The Application of Projected Conjugate Gradient Solvers on Graphical Processing Units

Youzuo Lin[*]
Mail Stop D443
Los Alamos National Laboratory
Los Alamos, NM 87545
ylin@lanl.gov

Rosemary Renaut[†]
School of Mathematical and Statistical Sciences
Arizona State University
Tempe, AZ 85287-1804
renaut@asu.edu

ABSTRACT
Graphical processing units introduce the capability for large scale computation at the desktop. Presented numerical results verify that efficiencies and accuracies of basic linear algebra subroutines of all levels when implemented in CUDA and Jacket are comparable. But experimental results demonstrate that the basic linear algebra subroutines of level three offer the greatest potential for improving efficiency of basic numerical algorithms. We consider the solution of the multiple right hand side set of linear equations using Krylov subspace-based solvers. Thus, for the multiple right hand side case, it is more efficient to make use of a block implementation of the conjugate gradient algorithm, rather than to solve each system independently. Jacket is used for the implementation. Furthermore, including projection from one system to another improves efficiency. A relevant example, for which simulated results are provided, is the reconstruction of a three dimensional medical image volume acquired from a positron emission tomography scanner. Efficiency of the reconstruction is improved by using projection across nearby slices.

Keywords
High Performance Computing, GPU, Krylov Subspace Methods, Lanczos-Galerkin Projection, Conjugate Gradient

1. INTRODUCTION
In recent years there has been increasing interest in many disciplines on the use of many-core based high performance computing (HPC) architectures. The NVIDIA® graphical processing unit (GPU) is one such commonly available chip. Unlike traditional graphical cards, the current generation of GPUs is much better suited to scientific computation. Initially, GPUs were notorious for requiring users to adopt tedious programing techniques utilizing the graphic application programming interface (API) to access the processor cores. There was also weak support for floating point accuracy as needed for scientific applications. However, the scenario changed completely with the introduction of the NVIDIA® GeForce 8800 GTX (or G80) in 2006. This chip comes with the full support for the Institute of Electrical and Electronics Engineers (IEEE) single precision floating point standard. In the latest NVIDIA GPU series, the Tesla C1060, the IEEE double precision standard is also supported. Moreover, the introduction of CUDA (Computed Unified Device Architecture) [4] by NVIDIA in 2007 enables programmers to bypass the utilization of the graphics interface, which itself simplifies the programming significantly. At this point several GPU accelerated libraries are available to augment CUDA, such as CUBLAS and CUFFT [3, 5] which are the CUDA implementations of the Basic Linear Algebra Subprograms (BLAS) and the Fast Fourier transform (FFT), respectively. There are also several software development kits (SDK) from third party vendors specifically designed for Matlab scripting and built upon CUDA. Jacket[6] is of this type. Therefore the development cycle utilizing GPUs has been much shortened.

The enormous capability of a GPU for large computations is due to the many core hardware design and high bandwidth memory structure. Taking the Tesla C1060 as an example, as illustrated by Figure 1, there are 30 streaming multiprocessors (SM), each of which has 8 streaming processors (SP). The clock rate of each individual single precision is 1.24 GHz. The C1060 chip supports 1024 threads per SM, which can reach 30720 threads in the whole chip. In contrast, the CPU can only support 2 or 4 threads per core. With respect to the memory, in C1060 there is a 4 GB DDR3 DRAM, the bandwidth of which can be as fast as 102 GB/s, while the system DDR2 DRAM bandwidth can only reach 6 GB/s. To be fair, the communication bandwidth between the GPU and host computer through the PCI express bus (PCI express V2.0 x16) is 8 GB/s. Although the latency of the GPU DRAM is admittedly longer than the system DRAM, the large number of working threads can usually hide this latency [11]. Therefore, the Tesla C1060 possesses a tremendous computational capability. For single precision computation, its peak performance is 933 GFLOPS, and for double precision it is 78 GFLOPS. Obviously there is still a huge gap between single precision and double precision performance for the current GPU. Because of such differences,

[*]Corresponding author
[†]This research was supported by the NSF grant DMS 0652833, 0966270 and 0937737.

there may be some limitations for certain applications which are highly dependent on the accuracy.

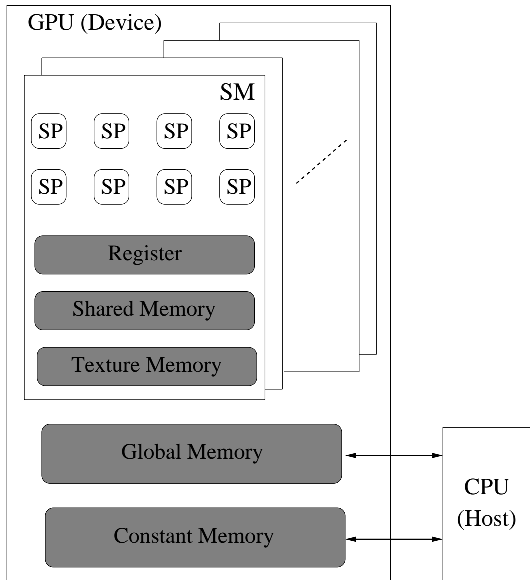


Figure 1: The structure of NVIDIA C1060 [4].

In order to fully take advantage of the computational capability provided by the GPU, it is reasonable to expect that a high portion of the computation should be accomplished in the GPU, while a small portion of the computation may be completed by the CPU. A significant portion of the application should be parallizable such that it can be implemented on the GPU. The remaining sequential part of the algorithm should run on the CPU. So far, GPUs have been widely applied to various applications, such as in medical imaging [11, 16] and for improving the efficiency of many numerical methods [7].

We will be focusing on the implementation and optimization of the conjugate gradient (CG) [9] algorithm on the GPU. Our main contributions are two fold, first, we reveal the fact that the current GPU has much better performance for BLAS 3 than BLAS 1 or BLAS 2, which can be taken advantage of to reorganize some existing algorithms so as to maximize use of BLAS 3 operations. Second we show by examples that the projection based CG method [18] can be utilized as a linear solver for a particular type of problem on a GPU based HPC environment. Some early results of this work have been presented in [12]. In this paper, we give the specific derivations of our work as well as more numerical results. The way that we organize the paper is as follows: Section 2 focuses on the main numerical background that is used in this paper. Section 3 discusses the parallelism model and the performance of BLAS on the GPU environment; Section 4 is concentrated on the specific details of the implementation and optimization of the CG solver utilizing Lanczos-Galerkin projection; and Section 5 gives the numerical results of our proposed algorithm for a 1D synthetic problem from [8] and an example for the reconstruction of slices of a 3D medical image.

2. NUMERICAL BACKGROUND AND RELATED WORK

2.1 The Multiple Right Hand Side Problem

We consider the solution of the multiple right hand systems of equations

$$AX = B = [\mathbf{b}^{(1)}, \dots \mathbf{b}^{(s)}], \tag{1}$$

where $\mathbf{b}^{(i)}$ are the columns of matrix B and solutions $\mathbf{x}^{(i)}$ are the columns of the matrix X. The challenge of solving (1) arises not only due to the massive computation required but also due to the desire for real time computation. For example, we takes as out test example the case of 3D image reconstruction from projection data, as is typical for data from positron emission tomographic scans. Solution of this problem using a traditional high performance computing (HPC) set up based on a cluster of workstations is not practical for a clinical setting. It is thus of interest to develop viable algorithms which take advantage of GPU computing. Here we consider the use of a projection based conjugate gradient solver for the solution of (1) which is optimized for the GPU. But first, we need to consider the conditioning of the problem which is determined by the system matrix A.

In many applications, particularly for many examples in image reconstruction or restoration, the formulation is ill-posed and the resulting matrix A is ill-conditioned. Thus in solving (1) some regularization needs to be imposed [10]. Because our focus is on the GPU implementation we consider here the basic Tikhonov regularization, which for the solution of a single system is given by

$$\min_{\mathbf{x}} \left\{ \|A\mathbf{x} - \mathbf{b}\|_2^2 + \lambda^2 \|\mathbf{x}\|_2^2 \right\}. \tag{2}$$

We note that this depends on a parameter λ, which is the regularization parameter. There is a vast literature on how to actually find suitable λ [13, 14], but here we assume that λ is provided. This is a reasonable assumption in the clinical environment, for which scan protocols are carefully determined for particular medical conditions and requirements. While (2) can be solved using direct methods, practical implementations require iterative formulations.

2.2 Conjugate Gradient Methods

For the single equation suppose that $\mathbf{r}^{(0)}$ is the initial residual $\mathbf{b} - A\mathbf{x}^{(0)}$ for some initial guess $\mathbf{x}^{(0)}$. Then the associated Krylov subspace generated when using a conjugate gradient algorithm, as given in **Algorithm** 1, is given by

$$\mathcal{K}_K(A, \mathbf{r^{(0)}}) = \mathrm{span}\{\mathbf{r^{(0)}}, A\mathbf{r^{(0)}}, A^2\mathbf{r^{(0)}}, \dots A^{K-1}\mathbf{r^{(0)}}\}. \tag{3}$$

The solution is in the space spanned by the direction vectors

$$\begin{aligned} \mathbf{x} &= \sum_{i=1}^{K} \alpha^{(i)} \mathbf{p}^{(i)}, \qquad (4) \\ \mathbf{x} &\in \mathrm{span}\{\mathbf{p}^{(1)}, \mathbf{p}^{(2)}, \dots, \mathbf{p}^{(K)}\}, \\ &= \mathrm{span}\{\mathbf{r}^{(0)}, A\mathbf{r}^{(0)}, \dots, A^{K-1}\mathbf{r}^{(0)}\}. \end{aligned}$$

While this algorithm can be applied immediately for the solution of (1), we first investigate the efficiency of the basic linear algebra operations (BLAS) on the GPU.

Algorithm 1 Canonical CG to solve $A\mathbf{x} = \mathbf{b}$, [9]

Input: A, $\mathbf{b}$, TOL, $\mathbf{x}^{(0)}$
Output: $\mathbf{x}^{(k)}$
1: Initialize $k = 0$, $\mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)}$, RelRes $= 1$
2: **while** RelRes $>$ TOL **do**
3: $k = k + 1$
4: **if** $k = 1$ **then**
5: $\mathbf{p}^{(1)} = \mathbf{r}^{(0)}$
6: **else** $\{k > 1\}$
7: $\beta^{(k)} = \dfrac{<\mathbf{r}^{(k-1)}, \mathbf{r}^{(k-1)}>}{<\mathbf{r}^{(k-2)}, \mathbf{r}^{(k-2)}>}$
8: $\mathbf{p}^{(k)} = \mathbf{r}^{(k-1)} + \beta^{(k)}\mathbf{p}^{(k-1)}$
9: **end if**
10: $\alpha^{(k)} = \dfrac{<\mathbf{r}^{(k-1)}, \mathbf{r}^{(k-1)}>}{<\mathbf{p}^{(k)}, A\mathbf{p}^{(k)}>}$
11: $\mathbf{x}^{(k)} = \mathbf{x}^{(k-1)} + \alpha^{(k)}\mathbf{p}^{(k)}$
12: $\mathbf{r}^{(k)} = \mathbf{r}^{(k-1)} - \alpha^{(k)}A\mathbf{p}^{(k)}$
13: RelRes $= \|\mathbf{r}^{(k)}\|_2 / \|\mathbf{b}\|_2$
14: **end while**

3. PERFORMANCE OF BLAS ON GPU

We consider the BLAS 1-3 implemented on the GPU, implementing the matrix-vector operation in both CUDA and Jacket. These results are contrasted with Matlab running on the CPU. All our tests are running on a desktop with Intel Xeon® quad-core processor with 4M cache and 2.13 GHz clock rate. The GPU used is the NVIDIA® Tesla C1060. In the experiments three computational environments are contrasted. These are the standard single core CPU, the multiple four-core CPU and the GPU environment, denoted throughout by Device, Host (SingleThread) and Host (MultiThread), respectively. All the benchmarks are from [1], for matrix sizes varied from 817×817 to 15439×15439 as shown in Table 1; the top row is the benchmark matrix, and the second is its size. Evaluated is the cost for a matrix-vector multiply which is a standard BLAS 2 operation.

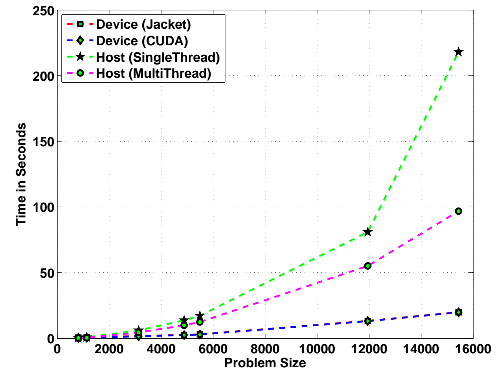
Figure 2 illustrates the comparison results in terms of the computational cost, speedup and relative error. To determine the relative error we take the double precision result as the more accurate solution and consider the single precision as the approximate solution to the double precision solution. Mathematically, the relative error is computed as

$$\text{RelErr} = \left| \frac{\text{result}_{\text{SP}} - \text{result}_{\text{DP}}}{\text{result}_{\text{DP}}} \right|, \quad (5)$$

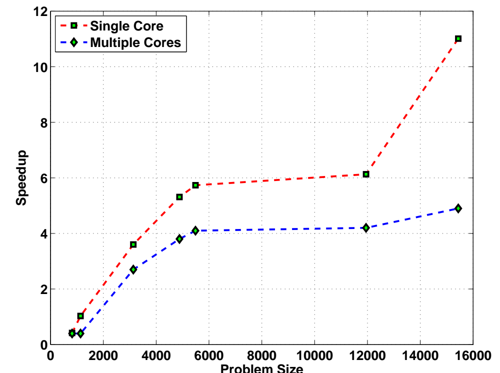
where $\text{result}_{\text{SP}}$ is the result in single precision and $\text{result}_{\text{DP}}$ is the result in double precision. Recalling that Device and Host aliases for the GPU and CPU respectively, the speedup is measured with respect to the CPU cost,

$$s(n) = \frac{\text{Execution time using Host (CPU)}}{\text{Execution time using Device (GPU)}}. \quad (6)$$

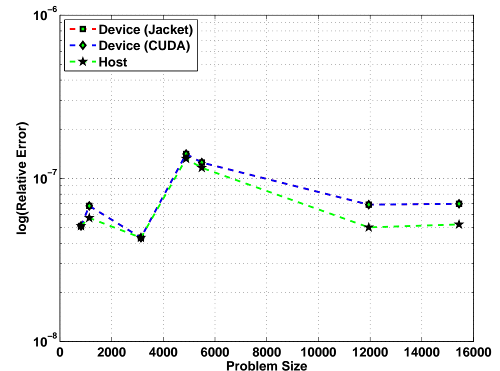
From Figures 2(a)-2(b), the speedup of BLAS 2 is evident, and improves with increasing problem size. Figure 2(c) indicates that the relative error is quite small, around 1.0×10^{-7} in all our benchmarks, which is on the order of single precision accuracy. Jacket and CUDA are comparable for this application in terms of computational cost and accuracy. Thus, given the relatively lower programming complexity for Jacket, it is selected for the further tests.



(a) Computational Cost



(b) Speedup



(c) Relative Error

Figure 2: Matrix vector multiplication results. Figures 2(a) to 2(c) compare the computational cost, the speedup (6) and the relative error, respectively. Note in Figure 2(a) that the cost on the Device is independent of implementation via Jacket or CUDA and the two graphs with symbol □ and ◇ are overlaid. The same is seen in Figure 2(c) for the relative error using CUDA or Jacket. Recall for these tests that the multiple core tests refer to the quadcore CPU.

Matrix name	bcsstm19	1138_bus	bcsstm23	bcsstk16	s1rmt3m1	bcsstk18	bcsstk18
Matrix size	817	1138	3134	4884	5489	11948	15439

Table 1: Test matrices selected from Matrix Market [1].

The performance of the GPU with respect to the choice of kernel is also of interest. In Figure 3 we report the floating point operations per unit time in seconds, or "GFLOP/s" for the BLAS 1 (SAXPY, inner product), BLAS 2 (matrix vector multiplication) and BLAS 3 (matrix matrix multiplication) kernels. In order to have a fair comparison among different kernels, the test problem is formulated so that all kernels have have the same number of flops. In particular, for BLAS 1, we choose the vectors of size $n^3 \times 1$, for BLAS 2, we choose the matrix of size $n^2 \times n$ and vector to be of size $n \times 1$, and for BLAS 3, we choose the matrices of size $n \times n$. In this way, all the kernels will result in the same FLOP count of $2n^3$. In Figure 3, the x-axis corresponds to the total number of Mega Flops, the y-axis corresponds to the Giga Flops per unit time measured in seconds. It is interesting to see that BLAS 3 kernels definitely outperform both BLAS 1 or BLAS 2 kernels. This is preserved as n increases, but the relative improvement levels off. This suggests that in optimizing the performance of an algorithm for use on the GPU is should be formulated to take advantage of BLAS 3 kernels.

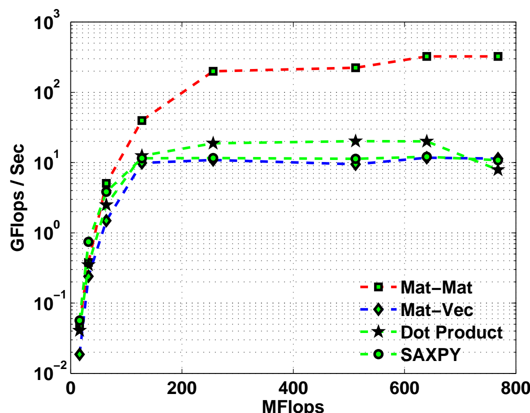


Figure 3: Computational Kernels Comparison. Results in GFLOPS/Sec for SAXPY, dot product, matrix vector multiplication (Mat-Vec) and matrix matrix multiplication (Mat-Mat).

4. A PROJECTION BASED KRYLOV SUBSPACE ALGORITHM ON GPU

4.1 Projection Based CG Implementation and Performance

While it is important to optimize for the GPU using BLAS 3, it is also relevant to consider the specific application in mind. Here we seek an algorithm which also takes advantage of the multiple right hand side feature in equation (1). Even though in [8] the projection idea is applied successfully, the potential loss of accuracy that results when using the GPU in single precision in order to optimize efficiency, must be considered. In particular, while the convergence of the Lanczos-Galerkin projection based solver has been extensively discussed in the literature [8, 18], the impact of single precision on convergence may be significant.

For clarity of exposition, the Projected Conjugate Gradients (PrCG) algorithm for a regular square linear system is presented in **Algorithm** 2. Here the subscript q is the index for the current system, while the superscript indicates the iteration step. Steps 5 to 11 of **Algorithm** 2 yield the solution $\mathbf{x}_q^{(K)}$ which is the solution of the qth system using projection onto the Krylov subspace for the 1st system, $q = 1$. If $\mathbf{x}_q^{(K)}$ does not satisfy the test for convergence, the CG algorithm is restarted using $\mathbf{x}_q^{(K)}$ as initial guess for the Krylov subspace associated with right hand side $\mathbf{b}_q$. The solution is thus obtained on an augmented Krylov subspace of length $K_q > K$. Clearly, the seed space in **Algorithm** 2 plays an important role, since it is used to generate solutions for all remaining systems. Building a good seed space $\mathcal{K}_K(A, \mathbf{r}^{(0)})$ is crucial for subsequent updates, but is beyond the scope of this work. It is discussed in [8] for the case in which the space can be updated each iteration. Here $\mathcal{K}_K(A, \mathbf{r}^{(0)})$ is generated based on the seed and not updated thereafter.

Algorithm 2 Projected CG (PrCG), [8]

Input: TOL, $i = 0$, RelRes $= 1$, K, A, $\mathbf{b}_q$, $\mathbf{x}_q^{(0)}$, $A\mathbf{p}_1^{(i)}$ and $\mathbf{p}_1^{(i)}$ $1 \leqslant i \leqslant k$
Output: $\mathbf{x}_q^{(K)}$
1: **if** seed system **then**
2: Canonical CG(); % and output vectors $A\mathbf{p}_1^{(i)}$ and $\mathbf{p}_1^{(i)}$ $1 \leqslant i \leqslant K$
3: **else**
4: % Project onto seed space
5: $\mathbf{r}_q^{(i)} = \mathbf{b}_q^{(i)} - A\mathbf{x}_q^{(i)}$
6: **for** $i = 1$ to K **do**
7: $\alpha_q^{(i)} = \dfrac{< \mathbf{p}_1^{(i)}, \mathbf{r}_q^{(i)} >}{< \mathbf{p}_1^{(i)}, A\mathbf{p}_1^{(i)} >}$
8: $\mathbf{x}_q^{(i)} = \mathbf{x}_q^{(i)} + \alpha_q^{(i)} \mathbf{p}_1^{(i)}$
9: $\mathbf{r}_q^{(i+1)} = \mathbf{r}_q^{(i)} - \alpha_q^{(i)} A\mathbf{p}_1^{(i)}$
10: **end for**
11: RelRes $= \|\mathbf{r}_q^{(i)}\|_2 / \|\mathbf{b}_q\|_2$
12: **end if**
13: **if** RelRes > TOL **then**
14: % Augment the seed space
15: Continue the iteration using canonical CG
16: **end if**

A test problem of size 10240 with 2 right hand sides is generated, this test problem is selected from Chan's paper [8] with some modifications. The system matrix A is a square matrix of size $n \times n$, the entries of which are defined as $A = \text{diag}(1, \ldots, n)$. The right-hand sides are $\mathbf{b}_i^{(j)}(t) = \sin(t + (i + j - 2)\Delta t)$, $i = 1, \ldots, n$ $\Delta t = 2\pi/100$

and $j = 1, 2$. The effect of using the PrCG for solving the test problem is obvious. The total cost is reduced to 1.91s (from 5.49s using **Algorithm** 1), and the computational cost for Mat-Vec is reduced to 1.01s (from 4.0s using **Algorithm** 1). Furthermore, as the cost of Mat-Vec is reduced, the costs of both Dot-Prod and SAXPY are more significant.This is due to the fact that **Algorithm** 2 mostly relies on either Dot-Prod or SAXPY operations.

4.2 Kernel Optimization: Optimized Projected CG

The operations in **Algorithm** 2 are mainly BLAS 1 operations. But we know that we should require that the projected CG is implemented using BLAS 3 operations. From $\mathbf{r}_q^{(i+1)} = \mathbf{r}_q^{(i)} - \alpha_q^{(i)} A\mathbf{p}_1^{(i)}$, where $\mathbf{r}_q^{(i)} = \mathbf{b}_q - A\mathbf{x}_q^{(i)}$, and using (4), gives

$$\mathbf{r}_q^{(i+1)} = \mathbf{b}_q - \sum_{c=1}^{i} \alpha_q^{(c)} A\mathbf{p}_1^{(c)}. \tag{7}$$

By the orthogonality conditions with respect to matrix A

$$\begin{aligned} < \mathbf{p}_1^{(j)},\ A\mathbf{p}_1^{(i)} > &= 0, \quad j \neq i, \text{ and} \\ < \mathbf{p}_1^{(j)},\ \mathbf{r}_q^{(i)} > &= 0, \quad j < i. \end{aligned}$$

In particular, $0 =< \mathbf{p}_1^{(i)},\ \mathbf{r}_q^{(i+1)} >$. Thus taking the dot product with $\mathbf{p}_1^{(i)}$ on both sides of (7), gives

$$\begin{aligned} 0 &= < \mathbf{p}_1^{(i)},\ \mathbf{b}_q > - < \mathbf{p}_1^{(i)},\ \sum_{c=1}^{i} \alpha_q^{(c)} A\mathbf{p}_1^{(c)} > \\ &= < \mathbf{p}_1^{(i)},\ \mathbf{b}_q > -\alpha_q^{(i)} < \mathbf{p}_1^{(i)},\ A\mathbf{p}_1^{(i)} >, \end{aligned}$$

and

$$\alpha_q^{(i)} = \frac{< \mathbf{p}_1^{(i)},\ \mathbf{b}_q >}{< \mathbf{p}_1^{(i)},\ A\mathbf{p}_1^{(i)} >}. \tag{8}$$

Let $P = [\mathbf{p}_1^{(0)}, \mathbf{p}_1^{(1)}, \ldots, \mathbf{p}_1^{(K)}]$ be the matrix of A- conjugate directions. By the A-conjugacy $P^T AP$ is diagonal, and $(P^T AP)_{ii} =< \mathbf{p}_1^{(i)},\ A\mathbf{p}_1^{(i)} >$. Thus $\alpha_q^{(i)}$ in (8) is given by

$$\alpha_q^{(i)} = \frac{< \mathbf{p}_1^{(i)},\ \mathbf{b}_q >}{(P^T AP)_{ii} + \epsilon}, \tag{9}$$

where ϵ is a tolerance parameter, $0 < \epsilon \ll 1$, which avoids division by zero.

Notice that matrices P and $AP = [A\mathbf{p}_1^{(0)},\ A\mathbf{p}_1^{(1)}, \ldots, A\mathbf{p}_1^{(K)}]$ can be stored as the intermediate results from solving the seed system. Thus $\alpha_q^{(i)}$ in (9) can be calculated using a matrix operation without updating $\mathbf{x}_q$

$$\boldsymbol{\alpha} = P^T \mathbf{b}_q \,./\, \text{Vecdiag}(P^T AP). \tag{10}$$

Here Vecdiag(T) is the vector of diagonal entries of T

$$\text{Vecdiag}(T) = [T(0,0), T(1,1), \ldots, T(n,n)]^T,$$

and element-wise division between two vectors is given by

$$\mathbf{v}./\mathbf{u} = [v(0)/u(0), v(1)/u(1), \ldots, v(n)/u(n)]^T.$$

Therefore solution $\mathbf{x}_q$ can be updated using a Mat-Vec update

$$\begin{aligned} \mathbf{x}_q^{(i)} &= \mathbf{x}_q^{(0)} + \sum_{c=1}^{i} \alpha_q^{(c)} \mathbf{p}_1^{(c)}, \\ &= \mathbf{x}_q^{(0)} + P\boldsymbol{\alpha}, \end{aligned} \tag{11}$$

which is the standard column version for a Mat-Vec operation. Equivalently, we rewrite (11) somewhat unconventionally using a BLAS 3 operation as,

$$\mathbf{x}_q^{(i)} = \mathbf{x}_q^{(0)} + \text{sum}(P \cdot \text{diag}(\boldsymbol{\alpha})), \tag{12}$$

where diag$(\mathbf{v})$ is the diagonal matrix with vector $\mathbf{v}$ along the diagonal and sum(T) denotes the column-sum of the columns of matrix T

$$\text{sum}(T) = \sum_{i=1}^{n} \mathbf{t}_i.$$

Similarly, we can calculate the residual as

$$\mathbf{r}_q^{(i)} = \mathbf{r}_q^{(0)} - AP\boldsymbol{\alpha}, \tag{13}$$

or,

$$\mathbf{r}_q^{(i)} = \mathbf{r}_q^{(0)} - \text{sum}(AP \cdot \text{diag}(\boldsymbol{\alpha})). \tag{14}$$

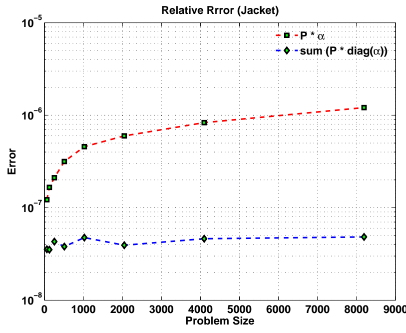
We illustrate the difference in both relative error and computational cost of (11) and (12) in Figure 4 and Figure 5. The test matrices and vectors are generated to have entries which are i.i.d. normal. Both CUDA + CUBLAS and Jacket computing environments are considered. Regardless of the computing environment, the scaled column sum formulation (11) is more efficient than (12). On the other hand, the relative error of (12) is much smaller than (11), which may be critical for applications in which the accuracy is paramount.

We now obtain the **Optimized Projected CG** (OPrCG), **Algorithm** 3, which is **Algorithm** 2 with BLAS 1 operations replaced by BLAS 2 and BLAS 3 operations.

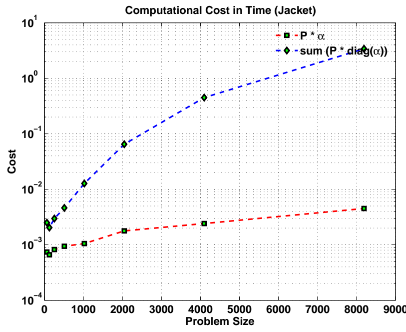
Algorithm 3 Optimized Projected CG (OPrCG)

Input: TOL, $i = 0$, RelRes $= 1$, K, A, $\mathbf{b}_q$, $\mathbf{x}_q^{(0)}$, P, and q
Output: $\mathbf{x}_q^{(K)}$
1: **if** seed system **then**
2: Canonical CG() % and output vectors $A\mathbf{p}_1^{(i)}$ and $\mathbf{p}_1^{(i)}$ $1 \leqslant i \leqslant K$
3: **else**
4: % Project onto seed space
5: $\mathbf{r}_q^{(i)} = \mathbf{b}_q - A\mathbf{x}_q^{(i)}$
6: $\boldsymbol{\alpha} =< P,\ \mathbf{b}_q > ./\text{Vecdiag}(< P,\ AP >)$
7: Update $\mathbf{x}_q$ utilizing either (11) or (12)
8: Update $\mathbf{r}_q$ utilizing either (13) or (14)
9: RelRes $= \|\mathbf{r}_q\|_2 / \|\mathbf{b}_q\|_2$
10: **end if**
11: **if** RelRes $>$ TOL **then**
12: % Augment the seed space
13: Continue the iteration using canonical CG
14: **end if**

Figure 6 shows the difference in computational cost using OPrCG and PrCG. The total computational cost using PrCG

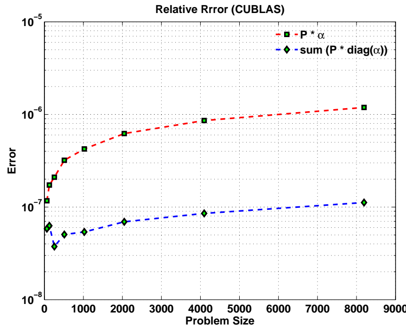


(a) Relative Error Using Jacket

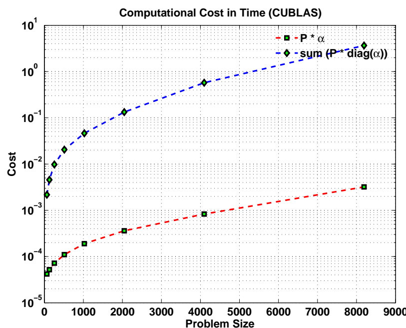


(b) Cost Using Jacket

Figure 4: Comparison of the relative error and computational cost using (11) and (12) in Jacket.



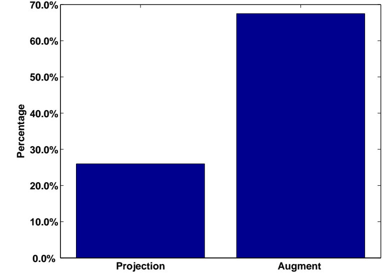
(a) Relative Error Using CUBLAS



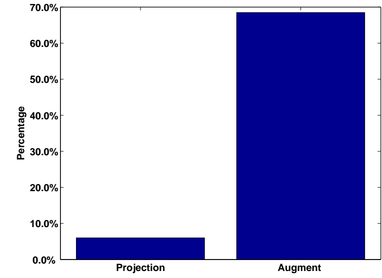
(b) Cost Using CUBLAS

Figure 5: Comparison of the relative error and computational cost using (11) and (12) in CUDA + CUBLAS.

is 1.91s, where the projection procedure takes about 27.18% as in Figure 6(a). By using OPrCG, the total computational cost is reduced to 1.50s, where the projection procedure is only about 8.02% as in Figure 6(b). A 20% improvement in computational cost is obtained in this example.



(a) Projected CG



(b) Optimized Projected CG

Figure 6: Comparison in computational cost using OPrCG in Algorithm 3 and PrCG in Algorithm 2. The only difference is due to the BLAS 3 implementation for the projection steps, which is more efficient.

5. NUMERICAL RESULTS

5.1 3D Shepp-Logan Phantom Reconstruction

We now turn our attention to the use of the GPU for the reconstruction of the slices for a 3D Shepp-Logan Phantom [2], see Figure 7. It is immediately clear that consecutive slices are similar and that a projection-based approach should be used in the reconstruction. We build a projection matrix of size 16650×16384, which corresponds to an image for one slice of size 128×128. The condition number of the system is 1.1×10^{32}. Tikhonov regularization (2) is used to regularize the reconstruction; the regularization parameter is set to be $\lambda^2 = 0.5$ throughout the tests. For this test, we chose to compute the reconstructions in batches of four slices. The performance of the algorithm therefore relies heavily on the accuracy of the projection steps. It is thus appropriate to use the more accurate update scheme and so we select (12) and (14) for updating in this example.

The loss of orthogonality can be a significant issue when the system is ill-posed. Specifically, as the problem size increases the loss of orthogonality starts to effect the performance of OPrCG. This is not surprising because the algorithm depends significantly on the orthogonality of the conjugate directions and residuals. When the problem is ill-posed, the loss of orthogonality for the conjugate directions becomes more severe [17] and reorthogonalization is needed. This

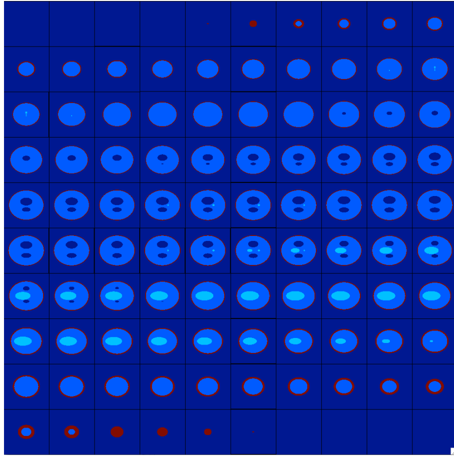


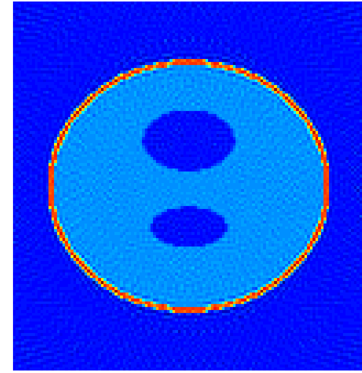
Figure 7: Slice Show of A 3D Shepp-Logan Phantom

can be accomplished by reorthogonalization applied to either the gradient vectors $\mathbf{p}^{(i)}$ or to the residual vectors $\mathbf{r}^{(i)}$. Here we reorthogonalize for the gradient vectors.

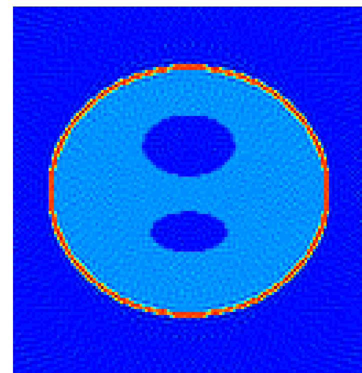
For the simulation we select slice 64 as the seed slice and use projection to obtain reconstructions of slices 65 to slice 68. The dimension of P used in both PrCG and OPrCG is 60, the dimension of the augmented space for each of the slices is reported in Table 2. Three different approaches and their results are reported in Table 2, where three reconstruction approaches are compared: CG (**Algorithm** 1), PrCG (**Algorithm** 2) and OPrCG (**Algorithm** 3). Again, the methods are contrasted through measurements of Cost, Speedup and SNR. In order to further contrast PrCG and OPrCG, we also compute the percentage Improvement Ratio given by

$$\text{ImpRate} = \left| \frac{\text{Cost}_{\text{PrCG}} - \text{Cost}_{\text{OPrCG}}}{\text{Cost}_{\text{PrCG}}} \right| \times 100\%. \tag{15}$$

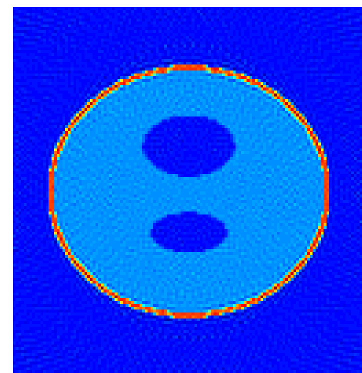
From Table 2, we see that in terms of computational cost the projection based approaches achieve a speedup from four to eight times depending on the location relative to the seed slice. The largest speedup for the two projection based methods is at slice 65, which is reasonable considering the largest similarity to the seed. As the slice location is further away from the seed slice, the speedup is reduced as illustrated in Table 2. This is consistent with the discussion in [8] that the right hand sides need to be close. To further compare between PrCG and OPrCG, we can see that OPrCG yields even further speedup than PrCG, while still preserving the SNR of the reconstruction. The quality of the reconstructed images using all three approaches is fairly comparable, the SNR is preserved independently of the algorithm. To further illustrate, we show the results for slice 66 in Figure 8, visually the results are comparable. We conclude from the results in Table 2 and the reconstruction in Figure 8, that the projection based algorithms yield good performance. Moreover OPrCG can further speed up the algorithm while maintaining good accuracy. We note that the seed system should be replaced after some number of steps, dependent on the distance of $\mathbf{b}^{(1)}$ from $\mathbf{b}^{(q)}$, $q > 1$ [8].



(a) Host



(b) Device - PrCG



(c) Device - OPrCG

Figure 8: Reconstruction Results on Host and Device Using Three Different Approaches. Slice index $= 66$**, image size** $= 128\times128$**. 8(a): Reconstruction using CG,** SNR $= 13.80$ **dB; 8(b): Reconstruction using PrCG,** SNR $= 13.83$ **dB; 8(c): Reconstruction using OPrCG,** SNR $= 13.83$ **dB.**

Solver / Slice	CG			PrCG				OPrCG				ImpRatio
	Cost	**SNR**	**Mat-Vec**	**Cost**	**SNR**	**Mat-Vec**	**Speedup**	**Cost**	**SNR**	**Mat-Vec**	**Speedup**	
65	15.79	13.67	83	2.17	13.67	36	7.27	2.06	13.67	36	7.67	5.07%
66	15.89	13.80	77	2.73	13.83	48	5.80	2.58	13.83	47	6.16	5.49%
67	13.58	13.91	76	3.10	13.95	56	4.37	2.80	13.95	52	4.84	9.68%
68	15.55	13.80	81	3.55	13.83	65	4.38	3.47	13.83	66	4.49	2.25%

Table 2: Reconstruction of Four Consecutive Slices from 65 to 68. The 64th slice is selected as seed and the the others are reconstructed utilizing CG (Algorithm 1), PrCG (Algorithm 2) and OPrCG (Algorithm 3). The results are the computational cost (Cost), SNR value (SNR, in dB), speed up (Speedup), the number of the matrix vector multiplications (Mat-Vec) and the improvement ratio (ImpRatio).

6. CONCLUSIONS AND FUTURE WORK

We discussed implementation issues for a projection based CG algorithm on the GPU. The performance of the GPU for BLAS 3 is much better than for BLAS 1 and 2. Thus the projected CG algorithm can be optimized to run on the GPU by taking advantage of BLAS 3 operations. The numerical tests verify that the optimized algorithm is feasible in terms of both accuracy and convergence. This is consistent with earlier work on the block CG (BCG) algorithm [15]. In particular, Saad showed that using the projected system is mathematically equivalent to starting the block CG method with the initial block $[\mathbf{b}^{(1)}, \mathbf{b}^{(2)}]$, [18, Theorem 4.1], which explains the relative success of the approach adopted here. It is worth noting, however, that the standard block CG approach assumes that the set of right hand sides is available simultaneously, which need not always be the case.

The speedup of the presented algorithm depends significantly on the efficiency of the Mat-Vec multiplication. It may be possible to optimize this operation by considering the special structure within the system matrix. Furthermore, in the discussion presented here, we acknowledge that we have neither addressed the additional storage requirements associated with storing the matrices P and AP, nor the communication costs associated with distributing P and AP. These considerations are topics for our future research. Trade-offs between utilizing structure, increasing storage and maintaining sufficient accuracy most likely exist. Recently, there is a new series of GPUs named “Fermi” with new architecture released on the market by NVIDIA®. It is said to have much better performance in double precision than all the previous generations of GPUs. It would be interesting to see the performance on the new card.

7. ACKNOWLEDGEMENT

This research is supported by the NSF research grants (DMS 0652833, 0966270 and 0937737) and the graduate student research grant from the Graduate Professional Student Association (GPSA), Arizona State University. We would like to thank NVIDIA® for the generous donation of a NVIDIA® Tesla C1060 GPU, Dr. Wolfgang Stefan from Rice University for discussions on the GPU and Jacket, and Dr. Nathan Bell from NVIDIA for his general advice on the paper.

8. REFERENCES

[1] Matrix market. A repository of matrix test data for use in comparative studies of algorithms. `http://math.nist.gov/MatrixMarket/`.

[2] 3D Shepp-Logan phantom, 2005. `http://www.mathworks.com/matlabcentral/fileexchange/9416-3d-shepp-logan-phantom`.

[3] NVIDIA CUBLAS Library, Version 2.3., 2009. `http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/CUBLAS_Library_2.3.pdf`.

[4] NVIDIA CUDA Computed Unified Device Architecture Programming Guide, Version 2.3.1., 2009. `http://developer.nvidia.com/object/cuda_2_3_downloads.html`.

[5] NVIDIA CUFFT Library, Version 2.3., 2009. `http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/CUFFT_Library_2.3.pdf`.

[6] Jacket - The GPU Acceleration Engine for MATLAB, Version 1.2.2., 2010. `http://www.accelereyes.com/`.

[7] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking*, 2009.

[8] T. F. Chan and W. L. Wan. *SIAM Journal on Scientific Computing*, 18(6):1698–1721, 1997.

[9] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, third edition, 1996.

[10] P. C. Hansen. *Rank-Deficient and Discrete Ill-Posed Problems*. SIAM, 1997.

[11] D. B. Kirk and W. Hwu. *Programming Massively Parallel Processors*. Morgan Kaumann, 2010.

[12] Y. Lin and R. Renaut. Projected conjugate gradient solvers on GPU and its applications. In *GPU Technology Conference*, 2010.

[13] Y. Lin, B. Wohlberg, and H. Guo. *Signal Processing*, 90(8):2546–2551, August 2010.

[14] J. Mead and R. A. Renaut. *Inverse Problems*, 25(2):025002 –025020, 2009.

[15] D. P. O’Leary. *Numerical Linear Algebra with Applications*, 29:293–322, 1980.

[16] D. Riabkov, X. Xue, D. Tubbs, and A. Cheryauka. Accelerated cone-beam backprojection using GPU-CPU hardware. In *Proceedings of the 9th IntâĂŹl Meeting Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine*, 2007.

[17] F.-X. Roux. Acceleration of the outer conjugate gradient by reorthogonalization for a domain decomposition method for structural analysis problems. In *International Conference on Supercomputing*, 1989.

[18] Y. Saad. *Mathematics of Computation*, 48(178):651–662, 1987.