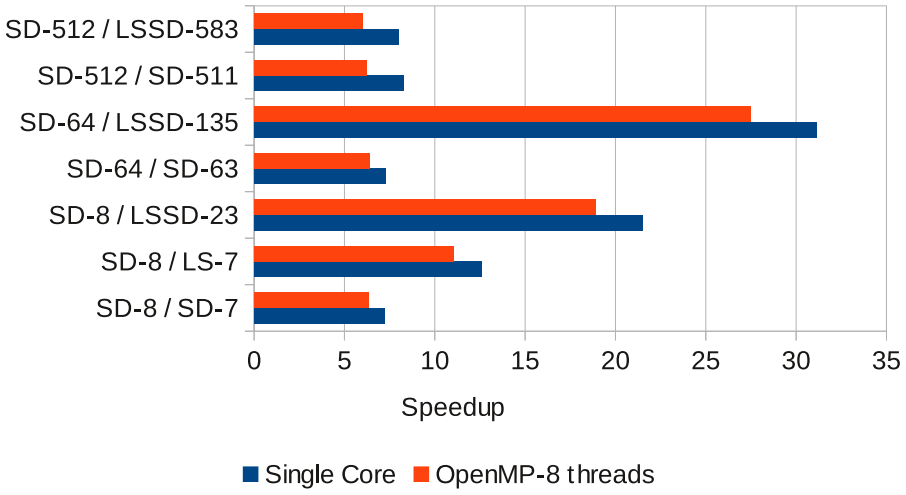


Figure 4.3: Comparison of speedup on the GPU with respect to the CPU with openMP parallelization

4.6 Conclusions

Our results show that with suitable deflation vectors derived from sub-domain based on blocks and level-set function can substantially reduce the setup time for calculating E^{-1} (since the number of vectors is small). Furthermore, these vectors better approximate the eigenvectors corresponding to the small eigenvalues which delay convergence. Therefore, the deflation step is effective in accelerating convergence. Deflation implemented in such a way in combination with TNS2 preconditioning for the first level can deliver up to 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.

In the next chapter we show how to further optimize the code so that setup

times can be reduced. In the subsequent experiments we 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.

CHAPTER 5

Extending DPCG to multiple GPUs and CPUs

5.1 Introduction

In the previous two chapters we have shown that deflation can be an effective choice to accelerate the preconditioned Conjugate Gradient method on the GPU. However, with increasing problem sizes more memory is required on the GPU. To overcome this limitation we must think of parallelization of the algorithm on multiple compute units (CUs). We define a compute unit as a single core of a multi-core CPU or a single GPU.

Parallelizing an algorithm on multiple CUs requires communication between the CUs. This communication adds to the execution time of the algorithm. As the number of CUs increases one has to make sure that the rising costs of communication can be kept under control.

In this chapter we show how we have parallelized the DPCG method on the GPU. We present the results on two different clusters where we tested our

---

The work presented in this chapter also appears in:

Rohit Gupta, Martin B. van Gijzen, and Cornelis Vuik. Multi-GPU/CPU deflated preconditioned conjugate gradient for bubbly flow solver. In *Proceedings of the High Performance Computing Symposium*, HPC '14, pages 14:1–14:8, San Diego, CA, USA, 2014. Society for Computer Simulation International.

implementation.

On the first cluster (DAS-4[1]) which has less GPUs we tested two partitioning strategies and discuss in detail about the more effective strategy. We outline the implementation details that we had to take care of while developing this multi-GPU, multi-CPU implementation with MPI.

On the second (Cartesius[2] cluster at SURFSARA[3]) cluster with more GPUs and multiple GPUs per node we experiment with different storage formats for the coefficient matrix and report the advantages of using one storage format over the other in a multi-GPU/CPU implementation. We compare the performance difference when one or more GPUs per node are used.

Our results on the first cluster consist of the execution times for the DPCG method with TNS-based preconditioning for the first level and deflation for the second level of preconditioning. For the results on the second cluster we also present results with diagonal preconditioning based DPCG and compare them with their TNS-based counterpart.

5.2 Problem definition

Throughout this chapter we consider the bubbly flow problem. Our implementation has two parts. The first part is the calling software and the second one is the linear system solver. The calling software is a software implementation of the Navier Stokes equations which are solved in discrete time steps. Every time-step, a linear system arising from the discretization of the pressure correction equation must be solved. The solution of the linear system is the most time-consuming part of the solution as it takes up to $70 - 80\%$ of the time. In this chapter we use two different calling softwares and for each calling software we use a specific data division scheme which we will explain in the next section. The calling softwares have been written in FORTRAN. The first one is the same code that was used in the previous chapters [62] and the second calling software forms the basis of the work of [45]. We have replaced the linear system solver in both these calling softwares with our preconditioned iterative solver that runs on multiple CUs and uses a particular data division.

We note the benefits of the level-set sub-domain approach as outlined in the previous chapter but in this chapter we use only sub-domain deflation vectors for all the experiments (on both clusters). The reason for this choice is a simpler software design that can also be implemented easily. We have focused exclusively on implementing our linear solver in a multi-GPU/CPU environment using MPI communication layer. Our effort has been dedicated

[1] http://www.cs.vu.nl/das4/

[2] https://surfsara.nl/systems/cartesius/description/

[3] https://surfsara.nl/

to finding out the merits of the data divisions, storage formats, overlapping communication and computation and using multiple GPUs (or multiple CPU cores) per node.

The problem discussed in the first part of the results (Section 5.5.1) on the DAS-4 cluster is still defined on a unit cube as introduced in Section 3.3 (Figure 3.3). Nine bubbles (Figure 4.1) are present inside the cube as mentioned previously in Section 4.2. Neumann boundary conditions are used on all faces of the cube. All other parameters (except for the problem size) remain the same as in the problem defined in Section 4.2. The calling software used is from the work of [62].

The problem which we solve on the Cartesius cluster in the second part of the results (Section 5.5.2) is defined on a cuboid and uses the calling software from [45]. This is mentioned in the subsequent Sections (Section 5.5.1) along with the bubble arrangement (Figure 5.4). Periodic boundary conditions are used for this problem.

In both problems the density contrast is 10^3 between the bubbles and the rest of the medium.

5.3 Data divisions

We examine two different approaches for distributing data amongst the CUs to achieve load balancing.

5.3.1 Division by rows

In a division by rows, we divide the coefficient matrix and vector amongst the CUs. The vectors are divided into $\frac{N}{p}$ chunks where N is the number of unknowns and p is the number of CUs. It is required for the vector to have a ghost region or halo cells around it in order to account for the columns of the matrix that lie outside the $\frac{N}{p}$ column range (because of the off-diagonal values). In this division there can be three specific cases for inclusion of these ghost regions in the vector. For the first processor the halo cells are at the end of the $\frac{N}{p}$ rows of the vector. For the p^{th} processor this region is in the beginning and for CUs in between it is on both sides of the vector. Figure 5.1 shows an example of these regions for the p^{th} processor.

5.3.2 Division by blocks

In a division by blocks we divide the grid composed of individual cells in 3D. It can be broken down into cuboids which can be distributed to individual CUs. Furthermore, local coefficient matrices can be constructed which are then appended with halo values for the cells that lie on the boundaries of

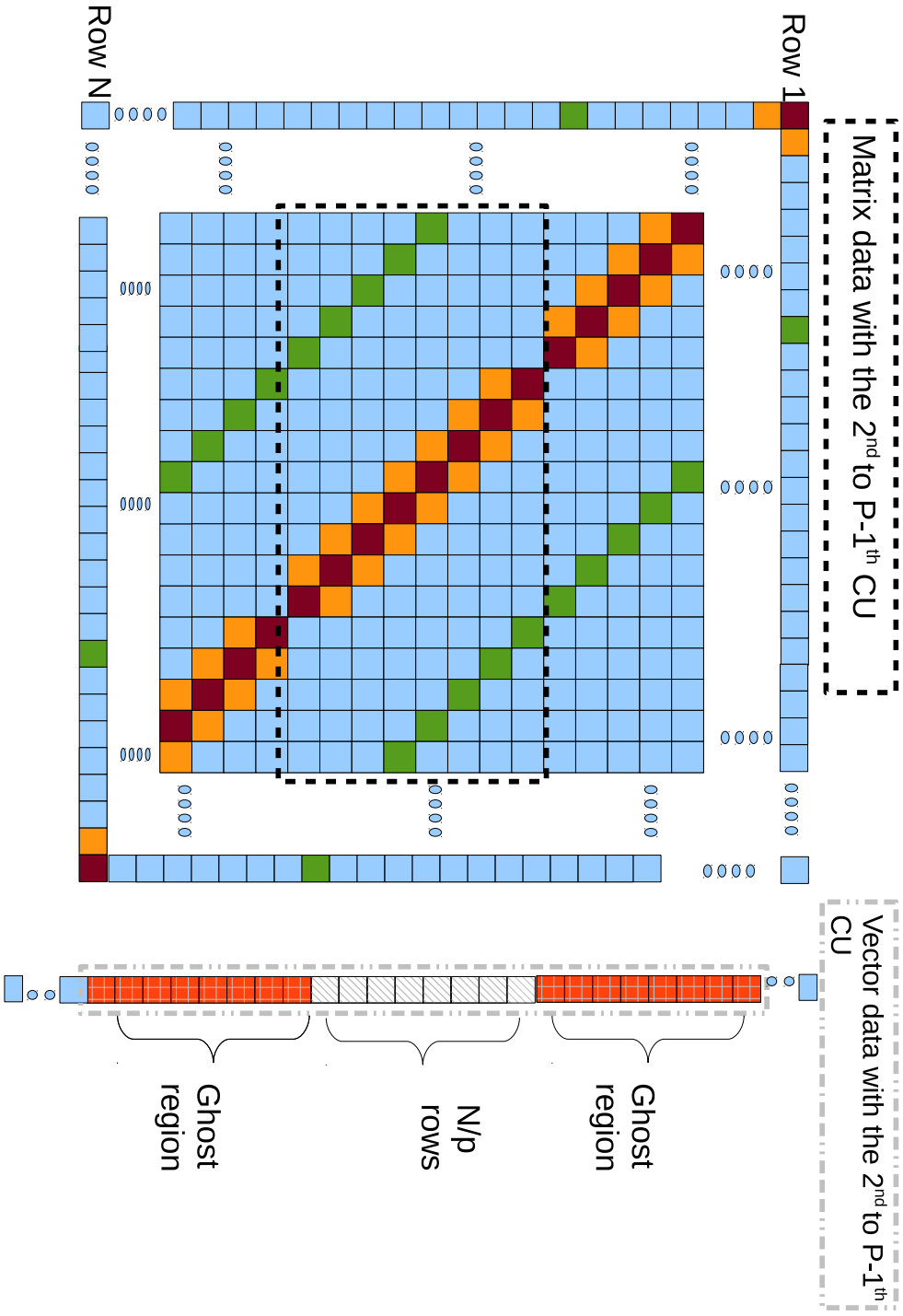


Figure 5.1: Data layout for CUs 2 to P-1.

the cuboid. In this division the vector also needs to be supplemented with values from neighboring CUs each and every time a matrix-vector operation is performed.

5.3.3 Communication scaling

The technique mentioned in Section 5.3.1 is simpler to construct than the one in Section 5.3.2, however, it involves a serious cost as the problem size increases. We briefly explain it here.

The Sparse-Matrix Vector Product (SpMV) operation can be used to understand the difference in behavior for these two data division with respect to scaling. We calculate communication costs involved in this operation to provide an estimate of how this cost changes with increase in problem size and number of processors for the two data divisions. In the block division scheme of Section 5.3.2 the communication happens on all the faces. Each face is of dimension $N^{\frac{1}{3}}$, where $N = n^3$, (for $3D$ case) where n is the size of the grid in any one dimension. If we assume for simplicity that the number of processors in each dimension is $p^{\frac{1}{3}}$, where p is the total number of processors amongst which the problem is divided. Then, the number of values exchanged at any face of the cube/cuboid of the problem with each processor with another adjacent processor is $\frac{N}{p}^{\frac{1}{3}^2}$. For the 6 faces of the cube it would be a multiple of 6 times this number. Since it depends on p, this division of work scales better with increasing p.

In the case of row-based division of work, the order of communication is always $N^{\frac{2}{3}}$ per block so it does not scale with increasing number of CUs.

5.4 Implementation

In this section we explain some of the choices we have made in implementing the algorithm with two different data division schemes. In both codes on the GPU as well as on the CPU (and across both cluster implementations) the same algorithm is used. As a reference we present the (abridged) Deflated Preconditioned Conjugate Gradient (DPCG) Algorithm for a single compute unit (complete Algorithm is given in [62]). In our multi-GPU/CPU implementation of this algorithm, a number of synchronization and communication steps can be recognized. In Algorithm 4 we mark steps that involve global communication with cyan and those involving nearest neighbor communication with magenta. Specifically, SpMV operations involving A, L or L^T involve nearest neighbor communication and dot products, verification of termination criteria and calculation of vector a_1 (from (2.34a)) require global communication.

In the next section we give information about the calling softwares for both data divisions.

Algorithm 4 Deflated Preconditioned Conjugate Gradient (abridged)

1: Select x_0. Compute $r_0 := b - Ax_0$, $r_0 = Pr_0$
2: **for** i:=0,..., until convergence **do**
3: Solve $Kw_{i-1} := r_{i-1}$ // *Preconditioning*
4: $\rho_{i-1} = {r_{i-1}}^T w_{i-1}$
5: Calculate β and p_i.
6: $q_i = PAp_i$ // *Deflation*
7: $\alpha_i = \rho_{i-1}/{p_i}^T q_i$
8: Update x and r.
9: Check convergence.
10: **end for**
11: Correction step.

5.4.1 Calling software and solver routine

In this section a description of the implementations of the different steps in Algorithm 4 are given. Some of these are common to both data divisions and others are implemented differently for both divisions. The calling softwares for block-based and row-based data divisions are FORTRAN codes which have been written for simulating the Navier-Stokes equations to describe bubbly flow. One of them is based on the work in [45] and the other on the work in [62]. We use the software developed in [62] to call the row-based solver and the software that formed the basis of the research in [45] is used to call the block-based DPCG linear system solver. Through these experiments we want to experimentally verify the effect of the data division schemes on multi-CPU and multi-GPU implementations. Both calling software pieces provide a coefficient matrix, right-hand side and an initial guess to the linear system solver which then applies DPCG to arrive at a solution. However, there are still some differences in how these three quantities are passed to the linear solver and that makes its software design significantly different from each other.

In one of the calling softwares the division of data is done first and then the solver is launched whereas in the other the data division is implicit. We provide more details in the following subsection.

Row-based division

In case of row-based division the entire coefficient matrix A, and the right-hand side b, are generated in a FORTRAN code that solves the Navier-Stokes

equations for fluid flow. The FORTRAN software passes these variables to the C/CUDA code. The matrix and the variables are then divided into parts and sent to all compute units (including the originating compute unit). Using these parts and with some communication amongst the CUs, parts of the matrices (AZ, AZ^T, E^{-1}, Z, L and L^T) are setup on each CU. This constitutes the setup phase of the solver with row-based data division.

The deflation vectors in the code having row-based data division are of the sub-domain kind (for details refer Section 2.4.2, [62] and [64]).

Block-based division

In case of block-based division also the GPU/CPU solver is called as a function from FORTRAN. However, the FORTRAN code is already running on multiple compute units and each compute unit receives a part of the coefficient matrix and the right-hand side relevant to it. So each node can immediately calculate its local part of the matrices like in the row-based case with lesser communication. Hence, the setup phase in the block-based division is shorter (and computationally cheaper) compared to the row-based for the same number of unknowns. The FORTRAN code provides a coefficient matrix A with the external coefficients embedded in it. These coefficients are stored in separate arrays once and then can be used every time an SpMV operation is to be performed using A, L or L^T.

It must be noted that in both cases while making the matrix E^{-1} (for deflation since we solve the coarse grid using the `gemv` operation) some global communication is required since each CU has a part of E (since they have a part of A and Z and $E = Z^T AZ$) that is generated independently. With an `MPI_Alltoall` communication step, E is aggregated and its inverse is independently calculated by each CU. The deflation vectors in the block-based code are of the sub-domain variety (refer Section 2.4.2, [64], and for details refer [62]). Also, the number of these vectors is one less than the number of the CUs, e.g., if there are 8 CUs used to solve the problem then there are only 7 deflation vectors. Discarding one deflation vector is required since otherwise the matrix E will be singular for the problem with Neumann boundary conditions. This in itself may not be a problem (if the inner system is solved using the CG method) but since we solve the inner system using the explicit inverse of E we need an invertible E. Removing one vector ensures non-singularity of E for our problem.

The choice to make one deflation vector (sub-domain) per CU was made to simplify the deflation operation. If there is only one sub-domain on each cuboid which is the domain of a single CU, the operation AZ is a simple summing of rows of A. Depending on the boundary conditions and also because of the off-diagonal coefficients belonging to the rows corresponding to the cells on

the outside boundaries of each CU's cuboid, there are non-zero elements in the rows of AZ.

The multiplication of $Z^T \times AZ$ is a matrix-matrix multiplication which can be implemented easily as compared to the case when the sub-domains span over a portion of the grid that belong to more than one processor.

The solver routine in both data divisions for GPU and CPU execution makes use of libraries. On the CPU we use the MKL library and on the GPU we use CUBLAS and CUSPARSE. We decided to use them instead of CUSP which was used in the previous chapter since CUSPARSE provides a better ecosystem (in terms of support and knowledge base) for maintaining and further developing our software.

The BLAS operations and SpMV operations are handled by these libraries and hence the data available from the FORTRAN code has to be translated into appropriate data structures which the libraries can interpret.

5.4.2 Communication outline in multi-compute unit implementation

In the DPCG algorithm distributed over CUs, communication can be of two varieties. Global communication and communication with nearest neighbors. When calculating dot products in steps 4 and 7 of Algorithm 4, global communication is needed. `MPI_Allgather` is used in step 9 when the stopping criteria has to be evaluated. The individual parts of the norm are calculated (sum of squares of the parts of the residual on each processor). These are then propagated to all CUs. Once the complete sum is available, a square root is calculated to get the norm at every step of the iteration. In step 1 and 6 the deflation operation is performed which involves assembling a vector (a_1 from (2.34a)) for the coarse grid. All the compute units are required to share their parts of the coarse grid vector using an `MPI_Allgather` call.

Examples of nearest neighbor communication are in steps 1, 3 and 6 of Algorithm 4 where SpMV operation is performed.

Row-based division - SpMV and preconditioning operation

In the case when the coefficient matrix and vector have been divided by rows, one CU acts as a master. This CU at the beginning sends out data to all other CUs (including itself) an extension of parts of A so that L and L^T can be made with overlap regions around them. After this step, all CUs begin their computations. This ensures that each time before the preconditioning step 3 in Algorithm 4, each CU has to only request part of the r_{i-1} *once* from its nearest neighbors. This part of the residual, r_{i-1} is, however, extended with ghost regions (e.g. up to 3 levels on each side for CUs between the 1^{st} and p^{th}).

A multiplication of $LD^{-1}/D^{-1}L^T$ (with dimensions $(\frac{N}{p} + k \times n^2) \times \frac{N}{p} + n^2$ in 3D, where k can be 4 or 2 depending on whether it is a compute unit with index between 1 and p or its index is 1 or p) with r_{i-1} is done every time the preconditioning is applied. This operation results in a vector that has dimensions $\frac{N}{p} \times 1$ so r_{i-1} must have extended ghost regions in order to calculate

$$(I - \tilde{L^T} + \tilde{L^T}^2)D^{-1}(I - \tilde{L} + \tilde{L}^2)r_{i-1} \tag{5.1}$$

where $\tilde{L^T} = D^{-1}L^T$ and $\tilde{L} = LD^{-1}$. Expression (5.1) involves four SpMVs (two with G and two with U). For the SpMV operation in Step 6 within the iteration loop the ghost region required for Ap_i follows from the ghost regions that were received from the preconditioning step and hence it can be done without an additional nearest neighbor communication.

Block-based division

In the case of Block-based division, the preconditioning scheme we have used, (TNS-based preconditioning, Section 3.2.2), requires four matrix vector multiplications where LD^{-1} or $D^{-1}L^T$ is multiplied with a vector to generate one of the terms needed in the calculation of the final result.

The SpMV operation in case of the block-based data division forms a major component of the computation. Other than steps 1 and 6 of Algorithm 4 where it is used to multiply the coefficient matrix with a vector, it is also used in the preconditioning operation of step 3. This is because preconditioning is implemented as a sequence of following steps:

$$f_1 = LD^{-1}r_{i-1} \qquad f_2 = LD^{-1}f_1 \tag{5.2a}$$

$$f_3 = r_{i-1} - f_1 + f_2 \qquad f_4 = D^{-1}f_3 \tag{5.2b}$$

$$f_5 = D^{-1}L^T f_4 \qquad f_6 = D^{-1}L^T f_5 \tag{5.2c}$$

$$w_{i-1} = f_4 - f_5 + f_6. \tag{5.2d}$$

To make the SpMV operation efficient we do an overlap in communication and computation. The complete result requires generating a product of the local matrix with the local vector and summing it up with the products of the coefficients that lie outside the local block with the corresponding components of the vector required from neighboring CUs. First, we issue a communication request to get the components of the vector required from nearest neighbors and since we do it using a non-blocking asynchronous MPI call, the control returns immediately. In case when the CU is a GPU, before and after the MPI transfers a device-host or host-device transfer is also involved. Then we calculate the SpMV for the local data (this can be on the GPU or the CPU

depending on which kind of CU the code is being executed upon). Hence, while the vector is being received from the neighboring CUs we do some computation instead of waiting for data. After finishing the computation and making sure that all components of the vector have arrived we calculate the rest of the SpMV with coefficients belonging to outside points with the received parts of the vector. The SpMV operation in step 1 and 6 of Algorithm 4 uses the same SpMV routine as is used by the preconditioning scheme.

5.5 Experiments and results

We present the results of our research in this section, starting with a brief description of the hardware we used. We first present the results of comparison of the data division schemes on the DAS-4 cluster followed by scaling experiments on the Cartesius cluster.

Definitions and reporting schema

A short description of how we report the timing under different headings is now presented.

1. Number of cores/CPU - Each node has a CPU with certain number of cores on it. We mention the number of cores used under this heading.

2. Number of nodes - A particular experiment utilizes a certain number of nodes with all its cores (or in case of multi-GPU experiments 1 or 2 cores per node). These nodes communicate over MPI with other nodes.

3. Vectors - These are the number of sub-domains into which the domain is sub-divided. Deflation vectors of the sub-domain variety are then defined on these domains. This value changes for different executions in the block-based division (both of Cartesius and DAS-4) but in row-based division (DAS-4 results) the number of vectors are kept constant at 7. The number of CUs is equal to the product of number of nodes and number of cores/node or number of GPUs/node.

4. Configuration - This refers to how the sub-domains are arranged in 3D. Only true for block-based division of code.

5. Iterations - number of iterations required for convergence.

6. Global setup - This is the time required before the beginning of the CG algorithm and it includes the time to convert the data coming from the

FORTRAN code into C/MKL data structures. Conversions to appropriate data formats. Preparing matrices L, L^T, Z, Z^T, AZ, AZ^T and E that are required by the algorithm.

7. Local setup - This includes the time required by the CG algorithm before starting line 1 of Algorithm 4. It also includes the time to translate C data structures to CUSPARSE data structures. This includes time taken to transfer data to the GPU.

8. CG - This is the time taken from step 1 to step 10 of Algorithm 4 and includes time to write back the result to the host.

9. spmv(Ax) - This includes the time taken to do a sparse matrix vector multiplication of the coefficient matrix with a vector A. For the results on the Cartesius cluster this also includes the data setup, transfer (communication) and computation times. This is because of the implementation on Cartesius cluster which involves overlapping communication and computation (refer Section 5.4.2).

10. precon - This includes the time for preconditioning operation (including communication for results on Cartesius cluster only).

11. Dot-daxpy-copy - This includes the time required for doing dot product operations, copy and axpy operations on vectors. Dot products require global communication.

12. deflation - The time required for doing deflation and correction operation in the last step including communication.

13. comm-mpi-glbl - Time spent in global communication with MPI (`MPI_Allgather` and `MPI_Allreduce`).

14. comm-mpi-NN - Time spent in nearest neighbor communication. On the DAS-4 cluster for row-based divisions, nearest neighbor communication is done once every iteration before the preconditioing operation and it gathers the required overlap regions for preconditioning and SpMV operations. On Cartesius and DAS-4 cluster for block-based divisions nearest neighbor communication is used in spmv(Ax) and in precon operations when TNS-based preconditioning is used).

15. comm-h2dd2h - Time spent in host-device device-host transfers. Applicable only to GPU implementations when MPI data has to be exchanged between two CUs which are GPUs.

16. comm-init - This is the time spent in initial transfer of data from the master CU to the rest of the CUs that happens in the results for row-based division on the DAS-4 cluster.

    In the Tables that follow these timings can be grouped under sub-heads. Items 6 and 7 combined give the total Setup time. Item 8 is the execution time or the time it takes to converge for the DPCG method. Items 9-13 are the break-up of the total execution time. This means that the numbers appearing against CG in the tables are the sum of the numbers appearing against spmv(Ax), precon, Dot-daxpy-copy, deflation and comm-mpi-glbl. Items 13-16 give the break-up of the communication time.

    In the charts that are presented in subsequent sections we provide insets that are useful in noting the smaller numbers for experiments which are much faster (up to an order of magnitude) than say e.g. a single node experiment.

5.5.1 Results on the DAS-4 cluster

Experimental setup

In this section we present results obtained on the DAS-4 cluster in the Netherlands. Each node of a DAS-4 node which we use has one GPU and a CPU. Specifically, each node has a K20 GPU (with around 5.25 GB memory on board, ECC (error-correcting code) on). The CPUs used are dual quad core Xeon processors (running at 2.4GHz). Each CPU has 24GB of main memory. The communication fabric through which the nodes are connected at DAS-4 is Infiniband.

As for the software we use CUDA 5.5 and openMPI (version 1.4.4) for MPI communication. For compiling the CPU version of the code we use an Intel compiler (Composer XE 13.3) and Intel MKL (version 11). On the CPU we use openMPI directives `--bind=to-core, --bycore` in order to make sure that we do not have any performance loss due to process movement.

When conducting experiments on the GPU we use one core of the CPU per node as a control processor to offload entire computation to the GPU and also act as a bridge to communicate with other GPUs (on other nodes) via their CPUs over MPI. While conducting experiments on the CPU we use a variety of configurations where we use 1, 2, 4 or 8 nodes. The CPU cores or GPUs individually are called compute units or CUs. We also report the time spent in MPI communications and the time it takes on the GPU to do device-host-device data transfers while running the DPCG code.

Speedup, stopping criteria and tolerance

In our reporting we mention speedup as the total time (global setup + local setup + iteration time) spent in the multi-CU (CPU) vs. multi-CU (GPU) based implementation.

$$Speedup = \frac{Total\ Time\ on\ \mathbf{CPU}}{Total\ Time\ on\ \mathbf{GPU}} \tag{5.3}$$

We use the stopping criteria given by

$$\| r_i \|_2 \leq \| b \|_2 \epsilon, \tag{5.4}$$

where $\epsilon = 10^{-6}$ unless otherwise specified.

Division by rows

In Table 5.1 to 5.4 we present the results for grid sizes 128^3 and 256^3 on the CPU and the GPU. The number of iterations for grid size 128^3 is 630 and for grid size 256^3 is 1159. The number of iterations stays constant even if we increase the number of CUs because the number of deflation vectors is constant at 7 (also mentioned at the end of experimental setup, Section 5.5.1).

128^3					
	Number of cores/CPU	8			1
	Number of nodes (Total cores)	1(8)	2(16)	4(32)	8(8)
Total setup	Global setup	0.2	1	1.2	0.6
	Local setup	0.19	0.13	0.1	0.19
Execution	CG	37.64	21.11	13.23	14.6
Break-Up	spmv	6.96	3.42	1.71	4.04
	precon	14.83	8.35	4.9	4.97
	Dot-daxpy-copy	7.31	3.37	1.52	1.87
	deflation	4.14	1.77	0.86	1.68
Communication	Comm-mpi-glbl	3.3	3.48	3.72	1.62
	Comm-mpi-NN	1.35	0.81	0.62	0.49
	Comm-init	0.3	0.69	1.43	0.42

Table 5.1: Results for grid sizes 128^3 on the CPU. Row-based domains.

In our results there are two important exceptions. Firstly, for the GPU results we are not able to show the results for a single GPU since for this problem size, 256^3 all the data structures required to run our implementation do not fit on a single GPU. Secondly, in the case of problem size 128^3 on

256^3						
	Number of cores/CPU	8				1
	Number of nodes(Total cores)	1(8)	2(16)	4(32)	8(64)	8(8)
Total setup	Global setup	5.7	6	8	17	4
	Local setup	1.85	0.98	0.84	0.86	1.25
Execution	CG	626.55	416.28	234.79	153.25	218.09
	spmv	116.21	70	29.27	16.97	59.78
	precon	252.9	177.66	102.58	51.97	80.91
	Dot-daxpy-copy	135.05	87.44	34.51	15.81	36.34
	deflation	78.95	39.4	24.96	11.72	27.25
Communication	Comm-mpi-glbl	30.84	34.77	43.57	56.18	8.46
	Comm-mpi-NN	14.76	10.82	7.7	5.24	6.2
	Comm-init	2.5	4.08	6.91	15.47	1.99

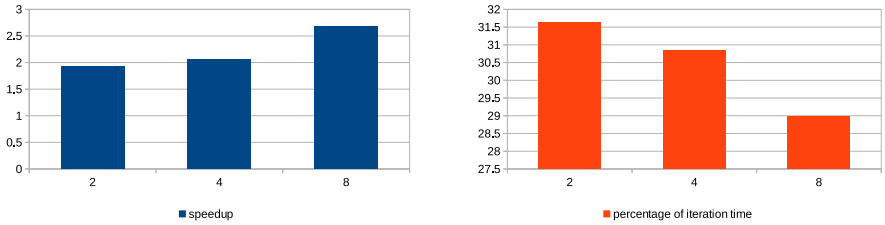
Table 5.2: Results for grid sizes 256^3 on the CPU. Row-based domains.

the CPU we do not present the results for the 64 core case. This is because of a choice we made in implementing the preconditioning for row-based data division as mentioned in Section 5.4.2. We need a ghost region of length $4 \times n^2$ on each CU and we do only one transfer in order to reduce the communication cost. In the case when problem size is 128^3 and we have 64 CUs on which the problem is divided, each CU has only $2 \times 128 \times 128$ rows so it is not possible for one neighbor to satisfy the demand for the ghost region of a neighboring CU. More MPI transfers must be done in order to get the required size of data. This can turn out to be costlier (and defeats the purpose of our one-time communication as suggested in Section 5.4.2) if the number of CUs rises. This communication can hurt the performance of our implementation and hence we have a limit (in the row-based case) for the number of CUs that can be used for a given problem size. Specifically, if problem size is n^3 then our software cannot use more than $n/4$ CUs.

Comparing Tables 5.2 with 5.4 we see that the speedup for the case when 8 GPUs are used compared to when 64 cores are used is up to 2.5 times when the number of unknowns is 256^3. Speedup is defined as the ratio of the total time (setup(global+local) + CG time) taken on the CPU divided by the total time on the GPU. This is summarized in Figure 5.2(a).

The cost of setup time in the results presented in Tables 5.1 to 5.4 can be amortized if the DPCG solver (with row-based division) is used in a time-stepping simulation where the solution of an ill-conditioned linear system is required in every time-step. Moreover, if the simulation involves a coefficient matrix that does not change over time (e.g. in case of seismic modeling of the earths' interior) then the setup time is only a one time cost.

For CPU implementations the computation times drop with increasing



(a) Speedup

(b) Memory transfers between GPU and CPU

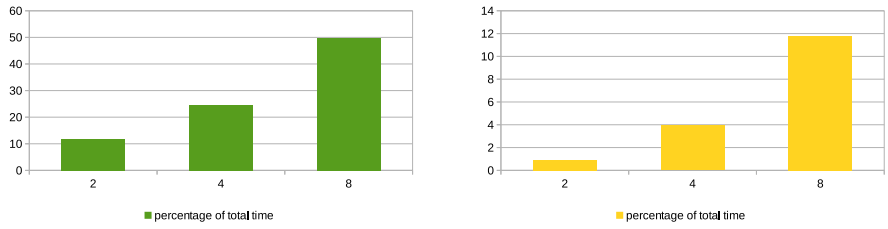
Figure 5.2: Speedup (ref. Equation (5.3)) and memory transfers on the GPU for row-based division. Problem size 256^3.

128^3				
	Number of GPUs/node	1	1	1
	Number of nodes	2	4	8
Total setup	Global setup	5.3	4.15	3.77
	Local setup	4.89	3.61	2.91
Execution	CG	12.29	7.9	5.17
Break-Up	spmv(Ax)	0.72	0.38	0.2
	precon	1.08	0.6	0.35
	Dot-daxpy-copy	0.7	0.43	0.3
	deflation	5.96	3.03	1.6
Communication	Comm-mpi-glbl	0.06	1.38	1.44
	Comm-mpi-NN	0.11	0.21	0.19
	Comm-h2dd2h	3.7	2.01	1.16
	Comm-init	0.12	0.29	0.55

Table 5.3: Results for grid sizes 128^3 on the GPU. Row-based domains.

256^3				
	No. of GPUs/node	1	1	1
	No. of nodes	2	4	8
Total setup	Global setup	40	24	11
	Local setup	38.34	20.38	7.92
Execution	CG	178.31	93.82	49.67
Break-Up	spmv(Ax)	10.25	5.18	2.6
	precon	16.5	8.5	4.49
	Dot-daxpy-copy	8.46	4.45	2.33
	deflation	85.91	43.03	21.66
Communication	Comm-mpi-glbl	0.45	2.73	2.77
	Comm-mpi-NN	0.75	1.68	1.44
	Comm-h2dd2h	56.38	28.41	14.38
	Comm-init	0.72	1.63	2.56

Table 5.4: Results for grid sizes 256^3 on the GPU. Row-based domains.



(a) CPU (b) GPU

Figure 5.3: MPI communication as a percentage of total time. Problem size 256^3. Row-based domains

number of CPU cores as individual computations (that become small enough) can benefit from the caches on the CPUs (Tables 5.1 and 5.2).

In addition we see that communication times increase (shown as percentage of total time in Figure 5.3(a), they do not form a large part of the GPU times but increase with the number of nodes. In Figure 5.3(b) we can observe that as the number of CUs increase in the case of CPUs the percentage time spent in communication also increases(this is in line with the predictions of communication costs for row-based division as pointed out in Section 5.4.2). A significant portion of this time is accounted for in the setup when the CU that receives all the data has to distribute it to other CUs which maybe on the same node (then transfers are done via memory copies) or on another node (then transfers are done via MPI).

The computation times are quite high since the size of the problem per CU compared to using 64 cores is much bigger. This is because, when all 8 cores on a single CPU are used there are many more cache misses. Each core is handling large sets of data (16MB for a vector and around 7 times more when accessing the part of the coefficient matrix each CU has) and therefore, there is much more memory contention compared to when only one core is being used.

Examining the GPU results, around one-third of the time spent in the execution is consumed by the memory transfers (Figure 5.2(b)) between host-to-device and vice-versa. For the results shown in Tables 5.1 to 5.4 the total times is a sum of the operations spmv, precon, Dot-daxpy-copy, deflation, comm-mpi-NN and comm-mpi-glbl. It must be also noted that all these timings are for the master node. For each node the timing is different as it depends on the time spent in synchronizing with other nodes also.

Division by blocks

In the case of row-based data division one can divide the coefficient matrix in only one way. That is to divide amongst all the compute units N/p rows of the matrix A vector with appropriate ghost layers/cells. However, in the case of block-based division the grid is distributed amongst individual CUs. This gives the possibility of having many configurations when p CUs are available. A configuration of a grid size n^3 is denoted by three numbers (n_x, n_y, n_z) which stand for the number of CUs logically in each direction of such a grid division. This leads to a local grid on each CU with dimensions ($\frac{n}{n_x}$,$\frac{n}{n_y}$, $\frac{n}{n_z}$). In Table 5.5 we show for different values of cores/node (CU is a CPU core) what configurations are possible for a grid size of 256^3 on the CPU (when number of CUs is fixed at 16).

The choice of a particular configuration and how many cores per node are used can affect timings and we report these changes. When more cores per node are used the computation times rise. In Table 5.5 we notice that the number of iterations are more for the configuration $2, 2, 4$ and $4, 2, 2$ in comparison with $2, 4, 2$. This is caused by the difference in the way deflation vectors are made by the virtue of the domain decomposition. The deflation vectors are chosen piecewise constant per sub-domain and the number of sub-domains is equal to the number of CUs in our experiments, e.g. in configuration $2, 4, 2$ they are $2 \times 4 \times 2 = 16$ and so on for the rest of the configurations. The individual numbers determine how many of them are in each co-ordinate direction x, y or z.

In Figure 5.4 we show how the bubbles are placed when we divide the grid amongst CUs using blocks. There are two bubbles with one slightly displaced with respect to the other one in z and y direction. For the $2, 2, 4$ or $4, 2, 2$

Number of nodes=2				
CUs/node	8			
Config.	Iter.	Setup	Execution	Comm.
$2, 2, 4$	376	3.2	86.6	1.4
$2, 4, 2$	331	3.2	76.2	1.4
$4, 2, 2$	376	3.3	87.7	1.8
Number of nodes=4				
CUs/node	4			
Config.	Iter.	Setup	Execution	Comm.
$2, 2, 4$	376	3.2	80.2	1.2
$2, 4, 2$	331	3.1	70.7	1.1
$4, 2, 2$	376	3.2	82.5	1.2
Number of nodes=16				
CUs/node	1			
Config.	Iter.	Setup	Execution	Comm.
$2, 2, 4$	376	3.1	38.5	1
$2, 4, 2$	331	3.1	33.0	0.82
$4, 2, 2$	376	3.1	36.8	1.1

Table 5.5: Different configurations. Block-based domains. Grid size 256^3 on the CPU. Using 16 cores across different number of nodes.

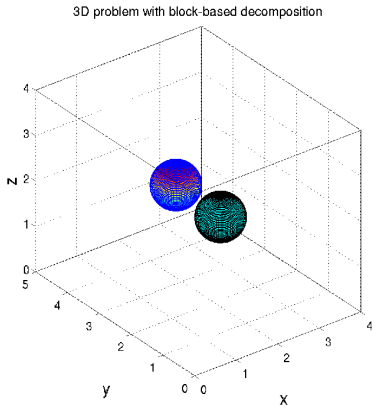


Figure 5.4: 3D problem model with 2 bubbles

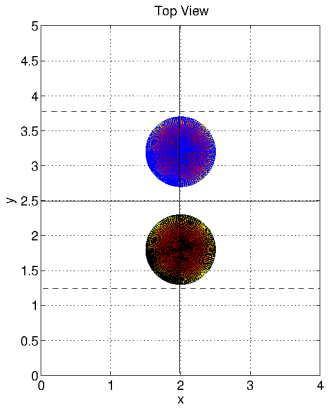


Figure 5.5: Top view with deflation vectors

configuration there are two vectors in y and for the configuration $2, 4, 2$ there are 4 vectors in y. The top view for the case when y direction has 2 or 4 vectors is shown in Figure 5.5. It must be noted here that the problem is defined on a cuboid of physical dimensions $4, 5, 4$ (Length, Width, Height) units. Hence, the deflation vectors in y direction are of different width compared to those in x and z direction. This is one reason for the sensitivity of the vectors in the y direction. A more convincing reason for the difference in number of iterations between the three configurations can be seen in the plot of the 2-norm of the residual for the convergence of DPCG method based on these three configurations of deflation vectors (Figure 5.6).

We can see that for the configurations $(2, 2, 4)$ and $(4, 2, 2)$ the residual is always larger than for the configuration $(2, 4, 2)$. One can conclude that the configuration $(2, 4, 2)$ is a better choice for the deflation subspace than the other two cases and this is the reason why convergence is achieved faster.

As can be seen in Table 5.5, configurations can make some difference in the number of iterations and execution times but the most striking feature is that when the number of cores are reduced per node then we see a drop in execution times up to a factor of about two and a half times.

We devised an experiment to understand the difference in execution times we see in Table 5.5. In the following section we describe this experiment and comment on the findings.

Calculation times (`axpy`) and point-wise multiply variance with different number of cores per node in use

For this experiment, an average of 20 runs per case is taken. Every such av-

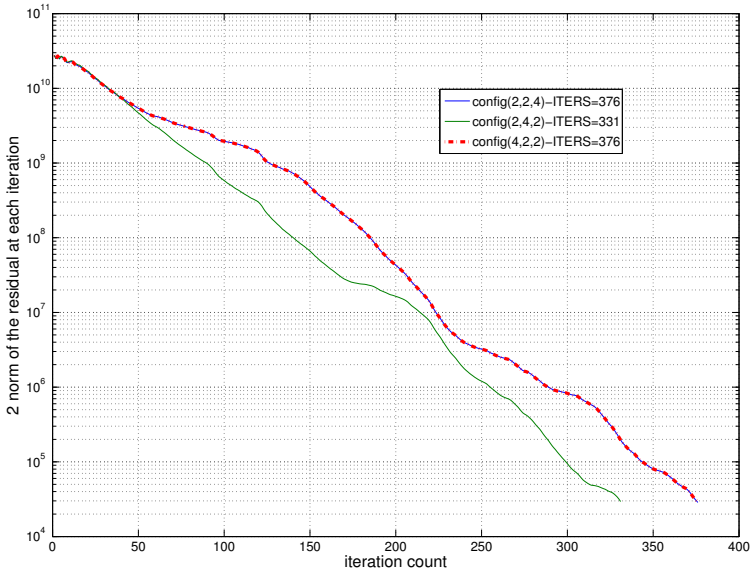


Figure 5.6: 2-norm of the residual for DPCG method across different configurations

erage is calculated 20 more times and the minimum value out of these 20 is taken. This is done since the individual times to complete one transfer can be of the order of nanoseconds. Measuring them only a few times can increase chances of error. Moreover, taking the smallest of the numbers ensures that the best case performance is recorded. The calculation times for the case when we use all cores on the CU compared to the case when only one core is used per CU in case the CUs are CPU cores can vary by up to 2.2 times. This is visible in the results from Table 5.5. This can be verified from the results in Figure 5.7 and 5.8 where the element-wise multiplication of two vectors and calculation of `axpy` operations is tested on various vector sizes. In Figure 5.7 and 5.8 xCU means how many cores are being used out of 8 available in a single CPU. The x-axis shows the problem size, N in powers of two. It can be observed that when all cores are used the computation time for these operations is almost four times with respect to the case when only one core is used. This is because when all cores are used there are more memory contentions. This is after we have used openMPI options of `--bind=to-core, --bycore` that enforce locality.

In our implementation of the SpMV (for the block-based data division), the product of the parts of the vector obtained after communication along with the corresponding coefficients is calculated separately and the results are

added to the SpMV result of the interior cells per processor. The interior SpMV is done using a call to the MKL library.

The preconditioning operation as outlined under the description of the block-based division in Section 5.4.2 involves scaling operations of the vector with the matrix D^{-1}, SpMV operations of $LD^{-1}x$ (and analogous operations in case of $D^{-1}L^T$) and axpy operations when combining individual terms to make the final product of the residual with the preconditioning matrix.

These three operations take the bulk of the time on the CPU implementations. In general, it can be expected that preconditioning takes twice the amount of time as the SpmV operation involving the coefficient matrix with a vector but in our results we see that it is between three to four times.

In order to understand how the total time is distributed amongst different operations of Algorithm 4 we did an experiment with 8 CU's and used 2 CU's in each direction. The configuration is $(2, 2, 2)$. In Table 5.6 we see how preconditioning dominates the total CG time followed by SpMV involving the coefficient matrix. More cores used per CPU can cause both these times to increase.

Going back to Table 5.5 the ratio of execution times is around 2.2 when 8 cores are used per node compared to one. Preconditioning alone takes close to 70% of the time and involves sparse matrix-vector products, scaling operations and vector updates. If this alone is reduced by four times then the effect on total time would be a reduction of around two times (using Amdahl's Law). With this perspective if we weigh the results shown in our test for element-wise multiplications (which are the primary operations in the preconditioning step) a possible explanation for the lower execution times when fewer cores per node are used can be offered for the results in Table 5.5.

As a closing remark to this experiment it must be said that the SpMV is done using a library and the matrix is stored in COO format, since it is a generic format. Furthermore, the details of the implementation are hidden from users so these experiments can provide the best approximation of the behavior seen in the results.

Now we continue presenting results for multi-CPU and multi-GPU implementations using block-based data division.

In Table 5.7 and 5.8 we show the results for the block-based decomposition on CPU and GPU.

The GPU version almost equals the 64 core CPU version if we consider the timings per iteration (GPU (total) time for 252 iterations would be 14.72 seconds as compared to CPU (total) time of 15.1). It must be noted that the 64 core CPU version has a larger number (8 times more) of deflation vectors compared to the GPU version and hence its convergence is accelerated

	Number of cores/CPU	4	2	1	8
	Number of nodes(Total Cores)	2(8)	4(8)	8(8)	1(8)
128^3					
Total Setup	Global setup	0.45	0.45	0.44	0.5
	Local setup	0.009	0.0026	0.002	0.002
Execution	CG	7.07	4.97	4.55	10.78
Break-Up	spmv	1.34	1.07	1.06	1.66
	precon	4.27	2.82	2.62	6.26
	Dot-daxpy-copy	0.99	0.0.54	0.55	2.13
	deflation	0.27	0.26	0.21	0.6
Communication	Comm-mpi-glbl	0.05	0.23	0.02	0.06
	Comm-mpi-NN	0.18	0.245	0.1	0.17
256^3					
Total setup	Global setup	7.38	6.92	6.9	7.2
	Local setup	0.075	0.006	0.007	0.009
Execution	CG	122.69	95.8	87.2	194
Break-Up	spmv	21.55	18.7	17.8	27.87
	precon	68.4	52.5	48.4	111.1
	Dot-daxpy-copy	22.76	18.4	15.9	41
	deflation	7.5	4.65	4.67	11.51
Communication	Comm-mpi-glbl	0.7	1.3	0.23	0.14
	Comm-mpi-NN	3.4	3.65	0.62	1.1

Table 5.6: Configuration $(2, 2, 2)$. Grid sizes 128^3 and 256^3 on the CPU. Block-based domains. Iterations for $128^3 = 188$ and for $256^3 = 375$.

	Number of cores/CPU	8		
	Number of nodes(Total cores)	2(16)	4(32)	8(64)
	Arrangement	(2,4,2)	(4,4,2)	(4,4,4)
	Vectors	16	32	64
	Grid size	128^3		
	Iterations	166	161	126
Total setup	Global setup	0.29	0.21	0.061
	Local Setup	0.009	0.003	0.024
Execution	CG	4.83	2.8	0.67
Break-Up	spmv	0.7	0.35	0.13
	precon	2.84	1.4	0.33
	Dot-daxpy-copy	0.79	0.3	0.1
	deflation	0.21	0.15	0.05
Communication	Comm-mpi-glbl	0.03	0.15	0.06
	Comm-mpi-NN	0.096	0.085	0.05
	Grid size	256^3		
	Iterations	331	323	252
Total setup	Global setup	3.12	1.26	0.45
	Local setup	0.01	0	0
	CG	72.65	37.74	15.1
Break-Up	spmv	10.73	5.44	2.15
	precon	42.9	21.65	8.6
	Dot-daxpy-copy	15.04	7.34	2.75
	deflation	4.57	2.41	1.11
Communication	Comm-mpi-glbl	0.27	0.85	0.8
	Comm-mpi-NN	0.59	0.56	0.36

Table 5.7: Results for grid sizes 128^3 and 256^3 on the CPU. Block-based domains. 8 cores/node.

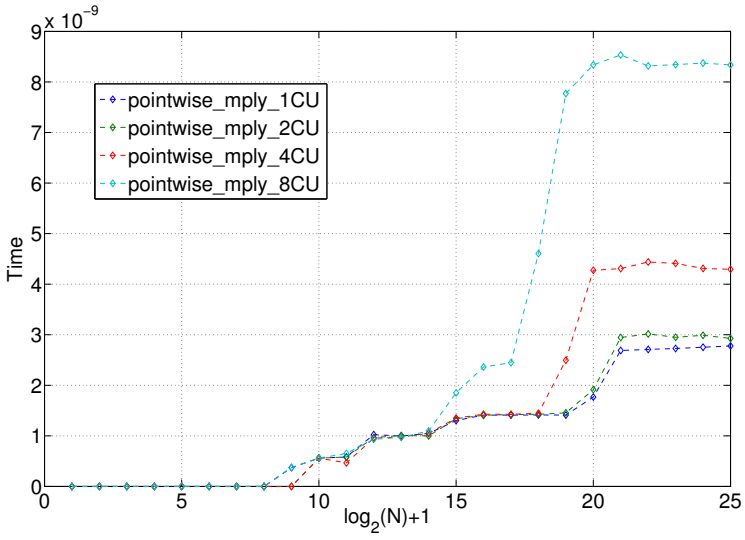


Figure 5.7: Calculation times for element-wise multiplication of two vectors

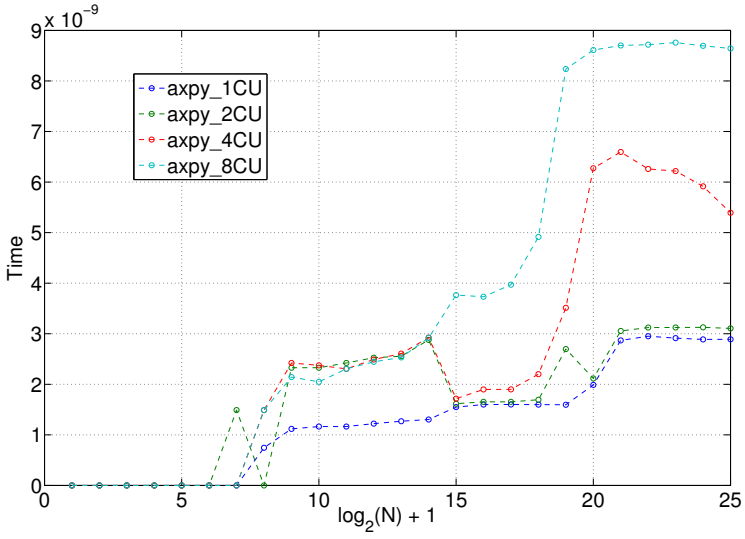


Figure 5.8: Calculation times for `axpy`

compared to the 8 GPU version.

	128^3	256^3
Iterations	188	375
Global setup	0.54	8.74
Local setup	0.03	0.156
CG	1.89	21.9
spmv(Ax)	1.15	16.3
precon	0.57	4.3
Dot-daxpy-copy	0.08	0.648
deflation	0.08	0.61
Comm-mpi-glbl	0.04	0.07
Comm-mpi-NN	0.21	0.8
Comm-h2dd2h	0.19	1.35

Table 5.8: Results for grid sizes 128^3 and 256^3 on the GPU. Block-based domains. Configuration $2, 2, 2$. 8 nodes (1GPU/node).

5.5.2 Experiments on Cartesius cluster

In this section we present the results that were generated on the Cartesius cluster which has been installed by SURFSARA. We only present results from problems with 256^3 unknowns for the problem setup mentioned in the previous section. The contrast in density, stopping criteria and tolerance is the same as for the previous set of experiments presented in this chapter.

In this set of experiments we show the effect of using different storage formats on the performance of the DPCG algorithm. Furthermore, we show the advantages of using simpler first level preconditioning schemes. Finally, we also consider the benefits of using multiple GPUs per node.

Experimental setup

We have done our experiments on the accelerator island at the Cartesius cluster of SURFSARA. Each node has the following configuration

1. 2 × 8-core 2.5 GHz Intel Xeon E5-2450 v2 (Ivy Bridge) CPUs/node,
2. 2 × NVIDIA Tesla K40m GPUs/node (up to 11.25GB DDR5 main memory with ECC on);
3. 96 GB memory per node.

The interconnect used is Mellanox ConnectX-3 Infiniband adapter providing 4 × FDR (Fourteen Data Rate) resulting in 56 Gbit/s inter-node bandwidth, with an inter-island latency of 3μs.

We use CUDA 6 and Intel compiler version 13.1 as the GPU and CPU software suites. For the CPU version we use MKL (version 11.0.2) that ships with the Intel compiler. For the GPU we use CUBLAS and CUSPARSE in CUDA 6. The nodes communicate amongst each other using MPI libraries from Intel.

In the results that follow we only use block-based division and corresponding calling software (discussed in Sections 5.3.2, 5.4.1, 5.4.2 and 5.5.1).

Results

In Tables 5.9 and 5.10 we show the results for multi-CPU implementation with diagonal and TNS-based preconditioning based DPCG where the matrices A, L and L^T are stored in the COO format on the CPU.

Number of cores/CPU	16						
Number of nodes	1	2	4	8	16	32	64
Vectors	16	32	64	128	256	512	1024
Configuration	(2,4,2)	(4,4,2)	(4,4,4)	(4,4,8)	(8,4,8)	(8,8,8)	(8,8,16)
Iterations	771	756	552	556	516	299	300
Global setup	2.99	1.18	0.391	0.191	0.1177	0.257	3.12
Local setup	0.002	0.001	0.001	0.0001	0	0.0001	0.017
CG	78.22	39.21	14.19	6.595	2.818	0.84	1.088
spmv(Ax)	25.19	12.71	4.81	2.469	1.221	0.203	0.106
precon(scaling)	7.11	3.59	1.407	0.415	0.176	0.047	0.013
Dot-daxpy-copy	34.74	16.83	5.751	2.85	0.893	0.13	0.055
deflation	11.11	5.91	2.1419	0.71	0.386	0.37	0.7955
Comm-mpi-glbl	0.05	0.18	0.087	0.224	0.24	0.143	0.208
Comm-mpi-NN	0.402	0.229	0.07	0.06	0.04	0.02	0.016

Table 5.9: Diagonal preconditioning based DPCG. Multi-CPU implementations. Storage of matrices in COO format.

Comparing the results shown in Tables 5.9 and 5.10 we can make the following observations.

1. TNS-based preconditioning takes roughly half as many iterations for convergence compared to diagonal preconditioning.

2. For a smaller number of nodes diagonal preconditioning takes less total time compared to TNS-based DPCG.

These findings can also be noted in Figure 5.9. In Tables 5.11 and 5.12 the difference from Tables 5.9 and 5.10 is that instead of COO format of storage for A, L and L^T, DIA format is used and corresponding SpMV routines are utilized. The matrices A, L and L^T have non-zeros only on a few diagonals

Number of cores/CPU	16						
Number of nodes	1	2	4	8	16	32	64
Vectors	16	32	64	128	256	512	1024
Configuration	(2,4,2)	(4,4,2)	(4,4,4)	(4,4,8)	(8,4,8)	(8,8,8)	(8,8,16)
Iterations	331	323	252	251	245	131	132
Global setup	2.91	1.16	0.395	0.198	0.116	0.2169	3.12
Local setup	0.004	0.004	0.002	0.0015	0.001	0.018	0.0305
CG	67.25	33.9	13.21	6.31	2.73	0.674	0.694
spmv(Ax)	9.04	4.67	1.875	0.961	0.494	0.1239	0.0533
precon	39.07	19.77	7.72	3.82	1.611	0.293	0.129
Dot-daxpy-copy	14.81	7.21	2.755	1.17	0.377	0.066	0.029
deflation	4.27	2.26	0.778	0.255	0.151	0.136	0.372
Comm-mpi-glbl	0.026	0.069	0.101	0.117	0.133	0.101	0.197
Comm-mpi-NN	0.68	0.395	0.275	0.127	0.088	0.02	0.018

Table 5.10: TNS-based preconditioning based DPCG. Multi-CPU implementations. Storage of matrices in COO format.

and therefore they can benefit from this storage format. These routines were discussed in [6].

Number of cores/CPU	16						
Number of nodes	1	2	4	8	16	32	64
Vectors	16	32	64	128	256	512	1024
Configuration	(2,4,2)	(4,4,2)	(4,4,4)	(4,4,8)	(8,4,8)	(8,8,8)	(8,8,16)
Iterations	771	756	552	556	516	299	300
Global setup	3.09	1.252	0.455	0.217	0.131	0.22	3.12
Local setup	0.001	0.0009	0.0007	0.0008	0	0.019	0.022
CG	69.865	34.97	12.68	5.77	2.39	0.755	1.05
spmv(Ax)	16.67	8.392	3.238	1.677	0.792	0.165	0.071
precon(scaling)	7	3.54	1.354	0.44	0.198	0.0408	0.016
Dot-daxpy-copy	35.21	17.054	5.93	2.826	0.897	0.119	0.053
deflation	10.92	5.878	2.06	0.714	0.328	0.333	0.802
Comm-mpi-glbl	0.039	0.114	0.1269	0.154	0.226	0.179	0.203
Comm-mpi-NN	0.41	0.239	0.07	0.06	0.04	0.022	0.013

Table 5.11: Diagonal preconditioning based DPCG. Multi-CPU implementations. Storage of matrices in DIA format.

Comparing Tables 5.9 and 5.10 with 5.11 and 5.12 we can make an observation. The time for iterations has reduced between $10-15\%$ in some cases for the DPCG implementations with storage of matrices in DIA format for both DPCG variants (with TNS-based and diagonal preconditioning) (summarized in Figure 5.9). This time can be traced back to the reduction in times for the spmv(Ax) and precon operations in Tables 5.10 and 5.12. For diagonal precon-

Number of cores/CPU	16						
Number of nodes	1	2	4	8	16	32	64
Vectors	16	32	64	128	256	512	1024
Configuration	(2,4,2)	(4,4,2)	(4,4,4)	(4,4,8)	(8,4,8)	(8,8,8)	(8,8,16)
Iterations	331	323	252	251	245	131	132
Global setup	3.2	1.3	0.467	0.232	0.132	0.223	3.162
Local setup	0.0048	0.0052	0.0039	0.002	0.0018	0.017	0.022
CG	57	28.67	11.23	5.168	1.945	0.537	0.57
spmv(Ax)	6.382	3.28	1.382	0.684	0.31	0.077	0.035
precon	31.19	15.689	6.253	2.95	1	0.195	0.084
Dot-daxpy-copy	14.978	7.293	2.776	1.191	0.397	0.068	0.025
deflation	4.31	2.26	0.743	0.266	0.152	0.143	0.367
Comm-mpi-glbl	0.212	0.202	0.091	0.088	0.108	0.095	0.123
Comm-mpi-NN	0.74	0.451	0.287	0.137	0.086	0.025	0.016

Table 5.12: TNS-based preconditioning based DPCG. Multi-CPU implementations. Storage of matrices in DIA format.

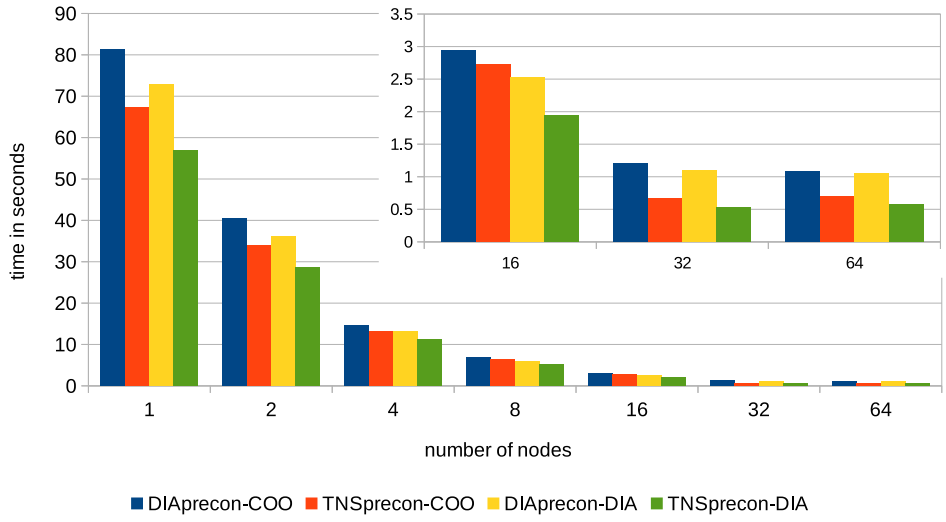


Figure 5.9: Reduction in total time when storage formats are changed on the CPU

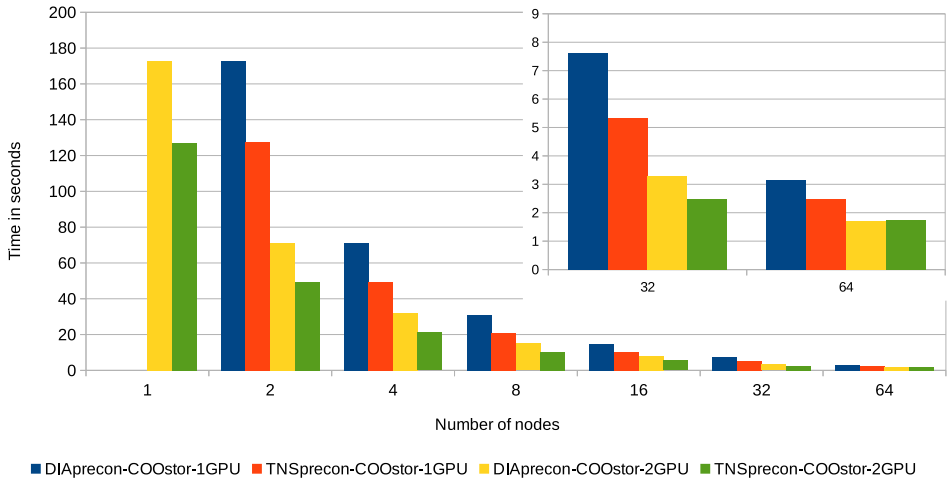


Figure 5.10: Total time when using COO storage formats on the GPU for A, L and L^T matrices

ditioning based DPCG this is true as well but comparing Tables 5.9 and 5.11 we notice that the execution time is significantly reduced for the spmv(Ax) operation. This is expected because of the suitability of the system/coefficient matrix and the L and L^T matrices for the DIA format. The setup time for 64 nodes are an order of magnitude higher than those for 32 nodes. This time is mostly spent in the calculation of local inverse of the matrix E for all the individual cores. Due to the large number of processors and a larger size for E (since there are 1024 cores the size of E for each core is 1024×1024) the calculation of E^{-1} can cause memory contention when all 64 cores do the calculation.

The benefit of changing to the DIA format on the GPU is even more pronounced as can be seen in Figures 5.10 and 5.11(detailed results for COO format based multi-GPU implementations appear in Appendix C). The total time is halved.

In Tables 5.13 and 5.14 we present multi-GPU results for one or two GPUs/node with diagonal preconditioning based DPCG.

In Tables 5.15 and 5.16 we present the results of TNS-based preconditioning based DPCG on multiple GPUs connected via MPI. We show the results for the case when two GPUs per node are used in Tables 5.16.

Comparing Tables 5.13 with 5.14 and 5.15 with 5.16 we see that the two GPU per node approach halves the execution time compared to only using one GPU per node. It must be also kept in mind that increasing the number of GPUs/node leads to an increase in the number of vectors as well.

For the same number of nodes if we compare the results of using all the

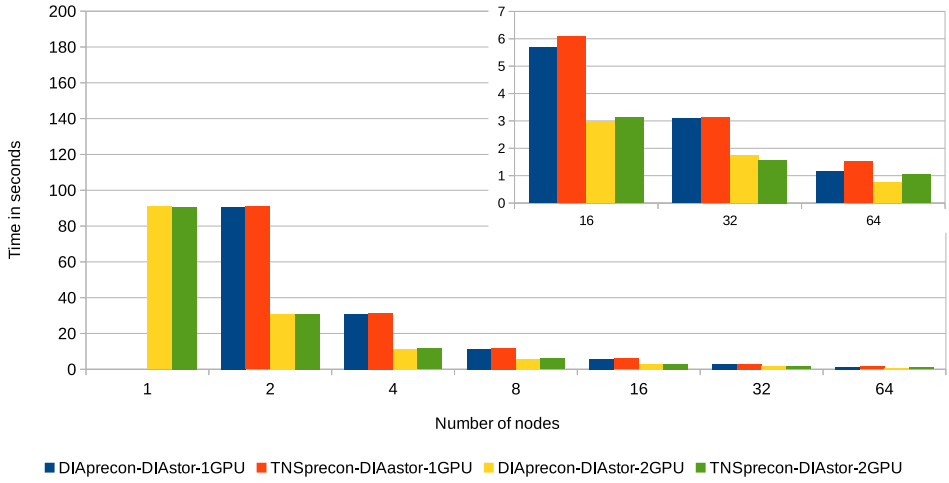


Figure 5.11: Total time when using DIA storage formats on the GPU for A, L and L^T matrices

Number of GPUs/node	1					
Vectors	2	4	8	16	32	64
Configuration	(2,1,1)	(2,1,2)	(2,2,2)	(2,4,2)	(4,4,2)	(4,4,4)
Iterations	878	878	875	771	756	552
Global setup	73.8	21.93	6.469	2.882	1.146	0.433
Local setup	0.243	0.122	0.068	0.063	0.128	0.08
CG	16.779	8.879	4.849	2.739	1.818	0.877
spmv(Ax)	7.086	3.831	2.099	1.315	0.785	0.375
precon	0.931	0.473	0.243	0.113	0.0622	0.027
Dot-daxpy-copy	4.866	2.529	1.351	0.679	0.416	0.216
deflation	3.83	1.989	1.085	0.563	0.413	0.201
Comm-mpi-glbl	0.0864	0.116	0.166	0.194	0.4359	0.13
Comm-mpi-NN	0.435	0.316	0.254	0.17	0.113	0.058
Comm-h2dd2h	2.02	1.109	0.597	0.509	0.268	0.108

Table 5.13: Diagonal preconditioning based DPCG. Multi-GPU implementations. Storage of matrices in DIA format. 1 GPU/node

Number of GPUs/node	2						
Vectors	2	4	8	16	32	64	128
Arrangement	(2,1,1)	(2,1,2)	(2,2,2)	(2,4,2)	(4,4,2)	(4,4,4)	(4,4,8)
Iterations	878	878	875	771	756	552	556
Global setup	73.79	21.91	6.46	2.88	1.145	0.401	0.208
Local setup	0.223	0.111	0.067	0.0607	0.104	0.458	0.04
CG	16.92	8.97	4.943	2.74	1.73	0.883	0.52
spmv(Ax)	7.298	3.932	2.12	1.34	0.823	0.377	0.218
precon(scaling)	0.93	0.472	0.242	0.112	0.061	0.026	0.012
Dot-daxpy-copy	4.85	2.522	1.346	0.673	0.409	0.21	0.118
deflation	3.8	2	1.09	0.565	0.379	0.215	0.12
Comm-mpi-glbl	0.065	0.115	0.158	0.178	0.268	0.32	0.11
Comm-mpi-NN	1.08	0.647	0.472	0.285	0.199	0.084	0.037
Comm-h2dd2h	1.59	0.892	0.496	0.433	0.229	0.093	0.064

Table 5.14: Diagonal preconditioning based DPCG. Multi-GPU implementations. Storage of matrices in DIA format. 2 GPUs/node

Number of GPUs/node	1					
Vectors	2	4	8	16	32	64
Configuration	(2,1,1)	(2,1,2)	(2,2,2)	(2,4,2)	(4,4,2)	(4,4,4)
Iterations	377	377	375	331	323	251
Global setup	74.79	22.44	6.72	3.01	1.21	0.434
Local setup	0.398	0.2	0.18	0.08	0.09	0.12
CG	15.78	8.64	4.85	3.01	1.814	0.954
spmv(Ax)	3.07	1.69	0.93	0.573	0.33	0.17
precon(scaling)	8.908	4.92	2.81	1.86	1.09	0.557
Dot-daxpy-copy	2.09	1.11	0.597	0.299	0.18	0.099
deflation	1.66	0.88	0.486	0.249	0.164	0.093
Comm-mpi-glbl	0.048	0.1	0.105	0.12	0.139	0.18
Comm-mpi-NN	0.653	0.487	0.432	0.302	0.188	0.102
Comm-h2dd2h	4.43	2.44	1.29	1.089	0.562	0.237

Table 5.15: TNS-based preconditioning based DPCG. Multi-GPU implementations. Storage of matrices in DIA format. 1 GPU/node

Number of GPUs/node	2						
Vectors	2	4	8	16	32	64	128
Arrangement	(2,1,1)	(2,1,2)	(2,2,2)	(2,4,2)	(4,4,2)	(4,4,4)	(4,4,8)
Iterations	377	377	375	331	323	252	251
Global setup	74.74	22.4	6.7	3	1.206	0.431	0.2079
Local setup	0.3	0.103	0.07	0.08	0.11	0.19	0.051
CG	15.72	8.448	4.81	2.986	1.8	0.951	0.791
spmv(Ax)	3.17	1.718	0.955	0.583	0.356	0.176	0.136
precon	8.75	4.76	2.76	1.827	1.05	0.542	0.454
Dot-daxpy-copy	2.09	1.084	0.581	0.291	0.176	0.096	0.075
deflation	1.65	0.876	0.478	0.252	0.169	0.096	0.078
Comm-mpi-glbl	0.04	0.08	0.087	0.16	0.13	0.254	0.131
Comm-mpi-NN	1.5	0.9	0.7	0.469	0.27	0.1409	0.083
Comm-h2dd2h	3.53	1.95	1.07	0.925	0.48	0.205	0.196

Table 5.16: TNS-based Preconditioning based DPCG. Multi-GPU implementations. Storage of matrices in DIA format. 2 GPUs/node

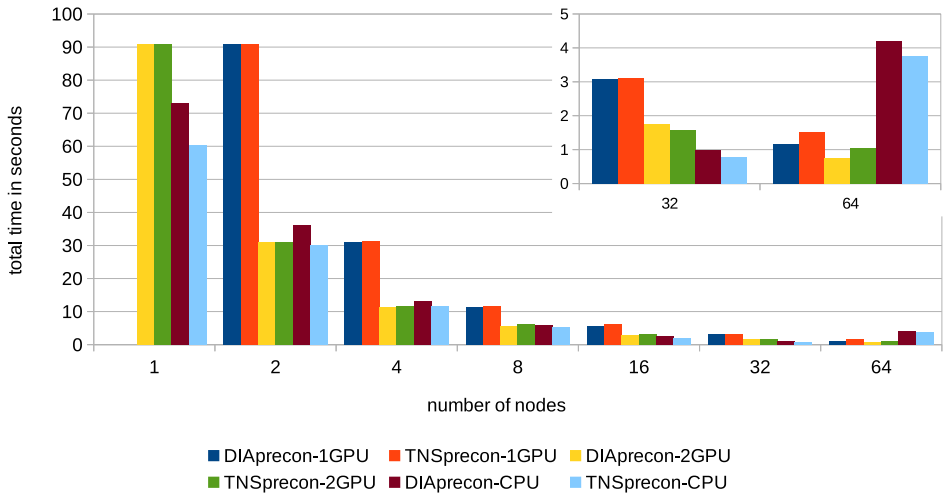


Figure 5.12: Total times across GPU and CPU implementations

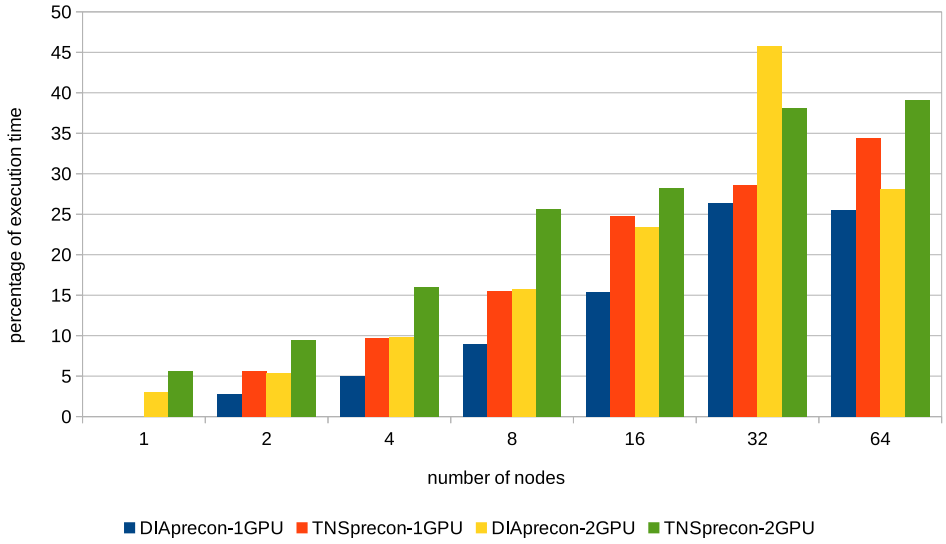


Figure 5.13: Communication times as a percentage of total time.

cores of the CPUs versus using two GPUs per node then the GPU versions of the code for diagonal preconditioning can be up to 3 times faster than the CPU code. This is because of the higher setup times the CPU takes (because of increased number of vectors and corresponding larger size of E). However for the iteration times on the CPU versus those on the fastest version of GPU implementation (diagonal preconditioning with 2 GPUs/node Table 5.14) the GPU and CPU times are almost equal (Table 5.12). These results have been summarized in Figure 5.12. This figure only considers the results when matrices are stored in DIA format across CPU and GPU for both preconditioning techniques (DIA and TNS). Total time is the sum of the time under the headings CG, local setup and global setup. This can be verified from the Tables 5.11 and 5.14.

For the TNS based preconditioning however the CPU can be (for execution times) 33% faster than the multi-GPU version with similar preconditioning scheme (e.g. for the 64 node case on GPU and CPU implementations). In fact the GPU version (with 2 GPUs per node) with diagonal preconditioning that uses 64 nodes takes the least amount of time to converge across all multi-GPU/CPU implementations for this problem. This is evident in Tables 5.11 to 5.16.

In Table 5.16 we notice how communication can become up to 40% of the total time for the case when 2 GPUs are used per node for a case when 128 GPUs are used across 64 nodes. This is 50% longer than the numbers in Table 5.14 where the total communication across 64 nodes is 27%. These

observations have been summarized in Figure 5.13. Total time is the time mentioned under the heading CG plus the Global and Local setup times in the tables referred and the communication time is the sum of the communication time mentioned as comm-mpi-glbl, comm-mpi-NN and comm-h2dd2h.

In the case of TNS-based preconditioning we see that the time spent in communication climbs up (Tables 5.15 and 5.16 and the plot in Figure 5.13). When 128 GPUs are used across 64 nodes the time that communication consumes is 45% for the two GPU per node case and 35% for the one GPU per node case. This is in contrast to when diagonal preconditioning is used in the same experiments. The maximum time taken up by communication is less.

An observation that can be made by examining the results for the TNS-based preconditioner is that it reduces the number of iterations thereby effectively reducing the percentage of global communications also. At the same time since it requires nearest neighbor communication the percentage of time spent in nearest neighbor communication increases. With this observation in mind one can think of a higher degree of the polynomial (greater than 3 that we use) in the approximation of the inverse of $(I - LD^{-1})^{-1}$ could be beneficial (since it is a better approximation) if communication costs become predominant in the time it takes to converge for a particular problem.

5.6 Conclusions

We have evaluated a variety of multi-GPU and multi-core CPU implementations in this chapter. Through our experiences we conclude that block-based data divisions scale much better than row-based division of data. For the block-based data division one has to be aware of the configurations (the number of deflation vectors (and consequently number of CUs, since we use one vector/sub-domain per CU) that are used in each direction to divide the domain) in case the geometry is like the one we used in our experiments.

We have also shown that it is possible to have a strong scaling both on CPU and GPU for our method of preconditioned CG. Combinations of highly parallel simple preconditioners along with simple variants of deflation vectors can already benefit both implementations. Sparse-matrix vector multiplication takes the longest amount of time when the preconditioning scheme is as simple as diagonal preconditioning. If TNS-based preconditioning is used then preconditioning takes the most amount of time, however a higher degree of polynomial in the TNS-based approximation of the preconditioner can be beneficial for communication time dominant problems.

CHAPTER 6

Comparing DPCG on GPUs and CPUs for different problems

6.1 Introduction

In the preceding chapters we have seen how deflation can be adapted to the GPU and its performance can be improved using better deflation vectors. We have shown in the previous chapter how deflated PCG can be implemented on multiple GPUs and CPUs.

In this chapter we show how deflated PCG can be used for different problems and compare variation in performance of the DPCG method when different first level preconditioning techniques are used. We investigate the linear system arising from bubbly flow and porous media flow. Specifically, the linear system (1.1) that results after discretizations of either (1.5) or (1.6). It is solved using the DPCG method (Algorithm 3).

The solution approach for bubbly flow is the Mass-Conserving Level-Set (MCLS) method [68]. This approach has been used in the software that was

---

The work presented in this chapter also appears in:

R. Gupta, D. Lukarski, M. B. van Gijzen, and C. Vuik. Evaluation of the deflated preconditioned CG method to solve bubbly and porous media flow problems on GPU and CPU. *International Journal for Numerical Methods in Fluids*, 2015.

used to generate results in chapters 3, 4 and 5 (row-based division). For the porous media flow problem we follow the solution approach described in [76].

The ill-conditioning of both linear systems (arising from bubbly and porous media flow) arises from small eigenvalues in the spectrum of the coefficient matrix which can be removed using deflation.

In this chapter we want to examine how deflation coupled with a suitable preconditioner on the first level can be used to accelerate the convergence of the DPCG method for these problems. In addition, we also provide a comparison with an optimized CPU implementation and show that GPUs can be advantageous.

We consider a variety of first-level preconditioning techniques [49] along with deflation [31] for a two-level Preconditioned Conjugate Gradient (PCG) implementation.

In this chapter we use the DPCG method implemented within the PARALUTION library (refer [48]). PARALUTION provides an Application Programming Interface (API) for various sparse iterative solvers and preconditioners (including DPCG). It includes several preconditioners based on incomplete LU-type decomposition (and other preconditioning schemes) which we have used in the set of experiments presented in this chapter.

6.2 First-level preconditioning techniques

In this section we describe the first level preconditioning techniques we use to accelerate convergence of CG. They exhibit fine-grain parallelism and hence can be executed efficiently on a GPU. We briefly explain the preconditioners based on classical incomplete LU-based factorizations followed by approximate inverse-based preconditioners. We also refer to the Truncated Neumann Series (TNS)-based preconditioners which we use, and refer to an earlier chapter (chapter 3, section 3.2.2) for description. The coefficient matrix A for both problems (bubbly and porous media flow) is symmetric positive definite (SPD).

In PARALUTION instead of Cholesky factorizations, ILU factorizations are used because

- 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.

- The incomplete Cholesky factorization requires positive diagonal elements. The classical incomplete Cholesky factorization does not always exist for all sparse SPD matrices (refer [77]). To solve that problem,

there are non-symmetric permutations which 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 GMRES etc.).

6.2.1 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 an upper triangular matrix. The sparsity pattern after factorization depends on the 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 [58]. 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 since the transpose operation is not very fast when considering multiple storage formats like ELL, DIA etc.

Level-scheduling method

Due to 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 [58]. To determine the levels, this technique requires an additional pre-processing step which calculates the element dependency.

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 [49], 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 [49].

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, which are calculated by dividing the problem size by number of colors used) for solving the forward and backward substitutions does not depend only on the discretization size (number of unknowns in the system).

Multi-elimination ILU

We had introduced Multi-Elimination ILU earlier in section 2.3.5. In this section we present the ideas used in PARALUTION to expose parallelism in this preconditioning technique.

As an alternative to the usual LU-decomposition we can factorize the system matrix A in the block form

$$A = \begin{bmatrix} T & F \\ E & C \end{bmatrix} = \begin{bmatrix} I & 0 \\ ET^{-1} & I \end{bmatrix} \times \begin{bmatrix} T & F \\ 0 & \hat{A} \end{bmatrix}, \tag{6.1}$$

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 to the matrix C due to the additional fill-ins coming from the matrix-matrix multiplication. We can recursively apply this decomposition 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$(0,1)$.

6.2.2 Multi-colored symmetric Gauss-Seidel

We parallelize the symmetric Gauss-Seidel (SGS) preconditioner using a multi-coloring decomposition of the original matrix A. The multi-coloring exposes additional parallelism in the forward and backward substitution. This gives 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 [49]. In our results this preconditioner is abbreviated as MCSGS.

6.2.3 Truncated Neumann series (TNS)-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,

$$M = (I + LD^{-1})D(I + D^{-1}L^T), \tag{6.2}$$

where I is the identity matrix and L and D follow from the SGS preconditioner. This preconditioner has been used in previous chapters and has been explained in section 3.2.2. The preconditoner we use in our results that is based on the trucated neumann series appears under the abbreviation TNS and is given by

$$M_{TNS}^{-1} = (I - D^{-1}L^T + (D^{-1}L^T)^2)D^{-1}(I - LD^{-1} + (LD^{-1})^2). \tag{6.3}$$

6.2.4 Factorized Sparse Approximate Inverse (FSAI)-based preconditioners

An efficient algorithm for construction of a sparse approximate inverse [25] with respect to the complexity of the building phase is the Factorized Sparse Approximate Inverse (FSAI) method [43]. 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

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

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 [43]. 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 [10]. AINV approximates the matrix A^{-1} by a bi-conjugation process applied to a linearly independent set of vectors.

6.3 Second-level preconditioning

For the second level preconditioning we use Deflation. This technique has been explained previously in section 2.4. For effective deflation one has to have a good approximation of the deflation subspace. This is achieved by approximating eigenvectors corresponding to the small eigenvalues in spectrum of A that delay convergence. These approximate eigenvectors are the columns of the matrix Z. We have enumerated some of the choices to the approximations of eigenvectors in question in sections 2.4.2, 4.3, 4.3.1 and 4.3.1. In this section we describe one of the variants of these approximations that we use for the experiments in this chapter for the porous media flow problems.

6.3.1 Physics based deflation vectors

Porous media flow problem

Sub-domain vectors with weighted overlapping - Deflation vectors we use for porous media flow are similar in construction to the sub-domain deflation vectors used for bubbly flow. They use additional 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. The cells with a certain value of permeability are initialized with ones and for the other medium they have all zeros. We make these vectors according to the structure of the layers. This way they capture the physics of the problem well. For details we refer the interested reader to [76].

6.4 Implementation details

All solvers and preconditioners are implemented in the PARALUTION library (version 0.7.0) [48]. The methods can be executed on different back-ends such as OpenMP, CUDA or OpenCL.

6.4.1 Sparse matrix storage

In Table 6.1 we outline the storage formats (refer [6] for information on formats) we use on the GPU. We keep the matrix A in the DIA format on the GPU for the bubbly flow problem. Keeping it in this format is instrumental in achieving high-throughput for SpMVs using A. In Table 6.1 we outline the formats 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) gives up to 20% improvements in wall-clock times over using the CSR format on the GPU. On the CPU we keep the CSR format for all matrices.

A	Preconditioners			
DIA	TNS	FSAI	MCSGS	Others
	DIA	HYB		CSR

Table 6.1: Sparse Storage Formats on GPU.

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.

6.4.2 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

$$\| r_i \|_2 \leq \| b \|_2 \epsilon \tag{6.4}$$

where r_i is the residual at the i^{th} iteration and b is the right-hand side.

6.4.3 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 parsed for each traversing step.

For higher degree of parallelism in the ILU(p,q) and symmetric Gauss-Seidel (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

$$\begin{aligned} x_i &:= D^{-1}(r_i - \sum_{j=1}^{i-1} L_{i,j} x_j) \\ z_i &:= x_i - \sum_{j=1}^{B-i} U_{i,j} z_{i+j} \end{aligned} \tag{6.5}$$

for the preconditioner $LDUz = r$, where matrices are divided into B sub-blocks and the indices describe the corresponding blocks. In Equation (6.5) D

is the main diagonal and I is the identity matrix. The first equation is solved forward $i = 1...B$ and the second is solved backward $i = B...1$. The inversion of D matrix is trivial. 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 $\hat{A}$ (see (6.1)) is done 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).

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.

6.4.4 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.

6.4.5 Truncated Neumann series (TNS)-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. $D^{-1}L^T$ is stored explicitly in order to use the library functions which do not expose the internals of the L matrix to the user. This storage space can be avoided if custom kernels are used for DIA SpMV operation.

TNS-based preconditioning is applied in steps. We use three terms of the Neumann series for approximating $(I + LD^{-1})^{-1}$. Four matrix-vector products (2 with LD^{-1} and 2 with $D^{-1}L^T$) are required in the variant 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.

6.4.6 Deflation

For the deflation algorithm listed in Section 2.4 we can categorize the operations into sparse matrix - dense vector multiplications (for Ax, AZv), sparse matrix - matrix products (for calculating AZ) and dense matrix - vector products (for calculating $E^{-1}v$). Where x is a $N \times 1$ 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. The inner system is solved using the explicit inverse of E.

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. For the bubbly flow problem the storage format is converted after this step 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 6.4.5 for storing $D^{-1}L^T$. The matrices AZ and E are then constructed using matrix multiplication routines in PARALUTION. Due to 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 format is done. 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.

6.5 Numerical experiments

The experiments are performed on an 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 chapter we present results for level-set sub-domain (LSSD) deflation vectors discussed in section 4.3.1 for the bubbly flow problem. For porous media flow problem we use the weighted overlapping based deflation vector previously mentioned in Section 6.3.1.

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 we consider in both problems are numbered lexicographically with the most varying index being the x-direction followed by y and then z.

6.5.1 Bubbly flow problem

For the computational domain we use a 3D cube with dimensions $128 \times 128 \times 128$. This cube (Figure 4.1 Section 4.2) has nine bubbles whose density is different from the rest of the cube. The density contrast between outside and inside the bubble is 10^3. The domain is discretized using finite-difference discretization. This results in a septa-diagonal matrix for the 3D problem we have considered. For the iterative method, we use a relative stopping criterion 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 due to the fact that some sub-domains intersect the bubbles (refer section 4.3.1). All these vectors make the matrix Z which is then used for LSSD deflation.

In interest of the space constraints, in Table 6.4 to 6.3 we use some abbreviations for the preconditioning schemes. We provide their complete names before presenting the results.

1. MCSGS stands for multi-colored symmetric Gauss-Seidel 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 symmetric Gauss-Seidel 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.

We first present the results for the PCG method (Tables 6.2 and 6.3) and then for the DPCG method (Tables 6.4 and 6.5). We combine different preconditioners available in the PARALUTION library with deflation and compare their performance. In Tables 6.2 and 6.3 we see that the PCG method can be up to 5 times fast on the GPU compared to the CPU. Diagonal preconditioning emerges as the fastest preconditioning technique.

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

Table 6.2: Comparison of PCG schemes on the **CPU**. Bubbly flow problem.

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

Table 6.3: Comparison of PCG schemes on the **GPU**. Bubbly flow problem.

For the DPCG method the best speedup (2.5x) is close to half as compared to best speedup (5x) of the PCG schemes as observed in Tables 6.2 and 6.3. However, if we only consider iteration times then for the GPU version of deflated PCG method we still have up to four times speedup. The ME-ILU-ILU(0,1) preconditioner achieves fastest convergence (in 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 Table 6.2 and 6.3 with those in Table 6.4 and 6.5

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**

Table 6.4: Comparing **deflated** PCG for different preconditioners on **CPU**. Bubbly flow problem.

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

Table 6.5: Comparing **deflated** PCG for different preconditioners on **GPU**. Bubbly flow problem.

we see that deflation helps accelerate convergence in terms of 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 to other preconditioning techniques. With the addition of deflation it is further accelerated in terms of convergence. It still lacks in wall-clock time compared to 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 are 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 Ritzvalues (refer [58] and [70]). For this experiment we reduced the grid size to 32^3 while keeping the

number of bubbles the same. In Figure 6.1 we compare two different DPCG implementations with their PCG counterparts.

The spectrum of A has 8 small eigenvalues that correspond to the 9 bubbles in the system. We see in Figure 6.1 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 Table 6.2 to Table 6.5.

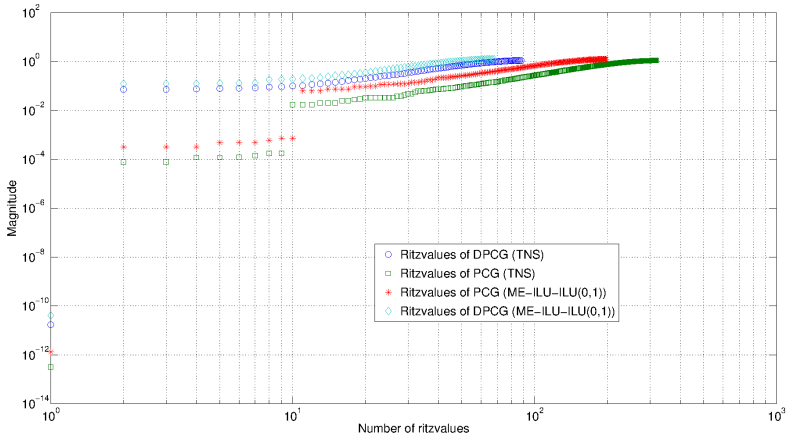


Figure 6.1: Comparison of Ritz values for DPCG and PCG. Preconditioners used are TNS-based and ME-ILU-ILU$(0,1)$.

6.5.2 Porous Media Flows

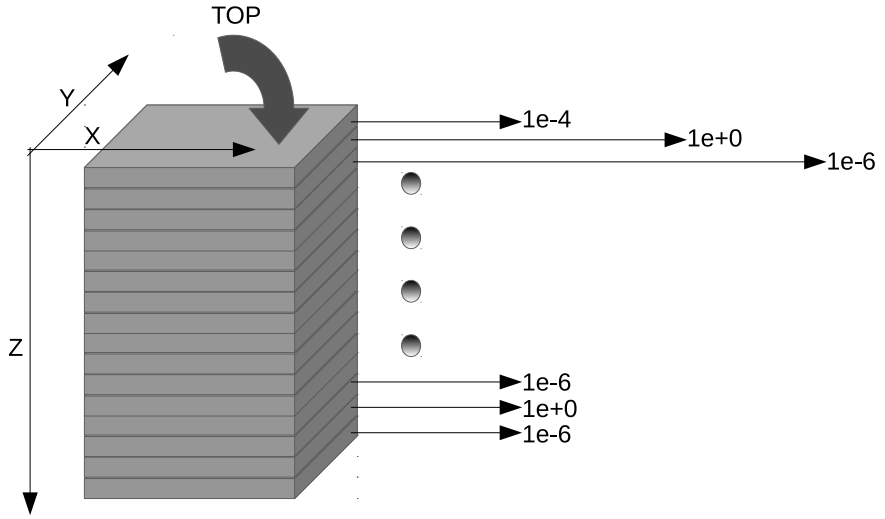
For porous media flow we consider two problems. One that is defined on a regular domain and one with an irregular geometry. For both problems we use finite-element discretization using parallelepiped elements. It must be noted that for a regular problem defined on a cuboidal domain these elements are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

The deflation vectors we use for this problem have been briefly described in Section 6.3.1. They are piece-wise constant but with additional factors added near the interface.

Problem defined on a regular geometry

The first problem we consider is defined on a regular geometry. This problem has 15 layers with contrast distribution as given in Figure 6.2. These 15 layers are arranged in slabs with dimensions $63 \times 64 \times 64$. Therefore the total number

of unknowns is $15 \times 63 \times 64 \times 64$. 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.



- Dirichlet boundary conditions on top
- Neumann boundary conditions on the rest of the sides

Figure 6.2: Layered problem. 15 layers with variable contrasts

In Tables 6.6 and 6.7 we see that the GPU implementations of PCG method for this problem can be between 3 to 4 times as fast as compared to their CPU implementations.

In Tables 6.8 and 6.9 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 to the CPU. This is the reason why the speedup is small. The setup times are larger due to the fact that 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 since it uses CSR storage format for all the matrices involved and uses latest versions of the CUSPARSE library. The reason why we use this method of general storage for the AZ matrix is because we use Z that is derived from the approximation of the eigenvectors of the problem. Such a Z may or may not have a structure in terms of distribution of non-zeros. However, if the non-zero pattern is known apriori (e.g. for sub-domain deflation vectors, Section 4.3), then it is possible to construct an AZ matrix with much less operations (and consequently also less

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

Table 6.6: Comparison of PCG schemes on the **CPU**. Layered problem.

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

Table 6.7: Comparison of PCG schemes on the **GPU**. Layered problem.

operations for $AZ \times x$) and therefore save on this setup time (and consequently solve time). Such an implementation has been studied in the appendices of the thesis work presented in [62].

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 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 done only once, e.g. problems coming from the study of seismic processes in the earths' interior the coefficient matrix and the other associated matrices are made only once, the GPU implementations can give a significant saving in execution time.

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

Table 6.8: Comparing **deflated** PCG for different preconditioners on **CPU**. Layered problem.

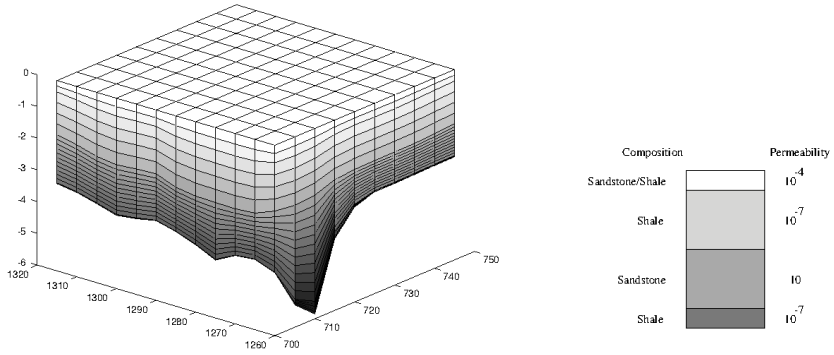
Preconditioner	Iterations	Setup time	Solve time	Total time
MCSGS	265	7.41	6.69	14.1
FSAI	221	14.32	6.41	20.73
ILU(0)	178	7.24	12.49	19.73
ME-ILU-J	254	7.37	6.06	13.43
ME-ILU-SGS	252	7.22	6.28	13.5
ME-ILU-ILU(0,1)	**139**	8.67	**5.03**	13.7
ILU(0,1)	256	7.41	6.12	13.53
DIAGONAL	528	**6.61**	9.75	16.36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

Table 6.9: Comparing **deflated** PCG for different preconditioners on **GPU**. Layered problem.

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 6.3 and refer [76] for details). The total number of unknowns in this problem are 146520. It contains 9 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. We only present results for ILU(0) based preconditioner in this section as for all other preconditioners the DPCG or the PCG method do not converge.

In Tables 6.10 and 6.11 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 (due to sparse matrix-matrix multiplications) in comparison to



(a) Geometry (b) Legend

Figure 6.3: Unstructured problem from oil industry

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

Table 6.10: PCG (with ILU(0) preconditioner) on **CPU** and **GPU**. Problem from the oil industry.

the CPU version and also compared to both versions of PCG implementation in Table 6.10.

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 re-numbering) to extract parallelism. However, for the other first-level preconditioners a reordering is done, 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 (MIS) ordering (available in PARALUTION) for 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 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).

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

Table 6.11: **deflated** PCG (with ILU(0) preconditioner) on **CPU** and **GPU**. Problem from the oil industry.

We also concluded that other preconditioners use sophisticated techniques to extract parallelization also comes in the way of convergence.

6.6 Experiments with varying grid sizes and density ratios

For the layered problem introduced in Section 6.5.2 Figure 6.2 we conducted additional experiments.

In Figures 6.4 and 6.5 we consider two different scenarios. In Figure 6.4 we observe how changing the contrasts in the layers of the problem affects the iteration count. In Figure 6.5 we show how the iterations required for convergence change with increasing number of unknowns.

The individual bars in Figures 6.4 and 6.5 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.

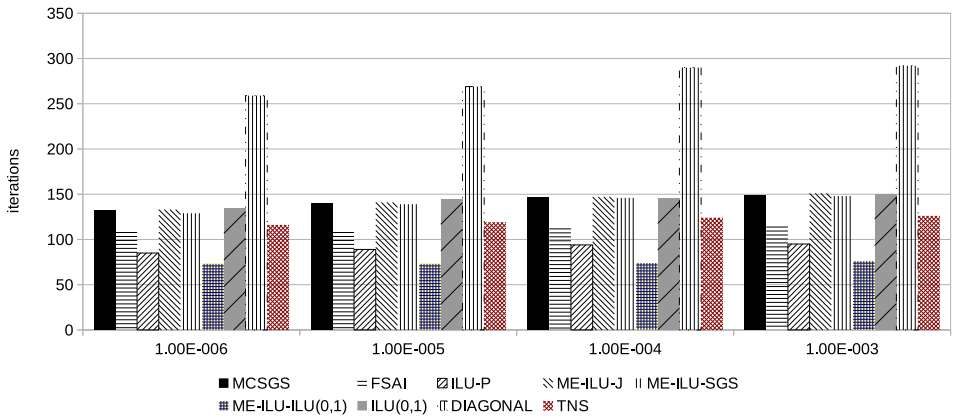


Figure 6.4: Layered Problem. 15 layers. The contrasts vary as $10^{-4}, [1, 10^{-k}]$ []repeats 6 more times where $k = 3, 4, 5, 6$.

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 we have used (in Figure 6.4) 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.

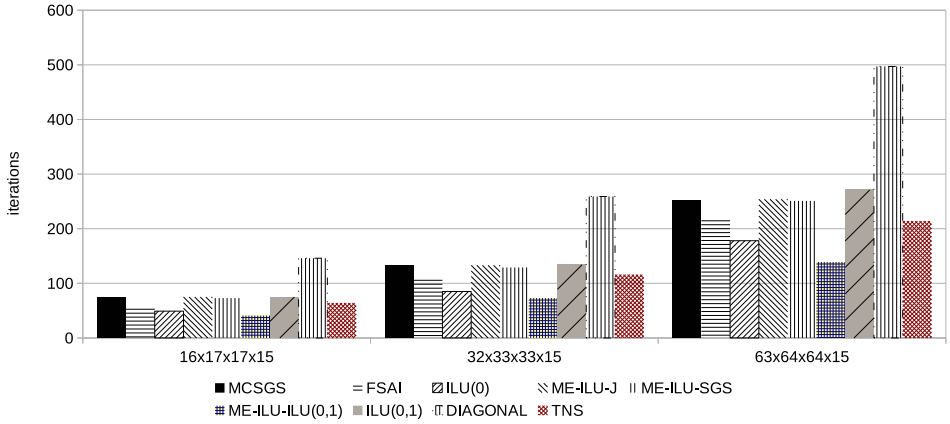


Figure 6.5: Layered Problem. 15 layers. The contrasts vary as $10^{-4}, [1, 10^{-6}]$ []repeats 6 more times. Three grid sizes are considered.

In Figure 6.5 we observe that ILU-type preconditioners require twice the number of iterations if grid sizes are increased by a factor two.

6.6.1 Using CG with Algebraic Multigrid (AMG) preconditioner for the layered problem and the problem from oil industry

Layered Problem

Corresponding to Figures 6.4 and 6.5 we also have results (Tables 6.12 and 6.13) for the implementation of Conjugate Gradient Method within PARALUTION along with Algebraic Multigrid(AMG) as a preconditioner.

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

Table 6.12: **CG-AMG results**. Layered Problem. 15 layers. The contrasts vary as $10^{-4}, [1, 10^{-k}]$ []repeats 6 more times where $k = 3, 4, 5, 6$.

As can be seen in Tables 6.12 and 6.13 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.

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

Table 6.13: **CG-AMG results**. Layered Problem. 15 layers. The contrasts vary as $10^{-4}, [1, 10^{-6}]$ []repeats 6 more times. Three grid sizes are considered.

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 to the fastest DPCG method on the CPU for the same problem (refer Table 6.8). On the other hand on the GPU (Table 6.13) 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 (refer Table 6.9) is $1.5\times$ better as compared to CG with AMG preconditioner.

Problem from oil industry

For this problem we also did experiments with the CG-AMG variant introduced for the previous problem which has a regular geometry (Table 6.14). The CG-AMG method converges for this problem. However, it takes a lot of iterations and takes much longer to converge. It provides no advantage over the DPCG method with ILU(0) preconditioning (refer Table 6.10). So, for this problem the DPCG method with ILU(0) is the best option.

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

Table 6.14: CG with AMG preconditioner on **CPU** and **GPU**. Problem from the Oil Industry.

6.7 Conclusion

In this chapter we have surveyed various preconditioning techniques one can use in the DPCG method to solve ill-conditioned linear systems arising from two different flow problems. We have compared 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 these problems.

Through our results we have shown that the combination of a simple preconditioner (like diagonal preconditioning for some problems) 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. This is also true for a stationary problem (e.g. from the domain of seismic processes) where setup is essentially done once. Such problems can benefit from our implementations.

CHAPTER 7

Conclusions

7.1 Introduction

In this thesis we have studied the implementation and optimization of the two-level preconditioned Conjugate Gradient method on the GPU. We have seen how it can accelerate convergence for different problems when effective deflation vectors are used. The two-level preconditioned method has also shown scalability in a parallel setup where the problem is broken down into many parts and the compute units co-operate to find a solution. In this chapter we highlight the important observations made from this research and provide ideas for future research on this topic.

7.2 Suitability

The Deflated Preconditioned Conjugate Gradient (DPCG) method can be efficiently mapped to the GPU. Specifically, the deflation operator can be broken down to individual steps (refer (2.34)) in order to achieve faster execution. In our implementation we have chosen to solve the inner system using the explicit inverse of the coarse system matrix E. This approach has proven to be very effective at bringing down the time it takes to do this operation in comparison to, e.g., solving the coarse system with an iterative method. At the same time, we have seen that this method has its pitfalls for deflation vectors that are bad approximations of the eigenvectors (row-based vectors, refer Chapter 3) corresponding to the small eigenvalues which delay convergence. Using vectors based on block sub-domains (refer Chapter 4) improves the reliability (since it is possible to solve for a much higher accuracy) of the choice

to solve the coarse system using the explicit inverse of E. Moreover, if we choose additional information from the problem, e.g. the level-set information then it is possible to construct vectors which are better approximations to the eigenvectors corresponding to the eigenvalues which we wish to deflate from the spectrum of the coefficient matrix A.

Other than deflation, using a preconditioning scheme that shows a higher degree of parallelism and comparable convergence behavior we get a further reduction in execution time. Compared to the implementation of well-known preconditioning techniques like (block) incomplete Cholesky on the GPU, the DPCG method using Truncated Neumann Series (TNS)-based preconditioner utilizes GPU resources much better.

7.3 Scalability

Even though the DPCG method is bandwidth-bound we have seen that it can demonstrate strong scalability. Through our experiments presented in Chapter 6 we could observe how this scalability can be affected. Namely, two important things must be taken care of to get better performance,

1. Choice of storage scheme for matrices A, L and L^T; and
2. Choice of preconditioning technique.

Choosing the storage format that maximizes the utilization of GPU memory bandwidth can give direct benefits in terms of reduced execution time for the SpMV operation. The Sparse Matrix Vector (SpMV) product operation emerges as the most expensive step in our implementation of the DPCG method. This is true because the preconditioning operation is implemented as repeated sparse matrix-vector products owing to the similarity of operations required to implement the preconditioner. The improvement in execution time is visible both for the TNS preconditioner and the diagonal preconditioner which we consider as first-level preconditioners in our experiments.

The choice of preconditioner can be decisive in the case when the number of processors is very high. We have tested the implementation of the DPCG method on the CPU for up to 1000 processors. For such a parallel code we observe that the communication time quickly becomes the most time-consuming part of the entire execution. This is where the choice of preconditioning becomes more important. In our case for the bubbly flow problem this is especially true when truncated Neumann series based preconditioner is used in the block-based data division scheme. Since the preconditioning operation involves communication with nearest neighbors, communication cost is inherent in this preconditioner. Not to mention the cost of global communication required for dot products.

If, however, diagonal preconditioning is chosen, the number of global dot products is almost twice that of the case when TNS preconditioner is used. This directly translates to a larger communication time percentage in the total time. However, in the case of TNS preconditioner as the first-level preconditioner, nearest neighbor communication is required and less global dot products (since TNS preconditioner converges at a fraction of the time required for DPCG with diagonal preconditioning) are needed. It is also possible to improve the TNS preconditioner by adding more terms to the approximation if global communication becomes prohibitive but at the same time the rising cost of nearest neighbor communication (and computation of additional terms) must be kept in check. Our choice of having three terms for the TNS preconditioner keeps the global communication much less than when diagonal preconditioning is used for first-level preconditioning.

7.4 Usability

We have tested our method for problems from porous media flow and bubbly flow within the framework of an open-source library (PARALUTION). Our comparisons for difficult problems show that the DPCG method (with a simple preconditioning scheme) has the potential to perform well both in terms of rate of convergence and computational wall-clock time. This was verified in Chapter 6 when we did comparative tests of the DPCG method using different first-level preconditioners. The iteration times for our implementations on the GPU are better in comparison to the CPU implementations and this extends the applicability of the DPCG method to problems with multiple right-hand sides. In such a situation the setup is done once and hence the cost of setup will be very well hidden by the advantage that solving multiple systems quickly can provide.

7.5 Suggestions for future research

7.5.1 Using newer programming paradigms

Through the course of this research we have tried different libraries for our implementations. Each of them have certain benefits but we have chosen the vendor specific libraries from NVIDIA since we have focused our code development for NVIDIA GPUs. However, several new programming paradigms have evolved during the span of this research. Most notably, OpenACC and OpenCL which can be used to write accelerated code for a larger number of parallel processing platforms. OpenACC can also be incorporated into our software as it is relatively simple to add (directives like OpenMP exist for par-

allel operations). OpenCL on the other hand will require much more coding effort as it is very detailed and verbose (much more than CUDA).

It would be also interesting to explore the implementation of our work onto FPGA (Field Programmable Gate Array) devices and benefit from their specialized hardware for some aspects of our implementation (e.g. level-set sub-domain deflation or a specific first-level preconditioning scheme).

7.5.2 Investigation of problems with irregular domains

In Chapter 6 we saw that for an irregular domain only ILU based first-level preconditioning works well. It would be interesting to explore why and how ILU based first level preconditioner performs well and a simple preconditioner like TNS or diagonal does not converge at all. It would also be worthwhile to explore how the irregularity of the domain could affect the choice of tolerance to which a system can be solved with the DPCG method with a particular choice of first-level preconditioning.

7.5.3 Improving scaling on multi-GPU

Through our results in the multi-GPU section we found out that communication could be the most time-consuming of all operations in the DPCG implementation. In order to improve scaling and to keep communication costs low it is possible to avoid communication by delaying it. One of the ways to achieve this is to delay the updation of residual and continue with the computation. The overall effect is of an improvement in execution time since the computation is relatively fast. Such techniques (refer [23]) can be employed for our DPCG implementation on large numbers of processors for better results.

7.5.4 Using better deflation vectors for multi-GPU implementations

In our multi-GPU and CPU implementations we have used a very simple method to construct deflation vectors for bubbly flow problems. Each compute unit was assigned one sub-domain so that construction of the intermediate matrices for deflation was simplified. However, we know from previous experiments that utilization of the level-set information coupled with block based sub-domains can accelerate convergence for such problems drastically. The work presented in [45] can be used in a parallel processing setup to make and track the level-set function across a time stepping simulation for better deflation implementation.

7.5.5 Applicability to other problems

Through out most of our research we have used a symmetric positive definite system that is based on a structured domain. A possible direction for an extension to this research could be more challenging problems which are non-symmetric and have irregular domains or use different schemes for discretization etc. An attempt in this direction is made in the appendices of this dissertation.

APPENDIX A

IDR(s) implementation in NVIDIA AmgX

A.1 Introduction

This Appendix describes the implementation of the IDR(s) method ([59] [72]) in the NVIDIA library AmgX[1].

IDR(s) is a Krylov subspace method that approximates the solution of a linear system by successively generating residuals in nested subspaces.

A.2 The IDR(s) method

The original IDR theorem (with a generalization to complex matrices) is stated as follows.

Theorem 1 (Induced Dimension Reduction (IDR))**.** *Let $A \in \mathbb{C}^{N\times N}$, $B \in \mathbb{C}^{N\times N}$ be a preconditioning matrix, let $Q \in \mathbb{C}^{N\times s}$ be a fixed matrix of full rank, and let $\mathcal{G}_0$ be any non–trivial invariant linear subspace of A. Define the sequence of subspaces $(\mathcal{G}_j)$ recursively as*

$$\mathcal{G}_{j+1} \equiv (I - \omega_{j+1}AB^{-1})(\mathcal{G}_j \cap Q^{\perp}) \quad for \quad j = 0, 1, \ldots, \tag{A.1}$$

where (ω_j) is a sequence in $\mathbb{C}^N$. If $Q^{\perp}$ does not contain an eigenvector of AB^{-1}, then for all $j \geq 0$

- $\mathcal{G}_{j+1} \subset \mathcal{G}_j$;

[1] https://developer.nvidia.com/amgx

- $\dim \mathcal{G}_{j+1} < \dim \mathcal{G}_j$ *unless* $\mathcal{G}_j = \{0\}$.

Proof. The proof for the IDR theorem is given in [59]. □

In the NVIDIA AmgX library we have implemented two variants of the IDR Algorithm. Algorithm 5 is called IDR(s) and Algorithm 6 is named as IDR-*minsync*(s).

In Algorithm 6 the IDR(s) algorithm as it appears in Algorithm 5 is reformulated to allow a reduction in communication. This is beneficial for implementation of the IDR(s) algorithm on distributed computing systems as it reduces the need for global communication. We use the so called IDR-*minsync*(s) algorithm for our multi-GPU implementation in AmgX.

The main difference between the two IDR(s) implementations is the way inner products are calculated. This is visible in the steps 14 to 19 of Algorithm 5 and in steps 15 to 18 of Algorithm 6. The IDR(s) Algorithm that appears in [72] and Algorithm 5, is the basis of our implementation of the IDR(s) Algorithm in AmgX. The IDR-*minsync*(s) Algorithm that appears in [16] and is cited in this appendix as Algorithm 6 forms the basis of our implementation in AmgX. For the IDR(s) Algorithm the steps 14 to 19 have a close resemblance to the Classical Gram-Schmidt (CGS) technique of orthogonalization whereas for the IDR-*minsync*(s) steps 15 to 18 are similar to the Modified Gram-Schmidt (MGS) approach.

A.3 AmgX

The NVIDIA AmgX library provides a number of iterative linear solvers. It is highly optimized for NVIDIA GPUs and aims to deliver up to an order of magnitude speedup on these GPUs. It supports openMP and MPI. It has a customizable structure so that many solvers can be combined with many different preconditioners. It contains an implementation of the Algebraic Multigrid method. The interface for this library is in C and C++.

A.3.1 Implementation of IDR(s) method in AmgX

The AmgX library already had the API for all of the BLAS routines (based on CUBLAS and CUSPARSE) when we started implementing the IDR(s) method within its framework. However, in the IDR(s) method it is often required to e.g. do a `gemv` operation from a sub-matrix of a larger matrix or do a `dot` operation over a certain range within a vector or within a column of a matrix. For these operations we had to write new function prototypes using templates.

In addition to this, for the multi-GPU implementations it was required to calculate partial dot products on different processors, combine them and then

Algorithm 5 IDR(s)-biortho with bi–orthogonalization of intermediate residuals

INPUT: $A \in \mathbb{C}^{N\times N}; x, b \in \mathbb{C}^N; Q \in \mathbb{C}^{N\times s}$; preconditioner $B \in \mathbb{C}^{N\times N}$; parameter s; accuracy ε.

OUTPUT: Approximate solution x such that $||b - Ax|| \leq \varepsilon$.

1: *// Initialisation*
2: Set $G = U = 0 \in \mathbb{C}^{N\times s}; M = [\mu_{i,j}] = I \in \mathbb{C}^{s\times s}; \omega = 1$
3: Compute $r = b - Ax$
4: *// Loop over nested $\mathcal{G}_j$ spaces, $j = 0, 1, \ldots$*
5: **while** $||r|| \geq \varepsilon$ **do**
6: *// Compute s linearly independent vectors g_k in $\mathcal{G}_j$*
7: $\boxed{\phi = Q^H r}$, $\phi = (\phi_1, \ldots, \phi_s)^T$ *// s inner products (combined)*
8: **for** $k = 1$ to s **do**
9: Solve $M\gamma = \phi$ for γ, $\gamma = (\gamma_k, \ldots, \gamma_s)^T$
10: $v = r - \sum_{i=k}^{s} \gamma_i g_i$
11: $\widetilde{v} = B^{-1}v$ *// Preconditioning step*
12: $u_k = \sum_{i=k}^{s} \gamma_i u_i + \omega\widetilde{v}$
13: $g_k = Au_k$
14: *// Make g_k orthogonal to $q_1, \ldots, q_{k-1}$*
15: **for** $i = 1$ to $k - 1$ **do**
16: $\boxed{\alpha = q_i^H g_k / \mu_{i,i}}$ *// $k - 1$ inner products (separate)*
17: $g_k \leftarrow g_k - \alpha g_i$
18: $u_k \leftarrow u_k - \alpha u_i$
19: **end for**
20: *// Update column k of M*
21: $\boxed{\mu_{i,k} = q_i^H g_k}$ for $i = k, \ldots, s$ *// $s - k + 1$ inner products (combined)*
22: *// Make the residual orthogonal to $q_1, \ldots, q_k$*
23: $\beta = \phi_k / \mu_{k,k}$
24: $r \leftarrow r - \beta g_k$
25: $x \leftarrow x + \beta u_k$
26: *// Update $\phi = Q^H r$*
27: **if** $k + 1 \leq s$ **then**
28: $\phi_i = 0$ for $i = 1, \ldots, k$
29: $\phi_i = \phi_i - \beta\mu_{i,k}$ for $i = k + 1, \ldots, s$
30: **end if**
31: **end for**

32: *// Entering $\mathcal{G}_{j+1}$, the dimension reduction step*
33: $\widetilde{v} = B^{-1}r$ *// Preconditioning step*
34: $t = A\widetilde{v}$
35: $\boxed{\omega = (t^H r)/(t^H t)}$ *// Two inner products (combined)*
36: $r \leftarrow r - \omega t$
37: $x \leftarrow x + \omega\widetilde{v}$
38: **end while**

redistribute them to different processors. Some wrappers for such operations with templates were also added within the library.

A.4 Experiments

For the experiments we consider two cases. One is a set of matrices called `atmosmodX` (where $X = d, l, j, m$). These matrices have been downloaded from the Florida matrix collection [18][2] . The other matrix results from the discretization of the reaction-convection-diffusion equation. We present results for two different kinds of preconditioning Multi-color-DILU (MDILU) and Block-Jacobi (BJAC) preconditioning. We also present results for one and four GPUs. In all our tables we report the number of matrix-vector multiplications performed by each method to achieve convergence followed by the total time taken in brackets. We compare the IDR(s) implementation in AmgX against Bi-CGSTAB and FGMRES implementations (in AmgX) for all our experiments. The FGMRES method we have used has a restart value of 100.

A.4.1 Setup

We perform our experiments on the PSG cluster at the NVIDIA Corporation headquarters in Santa Clara, California, USA. The machine we use has four K40 GPUs and a CPU. We do our experiments using one or all four of the GPUs.

A.4.2 Atmospheric problems - from Florida matrix collection

We choose four matrices from the Florida collection (Table A.1). These matrices have at least one million degrees of freedom and are not symmetric.

In Tables A.2 and A.3 we show the results of the four matrices introduced in Table A.1 for four different solution methods. In Table A.2 we present the

[2] http://www.cise.ufl.edu/research/sparse/matrices/

Algorithm 6 IDR(s)-minsync with bi–orthogonalization of intermediate residuals and with minimal number of synchronization points

INPUT: $A \in \mathbb{C}^{N\times N}; x, b \in \mathbb{C}^N; Q \in \mathbb{C}^{N\times s}$; preconditioner $B \in \mathbb{C}^{N\times N}$; accuracy ε.

OUTPUT: Approximate solution x such that $||b - Ax|| \leq \varepsilon$.

1: *// Initialisation*
2: $G = U = 0 \in \mathbb{C}^{N\times s}; M_l = I \in \mathbb{C}^{s\times s}, M_t = M_c = 0$: $\omega = 1$
3: Compute $r = b - Ax$
4: $\phi = Q^H r, \phi = (\phi_1, \ldots, \phi_s)^T$
5: *// Loop over nested $\mathcal{G}_j$ spaces, $j = 0, 1, \ldots$*
6: **while** $||r|| > \varepsilon$ **do**
7: *// Compute s linearly independent vectors g_k in $\mathcal{G}_j$ // s inner products (combined)*
8: **for** $k = 1$ to s **do**
9: *// Compute $v \in \mathcal{G}_j \cap Q^\perp$*
10: Solve $M_l \gamma_{(k:s)} = \phi_{(k:s)}$
11: $v = r - \sum_{i=k}^{s} \gamma_i g_i$
12: $\widetilde{v} = B^{-1} v$ *// Preconditioning step*
13: $\widehat{u}_k = \sum_{i=k}^{s} \gamma_i u_i + \omega \widetilde{v}$ *// Intermediate vector $\widehat{u}_k$*
14: $\widehat{g}_k = A\widehat{u}_k$ *// Intermediate vector $\widehat{g}_k$*
15: $\boxed{\psi = Q^H \widehat{g}_k}$ *// s inner products (combined)*
16: Solve $M_t \alpha_{(1:k-1)} = \psi_{(1:k-1)}$
17: *// Make $\widehat{g}_k$ orthogonal to $q_1, \ldots, q_{k-1}$ and update $\widehat{u}_k$ accordingly*
18: $g_k = \widehat{g}_k - \sum_{i=1}^{k-1} \alpha_i g_i, \quad u_k = \widehat{u}_k - \sum_{i=1}^{k-1} \alpha_i u_i$
19: *// Update column k of M_l*
20: $\mu^l_{i,k} = \psi_i - \sum_{j=1}^{k-1} \alpha_j \mu^c_{i,j}$ for $i = k, \ldots, s$
21: *// Make r orthogonal to $q_1, \ldots, q_k$ and update x accordingly*
22: $\beta = \phi_k / \mu^l_{k,k}$
23: $r \leftarrow r - \beta g_k$
24: $x \leftarrow x + \beta u_k$
25: *// Update $\phi \equiv Q^H r$*
26: **if** $k + 1 \leq s$ **then**
27: $\phi_i = 0$ for $i = 1, \ldots, k$
28: $\phi_i \leftarrow \phi_i - \beta \mu^l_{i,k}$ for $i = k + 1, \ldots, s$
29: **end if**
30: **end for**

31: *// Entering $\mathcal{G}_{j+1}$. Note: $r \perp Q$*
32: $\widetilde{v} = B^{-1}r$ *// Preconditioning step*
33: $t = A\widetilde{v}$
34: $\boxed{\omega = (t^H r)/(t^H t); \phi = -Q^H t}$ *// $s+2$ inner products (combined)*
35: $r \leftarrow r - \omega t$
36: $x \leftarrow x + \omega\widetilde{v}$
37: $\phi \leftarrow \omega\phi$
38: **end while**

Name	Non-zeros	dimensions
atmosmodd	8814880	1270432
atmosmodj	8814880	1270432
atmosmodl	10319760	1489752
atmosmodm	10319760	1489752

Table A.1: Matrix properties

results comparing Bi-CGSTAB, IDR(s) with subspace size one and eight and FGMRES. In Table A.3 we present the results for the IDR-*minsync*(s) version with same subspace sizes as IDR(s) method compared with Bi-CGSTAB and FGMRES on four GPUs.

In the results presented in Tables A.2 and A.3 we can see that the IDR(s) method is either comparable or better (in terms of required matvecs for convergence) than the Bi-CGSTAB and FGMRES methods for almost all matrices with or without preconditioning. However, the IDR(s) and IDR-*minsync*(s) methods take more time in comparison to the fastest method (Bi-CGSTAB with or without preconditioning).

In order to understand this behavior we profiled all the executions presented in Table A.2 and A.3.

In the following list we explain the abbreviations for the kernel names that are reported in the Tables A.4, A.5.

1. csrMv - This is the kernel responsible for calculation of the sparse matrix vector product between the coefficient matrix A and a vector.

2. gemv2N - This is a kernel that is called when a tall matrix is multiplied with a vector.

3. axpy - This is the CUBLAS call for the BLAS routine axpy.

4. copy - This is the CUBLAS call for the BLAS routine copy.

5. dot - This is the CUBLAS call for the BLAS routine dot.

	Bi-CGSTAB	IDR(1)	IDR(8)	FGMRES
No preconditioning				
atmosmodd	560(1.19)	540(1.72)	493(3.26)	1072(5.37)
atmosmodj	634(1.35)	504(1.6)	493(3.26)	3005(15.07)
atmosmodl	354(0.889)	394(1.46)	332(2.7)	368(2.139)
atmosmodm	236(0.595)	248(0.919)	239(2.30)	247(1.434)
MDILU-preconditioning				
atmosmodd	360(5.03)	376(5.75)	284(5.42)	451(7.62)
atmosmodj	398(5.56)	362(5.54)	299(5.69)	676(11.4)
atmosmodl	184(3.17)	182(3.39)	171(4.1)	203(4.18)
atmosmodm	152(2.61)	152(2.83)	126(3.1)	130(2.68)
BJAC-preconditioning				
atmosmodd	538(2.15)	538(2.83)	493(4.31)	1072(7.38)
atmosmodj	554(2.22)	578(3.03)	491(4.27)	3005(20.6)
atmosmodl	354(1.65)	426(2.60)	332(3.5)	368(2.93)
atmosmodm	256(1.19)	240(1.47)	239(2.6)	247(1.97)

Table A.2: Single GPU results IDR(s) with and without preconditioning for matrices from the Florida collection. Number of matvecs and time for execution in brackets.

	Bi-CGSTAB	IDR-*minsync*(1)	IDR-*minsync*(8)	FGMRES
No preconditioning				
atmosmodd	604(0.728)	746(1.08)	494(1.34)	1072(2.41)
atmosmodj	582(0.659)	1008(1.46)	497(1.34)	3005(6.73)
atmosmodl	352(0.541)	432(0.82)	323(1.12)	368(1.03)
atmosmodm	234(0.361)	272(0.526)	230(0.81)	247(0.70)
MDILU-preconditioning				
atmosmodd	352(1.72)	442(2.46)	279(1.94)	425(2.52)
atmosmodj	344(1.61)	582(3.24)	295(2.12)	664(3.88)
atmosmodl	186(1.14)	236(1.74)	185(1.74)	207(1.56)
atmosmodm	176(1.07)	196(1.44)	142(1.30)	147(1.12)
BJAC-preconditioning				
atmosmodd	574(1.26)	808(2.62)	494(2.19)	1072(3.58)
atmosmodj	518(1.32)	780(2.49)	493(2.18)	3005(10.36)
atmosmodl	362(1.14)	444(1.99)	323(2.07)	368(1.67)
atmosmodm	246(0.84)	286(1.29)	230(1.39)	247(1.14)

Table A.3: Multi(4)-GPU results. IDR-*minsync*(s) with and without preconditioning for matrices from the Florida collection. Number of matvecs and time for execution in brackets.

6. scal - This is the CUBLAS call for the BLAS routine scal.

7. thrust_calls - This is a set of kernels that are used by preconditioners implemented within AmgX for calculating sum of a vector or the result of a dot product.

8. misc - These are miscellaneous kernels that are called by the AmgX library.

Profiling, using the NVVP (NVIDIA visual profiler), generates the percentages of time spent in individual operations on the GPU. We have presented profiles only for the `atmosmodd` matrix.

We notice that for the IDR(s) method with block-Jacobi preconditioning a chunk of the time is spent in doing the `gemv` operation when a tall matrix (large number of rows and few columns) is multiplied with a vector. The `axpy` and `dot` routines also consume a large amount of the time for the IDR(s) implementations. This is not true for the multi-color-DILU preconditioning as then the preconditioner takes up the majority of the execution time.

A.4.3 Reaction-Convection-Diffusion equation

The linear reaction-convection-diffusion equation can be represented by

$$\frac{\partial u}{\partial t} = \mu \bigtriangledown^2 u + \bigtriangledown.(u\mathbf{v}) + au, \tag{A.2}$$

where $\mu > 0$ is the diffusivity of the temperature or concentration of some material which is represented by u. $\mathbf{v}$ is the velocity field of the flow. $a(\mathbf{x})$ is the reaction rate. $\bigtriangledown^2$ is the Laplace operator and $\bigtriangledown.(u\mathbf{v}) = div(u\mathbf{v}) = \sum_{i=1}^{n} \frac{\partial(uv_i)}{\partial x_i}$. This matrix has more than four million unknowns. It is discretized on a 3D-cartesian grid.

For the results presented in Table A.6 and Figure A.1 we notice how the IDR-*minsync*(s) method can be beneficial against the Bi-CGSTAB and FGMRES methods. Increasing s also improves the performance both in terms of wall-clock time and in convergence speed.

A.4.4 Variance in iterations between single and multi-GPU implementations

In Table A.6 we notice that the number of iterations between single and multiple GPUs differs by a considerable margin. In order to understand this we plotted the 2-norm of the residual for Bi-CGSTAB (refer Figure A.2) implementation with two different preconditioners on one and four GPUs. We saw that the residual has a very erratic behavior locally but the overall tendency

	No preconditioning			
Kernel Name	Bi-CGSTAB	IDR(1)	IDR(8)	FGMRES
csrMv	60.1	39.5	21.1	27.4
gemv2N		26.7	27.2	
axpy		13	25.3	39.4
copy	5.4	7.1	5.2	2.3
dot	10.2	5.5	16	25.8
scal				2.4
thrust_calls	21.5	7.6	4.3	1.9
misc	0.4	0.4	0.7	0.8
	MDILU-preconditioning			
Kernel Name	Bi-CGSTAB	IDR(1)	IDR(8)	FGMRES
DILU_precon	74.1	67.5	63.1	71.9
csrMv	18	16.4	13.9	15.4
gemv2N		5.6	8.9	
axpy		2.7	8.3	11.3
copy		2.2	2.3	
dot	1.5	1.1	5.2	7.4
scal				0.7
thrust_calls	6.3	4	3.5	2.8
misc				0.2
	Block-Jacobi preconditioning			
Kernel Name	Bi-CGSTAB	IDR(1)	IDR(8)	FGMRES
csrMv	67	49.6	32	39.8
gemv2N		16.9	20.7	
axpy		8.2	19.2	29.3
copy		6.6	5.4	
dot	5.7	3.5	12.2	19.1
scal				1.8
thrust_calls	26.9	14.7	8.9	9.4
misc	0.2	0.2	0.6	0.6

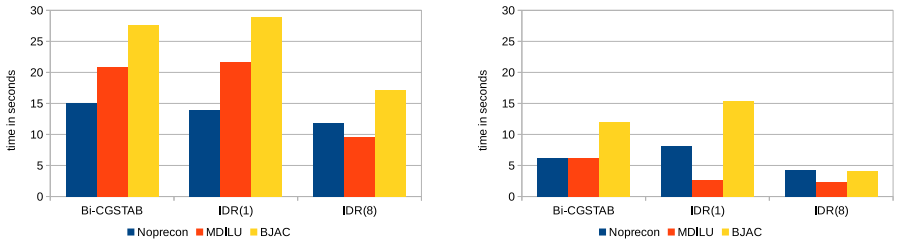
Table A.4: Percentage times taken by individual kernels for the `atmosmodd` matrix using different iterative methods.

	No preconditioning			
Kernel Name	Bi-CGSTAB	IDR-*minsync*(1)	IDR-*minsync*(8)	FGMRES
csrMv	55.1	37.4	17.9	25.3
gemv2N		25.1	46	
axpy		12.6	7.2	37.2
copy	5.2	5.6	3.6	2.3
dot	10.6	5.9	15.5	26
scal				2.3
thrust_calls	23.1	8.9	3.8	2.3
misc	4.4	2.8	4.5	4.2
	MDILU-preconditioning			
Kernel Name	Bi-CGSTAB	IDR-*minsync*(1)	IDR-*minsync*(8)	FGMRES
DILU_precon	69.7	63.1	50.7	57.2
csrMv	18.9	17.1	13.7	15.9
gemv2N		5.8	17.7	
axpy		2.9	2.8	11.9
copy		2.1	2	
dot		1.3	6	8.5
scal				0.7
thrust_calls	7	4.1	3.5	2.8
misc	1.1	0.9	0.7	1.6
	Block-Jacobi preconditioning			
Kernel Name	Bi-CGSTAB	IDR-*minsync*(1)	IDR-*minsync*(8)	FGMRES
csrMv	61.9	46.5	27.7	36.5
gemv2N		15.7	35.7	
axpy		7.8	5.6	27.9
copy		5.6	4	
dot	5.9	3.6	12.1	19.6
scal				1.7
thrust_calls	27.3	16.4	9.1	6.1
misc	4.1	3.1	3.9	3.8

Table A.5: Percentage times taken by individual kernels for the `atmosmodd` matrix using different iterative methods on 4 GPUs.

1 GPU		
Bi-CGSTAB	IDR(1)	IDR(8)
No preconditioning		
2322	1434	584
MDILU-preconditioning		
380	370	126
BJAC-preconditioning		
2282	1814	650
4 GPUs		
Bi-CGSTAB	IDR-*minsync*(1)	IDR-*minsync*(8)
No preconditioning		
2610	2612	698
MDILU-preconditioning		
486	188	126
BJAC-preconditioning		
2692	2640	441

Table A.6: Number of matvecs for IDR(s) (on 1 GPU) and IDR-*minsync*(s) (on 4 GPUs) results with and without preconditioning. The red colored results indicate no convergence due to breakdown. FGMRES does not converge for any of the cases.



(a) 1 GPU (b) 4 GPUs

Figure A.1: Execution times for `convection-diffusion reaction` equation using different iterative methods on 1 and 4 GPUs.

Percentage Time			
	No preconditioning		
Kernel Name	Bi-CGSTAB	IDR-*minsync*(1)	IDR-*minsync*(8)
csrMv	59.4	39.9	19
gemv_2N		26.9	49
axpy		13.2	7.3
copy	5.3	5.5	3.3
dot	10.1	5.6	15.3
thrust_calls	23.3	7.7	3.7
misc	0.5	0.5	1.6
	MDILU-preconditioning		
Kernel Name	Bi-CGSTAB	IDR-*minsync*(1)	IDR-*minsync*(8)
DILU_precon	73.9	67.3	54.5
csrMv	17.4	15.8	12.9
gemv2N		5.4	16.8
axpy		2.6	2.5
copy		1.8	1.7
dot	1.5	1.1	5.2
thrust_calls	6.2	3.4	2.9
misc	0.4	0.5	0.7
	Block-Jacobi preconditioning		
Kernel Name	Bi-CGSTAB	IDR-*minsync*(1)	IDR-*minsync*(8)
csrMv	66.3	49.6	29.3
gemv_2N		16.7	38
axpy		8.1	5.6
copy		5.6	3.9
dot	5.8	3.6	11.3
thrust_calls	26.5	14.8	8.7
misc	1.3	1.1	1.5

Table A.7: Percentage times taken by individual kernels for the matrix based on the discretization of the `convection-diffusion reaction` equation using different iterative methods on 4 GPUs.

is to converge. With a stronger preconditioning (DILU) the convergence is achieved faster but the behavior still seems erratic. This has to do with the parameters used in the construction used in the convection-diffusion-reaction matrix. The matrix is highly non-diagonally dominant and is very badly conditioned. This is the reason why we see the residual in Bi-CGSTAB method oscillating. It is because of the, round-off errors in the local parts of dot products and vector updates that are calculated in parts on multiple GPUs combined with the sensitivity of the method and the tough problem, that we see a difference in the number of iterations when number of GPUs are increased from one to four.

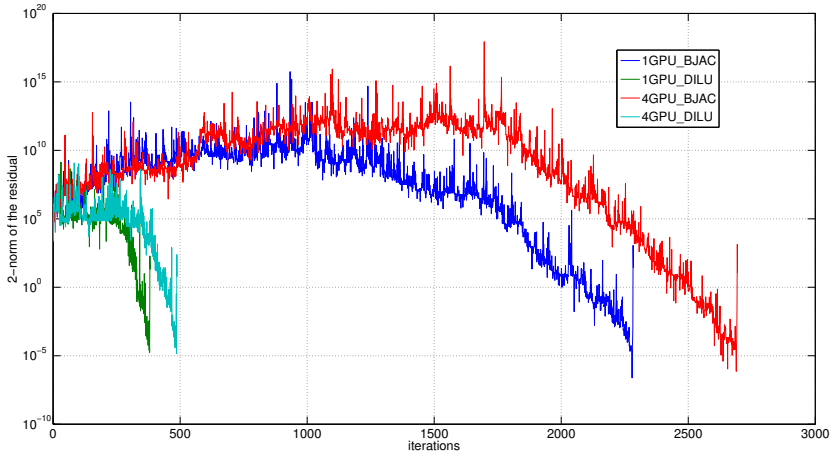


Figure A.2: Difference in iterations for Bi-CGSTAB across 1 and 4 GPUs for the `convection-diffusion-reaction` equation.

In Table A.6 the IDR method with subspace size 1 also seems to have a difference in number of iterations when executed on one and four GPUs. This is because for the single GPU case the IDR-(s) method without preconditioning and with block-Jacobi preconditioning the method breaks down in the Algorithm 5 at line 23 .

A.4.5 Profiling using NVVP

On profiling the execution on 4 GPUs (Table A.7) we found that for the matrix derived from the discretization of the reaction-convection-diffusion equation there is not much change in the profile compared to the 4 GPU profile for the `atmosmodd` matrix. However, because for this matrix the IDR(s) method is very well suited we see a reduction in execution times when we use four GPUs instead of one and also the IDR(s) methods have a faster convergence.

A.5 Conclusions

The implementation of the IDR(s) and IDR-*minsync*(s) methods in the AmgX library proves to be beneficial for matrices arising from non-trivial problems, compared to known methods like Bi-CGSTAB and FGMRES. Moreover, they show good scaling when multiple GPUs are used.

APPENDIX B

Using DPCG with matrices originating from Discontinuous Galerkin (DG) discretizations

In this chapter we present results of the DPCG method applied to matrices corresponding to physical problems with strong discontinuities. These matrices have been discretized using the Discontinuous Galerkin (DG) method. This discretization scheme is different from the discretizations used earlier in this thesis (finite-difference and finite element). We have restricted out test problem to two dimensions.

B.1 Introduction

Discontinuous Galerkin methods can be used to discretize partial differential equations. Mesh elements in DG methods are chosen as piece-wise polynomials instead of piece-wise constants. The degree (p) of this polynomial can be chosen for each mesh element. The polynomials can be discontinuous at element boundaries.

By increasing the degree of the polynomial per mesh element DG methods can achieve both a better accuracy and convergence compared to other methods for, e.g. Finite volume methods. The downside, however, is that with increasing polynomial degrees the number of unknowns per mesh element increases which results in a denser coefficient matrix. For problems having highly varying coefficients, the process of finding the solution to a linear system, that uses such a discretized matrix, becomes very challenging.

The coefficient matrix resulting from DG discretization has a block structure. Each block represents the discretization of a mesh element using a particular degree of a polynomial. If the polynomial degree is defined as p then the size of the block m, is given by

$$m = \frac{(p+1)(p+2)}{2}. \tag{B.1}$$

This structure of the coefficient matrix is different from the methods discussed earlier in this thesis which have 7 (3D) or 5 non-zero diagonals because of the finite difference discretization.

To solve the linear system attached to this coefficient matrix one can use the method of Conjugate Gradients with deflation and preconditioning. For the results presented in this chapter we use the DPCG method as discussed in the work of [73] (Algorithm 7) which has minor differences with the DPCG algorithm (Section 2.4 Algorithm 3) mentioned earlier in this thesis. The first difference is the way the coarse space is generated. For the problems presented in this Appendix and solved using the DPCG variant referenced in this Appendix from [73] the coarse space is chosen to correspond to the constant polynomial (zero-order). So the coarse system matrix $A_0 = RAR^T$. R is a restriction operator and is an $N \times N$ matrix where N is the number of unknowns. R is defined as

$$R = \begin{bmatrix} R_{11} & R_{12} & \cdots & R_{1N} \\ R_{21} & R_{22} & \cdots & R_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ R_{N1} & R_{N2} & \cdots & R_{NN} \end{bmatrix} \tag{B.2}$$

where the blocks have size $1 \times m$:

$$R_{ii} = \begin{bmatrix} 1 & 0 & \cdots & 0 \end{bmatrix}, R_{ij} = \begin{bmatrix} 0 & 0 & \cdots & 0 \end{bmatrix} \tag{B.3}$$

The second difference is the application of the deflation operator. Algorithm 3 involves separate application of the deflation matrix and then the preconditioning operation. In Algorithm 7, however, the deflation step Pr is implemented as follows

$$y_1 = \omega M^{-1} r \text{ pre-smoothing} \tag{B.4}$$

$$y = y_1 + Q(r - Ay_1) \text{ coarse correction} \tag{B.5}$$

where $Q = R^T A_0^{-1} R$ and $\omega = 1$. Matrix R is never stored but instead the operation involving the multiplication of a vector with R or R^T is performed using a selective copying of the source vector to a destination vector. This copy operation could be time-consuming on the GPU as values are picked

Algorithm 7 Deflated Preconditioned Conjugate Gradient Algorithm used in this chapter

1: Select x_0. Compute $r_0 := b - Ax_0$ and $\hat{r}_0 = Pr_0$, and set $p_0 := y_0$.
2: **for** i:=0,..., until convergence **do**
3: $\hat{w}_i := Ap_i$
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 $y_{i+1} = P\hat{r}_{i+1}$
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: $x_{it} := Qb + P^T x_{i+1}$

from one vector with offsets into another vector sequentially or vice versa. We will examine and comment about this operation in our results. Due to the block structure of the coefficient matrix we use a block-Jacobi preconditioner, M, also as it preserves the inherent structure of the matrix where each block corresponds to a mesh element. This has been previously studied in [73].

The matrix A_0 is similar to the matrix that results from the finite-difference discretization of the Poisson problem. It has 5 non-zero diagonals and is symmetric positive-definite. To solve the coarse system A_0 which results every time Q matrix is applied to a vector we use the DPCG method as discussed in Algorithm 3. We use truncated Neumann series based preconditioning for the first level and sub-domain deflation vectors for this system. The coarse system resulting from the deflation of the inner system associated with A_0 is solved explicitly using the inverse of the coarse matrix E (refer Section 2.34).

At a global level one can see this method as a nested deflation implementation since we use a variant of deflation (with the restriction and prolongation operators R and R^T) which results in a coarse system. This coarse system is then further coarsened using sub-domain deflation.

B.2 Problem definition

The problems we have chosen to test within our software are

1. Poisson Problem on a 2D grid,
2. Bubbly Flow problem on a 2D grid.
3. Inverse Bubbly Flow problem on a 2D grid.

The difference between the bubbly and the inverse bubbly problem is that the density contrast is inverted in between the bubbles and the surrounding medium. In the inverse bubbly problem the bubbles have a density of 10^{-5} and the surrounding medium has a density of 1.

B.2.1 Brief description about the design of custom software

The results presented in Section B.3 have been generated using software written for solving matrices (generated by DG discretization) using Algorithm 7. It has been implemented for the CPU and the GPU. It has been written in C++ and some parts use the API available in the PARALUTION library. The outer matrix A is stored in the CSR format on the CPU and GPU. The inner matrix A_0 is stored in the DIA format on the CPU and GPU. To solve the inner system we use the DPCG implementation within PARALUTION and write some helper functions in the DPCG class to make the Z matrix (in 2D) required for deflation.

The setup phase of the outer problem is executed entirely on the CPU (for both CPU and GPU implementations) and for the inner problem this is done on the CPU or GPU. Once the setup phase is over the entire algorithm listed in Algorithm 7 is run on the GPU and the result is copied back to the CPU.

In our results we also consider the case where we re-order the matrix in order to keep all the first rows of each block matrix (corresponding to each mesh element, N in total) together. This is followed by all the second rows of each block which are placed sequentially one after another. This is repeated for the m rows in each block matrix. So at the end of this row re-ordering we have a matrix with m blocks with N rows each from each mesh element. This re-ordering is then repeated for the columns in order to keep the properties of the matrix the same for the re-ordered case.

Doing this operation saves on the copies one has to do from a large vector on the fine grid to a small vector on the coarse grid. With reordering the copy operation involves only swapping pointers as the required parts of the vectors on both (coarse and fine) grids are in the initial (length $= N \times 1$) part of the large vector.

B.3 Numerical experiments

The experiments reported in this section have been performed on a single node of the DAS-4 cluster. Each node is equipped with a dual quad core Xeon processor and a NVIDIA K20 GPU with 5GB of memory. For the CPU version openMP is used and all 8 cores of the processor are used. The problem sizes we choose are 160^2, 320^2 and 640^2 but with different polynomial degrees p. The block size m increases with increasing p which leads to a larger total

number of unknowns = (number of grid points(N) $\times$ number of unknowns per mesh element(m)).

With this in mind we would like to state that in our results we are not showing the specific case when grid size is $N = 640^2$ and $p = 3$ so $m = 10$. This makes the total number of unknowns 4096000 and in this case the GPU memory falls short of being able to accommodate all matrices required to run the method on the GPU.

B.4 Results

In this section we show the results that were obtained with custom software for the matrices discussed in Section B.1. In all the results that are presented in this section the outer tolerance is kept to 10^{-6} and the inner tolerance is kept to 10^{-3}. The choice for a higher inner tolerance may seem non-intuitive at first but it has been tested and was first reported in [73]. The inner system is solved with deflated preconditioned Conjugate Gradient method (refer Section 2.4) with truncated Neumann series preconditioning (refer Section 3.2.2). The deflation vectors used are of the sub-domain variety (refer Section 4.3).

We consider three problems in this set of results and for each kind of problem we present results on two or three grid sizes across GPU and CPU. For each result we show the setup and solve time and also the time taken by the inner solver and the time to refine or coarsen the vector (for both unordered and re-ordered problems). For the unordered case this involves copying from a vector of size $Nm \times 1$ into a vector of size $N \times 1$. Only elements of a vector which are multiples of m are chosen and put successively into the new vector. The opposite operation involves copying from an $N \times 1$ vector to $Nm \times 1$ vector. For the re-ordered case this involves swapping the pointers of the $Nm \times 1$ vector with the $N \times 1$ vector and doing the same for the copying in the other direction. We also present results for the case when the matrix is reordered.

B.4.1 Poisson problem

Examining the results in Tables B.1 to B.3 we see that the GPU implementation can be more than two times faster than the CPU implementation. The setup times on the GPU dominate the total time. However, if more right hand sides are to be solved this cost can be hidden by the advantage in the smaller solve time. Inner solves on the GPU take only 50% of the time but on the CPU they take up to 80% of the solve time. Contrary to our assumption the application of R or R^T to a vector does not take the bulk of the total time. However, after re-ordering this time (to apply R or R^T) is further reduced.

CPU						
	p=1		p=2		p=3	
	NO reorder	reorder	NO reorder	reorder	NO reorder	reorder
Iters(outer)	24	24	20	20	22	22
Setup	0.1	0.12	0.26	0.34	0.66	0.86
Solve	0.47	0.48	0.7	0.7	1.44	1.54
multiplyR_Rt	0.01	0.001	0.01	0.001	0.02	0.001
Inner solve	0.24	0.24	0.2	0.2	0.24	0.23
Total	0.57	0.6	0.96	1.04	2.1	2.4
GPU						
Iters(outer)	24	24	20	20	22	22
Setup	0.1	0.14	0.3	0.38	0.73	0.92
Solve	0.34	0.34	0.31	0.31	0.39	0.41
multiplyR_Rt	0.001	0.001	0.001	0.001	0.001	0.001
Inner solve	0.3	0.3	0.25	0.25	0.29	0.29
Total	0.44	0.48	0.6	0.69	1.12	1.33

Table B.1: Results for grid size 160×160. Poisson problem. Deflation vectors used are 63.

The maximum speedup on the GPU (compared to the CPU) in solve time for the 640×640 problem is around a factor of 5.

B.4.2 Bubbly problem

The results for the bubbly problem (Table B.4 to B.6) follow the pattern of the Poisson problem with the maximum speedup of around 2.3. The bubbly problem is simpler in comparison to the inverse bubbly case for which we show the results in the next section.

This problem has the same number of iterations for convergence (for most cases) as the Poisson problem.

B.4.3 Inverse bubbly problem

The number of iterations for the inverse bubbly problem is almost double that of the bubbly flow problems. This also reflects in the solve times. Setup times are higher for this case amongst all the three problems but solve times are almost twice that of the Poisson problem. For the inverse bubbly problem the speedup is close to three if the total time is considered but for solve time it is around 6.

CPU						
	p=1		p=2		p=3	
	NO reorder	reorder	NO reorder	reorder	NO reorder	reorder
Iters(outer)	25	25	21	21	20	20
Setup	0.37	0.44	1.05	1.35	2.6	3.42
Solve	2.17	2.22	3.15	3.19	5.53	6.18
multiplyR_Rt	0.02	0.009	0.05	0.01	0.08	0.05
Inner solve	1.24	1.25	1.08	1.09	1.07	1.07
Total	2.54	2.66	4.2	4.54	8.13	9.6
	GPU					
Iters(outer)	25	25	21	21	20	20
Setup	0.4	0.47	1.15	1.44	2.85	3.68
Solve	1	1	0.95	0.96	1.09	1.15
multiplyR_Rt	0.01	0.01	0.01	0.01	0.01	0.01
Inner solve	0.89	0.89	0.78	0.78	0.77	0.77
Total	1.4	1.47	2.1	2.4	3.94	4.83

Table B.2: Results for grid size 320×320. Poisson problem. Deflation vectors used are 99.

B.5 Observations

In the results presented in the previous section we see that for higher order problems (with a higher value for p) on the CPU the solve time is no longer dominated by the inner solves. This is because a majority of the time is spent in the preconditioning and the sparse matrix vector product operations for the coarse mesh. We have used a generic format for storage of these matrices but a block storage format may prove to be beneficial for this kind of sparse matrix. We also notice that although reordering does reduce the time spent in multiplying with R and R^T, it is not the most time-consuming operation.

Inner solves take a significant amount of the total time and with a better choice of deflation vectors or by using a method like Multigrid it can be possible to reduce the solve time. The high setup time noticeable in some of the results can be hidden if many systems are solved with the same coefficient matrix.

	CPU			
	p=1		p=2	
	NO reorder	reorder	NO reorder	reorder
Iters(outer)	18	18	17	17
Setup	1.51	1.81	4.38	5.59
Solve	8.82	8.91	13.19	13.32
multiplyR_Rt	0.09	0.02	0.17	0.06
Inner solve	5.93	5.92	6.14	6.09
Total	10.33	10.73	17.57	18.91
	GPU			
Iters(outer)	18	18	17	17
Setup	1.59	1.9	4.62	5.9
Solve	2.43	2.43	2.72	2.75
multiplyR_Rt	0.02	0.16	0.02	0.01
Inner solve	2.19	2.19	2.23	2.23
Total	4.02	4.33	7.34	8.65

Table B.3: Results for grid size 640×640. Poisson problem. Deflation vectors used are 255.

CPU						
	p=1		p=2		p=3	
	NO reorder	reorder	NO reorder	reorder	NO reorder	reorder
Iters(outer)	25	25	24	24	26	26
Setup	0.1	0.12	0.28	0.36	0.7	0.91
Solve	0.5	0.51	0.86	0.87	1.76	1.91
multiplyR_Rt	0.01	0.008	0.01	0.009	0.0085	0.0078
Inner solve	0.26	0.26	0.25	0.25	0.27	0.27
Total	0.6	0.63	1.14	1.22	2.45	2.82
	GPU					
Iters(outer)	25	25	24	24	26	26
Setup	0.16	0.13	0.32	0.39	0.76	0.96
Solve	0.37	0.38	0.38	0.38	0.46	0.48
multiplyR_Rt	0.01	0.0089	0.01	0.0086	0.0085	0.0093
Inner solve	0.33	0.31	0.31	0.31	0.33	0.34
Total	0.53	0.51	0.69	0.77	1.22	1.44

Table B.4: Results for grid size 160×160. Bubbly problem. Deflation vectors used are 63.

CPU						
	p=1		p=2		p=3	
	NO reorder	reorder	NO reorder	reorder	NO reorder	reorder
Iters(outer)	21	21	22	22	22	22
Setup	0.38	0.45	1.11	1.4	2.74	3.58
Solve	1.86	1.89	3.4	3.46	6.21	7.05
multiplyR_Rt	0.02	0.0076	0.05	0.01	0.08	0.03
Inner solve	1.07	1.07	1.17	1.18	1.18	1.18
Total	2.24	2.35	4.51	4.85	8.95	10.63
	GPU					
Iters(outer)	21	21	22	22	22	22
Setup	0.41	0.49	1.21	1.49	3.11	3.83
Solve	0.87	0.87	1.04	1.04	1.23	1.28
multiplyR_Rt	0.01	0.01	0.01	0.01	0.01	0.01
Inner solve	0.78	0.77	0.85	0.85	0.85	0.85
Total	1.29	1.36	2.25	2.53	4.34	5.11

Table B.5: Results for grid size 320×320. Bubbly problem. Deflation vectors used are 99.

	CPU			
	p=1		p=2	
	NO reorder	reorder	NO reorder	reorder
Iters(outer)	18	18	17	17
Setup	1.49	1.78	4.46	5.61
Solve	9.17	9.23	13.27	13.54
multiplyR_Rt	0.08	0.03	0.16	0.06
Inner solve	6.36	6.27	6.18	6.2
Total	10.66	11.01	17.73	19.15
	GPU			
Iters(outer)	18	18	17	17
Setup	1.6	1.86	5.24	6
Solve	2.59	2.57	2.81	2.83
multiplyR_Rt	0.02	0.02	0.02	0.01
Inner solve	2.34	2.33	2.31	2.31
Total	4.19	4.43	8.05	8.83

Table B.6: Results for grid size 640×640. Bubbly problem. Deflation vectors used are 255.

CPU						
	p=1		p=2		p=3	
	NO reorder	reorder	NO reorder	reorder	NO reorder	reorder
Iters(outer)	37	37	39	39	43	43
Setup	0.1	0.12	0.29	0.36	0.69	0.91
Solve	0.8	0.82	1.41	1.42	2.89	3.19
multiplyR_Rt	0.01	0.0093	0.02	0.0079	0.03	0.01
Inner solve	0.46	0.46	0.42	0.42	0.47	0.49
Total	0.9	0.94	1.7	1.78	3.58	4.1
	GPU					
Iters(outer)	37	37	39	39	43	43
Setup	0.1	0.14	0.32	0.4	0.75	0.99
Solve	0.66	0.66	0.65	0.66	0.82	0.86
multiplyR_Rt	0.01	0.01	0.01	0.01	0.0083	0.01
Inner solve	0.6	0.6	0.55	0.55	0.62	0.62
Total	0.76	0.8	0.97	1.05	1.57	1.85

Table B.7: Results for grid size 160×160. Inverse bubbly problem. Deflation vectors used are 63.

CPU						
	p=1		p=2		p=3	
	NO reorder	reorder	NO reorder	reorder	NO reorder	reorder
Iters(outer)	36	36	39	39	43	43
Setup	0.37	0.46	1.15	1.45	2.71	3.62
Solve	3.07	3.12	5.77	5.86	11.82	13.57
multiplyR_Rt	0.03	0.01	0.08	0.02	0.16	0.05
Inner solve	1.73	1.73	1.79	1.77	2	2
Total	3.44	3.58	6.91	7.31	14.53	17.19
	GPU					
Iters(outer)	36	36	39	39	43	43
Setup	0.49	0.5	1.2	1.55	4.99	4.09
Solve	1.44	1.45	1.63	1.65	2.29	2.19
multiplyR_Rt	0.01	0.01	0.01	0.02	0.01	0.01
Inner solve	1.29	1.3	1.32	1.32	1.46	1.46
Total	1.93	1.95	2.83	3.2	7.28	6.28

Table B.8: Results for grid size 320×320. Inverse bubbly problem. Deflation vectors used are 99.

APPENDIX C

Multi-GPU results when matrices A, L and L^T are stored in COO format

No. of GPUs/node	1					
Vectors	2	4	8	16	32	64
Arrangement	(2,1,1)	(2,1,2)	(2,2,2)	(2,4,2)	(4,4,2)	(4,4,4)
Iterations	878	878	875	771	756	552
Global Setup	78.87	23.15	6.75	3	1.18	0.409
Local Setup	1.186	0.598	0.305	0.21	0.232	0.19
CG	92.529	47.5	24.06	11.44	6.208	2.54
spmv(Ax)	82.836	42.5	21.3	9.975	5.229	2.026
precon(scaling)	0.006	0.005	0.005	0.005	0.005	0.0037
Dot-daxpy-copy	5.8	3.01	1.6	0.795	0.481	0.238
deflation	3.81	2.008	1.086	0.615	0.392	0.218
Comm-mpi-glbl	0.106	0.088	0.111	0.287	0.377	0.1
Comm-mpi-NN	0.405	0.4	0.216	0.151	0.098	0.06
Comm-h2dd2h	1.809	1.016	0.565	0.48	0.254	0.11

Table C.1: Diagonal preconditioning based DPCG. Multi-GPU implementations. Storage of matrices in COO format. 1 GPU/node

No. of GPUs/node	2						
Vectors	2	4	8	16	32	64	128
Arrangement	(2,1,1)	(2,1,2)	(2,2,2)	(2,4,2)	(4,4,2)	(4,4,4)	(4,4,8)
Iterations	878	878	875	771	756	552	556
Global Setup	78.24	23.1	6.75	2.99	1.18	0.408	0.197
Local Setup	1.13	0.571	0.295	0.148	0.108	0.149	0.13
CG	93.23	47.6	25.13	11.89	6.53	2.73	1.38
spmv(Ax)	83.58	42.5	21.948	10.479	5.55	2.2	1.03
precon(scaling)	0.005	0.005	0.005	0.005	0.0049	0.0034	0.0026
Dot-daxpy-copy	5.79	3	1.594	0.788	0.474	0.233	0.13
deflation	3.8	2.04	1.527	0.582	0.446	0.238	0.173
Comm-mpi-glbl	0.07	0.201	0.215	0.16	0.324	0.324	0.19
Comm-mpi-NN	1.055	0.608	0.502	0.292	0.166	0.096	0.054
Comm-h2dd2h	1.59	0.894	0.501	0.434	0.231	0.095	0.066

Table C.2: Diagonal preconditioning based DPCG. Multi-GPU implementations. Storage of matrices in COO format. 2 GPUs/node

No. of GPUs/node	1					
Vectors	2	4	8	16	32	64
Arrangement	(2,1,1)	(2,1,2)	(2,2,2)	(2,4,2)	(4,4,2)	(4,4,4)
Iterations	377	377	375	331	323	251
Global Setup	74.8	22.44	6.781	3.01	1.21	0.452
Local Setup	0.581	0.309	0.164	0.227	0.164	0.219
CG	52.32	26.82	13.88	7.18	3.95	1.818
spmv(Ax)	36.02	18.19	9.17	4.31	2.23	0.924
precon(scaling)	12.48	6.62	3.62	2.249	1.31	0.65
Dot-daxpy-copy	2.1	1.09	0.586	0.297	0.182	0.1
deflation	1.65	0.873	0.48	0.284	0.19	0.1
Comm-mpi-glbl	0.07	0.06	0.0603	0.277	0.237	0.28
Comm-mpi-NN	1.09	0.585	0.472	0.326	0.198	0.11
Comm-h2dd2h	4.41	2.4	1.26	1.1	0.562	0.239

Table C.3: TNS-based preconditioning based DPCG. Multi-GPU implementations. Storage of matrices in COO format. 1 GPU/node

No. of GPUs/node	2						
Vectors	2	4	8	16	32	64	128
Arrangement	(2,1,1)	(2,1,2)	(2,2,2)	(2,4,2)	(4,4,2)	(4,4,4)	(4,4,8)
Iterations	377	377	375	331	323	252	251
Global Setup	74.81	22.39	6.71	3	1.2	0.431	0.208
Local Setup	0.533	0.318	0.206	0.074	0.236	0.169	0.146
CG	51.9	26.73	14.24	7.323	4.07	1.89	1.385
spmv(Ax)	36	18.24	9.48	4.5	2.38	1.02	0.633
precon	12.1	6.5	3.65	2.23	1.26	0.637	0.505
Dot-daxpy-copy	2.1	1.09	0.583	0.293	0.18	0.097	0.075
deflation	1.66	0.871	0.487	0.265	0.193	0.105	0.113
Comm-mpi-glbl	0.0879	0.066	0.149	0.0938	0.204	0.234	0.2
Comm-mpi-NN	1.1	0.703	0.887	0.497	0.257	0.157	0.089
Comm-h2dd2h	3.97	2.25	1.07	0.927	0.484	0.208	0.199

Table C.4: TNS-based preconditioning based DPCG. Multi-GPU implementations. Storage of matrices in COO format. 2 GPUs/node

Curriculum vitae

Rohit Gupta was born on December 2, 1981, in Kanpur, Uttar Pradesh, India. He received his secondary education at Modern Public School in Delhi, 1997-1999. He obtained his Bachelor of Technology degree at Guru Gobind Singh Indraprastha University, Delhi in 2004. Immediately after completing his bachelors he worked in a startup (VirtualWire Technologies Pvt. Ltd., New Delhi) in the area of Embedded software and hardware development. After working, learning and saving for 4 years he applied for a Masters program and got a partial (NXP) scholarship for his masters studies at TUDelft. He obtained his Master of Science degree in Computer Engineering at Delft University of Technology, The Netherlands in 2010.

During his Masters study he was bitten by the bug of parallel computing and he found a kind mentor in the form of Prof. Kees Vuik who gave him the opportunity to conduct research in this area. Not able to believe his luck he jumped into the cause of research in the area of applied mathematics not knowing it was going to be a bumpy ride. He eventually learned to handle the responsibilities of a research candidate. Not to mention that for almost all of the 4 years (2010-2014) of his research, he has been involved in burning significant computing time. He was the member of the Numerical Analysis group within the Delft Institute of Applied Mathematics. He was supervised and course-corrected by Prof. Dr. Ir. C. Vuik and Dr. Ir. M.B. van Gijzen.

During his research he collaborated with Prof. Dr. Ir. Johan van de Koppel from the Netherlands Institute of Ecology (NIOZ) (2010-2011). He worked on accelerating ecological models which described growth of mussels in the Oosterschelde. He also completed an internship at NVIDIA corporation and contributed to the Computational Linear Algebra library AmgX released by NVIDIA in 2013.

List of publications and presentations

Journal Papers

- Gupta, R., van Gijzen, M. B., & Vuik, C. Evaluation of the Deflated Preconditioned CG method to solve Bubbly and Porous Media Flow Problems on GPU and CPU, International Journal of Numerical Methods in Fluids, 2015. Published online 24th September 2015.

- Johan van de Koppel, Rohit Gupta, Cornelis Vuik. Scaling-up spatially-explicit ecological models using graphics processors, Ecological Modelling, Volume 222, Issue 17, 10 September 2011, Pages 3011-3019.

- Quan-Xing Liu, Ellen J. Weerman, Rohit Gupta, Peter M. J. Herman, Han Olff, Johan van de Koppel. Biogenic gradients in algal density affect the emergent properties of spatially self-organized mussel beds, J. R. Soc. Interface, Vol. 11, Number 96, Published 23 April 2014.

Book Chapter

- Gupta, R., van Gijzen, M. B., & Vuik, C. Efficient two-level preconditioned conjugate gradient method on the GPU (pp. 36-49). Proceedings of VECPAR 2012, Springer Berlin Heidelberg (2013). Editors: Michel Daydé, Osni Marques and Kengo Nakajima.

Refereed proceedings at International Conferences

- Rohit Gupta, Martin B. van Gijzen, and Kees Vuik. Multi-GPU/CPU deflated preconditioned Conjugate Gradient for bubbly flow solver. In Proceedings of the High Performance Computing Symposium (HPC 2014). Society for Computer Simulation International, San Diego, CA, USA, April 13-16 2014. Article 14 , 8 pages.

- Gupta, R.; van Gijzen, M.B.; Vuik, C.. 3D Bubbly Flow Simulation on the GPU - Iterative Solution of a Linear System Using Sub-domain and Level-Set Deflation, 2013 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), Feb. 27 2013-March 1 2013, pp.359,366.

Talks at International Conferences

- Deflated Preconditioned Conjugate Gradient for Bubbly Flows: Multi-GPU/CPU Implementations. February 18-21. SIAM Parallel Processing Conference 2014, Portland, Oregon, United States.

- 3D bubbly flow simulation on the GPU - Iterative Solution of a linear system using sub-domain and level-set deflation. February 27 - March 1. PDP 2013, Belfast, Northern Ireland, United Kingdom.

- Efficient Two-Level Preconditioned Conjugate Gradient Method on the GPU. July 17-20. VECPAR 2012 Kobe, Japan.

- Robust Preconditioned Conjugate Gradient for the GPU and Parallel Implementations. May 14-17. GTC 2012, San Jose, United States.

- Towards Efficient Two-Level Preconditioned Conjugate Gradient on the GPU. September 28-30. Facing the Multi-Core Challenge II Conference 2011 Karlsruhe, Germany.

- Deflated Preconditioned Conjugate Gradient on the GPU. September 20-23. GTC 2010, San Jose, United States

Other Talks

- Two Phase Flow using two levels of preconditioning on the GPU. Burgersdag 2011, TUDelft. 13th January 2011 in Delft, The Netherlands.

Poster Presentations at International Conferences

- Two Level Preconditioned CG Method on the GPU presented at Facing th Multi-Core Challenge II Conference 2011. September 28-30. Karlsruhe, Germany

Other Poster Presentations

- Two Phase Flow using two levels of Preconditioning on the GPU - presented at BurgersDag January 13 2011.
- Towards Efficient Preconditioned CG Method on the GPU for bubbly flow problem - presented at NWO-JSPS Joint Seminar April 10-13 2012.

BIBLIOGRAPHY

[1] Fernando Alvarado, Hasan Dag, O Alvarado Hasan Da, et al. Sparsified and incomplete sparse factored inverse preconditioners. *Matrix*, 1:13, 1992.

[2] M. Ament, G. Knittel, D. Weiskopf, and W. Strasser. 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, pages 583–592, Washington, DC, USA, 2010. IEEE Computer Society.

[3] Steven F. Ashby and Robert D. Falgout. A parallel multigrid preconditioned Conjugate Gradient algorithm for groundwater flow simulations. *Nuclear Science and Engineering*, 124(1):145–159, 1996.

[4] Jacques M. Bahi, Raphaël Couturier, and Lilia Ziane Khodja. Parallel GMRES implementation for solving sparse linear systems on GPU clusters. In *Proceedings of the 19th High Performance Computing Symposia*, HPC '11, pages 12–19, San Diego, CA, USA, 2011. Society for Computer Simulation International.

[5] Muthu Manikandan Baskaran and Rajesh Bordawekar. Optimizing sparse matrix-vector multiplication on GPUs using compile-time and run-time strategies. *IBM Reserach Report, RC24704 (W0812-047)*, 2008.

[6] 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*, Supercomputing, pages 18:1–18:11, New York, NY, USA, 2009. ACM.

[7] N. Bell and M. Garland. Implementing Sparse Matrix-vector Multiplication on Throughput-oriented Processors. In *SC '09: Proc. of the Conf. on High Perf. Computing Networking, Storage and Analysis*, pages 1–11. ACM, 2009.

[8] M.W. Benson. Iterative solution of large scale linear systems. Master's thesis, Lakehead University, Thunder Bay, Canada, 1973.

[9] Michele Benzi. Preconditioning techniques for large linear systems: A survey. *J. Comput. Phys.*, 182(2):418–477, November 2002.

[10] Michele Benzi and Miroslav Tûma. A sparse approximate inverse preconditioner for nonsymmetric linear systems. *SIAM J. SCI. COMPUT*, 19(3):1135–1149, 1998.

[11] Michele Benzi and Miroslav Tûma. A comparative study of sparse approximate inverse preconditioners. *Applied Numerical Mathematics*, 30(2):305–340, 1999.

[12] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the GPU: Conjugate Gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, July 2003.

[13] Luc Buatois, Guillaume Caumon, and Bruno Levy. Concurrent number cruncher: A GPU implementation of a general sparse linear solver. *Int. J. Parallel Emerg. Distrib. Syst.*, 24(3):205–223, June 2009.

[14] Ali Cevahir, Akira Nukada, and Satoshi Matsuoka. High performance Conjugate Gradient solver on multi-GPU clusters using hypergraph partitioning. *Computer Science - Research and Development*, 25(1-2):83–91, 2010.

[15] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. *SIGPLAN Not.*, 45(5):115–126, January 2010.

[16] Tijmen P. Collignon and Martin B. van Gijzen. Minimizing synchronization in IDR (s). *Numerical Linear Algebra with Applications*, 18(5):805–825, 2011.

[17] NVIDIA Corporation. CUDA programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[18] Timothy A. Davis and Yifan Hu. The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[19] Zdenäok Dostãl. Conjugate Gradient method with preconditioning by projector. *International Journal of Computer Mathematics*, 23(3-4):315–323, 1988.

[20] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 38(8):391–407, 2012.

[21] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of CUDA and OpenCL. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216–225. IEEE, 2011.

[22] Zhuo Feng and Zhiyu Zeng. Parallel multigrid preconditioning on graphics processing units (GPUs) for robust power grid analysis. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 661–666, June 2010.

[23] P. Ghysels, T. Ashby, K. Meerbergen, and W. Vanroose. Hiding global communication latency in the GMRES algorithm on massively parallel machines. *SIAM journal on Scientific Computing, ExaScience Lab Technical Report 04.2012.1*, 35:C48–C71. (24 pages), 2013.

[24] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[25] Marcus J. Grote and Thomas Huckle. Parallel preconditioning with sparse approximate inverses. *SIAM J. Sci. Comput.*, 18(3):838–853, May 1997.

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

[27] R. Gupta, D. Lukarski, M. B. van Gijzen, and C. Vuik. Evaluation of the deflated preconditioned CG method to solve bubbly and porous media flow problems on GPU and CPU. *International Journal for Numerical Methods in Fluids*, 2015.

[28] R. Gupta, M. B. van Gijzen, and C. Vuik. Efficient Two-Level Preconditioned Conjugate Gradient method on the GPU. Technical report, Delft University of Technology, Delft, The Netherlands, 2011. DIAM Report 11-15.

[29] R. Gupta, M. B. van Gijzen, and C. Vuik. 3D bubbly flow simulation on the GPU - iterative solution of a linear system using sub-domain and level-set deflation. In *Proceedings of PDP 2013*, pages 359–366. IEEE CPS, 2013.

[30] Rohit Gupta, Martin B. van Gijzen, and Cornelis Vuik. Multi-GPU/CPU deflated preconditioned conjugate gradient for bubbly flow solver. In *Proceedings of the High Performance Computing Symposium*, HPC '14, pages 14:1–14:8, San Diego, CA, USA, 2014. Society for Computer Simulation International.

[31] Rohit Gupta, Martin B. van Gijzen, and Kees Vuik. Efficient two-level preconditioned Conjugate Gradient method on the GPU. In Michel Dayd, Osni Marques, and Kengo Nakajima, editors, *High Performance Computing for Computational Science - VECPAR 2012*, volume 7851 of *Lecture Notes in Computer Science*, pages 36–49. Springer Berlin Heidelberg, 2013.

[32] A. Hart, R. Ansaloni, and A. Gray. Porting and scaling OpenACC applications on massively-parallel, GPU-accelerated supercomputers. *The European Physical Journal-Special Topics*, 210(1):5–16, 2012.

[33] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition, 2003.

[34] M.R. Hestens and Stiefel E. Methods of Conjugate Gradients for solving linear systems. *J. Res. Nat. Bur. Stand.*, 49:409–436, 1952.

[35] Vincent Heuveline, Dimitar Lukarski, Nico Trost, and Jan-Philipp Weiss. Parallel smoothers for matrix-based geometric multigrid methods on locally refined meshes using multicore CPUs and GPUs. In Rainer Keller, David Kramer, and Jan-Philipp Weiss, editors, *Facing the Multicore - Challenge II*, volume 7174 of *Lecture Notes in Computer Science*, pages 158–171. Springer Berlin Heidelberg, 2012.

[36] University of Tennessee Innovative Computing Laboratory. Magmablas documentation. http://icl.cs.utk.edu/magma/docs/.

[37] Dana A. Jacobsen and Inanc Senocak. Multi-level parallelism for incompressible flow computations on GPU clusters. *Parallel Comput.*, 39(1):1–20, January 2013.

[38] T. Jönsthövel, M. B. van Gijzen, S. P. MacLachlan, C. Vuik, and A. Scarpas. Comparison of the Deflated Preconditioned Conjugate Gra-

dient method and Algebraic Multigrid for composite materials. *Computational Mechanics*, pages 1–13, 2011.

[39] A. A. Kamiabad and J. E. Tate. Polynomial preconditioning of power system matrices with graphics processing units. In Siddhartha Kumar Khaitan and Anshul Gupta, editors, *High Performance Computing in Power and Energy Systems*, Power Systems, pages 229–246. Springer Berlin Heidelberg, 2013.

[40] Kamran Karimi, Neil G Dickson, and Firas Hamze. A performance comparison of CUDA and OpenCL. *arXiv preprint arXiv:1005.2581*, 2010.

[41] 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.*, 236(3):281–293, September 2011.

[42] H. Knibbe, C.W. Oosterlee, and C. Vuik. 3D Helmholtz Krylov solver preconditioned by a shifted Laplace multigrid method on multi-GPUs. In Andrea Cangiani, Ruslan L. Davidchack, Emmanuil Georgoulis, Alexander N. Gorban, Jeremy Levesley, and Michael V. Tretyakov, editors, *Numerical Mathematics and Advanced Applications 2011*, pages 653–661. Springer Berlin Heidelberg, 2013.

[43] L. Yu. Kolotilina and A. Yu. Yeremin. Factorized Sparse Approximate Inverse Preconditionings I: Theory. *SIAM J. Matrix Anal. Appl.*, 14:45–58, January 1993.

[44] Dilip Krishnan and Richard Szeliski. Multigrid and multilevel preconditioners for computational photography. *ACM Trans. Graph.*, 30(6):177:1–177:10, December 2011.

[45] Marcel Kwakkel, Wim-Paul Breugem, and Bendiks Jan Boersma. An efficient multiple marker front-capturing method for two-phase flows. *Computers and Fluids*, 63(0):47 – 56, 2012.

[46] John M Levesque, Ramanan Sankaran, and Ray Grout. Hybridizing S3D into an exascale application using OpenACC: an approach for moving to multi-petaflops and beyond. In *Proceedings of the International conference on high performance computing, networking, storage and analysis*, pages 15:1–15:11. IEEE Computer Society Press, 2012.

[47] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. Efficient sparse matrix-vector multiplication on $x86$-based many-core processors. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 273–282, New York, NY, USA, 2013. ACM.

[48] Dimitar Lukarski. PARALUTION project. http://www.paralution.com/.

[49] Dimitar Lukarski. *Parallel Sparse Linear Algebra for Multi-core and Many-core Platforms – Parallel Solvers and Preconditioners*. PhD thesis, Karlsruhe Institute of Technology, January 2012. http://digbib.ubka.uni-karlsruhe.de/volltexte/1000026568.

[50] M. Manikandan Baskaran and R. Bordawekar. Optimizing sparse matrix-vector multiplication on GPUs. Technical report, IBM, 2009.

[51] 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*, 31(137):pp. 148–162, 1977.

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

[53] R. A. Nicolaides. Deflation of Conjugate Gradients with applications to boundary value problems. *SIAM Journal on Numerical Analysis*, 24(2):pp. 355–365, 1987.

[54] C. Oosterlee and T. Washio. An evaluation of parallel multigrid as a solver and a preconditioner for singularly perturbed problems. *SIAM Journal on Scientific Computing*, 19(1):87–110, 1998.

[55] Fábio Henrique Pereira, Sérgio Luís Lopes Verardi, and Silvio Ikuyo Nabeta. A fast algebraic multigrid preconditioned Conjugate Gradient solver. *Applied mathematics and computation*, 179(1):344–351, 2006.

[56] M. Pernice and M.D. Tocci. A multigrid-preconditioned Newton-Krylov method for the incompressible Navier–Stokes equations. *SIAM Journal on Scientific Computing*, 23(2):398–418, 2001.

[57] H. Rudi and J. Koko. Parallel preconditioned Conjugate Gradient algorithm on GPU. *Journal of Computational and Applied Mathematics*, 236(15):3584 – 3590, 2012.

[58] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.

[59] P. Sonneveld and M.B. van Gijzen. IDR(s): A family of simple and fast algorithms for solving large nonsymmetric systems of linear equations. *SIAM Journal on Scientific Computing*, 31(2):1035–1062, 2009.

[60] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[61] J. M. Tang, S.P. MacLachlan, R. Nabben, and C. Vuik. A comparison of two-level preconditioners based on multigrid and deflation. *SIAM Journal on Matrix Analysis and Applications*, 31(4):1715–1739, 2010.

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

[63] J.M. Tang and C. Vuik. Deflated ICCG method solving the singular and discontinuous diffusion equation derived from 3-D multi-phase flows. In *ECCOMAS CFD 2006: Proceedings of the European Conference on Computational Fluid Dynamics*, 2006.

[64] J.M. Tang and C. Vuik. Efficient deflation methods applied to 3-D bubbly flow problems. *Electronic Transactions on Numerical Analysis*, 26:330–349, 2007.

[65] J.M. Tang and C. Vuik. New variants of Deflation techniques for pressure correction in bubbly flow problems. *Journal of Numerical Analysis, Industrial and Applied Mathematics*, 2:227–249, 2007.

[66] J.M. Tang and C. Vuik. On deflation and singular symmetric positive semi-definite matrices. *Journal of Computational and Applied Mathematics*, 206(2):603 – 614, 2007.

[67] Julien C. Thibault and Inanc Senocak. Accelerating incompressible flow computations with a pthreads-CUDA implementation on small-footprint multi-gpu platforms. *J. Supercomput.*, 59(2):693–719, February 2012.

[68] S. P. van der Pijl, A. Segal, C. Vuik, and P. Wesseling. A mass-conserving level-set method for modelling of multi-phase flows. *International Journal for Numerical Methods in Fluids*, 47(4):339–361, 2005.

[69] S. P. van der Pijl, A. Segal, C. Vuik, and P. Wesseling. Computing three-dimensional two-phase flows with a mass-conserving level set method. *Comput. Vis. Sci.*, 11(4-6):221–235, July 2008.

[70] H.A. van der Vorst. *Iterative Krylov Methods for Large Linear systems.* Cambridge University Press, Cambridge, 2003.

[71] A. van Duin. Scalable parallel preconditioning with the sparse approximate inverse of triangular matrices. *SIAM Journal on Matrix Analysis and Applications*, 20(4):987–1006, 1999.

[72] Martin B. Van Gijzen and Peter Sonneveld. Algorithm 913: An elegant IDR(s) variant that efficiently exploits biorthogonality properties. *ACM Trans. Math. Softw.*, 38(1):5:1–5:19, December 2011.

[73] P. van Slingerland and C. Vuik. Fast linear solver for diffusion problems with applications to pressure computation in layered domains. *Computational Geosciences*, 18(3-4):343–356, 2014.

[74] Francisco Vaźquez, G. Ortega, José-Jesús Fernández, and Ester M. Garzón. Improving the performance of the sparse matrix vector product with GPUs. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 1146–1151. IEEE, 2010.

[75] Mickeal Verschoor and Andrei C. Jalba. Analysis and performance estimation of the conjugate gradient method on multiple GPUs. *Parallel Comput.*, 38(10-11):552–575, October 2012.

[76] C. Vuik, A. Segal, J. A. Meijerink, and G. T. Wijma. The construction of projection vectors for a deflated ICCG method applied to problems with extreme contrasts in the coefficients. *J. Comput. Phys*, pages 426–450, 2001.

[77] Xiaoge Wang, Randall Bramley, and Kyle A. Gallivan. A necessary and sufficient symbolic condition for the existence of incomplete cholesky factorization. Technical report, Indiana University, 1995.

[78] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. Openacc - first experiences with real-world applications. In Christos Kaklamanis, Theodore Papatheodorou, and Paul G. Spirakis, editors, *Euro-Par 2012 Parallel Processing*, volume 7484 of *Lecture Notes in Computer Science*, pages 859–870. Springer Berlin Heidelberg, 2012.

[79] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 38:1–38:12, New York, NY, USA, 2007. ACM.

[80] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[81] P. Zaspel and M. Griebel. Solving incompressible two-phase flows on multi-GPU clusters. *Computers & Fluids*, 80(0):356 – 364, 2013. Selected contributions of the 23rd International Conference on Parallel Fluid Dynamics ParCFD2011, also available as INS Preprint no. 1113.