DELFT UNIVERSITY OF TECHNOLOGY

REPORT 10-20

GPU Implementation of Deflated Preconditioned Conjugate Gradient

R. Gupta, C. Vuik, and C.W.J. Lemmens

ISSN 1389-6520

Reports of the Delft Institute of Applied Mathematics

Delft 2010

GPU Implementation of Deflated Preconditioned Conjugate Gradient

R. Gupta *, C. Vuik *, C.W.J. Lemmens *

October 18, 2010

Abstract

A Linear System of the pressure equation resulting from the Finite Difference Discretization of Two-Phase Flows has been implemented and solved on the GPU. Conjugate Gradient(CG) Method was used to solve the linear system with two levels of preconditioning. After implementing Block Incomplete Cholesky on CG, Deflation was applied as a second level preconditioner to improve the convergence rate. The GPU versions of the code benefit from the parallelism available in deflation step. For improving the parallel properties of the preconditioning step we use the novel, Incomplete Poisson Preconditioning. The final version with two levels of Preconditioning demonstrated up to 20 times speedup in the complete solution when compared to a single core CPU for a system of 1 million unknowns. Analysis of the results and a brief report on our experiments is presented.

Keyword: Conjugate Gradient, Preconditioning, Deflation, GPGPU, CUDA

1 Introduction

Large Sparse Linear Systems are mostly solved by Preconditioned Krylov Methods combined with some form of coarse grid acceleration. Specifically the convergence of Conjugate Gradient(CG) Method can be improved by Preconditioning. Many of the important building blocks of these algorithms could be optimized for execution on parallel architectures. Recently, with the advent of General Purpose Computing on the Graphical Processing Unit (GPU) it is possible to achieve $10 - 100$ times reduction in computing times. However, some of the methods involved in the Solution of the systems of interest do not run optimally on the GPU. In this paper we show that it is possible to achieve 10 fold speedup for a combination of suitable building blocks.

*Delft University of Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft Institute of Applied Mathematics, P.O. Box 5031, 2600 GA Delft, The Netherlands, (r.gupta@student.tudelft.nl, c.vuik@tudelft.nl, kees.lemmens@tudelft.nl)

In this paper we use two levels of Preconditioning with the CG method to solve a linear system. This system arises from the discretization of the Pressure Correction Equation. This equation is the most time-consuming step in the solution of the Incompressible Navier-Stokes Equation using the Level Set Method. This method as suggested in [Pijl, Segal, Vuik, and Wesseling, 2005], is of interest to us in modeling Physical Systems, especially Bubbly Flows. The Partial Differential Equations describing the Pressure Correction have been discretized through the use of finite differences. The linear system is of the form

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

where n is the number of degrees of freedom. We assume that A is symmetric positive definite(SPD), i.e.,

$$A = A^T,\ y^T A y > \ 0 \ \forall \ y \ \in \mathbb{R}\ y \neq 0. \tag{2}$$

The linear system given by (1) is usually sparse and ill-conditioned. This means that there are few non-zero elements per row of A and also that the condition number $\kappa(A)$ is usually large. Put in other words, the ratio of the largest eigenvalue to the smallest is large and this leads to slow convergence of the Conjugate Gradient Method.

$$\kappa(A) : \ = \frac{\lambda_n}{\lambda_1} \tag{3}$$

where $0 < \lambda_1 \leq \lambda_2 \leq ... \leq \lambda_n$ are eigenvalues of matrix A. See [Saad, 2003] for more details.

In order to have a smaller number of iterations for convergence, the matrix A is *pre-conditioned* to bring down the condition number from $\kappa(A)$ to $\kappa(M^{-1}A)$. The coefficient matrix A is multiplied by M^{-1}, the preconditioner (discussed in detail in [Meijerink and Vorst, 1977]). The original system (1) is then transformed into ,

$$M^{-1}Ax = M^{-1}b, \tag{4}$$

where M is SPD. M^{-1} is chosen in such a way that the cost of the operation $M^{-1}y$ with a vector y is computationally cheap. However, sometimes preconditioning might also not be enough. In that case we use second level of preconditioning or Deflation in order to reduce $\kappa(A)$. Details about the method could be found out in [Tang and Vuik, 2007].

In this work some previous results [Tang and Vuik, 2008] are used for implementation. The focus is to implement these methods on the Graphical Processing Unit (GPU). Recently Scientific Computing has largely benefited from the data parallel architecture of graphical processors. Many interesting problems which are computationally intensive are ideally suited to the GPU, especially matrix calculations. It is only intuitive to use them for solution of discretized partial equations. With the advent of the Component Unified Device Architecture (CUDA) paradigm of computing available on NVIDIA GPU devices, it has become easier to write such applications. First we take up the work already done on the GPU with respect to these iterative methods in the following section. We also bring out the contribution this study makes that is different from earlier results. Then we

briefly define the problem of Bubbly flows and a few words about the Two-Phase Matrix that we are interested in solving in Section 3. We take a short tour of the Iterative Solution Methods in Section 4 followed by some parallel implementations of preconditioners in Section 5. Finally we discuss implementation of the Conjugate Gradient Method with Preconditioning on the GPU in Section 6 supported by Numerical Results in Section 7 . We end this paper with conclusions in Section 8. For more details we refer to [Gupta, 2010].

2 Related Work

A variety of studies have been done to study Iterative Solution Methods on the GPU. Some of them discuss optimizations of basic building blocks and optimization techniques which we found useful for our work.

2.1 Sparse Matrix Vector Products- SpMVs

Sparse Matrix Vector(SpMV) Products take up the majority amount of execution time during the iterations of the Conjugate Gradient Algorithm. NVIDIA recently released a study, [Bell and Garland, 2008] in which they compare different sparse matrix storage formats and suggest some new methods and representative kernels for achieving upto 36 GFlop/s in Single Precision implementations of the SpMV kernel. They also compare the performance of the GPU with several architectures like STI Cell, and CPUs like Xeon, Opteron etc.

A recent study, [Monakov and Avetisyan, 2009] suggests to store a Sparse Matrix in a hybrid ELL-COO format to achieve maximum throughput in the SpMV kernel. Their method relies on an initial sweep on the matrix to find out the number of non-zero elements and the decision to divide the matrix into two different formats(ELL and COO) for storage.

The CUDA library can also be enriched with CUDPP [Harris, Sengupta, Owens, Tseng, Zhang, and Davidson, 2009] which provides a routine *cudppSparseMatrix*, for sparse matrix vector multiply which comes in handy when applying iterative methods. To use a method the user first declares a *Plan* in which the input output arrays, the number of elements etc. are specified

In [M. Baskaran and Bordawekar, 2008] improvements are demonstrated over the methods discussed above in [Bell and Garland, 2008] and [Harris, Sengupta, Owens, Tseng, Zhang, and Davidson, 2009] by exploiting some of the architectural optimizations to the Sparse Matrix-Vector Multiplication code. In particular the optimization efforts are centered on the following four guidelines:

- Exploiting Synchronization-Free Parallelism,
- Optimized Thread Mapping,
- Aligned Global Memory Access;

- Data-Reuse.

We utilize the knowledge given in [Bell and Garland, 2008] to write our own version of the Sparse Matrix Vector Multiplication Kernel and achieve a much higher memory throughput.

2.2 Conjugate Gradient

Aligning conjugate gradient method to the GPU has been discussed in [Georgescu and Okuda, 2007]. They also discuss the problems with precision and implementing preconditioners to accelerate convergence. In particular they state that for double precision calculations problems having condition numbers less than 10^5 may converge and give a speed-up also. They however warn that above a threshold value of the condition number the Conjugate Gradient Method will not converge. This observation relates to the limited(double) precision performance available on current GPUs.

Implementing single precision iterative solvers on the GPU is explored in [Buatois, Caumon, and Levy, 2009]. They limit the preconditioning to Jacobi-Type preconditioners. Further they report that for a limited number of iterations the GPU is able to provide a solution of comparable accuracy but as the iterations increase the precision drops in comparison to the CPU. In our results the relative error norm of the solution and the number of iterations required for convergence remain the same on the device(GPU) and the host(CPU).

2.3 Preconditioning

Techniques that are basically dependent on the Sparse Matrix Vector Multiply discussed in previous sections have been suggested in literature for accelerating Preconditioning of Iterative Solvers like GMRES and Conjugate Gradient. In [Wang, Klie, Parashar, and Sudan, 2009] an ILU Block Preconditioner is used, which has poor convergence qualities but is easier to parallelize, for solving a sparse linear system by the GMRES method. Coefficient matrix A is divided into equal sized sub-matrices which are then locally decomposed using ILU, as shown in Figure 1. The blocks shown in Figure 1 do not communicate to each other during the decomposition and also in solving it, this scheme fits well in the data parallel paradigm.

In [Asgasri and Tate, 2009] it is shown how a Chebyshev polynomial based preconditioner could be utilized for achieving speedups in the Conjugate Gradient method. The said preconditioner effectively reduces the condition number of the coefficient matrix thereby achieving convergence quickly. It approximates the inverse of the coefficient matrix with linear combinations of matrix-valued Chebyshev polynomials.

In [Ament, Knittel, Weiskopf, and Straβer, 2010] a new kind of preconditioning called the Incomplete Poisson Preconditioning is presented which takes the approximation of the preconditioner as follows

$$M^{-1} = KK^T \tag{5}$$

where

$$K = I - LD^{-1}. \tag{6}$$

In equation 6, L is the lower triangular part of A and D is the diagonal of A. This is comparable to an SSOR type preconditioner.

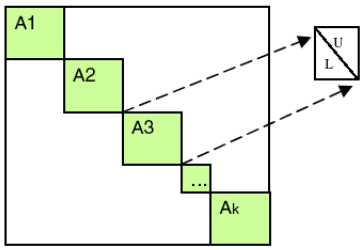

Figure 1: Block ILU preconditioner

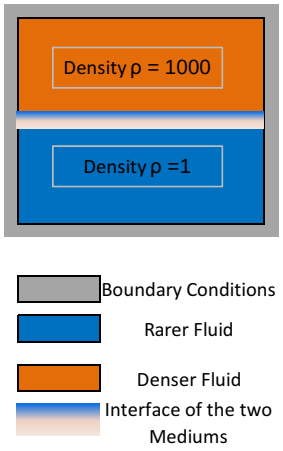

Figure 2: Two-Phase Flow Computational Model

This preconditioner introduces some fill-in (other than the normal sparsity pattern of A) in the multiplication of K with K^T. Their experiments suggest that the fill-in could be dropped due to it's comparatively small effect on the preconditioning process. The convergence might suffer however the method then maps well to the GPU. Hence the Incomplete nature of the stencil that emerges gives the method its name. We utilize a combination of IP Preconditioning and Deflation which shows significant performance benefits.

2.4 Precision Improvement

The GPUs have a comparatively low performance to Double Precision Computing. In [Baboulin, Buttari, Dongarra, Kurzak, Langou, Langou, Luszczek, and Tomov, 2008] double precision calculations are used for some part of the iterative method and single precision for others. Thus, achieving a trade-off that meets precision criteria and converges as good as the double precision case. At the same time the rate of convergence is also not affected very much. They have reported the results for a non-symmetric solver wherein the outer iteration is FMGRES and the inner one (for calculating M^{-1}) is a GMRES cycle. The idea here is that a single precision arithmetic matrix-vector product is used as a fast approximation of the double precision operator in the inner iterative solver.

All of our experiments have been in Single Precision. We progressively increase the performance gains by exposing higher levels of parallelism through the use of parallel Preconditioning Methods (IP) in place of Block Incomplete Cholesky Preconditioning which has comparatively less degree of parallelism. We also show how using architectural provisions within the GPU (efficient coalescing, better data structures, shared memory) one can improve the deflation kernels. The end result being that we are able to produce speedups of more than 20 times for a two level preconditioned Conjugate Gradient Solver. We have got up to 5 times speedup for the Block Incomplete Cholesky Preconditioned Deflated CG Solver. We are able to achieve up to 68 GFlops/s on NVIDIA Tesla Hardware. To our knowledge this is the first study that utilizes two level preconditioning on the GPU.

3 Problem Definition

Computations of Two-Phase (Bubbly) flows is the main application for this implementation. Two phase flows are complicated to simulate, because the geometry of the problem typically varies with time, and the fluids involved have very different material properties. This leads to large differences in the Matrix coefficients resulting from the discretized Pressure Correction Equation. Mathematically bubbly flows are modeled using the Navier Stokes equations including boundary and interface conditions, which can be approximated numerically using operator splitting techniques. In these schemes, equations for the velocity and pressure are solved sequentially at each time step. In many popular operator-splitting methods, the pressure correction is formulated implicitly, requiring the solution of a linear system (1) at each time step. This system takes the form of a Poisson equation with discontinuous coefficients and Neumann boundary conditions, i.e.,

$$-\bigtriangledown .\left(\frac{1}{\rho(x)}\bigtriangledown p(x)\right) = f(x), x \in \Omega, \tag{7}$$

$$\frac{\partial}{\partial \mathbf{n}} p(x) = g(x), x \in \partial\Omega, \tag{8}$$

where Ω, p, ρ, x and $\mathbf{n}$ denote the computational domain, pressure, density, spatial coordinates, and the unit normal vector to the boundary, $\partial\Omega$, respectively. Right-hand sides f and g follow explicitly from the operator-splitting method, where g is such that mass is conserved, leading to a singular but compatible linear system (1). In an earlier work [Pijl, Segal, Vuik, and Wesseling, 2005] the subject has been dealt at length about how the Navier Stokes Equation is utilized to model such a flow. In our experiments we are interested in Solving the Linear System that results from discretization of equation 7. In Figure 2 we present the simplified Computational Domain that we work with in our experiments for generating a Two-Phase Matrix. We use a 5-point Stencil for a 2-D grid($n \times n$) with $N = n \times n$ unknowns.

The square domain in the Figure 2 is divided into two parts and an interface and surrounded by Neumann Boundary conditions. Finite Difference Discretization translates the Grid (imposed over the domain) to a matrix which has coefficients placed on the 5

diagonals with the jumps appearing at the interface region. We follow a mass conserving approach while calculating the coefficients on the interface. There is some flux that enters horizontally and vertically and some of it leaves a cell. By taking the cell-centered approach (wherein the discretization point is at the center of the cell) and taking into account the contribution of all the flows through that point, we arrive on stencils for individual points on the grid. An example Stencil can be like

$$[-1 \qquad 0 \qquad (1\frac{1}{2} + 1\frac{1}{2}\epsilon) \qquad (-\frac{1}{2} - \frac{1}{2}\epsilon) \qquad -\epsilon]. \tag{9}$$

for a point on the interface and also adjoining the boundary. Here ϵ is the ratio of the densities of the two mediums.

4 Iterative Solvers

We choose the Conjugate Gradient Method for solving the Linear System arising from the discretization of the Pressure Equation.

4.1 Conjugate Gradient

The algorithm for Conjugate Gradient is given by [Saad, 2003].

Algorithm 1 Conjugate Gradient Algorithm

1: Compute $r_0 := b - Ax_0, p_0 := r_0$.
2: **for** $j = 0, 1, ...,$ until convergence **do**
3: $\alpha_j := (r_j, r_j)/(Ap_j, p_j)$
4: $x_{j+1} := x_j + \alpha_j p_j$
5: $r_{j+1} := r_j - \alpha_j Apj$
6: $\beta_j := (r_{j+1}, r_{j+1})/(r_j, r_j)$
7: $p_{j+1} := r_{j+1} + \beta_j p_j$
8: **end for**

To improve the rate of convergence of the CG method we apply subsequent levels of preconditioning.

4.2 Preconditioning

After Preconditioning we apply Conjugate Gradient to the system

$$M^{-1}Ax = M^{-1}b \tag{10}$$

where $M^{-1}A$ comes closer to I so that the method converges to the solution much faster as compared to plain CG. A host of Preconditioning methods are known [Saad, 2003]. ILU Preconditioning and Incomplete Cholesky are the most popular.

4.2.1 Diagonal Preconditioning

Diagonal or Jacobi Preconditioning is the simplest (and also the least effective) of the preconditioning methods that can be applied to the linear system $Ax = b$. The preconditioner matrix in this case is the main diagonal of A.

4.2.2 Incomplete Cholesky Preconditioning

Incomplete Cholesky Preconditioning involves a preconditioner of the form

$$M = LL^T \tag{11}$$

where L is lower triangular. It is made 'incomplete' by dropping off some of the elements. From the Cholesky factor we take the non-zeros that overlap with the sparsity pattern of the lower triangular part of A.

4.2.3 Second Level Preconditioning

To improve the convergence of our method we also use a second level of preconditioning. Deflation is an attempt to treat the remaining bad eigenvalues from the preconditioned matrix, $M^{-1}A$. This operation reduces the convergence iterations for the Preconditioned Conjugate Gradient (PCG) method and makes it more robust.

The linear system can be solved by employing the splitting

$$\begin{aligned} x = (I - P^T)x + P^T x \Leftrightarrow x &= Qb + P^T x && (12)\\ \Leftrightarrow Ax &= AQb + AP^T x && (13)\\ \Leftrightarrow b &= AQb + PAx && (14)\\ \Leftrightarrow Pb &= PAx, && (15) \end{aligned}$$

where

$$P = I - AQ, Q = ZE^{-1}Z^T, E = Z^T AZ. \tag{16}$$

Here $E \in \mathbb{R}^{k\times k}$ is the invertible Galerkin Matrix, $Q \in \mathbb{R}^{n\times n}$ is the correction Matrix, and $P \in \mathbb{R}^{n\times n}$ is the deflation operator. Z is the so-called 'deflation-subspace matrix' whose k columns are called 'deflation' vectors or 'projection' vectors. The x at the end of the expression is not necessarily a solution of the original linear system, since it might contain components of the null space of PA, $\mathcal{N}(PA)$. Therefor this 'deflated' solution is denoted as $\hat{x}$ rather than x. The deflated system is now

$$PA\hat{x} = Pb. \tag{17}$$

The Preconditioned deflated version of the Conjugate Gradient Method can now be presented. The deflated method (17) can be solved using a symmetric positive definite (SPD) preconditioner, M^{-1}. We therefore now seek a solution to

$$\tilde{P}\tilde{A}\hat{\tilde{x}} = \tilde{P}\tilde{b}, \tag{18}$$

where

$$\tilde{A} = M^{-\frac{1}{2}}AM^{-\frac{1}{2}}, \hat{\tilde{x}} = M^{\frac{1}{2}}\hat{x}, \tilde{b} := M^{-\frac{1}{2}}b, \tag{19}$$

and

$$\tilde{P} = I - \tilde{A}\tilde{Q}, \tilde{Q} = \tilde{Z}\tilde{E^{-1}}\tilde{Z^T}, \tilde{E} = \tilde{Z^T}\tilde{A}\tilde{Z}, \tag{20}$$

where $\tilde{Z} \in \mathbb{R}^{n*k}$ can be interpreted as a preconditioned deflation-subspace matrix. The resulting method is called the Deflated Preconditioned Conjugate Gradient (DPCG) method (details in [Vuik, Segal, and Meijerink, 1999]).

Algorithm 2 Deflated Preconditioned Conjugate Gradient Algorithm

1: Select x_0. Compute $r_0 := b - Ax_0$ and $\hat{r_0} = Pr_0$, Solve $My_0 = \hat{r_0}$ and set $p_0 := y_0$.
2: **for** j:=0,..., until convergence **do**
3: $\hat{w_j} := PAp_j$
4: $\alpha_j := \frac{(\hat{r_j}, y_j)}{(p_j, \hat{w_j})}$
5: $\hat{x}_{j+1} := \hat{x}_j + \alpha_j p_j$
6: $\hat{r}_{j+1} := \hat{r}_j - \alpha_j \hat{w}_j$
7: Solve $My_{j+1} = \hat{r}_{j+1}$
8: $\beta_j := \frac{(\hat{r}_{j+1}, y_{j+1})}{(\hat{r}_j, y_j)}$
9: $p_{j+1} := y_{j+1} + \beta_j p_j$
10: **end for**
11: $x_{it} := Qb + P^T x_{j+1}$

Note that $\tilde{P}$ or $M^{\frac{1}{2}}$ are never calculated explicitly. Hence the linear system is often denoted by

$$M^{-1}PA\hat{x} = M^{-1}Pb \tag{21}$$

Some Observations:

All known properties of Preconditioned Conjugate Gradient (PCG) also hold for DPCG, where PA can be interpreted as the coefficient matrix A after preconditioning. Moreover if $P = I$ is taken the algorithm above reduces to the Preconditioned Conjugate Gradient(PCG) algorithm.

Careful selection of Deflation vectors is required for this method to prove useful. Two methods, one based on eigenvector (of $M^{-1}A$) based subspace for Z and the other based on an arbitrary choice of the deflation subspace, are worth mentioning.

However to calculate the eigenvectors itself could be computationally intensive so an arbitrary choice which closely resembles part of the eigenspace is the way out. In short the ideal deflation method should satisfy the following criteria:

- The deflation-subspace matrix Z must be sparse;
- The deflation vectors approximate the eigenspace corresponding to the unfavorable eigenvalues;
- The cost of constructing deflation vectors is relatively low;
- The method has favorable parallel properties.

5 Parallel Preconditioning

In order to introduce parallelism in the completely sequential Incomplete Cholesky Preconditioner we use the Block-IC version. In this case we make blocks that grow in multiples of the grid dimension n for a grid with $n \times n$ points. This is very important for our implementation since we would like to expose (and utilize) parallelism in every step of the Algorithm.

5.1 Block-Incomplete Cholesky Preconditioning

Shown in Figure 3 is an 8×8 grid and the resulting matrix which has 64 rows. In our implementation the blocks have to be at least twice as big as grid dimension n or else a major part of the outer diagonals with offsets $\pm n$ is discarded and that leads to delayed convergence (and sometimes stagnation) of the iterative method.

5.2 Incomplete Poisson Preconditioning

Although Block Incomplete Cholesky Preconditioning is very effective in achieving convergence for the Conjugate Gradient Method. It is highly sequential within the block. Since in this study we implement Preconditioned Conjugate Gradient on a Data Parallel Architecture we also consider a recently suggested method of preconditioning called the Incomplete Poisson Preconditioning [Ament, Knittel, Weiskopf, and Straβer, 2010].

There is a price to pay for this 'parallelism' in terms of convergence speed. However, our experiments show that it is still at least as fast or comparable to the Block Incomplete Cholesky version (for a particular grid size) when the number of blocks is the maximum possible.

5.3 Domain Decomposition for Deflation

In Sub-domain deflation, the deflation vectors are chosen in an algebraic way. The computational domain is divided into several sub-domains, where each sub-domain corresponds to one or more deflation vectors. In our experiments we use Stripe-Wise domains as depicted in Figure 4.

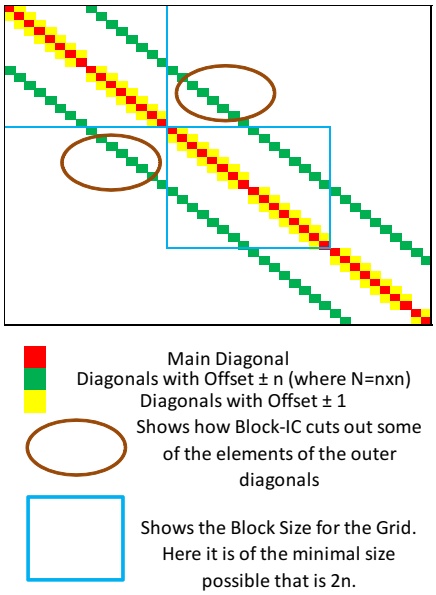

Figure 3: Matrix for an 8×8 Grid. 64 Unknowns. N=64, n=8 and BlockSize=$2n$

57	58	59	60	61	62	63	64
49	50	51	52	53	54	55	56
41	42	43	44	45	46	47	48
33	34	35	36	37	38	39	40
25	26	27	28	29	30	31	32
17	18	19	20	21	22	23	24
9	10	11	12	13	14	15	16
1	2	3	4	5	6	7	8

Figure 4: Stripe-Wise Deflation Domains. 4 domains in an 8×8 grid.

6 Implementation on the GPU

We have implemented our Iterative method on an NVIDIA GPU. We now give a primer of the various constructs provided by the language extensions on the GPU that are needed to write applications for the GPU. GPUs are based on the idea of Single Instruction Multiple Data (SIMD). This means that they can take a sequence of instructions and run them on a data set, dividing it amongst multiple processors and computing the result in parallel. Each unit of execution is called a thread. The sequence of instructions executed by every thread on a processor (there are many of these processors on the GPU) is called a kernel.

In order to utilize the GPU one has to identify these basic units of computation (i.e. kernels) inside the application and then launch the kernels on the 100s of processors simultaneously in parallel. The execution configuration before the kernel launch takes care of informing the GPU how many threads to launch and how to organize them in a logical block and further those blocks are organized in a logical grid.

GPU has a huge memory bandwidth and one of the keys to extracting performance out of the GPU is to utilize the above 100 Gb/s memory bandwidth available on the GPU. GPU also has different levels of memory like the CPU. However unlike the CPU the application developer has to manage these memories explicitly. Among these the most important is the Shared Memory which is as good as a small cache in which one can keep the heavily accessed data to minimize traffic to and fro between the chip and the global memory. Memory transfer between the CPU and the GPU should be kept to a minimum to maximize benefits from executing application code on the GPU.

6.1 Description of Kernel Design

Our complete iteration runs on the GPU with minimal transfers between the CPU and the GPU. This is one of the most important reasons for the performance boost we get from our implementation. Starting with the plain Conjugate Gradient Method and adding step-by-step the modules for preconditioning and Deflation we developed the code side by side on the GPU and the CPU. For this the following points were kept in mind.

1. Identifying Kernels of Computation.
2. Organizing code in form of kernels.
3. Prioritized Optimization of Kernels after analyzing the profiler results (% Time taken, Bandwidth utilised, Occupancy).

On the CPU we have used the Meschach BLAS Library for Dot Products and Saxpys. The kernels that were hand-coded are

1. Sparse Matrix Vector Multiply Kernel
2. Preconditioning Kernel(s)
3. Deflation Kernels

After testing with two levels of preconditioning on the CG method it was noticed that with increasing size of preconditioning blocks and increasing number of deflation vectors the number of iterations fall. On the GPU the *CUBLAS* library provided some useful functions for saxpy and dot products which we have used. For other operations custom kernels were written.

6.2 Sparse Matrix Vector Product (SpMV) - Kernel

Our matrix has a regular pattern that of a 5 point Laplacean Matrix in two dimensions. So there are 5 diagonals which contain the complete matrix. The storage format that we choose is called the Diagonal Storage format. All the diagonals are stored in a 1-D array, starting from the lowest sub-diagonal(with offset $-n$) followed by sub-diagonal with offset (-1), then the main diagonal and then the two super-diagonals. Also an important feature is that they all have the same length. This kind of uniformity of size makes coalesced access possible. So for example if say the sub-diagonal with offset -1 has one element less, then at that position a zero fills in to make it equal in size to the main diagonal.

Once stored in this way for each row of the matrix we have 5 fetches from the array holding the 5 diagonals and 5 from the vector. On the GPU we assign one thread to compute one element of the resulting Matrix-Vector Product. Additional optimizations include using shared memory and texture memory. The $offsets$ array is accessed by every thread and hence we store it on the shared memory to optimize the SpMV Kernel.

6.3 Preconditioning Kernel

The preconditioning kernel is the most sequential part of the entire algorithm. We initially begin with the Block Incomplete Cholesky Preconditioning. The Block Variant of Incomplete Cholesky Preconditioning basically exposes the parallelism at the block level. However each block has considerable amount of serial work to be done. One technique that we have employed is to break down the steps of preconditioning into three.

1. Forward Substitution
2. Diagonal Scaling.
3. Back Substitution

The diagonal scaling step can be heavily optimized using shared memory. This is possible since it is inherently parallel with two reads every thread and one multiplication all the calculations(N) are independent. For the first and the final step we can also use shared memory. The trick is to load the elements using a number of threads (number same as the block size) in parallel and then work on them and store them back in global memory. Later in the development process we used Incomplete Poisson(IP) Preconditioning to maximize benefits of parallelism. It has been discussed earlier in Section 5.2.

6.4 Deflation Kernels

For deflation we sub-divide the tasks into a couple of kernels at the outset. Namely,

1. Calculate $b = Z^T\mathrm{x}$
2. Calculate Matrix-Vector Product of E^{-1} with b.
3. Calculate Matrix-Vector Product of AZ with the result of the previous step and subtract from x.

For the first kernel $b = Z^T\mathrm{x}$ we have used the parallel sum approach as suggested in [SenGupta, Harris, Zhang, and Owens, 2007]. We only use the first part of the two part approach discussed in the paper. Details can be found out in the GPU Gems article [Harris, Sengupta, and Owens, 2007] for further optimizations to avoid divergence and warp-serialization.For the other two kernels it is useful to tailor the matrix multiplication example and use shared memory instead. This is better than the cublasSegmv (for some grid sizes) since we do not have an additional vector scaling and addition as required by cublasSgemv.

The decision to calculate E^{-1} explicitly is instrumental since it greatly reduces the time for the iterations. Though the setup time for the algorithm is affected but the overall gain in the running time of the method more than compensate the costly operation. If the number of deflation vectors become very high then, since E^{-1} is sparse, this approach might not be very efficient.

For the calculation of AZ times $E^{-1} \times b$ we used the *cublasSgemv* call. The final calculation $x_{it} = Qb + P^T x$ can also utilize the kernels discussed here and also the cublasSgemv. In the later stages of development we optimize the storage of AZ and re-write the kernels for calculations involving AZ.

7 Results and Discussion

Initially we perform our experiments using 5 point Laplacean matrix resulting from a 2D square grid. We talk about those experiments in the first subsection that follows and after that we report the results from a matrix that results from a Grid having two phases and an interface layer. For the two different matrices in question we performed several experiments on three grid sizes for three different preconditioning block sizes and three different choices of Deflation Vectors. All the experiments were done in single precision on the GPU as well as on the CPU. We use a Q9650 Intel Quad Core CPU however we only utilize a single core. We optimize it to use SSE instructions, unrolling loops and vectorizing using compiler switches. We also use the Meschach Blas Library for the Blas routines on the CPU. The GPU we use is a Tesla C1060 from NVIDIA. We use CUDA for writing our code on the GPU. We use the *CUBLAS* and *MAGMA* libraries when using Blas functions in the GPU version.

7.1 Numerical Experiments

We summarize our important findings in a speedup graph. It shows the speedups that we have got with different versions of the code. These results are for a grid size of 512×512. In the versions where we use Block Incomplete Cholesky preconditioning we use a block size of 1024. In the Deflated Preconditioned versions we use 4096 deflation vectors. The stopping criteria is $\frac{\|b - Ax_k\|_2}{\|b\|_2} \leq 10^{-5}$. Note that in this version we have the maximum degree of parallelism for our experiments. We have the largest grid size so more rows in parallel for Sparse Matrix Vector Multiplication. We have the largest number of Preconditioning Blocks and we take the highest number of deflation vectors ($8 \times n$). We now elaborate on the versions used:

1. (CGVV) Conjugate Gradient - Vanilla Version - The only kernel in this version of the solution is the Sparse matrix vector kernel. It takes the majority of the time in the execution time profile. However the kernel utilizes around 85 Gb/s of the Memory bandwidth on the GPU.

2. (CGBIC) Conjugate Gradient - Block Incomplete Cholesky Preconditioning - When we add preconditioning to the Conjugate Gradient version in the previous step the convergence is faster, however the speedup suffers due to the inherent serial nature of the Block Incomplete Cholesky Preconditioning within a block. We tried using shared memory for co-operative loading and writing of elements however that approach

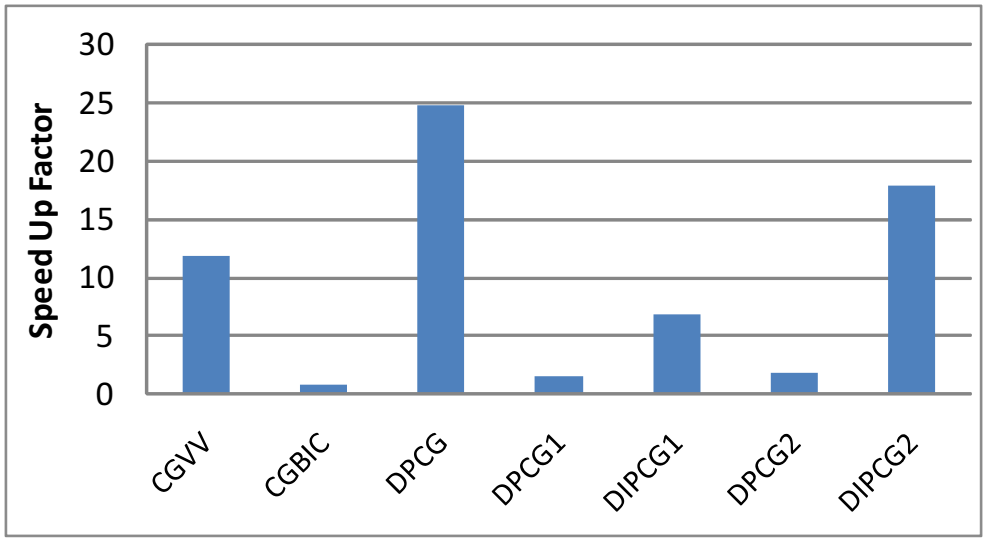

Figure 5: SpeedUp Graph across different Code Versions. Grid Size (512×512)

	Execution Times		No. of Iterations	
Code Version	CPU	GPU	CPU	GPU
CGVV	5.92	0.5004	652	649
CGBIC	5.0723	5.594	327	327
DPCG	110.45	4.45	42	42
DPCG1	1.5944	1.0102	41	41
DIPCG1	1.7285	0.2494	49	49
DPCG2	1.5938	0.8866	41	41
DIPCG2	1.7528	0.0975	49	49

Table 1: Comparison GPU vs. CPU. Number of iterations required for convergence and execution times.

suffers at larger block sizes (for e.g. 4096 elements mean 16384 bytes of shared memory).

3. (DPCG) Conjugate Gradient - Deflation and Block Incomplete Cholesky Preconditioning - Adding deflation to the Preconditioned Conjugate Gradient introduces considerable scope for parallelism. Also in order to leverage the computing power available of the GPU we use the explicit inverse of the matrix E and do dense matrix vector multiplication which can be done in parallel on the GPU. One important point that we found in our results is that the calculation involving the matrix AZ was taking most of the time so our focus became to optimize AZ storage and calculation. Table 1 shows that deflation decimates the number of iterations required for convergence.

4. (DPCG1) Conjugate Gradient - Deflation(Optimized - Level 1) and Block Incomplete

Cholesky Preconditioning - AZ is inherently a sparse matrix. Since A is sparse and Z has piece-wise constant deflation vectors. So we store AZ in a data structure $5 * N$ wide since it is also symmetric just like A and has 5 diagonals. However some of the diagonals are $\frac{d}{n}$ wide where, d is the number of deflation vectors and n is the dimension of the square grid ($N = n \times n$). In this version we also optimize the CPU version by use of some compiler flags to use SSE instructions, unrolling of loops etc. Result being that the CPU version gets very fast (up to 20 times) whereas the GPU version becomes 2 times as fast. The result is that speedup is decimated with respect to DPCG. The profiler in this version points to the Preconditioning as the most time consuming task.

5. (DIPCG1) Conjugate Gradient - Deflation(Optimized - Level 1) and Incomplete Poisson Preconditioning - We use a novel preconditioning method recently published. The parallel properties of this method are very well suited to the GPU. In effect it is as parallel as the Jacobi preconditioner, albeit much better mathematically and is as parallel as sparse matrix vector multiplication we used for doing the operation Ax. Its convergence rate is as good as the convergence of block-IC Preconditioner when the number of the blocks is maximum ($\frac{n}{2}$, where$N = n \times n$) as shown in Table 1. This version gives us a speedup that is almost 4 times that of the Block-IC version in the previous step. The profiler results show that now most of the time is take up by the computation step $E^{-1}b$ where E^{-1} is a dense $d \times d$ matrix and b is a $d \times 1$ vector.

6. (DPCG2) Conjugate Gradient - Deflation(Optimized - Level 2) and Block Incomplete Cholesky Preconditioning - We optimize the Matrix vector product $E^{-1}b$ by using the $MAGMA$ Blas library developed for CUDA. In some cases $MAGMA$ Blas delivers 3 times as much memory throughput for matrix vector multiplication compared to $CUBLAS$. Result being that we get almost double the speedup as we had for DPCG1 version of the code.

7. (DIPCG2) Conjugate Gradient - Deflation(Optimized - Level 2) and Incomplete Poisson Preconditioning - In this version we replace the Block-IC Preconditioning used in the previous step with Incomplete Poisson Preconditioning and we get much better speedup for this particular grid size (speedup grows across all grid sizes, deflation vectors and preconditioning blocks).

For all these versions we get the Relative error norm of the solution, $\frac{\|X_{exact} - X_k\|_2}{\|X_{exact}\|_2}$ at convergence (the k-th iteration) in the range of 10^{-3}.

We repeat experiments 1, 2, 6 and 7 for the Two-Phase Matrix as well. We have to set the stopping criterion at 10^{-2}. The Relative Error Norm of the Solution is also heavily affected and it stays at 10^{-1}. This is because of the very high condition number of the matrix A since the density contrast between the two mediums is $1000 : 1$. Also the deflation matrix P has an even worse condition number, so we see that as the number of deflation vectors increase the method misses convergence. The speedups remain the

same since all that we change is the matrix A and that does not change the number of computations involved. Please note that in case of missed convergence our method runs till 1000 iterations and the speedup is defined as

$$\text{SpeedUp} = \frac{\text{Time taken on the Host(CPU) to do 1000 iterations}}{\text{Time taken on the Device(GPU) to do 1000 iterations}} \tag{22}$$

Another interesting feature that we noticed in the results for Two-Phase Matrices was that of False convergence. This was noticeable in versions 1 and 2. This means that the relative norm of the residual reaches below the required tolerance

$$\frac{\parallel r_k \parallel}{\parallel r_0 \parallel} < \epsilon,\ \epsilon = 10^{-2} \tag{23}$$

but it rises and falls above and below this level (if we continue the iterations after that and record the residual). At one point the norm falls below machine precision and that does not make any sense. This behavior was consistent in Conjugate Gradient Method and Conjugate Gradient with Preconditioning for a two phase matrix.

7.2 Discussion

In this section we look at the different aspects of our implementation. We try to find out how much parallelism we exploit and how much bandwidth we are able to utilize on the GPU. We end this section with a discussion on what might be possibly limiting the achievable speedup and how far we are from that point. Throughout this section we analyze the results with a grid size of 512×512 and 4096 deflation vectors and Incomplete Poisson Preconditioning.

7.2.1 Static Analysis

In this section we calculate how many Floating Point Operations (FLOPs) each kernel does in each run and how many memory accesses happen both during loads and stores. We list this both for all the Kernels. The following notations are used.

- N, Number of Unknowns
- d, Number of Deflation Vectors
- m, Number of Iterations

From Table 2 one can find the number of FLOPs being performed in one complete run of the methods we have implemented.

We now elaborate some of the Kernel names:

$Z^T x$, $E^{-1}b$ and $AZ \times E^{-1}b$ form the steps of the deflation operation. Forward Substitution, Diagonal Scaling and Back Substitution form the steps of Block Incomplete Cholesky Preconditioning. Sdot is the Dot product function as named in BLAS libraries. We use

cublasSdot. Saxpy is the Saxpy Kernel as available in BLAS libraries. We use *cublasSaxpy* and also write custom kernels to club saxpy with scaling operations to minimize memory transfers. Sscal is the BLAS scaling operation and Snrm is the 2-Norm operation available in the BLAS libraries.

Let us take the case of the method DIPCG2 discussed in Section 7.1. It is the Deflated Preconditioned (Incomplete Poisson) Conjugate Gradient method that uses optimized AZ storage and the *gemv* routine from $MAGMA$ Blas library. It also has some optimizations that combine certain operations like scaling and saxpy for calculation of β as given in the step 9 of Algorithm 2.

The kernels involved in this variant then are listed in Table 3.

Summing up the FLOPs for m iterations we have

$$9N(m+1)+N(m+2)+d^2(m+3)+9N(m+1)+9N(m+1)+8Nm+6Nm+Nm+2Nm. \quad (24)$$

or

$$45Nm + d^2m + 29N + 3d^2 \quad (25)$$

So the computational intensity is governed by the first two factors of the expression in (25). Now let us take a specific case of $N = 262144$, $d = 4096$ and $m = 49$. These correspond to the experiment DIPCG2 discussed in 7.1 with grid size as 512×512 and the Number of Deflation Vectors = 4096. It takes the 49 iterations to converge both on the host and the device. The time on the device is 0.0987 seconds and on the host is 2.237 seconds. The speedup is 22.7 times.

Now the GPU theoretically(peak throughput) can deliver 933 GFlops/s. The CPU on the other hand, when talking about one core (which we use in our experiments), can deliver a peak throughput of 12 GFlops/s. The numbers for NVIDIA are available from the website which talks about the Tesla C1060 specifications [NVIDIA, 2010] . For Intel Processors also the numbers are provided on the website [Intel, 2010].

The computational load as calculated in (25) comes out to be 1.46 GFlops. Dividing this by the time taken we get 0.65GFlops/s for the CPU and 14.79 GFlops/s for the GPU.

These numbers can be further divided by the peak throughput to understand the Platform Utilization on the GPU as 1.585% and on the CPU as 5.41%.

7.2.2 Kernels- Performance

We refer to some of the works that outline how to effectively characterize a kernels' performance and its ability to scale across new generations of hardware that will have more processors to facilitate parallel execution. [Nickolls, Buck, Garland, and Skadron, 2008] and [Ryoo, Rodrigues, Baghsorkhi, Stone, Kirk, and Hwu, 2008] and [Komatitsch, Michéa, and Erlebacher, 2009] bring about certain methods by which we can find

1. How to find if a kernel is compute bound or bandwidth bound?
2. Expected Speedup from an application.

Kernel	Data Read In	Computations Done	Writes Performed	Degree of Parallelism in use	Number of Calls
Sparse-Matrix Vector Product and IP Preconditioning	$6N$	$9N$	N	N	$m+1$
$Z^T x$	N	N	d	d	$m+2$
$E^{-1}b$ (gemv)	$d(d+1)$	$d \times d$	d	d	$m+3$
$AZ \times E^{-1}b$	$5N+d$	$9N$	N	N	$m+1$
Forward and Back Substitution	$4N$	$3N$	N	$\frac{\sqrt{N}}{2}$	$m+1$
Diagonal Scaling	$2N$	N	N	N	$m+1$
$(AZ)^T x$	$6N$	$5N$	d	d	1
Sdot	N	$2N$	N	$-$	$4m$
Saxpy	$2N$	$2N$	N	$-$	$3m$
Sscal	N	N	N	$-$	m
Snrm	N	$2N$	N	$-$	m

Table 2: Kernels - Computation and Parallelism

SpMV	$Z^T x$	$E^{-1}b$(gemv)	$AZ \times E^{-1}b$	Sdot
Saxpy	IP Preconditioning	Snrm	$(AZ)^T x$	

Table 3: Kernels in the DIPCG2 version of the code in Section 7.1

Kernel Characteristics				
Method	Caching (Shared Memory)	Divergence	Shared-Memory Bank Conflicts	Warp Serialization
Magma_Sgemv	Yes	No	No	No
IPPreconditioning	Minimal	Yes	Yes	No
SpMV	Minimal	Yes	Yes	No
$AZE^{-1}b$	Minimal	Yes	Yes	No
Z^Tx	Yes	Yes	Yes	Yes
saxpy_alpha	Yes	No	No	No
saxpy_beta	No	No	No	No

Table 4: Grid of 512×512 points. Number of Deflation Vectors = 4096. With optimizations applied to AZ storage and calculation, $E^{-1}b$ with Magma_Sgemv and other optimizations.

3. Examination of PTX(CUDA assembly) code for finding percentage of code that is memory or compute intensive.

Also these documents detail important things to keep in mind when designing a kernel or optimizing it. These documents put to use, in their respective contexts, the Best Practices guide provided by NVIDIA [NVIDIA, 2009].

The most important factor in a kernels' effectiveness is its ability to do memory accesses in the best possible way. To this end a couple of important techniques are instrumental. This step comes obviously after the point of minimizing memory transfers as much as possible between the CPU and GPU.

1. coalesced memory access
2. caching
3. minimize divergence among threads within the same block

In Table 4 we list which techniques are used by the (except *CUBLAS*) kernels in our implementations. Memory coalescing has been used in all the kernels. We also list if there are shared memory conflicts.

7.2.3 Bandwidth Utilization

Let us take a look at the bandwidth utilization of the kernels in the most optimized version DIPCG2 (Section 7.1) of the code that we have. This is the Deflated Incomplete Poisson Preconditioned Conjugate Gradient Method with optimizations for AZ storage and calculation and also with the *gemv* operation from the *MAGMA* library.

In this version we consider the Grid Size 512×512 with 4096 deflation vectors. In Table 5 we list the Memory Throughput of Individual Kernels and the percentage of time they take of the total execution on the device. It also lists the occupancy of the kernels.

Kernel Statistics					
Method	%GPUTime	Read Throughput	Write Throughput	Overall Throughput	Occupancy
$MagmaSgemv$	44	85.2	0.02	85.4	50%
$IPPreconditioning$	9.6	66.39	5.75	72.15	100%
$SpMV$	9.6	66.44	5.76	72.2	100%
$AZE^{-1}b$	9.4	51.93	6.08	58.02	100%
Z^Tx	6.6	8.175	1.02	9.19	50%
$saxpy\_alpha$	3.2	34.85	34.85	69.7	100%
$saxpy\_beta$	2.5	42.64	21.3	63.67	100%
$cublas\_Sdot$	8.6	53.88	0.197	54.08	100%
$cublas\_Saxpy$	2.9	42.34	21.17	63.52	100%
$cublas\_2-Norm$	1.81	37.69	0.223	37.92	100%

Table 5: Grid of 512×512 points. Number of Deflation Vectors =4096. CG with Deflation and Incomplete Poisson Preconditioning. With optimizations applied to AZ storage and calculation, $E^{-1}b$ with MagmaSgemv and other optimizations.

The $CUBLAS$ Kernels are prefixed with $Cublas$ and other kernels have been hand-coded with exception of the $Magma\_Sgemv$ which is from the $MAGMA$ blas library. In Table 5 we show kernels that form more than 98% of the total execution time. The last 2% or so is taken up by transfers from Device to Host and a few calls to kernels used for correcting x at the end of the iteration by doing $x = Qb + P^Tx$ as the last step of Algorithm 2.

The Tesla system on which we have run all of our tests offers a memory bandwidth of 101Gb/s. As can be seen the $Gemv$ is utilizing a majority of the available bandwidth (85Gb/s). Followed closely by the IP Preconditioning and SpMV Kernels at 72 Gb/s. These three kernels form 60% of the total execution time. Except for the $CUBLAS$ call for calculating the 2-Norm of the updated residual (stopping criterion - required to be checked every iteration) and the call to calculate Z^Tx all the kernels utilise more than half of the available bandwidth. The average Memory throughput of this execution is 68 Gb/s.

7.2.4 Discussion on Possible Speedup Limits

Given that two of the kernels seem to be operating at 50% occupany we try to find out if they can deliver more performance and hence, a possibility of a higher speedup.

The current kernel for Z^Tx is trying to utilize both shared memory and parallel reduction in order to achieve its current bandwidth utilization. We have kept as many threads in the block as are the elements whose sum is required to make one element of the new vector y resulting from $y = Z^Tx$. Since in this kernel N/d elements have to be summed in

chunks to produce d elements where

$$N = \text{ Number of Unknowns}, \qquad d = \text{ Number of Deflation Vectors.} \tag{26}$$

$$y = d \times 1 \text{ vector}, \qquad x = n \times 1 \text{ vector.} \tag{27}$$

The occupancy varies according to the ratio of N/d but the bandwidth never crosses that indicated in Table 5. The kernel's occupancy varies with the factor N/d. For N/d above and equal to 128 (we have values like 16, 32, 64, 128, 256 and so on.) the occupancy is 100%. For the case under consideration the occupancy is 50% but for a lower number of deflation vectors (for e.g. 2048) it is 100% (since N/d becomes 128). Even then the bandwidth does not change. This means that the kernel cannot perform better than this. Trying to comment out the summing operations shows that the $Z^T x$ kernel can deliver a maximum of 28Gb/s and only takes 2% of the total execution time. The speedup varies by only 5%.

This kernel has a large amount of Shared Memory Bank conflicts. They can be overcome by changing the storage structure of the vector x however this is not useful since this would require changing many other kernels (which are already performing at 100% occupancy and are bandwidth limited) and also because this kernel is not the most time consuming kernel in the whole operation.

Other than this kernel ($y = Z^T x$) the other place where there is a possibility of improvement is the $Magma\_Sgemv$ kernel. Although it is utilizing most of the memory bandwidth it is still having an occupancy of 50%. A closer look at the occupancy for this kernel shows that it has an execution configuration of

$$\text{Grid Size } 64 \times 1 \times 1 \tag{28}$$

$$\text{Block Size } 64 \times 1 \times 1. \tag{29}$$

We used the code for double precision *gemv* posted on the the *MAGMA* forums which we change to single precision and verify that it is exactly similar.

By modifying the number of blocks in the code form 64 to 128 we get an occupancy of 100%. However the bandwidth stays around 85Gb/s. This shows that the kernel is bandwidth-bound. Since at maximum occupancy we see no change in the bandwidth.

All the other kernels are at 100% occupancy and are bandwidth bound since they have simple arithmetic operations and do not show changes in bandwidth with further increasing data sizes.

More elaborate analysis of Kernels and the cost of Inter-Warp Parallelism based on Memory Accesses and Computational overlap is possible. In [Hong and Kim, 2009] a detailed model for such analysis is discussed. However they do not address the issues with Shared Memory Bank Conflicts.

8 Conclusions

In this paper we investigate efficient implementations of the preconditioned Conjugate Gradient method on a GPU for very large, sparse systems of linear equations. We con-

sider linear systems which originate from a finite difference discretization of Poisson-like problems on a structured grid. As a typical example we consider the pressure equation which is used in simulations of multi-phase flows. Due to discontinuities in the density, the resulting matrix is ill-conditioned which leads to slow convergence.

To have an efficient implemention of the preconditioned Conjugate Gradient method we distinguish the following building blocks: vector update, inner product, sparse matrix vector product and the application of a preconditioner. For the first three operations efficient implementations on the GPU are available. The main bottleneck is a fast GPU implementation of the preconditioner. For this operator (except a diagonal preconditioner) the results are scarce in the literature. Our aim is to use a preconditioner, which is competitive in convergence with the best known serial preconditioners based on the incomplete Cholesky decomposition. In the implementation of the various preconditioners the following guidelines are used: exploit synchronization-free parallelism, optimize thread mapping, align global memory access, and reuse data as much as possible.

We start with a block-IC preconditioner, where the blocks are solved in parallel. To obtain good convergence for many blocks and for large ratios of the densities we add a second level preconditioner (deflation). It appears that the convergence is reasonable but the speedup is small due to the low level of parallelism. After that we use a recently developed Incomplete Poisson preconditioner, which has the same good parallelization properties as the matrix vector product. Combination of this preconditioner with deflation, which is also fast on the GPU, leads to a very efficient method.

We illustrate our work with some numerical experiments. From these experiments it appears that for large problems the GPU implementation of the Incomplete Poisson preconditioner combined with deflation is 20 times faster than ICCG on one node of a CPU. Finally, we observe that single precision arithmetic for multi-phase flow with large jumps in the density does not lead to reliable results. This is a point of future research.

References

M. Ament, G. Knittel, D. Weiskopf, and W. Straβer. A parallel preconditioned conjugate gradient solver for the poisson problem on a multi-GPU platform. http://www.vis.uni-stuttgart.de/ amentmo/docs/ament-pcgip-PDP-2010.pdf, 2010.

A. Asgasri and J. E. Tate. Implementing the Chebyshev Polynomial Preconditioner for the iterative solution of linear systems on massively parallel graphics processors. http://www.ele.utoronto.ca/ zeb/publications/, 2009.

M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov. Accelerating scientific computations with mixed precision algorithms. *CoRR*, abs/0808.2794, 2008. URL `http://dblp.uni-trier.de/db/journals/corr/corr0808.html`. informal publication.

N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. Technical Report NVR-2008-04, NVIDIA Corporation, December 2008.

L. Buatois, G. Caumon, and B. Levy. Concurrent number cruncher: a GPU implementation of a general sparse linear solver. *Int. J. Parallel Emerg. Distrib. Syst.*, 24(3):205–223, 2009.

S. Georgescu and H. Okuda. Gpgpu-enhanced conjugate gradient solver for finite element matrices. *Proceedings of The Second international Workshop on Automatic Performance Tuning*, 2007.

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, 2010. http://ta.twi.tudelft.nl/nw/users/vuik/numanal/gupta_afst.pdf.

M. Harris, S. Sengupta, and J. D. Owens. *Parallel Prefix Sum (Scan) with CUDA*, 2007. `http://developer.nvidia.com/GPUGems3/gpugems3_ch39.html`.

M. Harris, S. Sengupta, J. D. Owens, S. Tseng, Y. Zhang, and A. Davidson. Cudpp. `http://gpgpu.org/developer/cudpp`, 2009.

S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. *SIGARCH Comput. Archit. News*, 37(3):152–163, 2009. ISSN 0163-5964. doi: http://doi.acm.org/10.1145/1555815.1555775.

Intel. Processor specifications - by family. Website, 2010. `http://www.intel.com/support/processors/sb/cs-023143.htm`.

D. Komatitsch, D. Michéa, and G. Erlebacher. Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA. *J. Parallel Distrib. Comput.*, 69(5):451–460, 2009. ISSN 0743-7315. doi: http://dx.doi.org/10.1016/j.jpdc.2009.01.006.

M. M. Baskaran and R. Bordawekar. Optimizing sparse matrix-vector multiplication on GPUs. Technical report, IBM Research Division, NY, USA, December 2008. `http://gpgpu.org/2009/04/13/optimizing-sparse-matrix-vector-multiplication-on-gpus`.

J.A. Meijerink and H. A. Van der Vorst. An interactive solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Math. Comp.*, 31:148–162, 1977.

A. Monakov and A. Avetisyan. Implementing blocked sparse matrix-vector multiplication on NVIDIA GPUs. In *SAMOS '09: Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 289–297, Berlin, 2009. Springer-Verlag.

J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008. ISSN 1542-7730.

NVIDIA. *NVIDIA CUDA C Programming Best Practices Guide CUDA Toolkit v*2.3. NVIDIA Corporattion, Santa Clara, 2009.

NVIDIA. Tesla processor specifications. Website, 2010. `http://www.nvidia.com/object/product_tesla_c1060_us.html`.

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*, 47:339–361, 2005.

S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu. Optimization principles and application performance evaluation of a multithreaded *gpu* using CUDA. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-795-7.

Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics; 2 edition, Philadelphia, 2003.

S. SenGupta, M. Harris, Y. Zhang, and J.D. Owens. Scan primitives for GPU computing. *Graphics Hardware*, 2007.

J. M. Tang and C. Vuik. Acceleration of preconditioned krylov solvers for bubbly flow problems. *Lecture Notes in Computer Science, Parallel Processing and Applied Mathematics*, 4967(1):1323–1332, 2008.

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.

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

M. Wang, H. Klie, M. Parashar, and H. Sudan. Solving sparse linear systems on NVIDIA Tesla GPUs. In *ICCS '09: Proceedings of the 9th International Conference on Computational Science*, pages 864–873, Berlin, 2009. Springer-Verlag.