New Basic Linear Algebra Methods for Simulation on GPUs

Jian Shi\(^1\), Yiwei Zhang\(^1\), Blake Langland\(^2\), Jijun Tang\(^1\), Roger Dougal\(^2\)

1. Dept. of Computer Science and Engineering, University of South Carolina
2. Dept. of Electrical Engineering, University of South Carolina
Columbia SC, USA
Contact: dougal@cec.sc.edu

Keywords: Simulation, linear equation, GMRES, GPU

Abstract
We have used Graphics Processing Units (GPUs) to accelerate the solution of the types of equations typically encountered in dynamic system simulators. Compared to commercial matrix solvers that run on a CPU, we realized speedups ranging from 5 (for system size \(\approx 700\)) to 460 (for system size \(\approx 5800\)). While calculation time for the commercial matrix solver increased with matrix size \(\approx O(N)^2\cdot 3\), our new GPU-based Preconditioned Generalized Minimal Residual (PGMRES) technique yielded scaling as \(\approx O(N)^{1.2}\). A significant component of this performance was achieved by development of new Basic Linear Algebra routines for the NVIDIA Tesla GPU that directly address characteristics typical of matrices that describe the time domain response of naturally-coupled dynamic systems.

1. INTRODUCTION
Solving linear systems is the most frequently encountered problem in scientific and engineering calculations since nonlinear, time varying equations can be often linearized into their linear representations. For a large class of simulators for dynamic systems, the simulation time steps are normally set to small values to achieve high accuracy and stability, which requires multiple matrix calculation within quite limited time. Therefore, developing high-performance linear equation solvers is a key factor to the simulation performance. This problem normally contains two parts: algorithm improvement and parallel acceleration. Using Virtual Test Bed (VTB), a simulator for multidisciplinary dynamic systems, as the test case, we proposed and implemented a matrix solver utilizing Preconditioned Generalized Minimal RESidual (PGMRES) [23] on GPUs under a low-overhead hybrid parallelism model.

Recently, the trend has focused more on many core and on-chip parallelism than single core performance. A corollary of Moore’s Law suggests that it is the number of cores that doubles every 18 months [1]. Graphic Processing Units (GPUs) is taking the lead for its rather low cost and power consumption, comparing with many-core CPU cluster with the same amount of cores, as well as its quite impressive computing potential. However, for historical reasons, GPUs are designed for shader operations that involve applying a series of shader kernels to stream of data [24]. This structure, while making hardware implementation easier, makes GPU parallelism more difficult and adds more restrictions.

Existing methods to solve linear equations on GPUs share one common problem: they only gain an appreciable speed up when the input problems have relatively large size due to the parallelization overhead when utilizing GPUs. However, the size of the coefficient matrix\(^1\) in different simulation systems can vary in a wide range. In practice, very large systems can be segmented into a multiplicity of smaller systems by exploiting certain methods such as Latency Insertion Method (LIM) [9]. Taking our test bed as an example, the estimated sizes of system matrix range from 200 to 10,000 and a big amount of the systems are smaller than 2,000, which is significantly smaller than the size threshold of speed gain in existing methods. On the other hand, a typical scientific and engineering simulation needs to solve equations repeatedly. It means some expensive pre-processes for the input matrix, which plays a crucial role in some high-performance system solvers, is not an option. Based on the special needs of our test simulation system, we re-designed PGMRES, using Incomplete LU factorization as the preconditioner, on NVIDIA TESLA T10 GPUs. In addition, we developed a set of BLAS (Basic Linear Algebra Subprograms) routines to minimize the parallelization overhead. Compared to a general-purpose commercial matrix solver called Bluebit, we achieved about 3x to 5x speed up when the input problem size is less than 700 and up to 460x speed up when the input problem size is around 5,800.

2. BACKGROUND

2.1. GPU and CUDA C
The advent of multicore CPUs and manycore GPUs means that mainstream processor chips are now parallel systems. Furthermore, their parallelism continues to scale with Moore’s law. The challenge is to develop application software that transparently scales its parallelism to leverage the increasing number of processor cores, much as 3D graphics

---
\(^1\)Since the conductance matrices from our testing system are all square, we only discuss square matrix in this paper.
applications transparently scale their parallelism to manycore GPUs with widely varying numbers of cores.

Programming GPUs for general purpose computing is quite different from traditional style [17]. Earlier programming models for GPUs were based on shading languages[11][12][20] such as HLSL, GLSL [21] and Cg [16], which are graphics oriented and have to be used with OpenG L or DirectX APIs. In these models, computation was essentially organized as a sequence of shading operations (kernels) on graphics data (streams). Programmers had to make explicit calls to these graphics APIs to manage streams and launch kernel calls.

NVIDIA GPUs support four general-purpose parallel computing architectures. They are CUDA C, OpenCL, DirectCompute and CUDA FORTRAN. We choose CUDA C as our design platform since it is a convenient C-based programming interface extended only by a minimal set of additional constructs and in consistency with our testbed’s development environment.

2.2. Related Work

The great computing power of GPUs has attracted considerable interest in using them for general purpose computation. In the area of matrix computation, Galoppo et al. [7] first implement the LU factorization [5] to solve dense linear systems using GPUs, they reduced the problem of solving linear equation to a sets of rasterization problems on GPUs. Their implementation on a NVIDIA GeForce 7800 GPU beats a CPU-based ATLAS package: for \( N = 4,000 \) matrices it achieved up to 10 Gflops without pivoting and about 6 Gflops with pivoting. Vasily and James [25] studied the detailed benchmark of the GPU memory system and applied a few algorithmic optimizations aimed at increasing parallelism and regularity. Their implementation includes LU, QR and Cholesky factorizations of dense matrices on a single or double GPUs. For the case of sparse matrices, which are typical of power network problems, Garland [8] gave an introduction to sparse matrix algorithms on NVIDIA GPUs. These previous works showed impressive results and demonstrated great potential of using GPUs as coprocessors.

In various scientific and engineering calculations, such as accelerator physics, chemical process simulations, device and circuit simulations, fusion energy and structural analysis, etc. Sparse matrix is very frequently encountered. Conjugate gradient is an iterative method that can be applied to a sparse matrix which is symmetric and positive definite (SPD). Bolz et al. [3] implemented a sparse matrix solver on GPUs using conjugate gradients.

The Generalized Minimum Residual (GMRES) method is another iterative method for the numeric solution of linear systems. Zhang Y et al. implemented a parallel GMRES algorithm on grid systems [27]. Similar research was done by Dias, R et al., He, H et al. and Mingliang, W et al. [6][10][17] in favor of attempting to parallelize GMRES on various platforms. As compared to the method we have developed, those methods share a common problem: the speed up is trivial or even negative until the size of the input matrix is on the order of \( 10^5 \) or larger.

3. DIAGNOSING SYSTEM MODELS

A system modeled in VTB consists of a set of basic components that are connected at nodes to define the whole system. During the simulation, the physical circuit are characterized into the mathematic conductance matrix, each node corresponds to one element inside the conductance matrix. Therefore, these conductance matrices possess some common properties:

1) Sparsity: If a node is connected to other nodes, its corresponding value inside the matrix will be non-zero. Otherwise, the value is zero. Hence, it is not hard to imagine that most of the conductance matrices are sparse since the connections between components are sparse with respect to all the possible links between them. For some large systems, fewer than 1% of their elements are non-zero.

2) Size: The size of the conductance matrix is decided by the number of nodes, and the number of nodes of a system model is fully dependent on the level of detail of the system. To obtain better solution time of those very large systems, segmentation methods, such as LIM, can be applied to transfer solving one very large system into solving multiple smaller ones. Therefore, our focus is on systems of size \( \approx 10,000 \).

These matrices can be considered “small” as most methods published to date focus on large systems \((N > 100,000)\). However solving these matrices poses very different requirements than most existing methods:

- first, to achieve better accuracy and stability, it is frequently necessary to repeatedly solve the matrix many times within quite limited real time. In some case, the matrix needs to be solved about 1,000,000 times to simulate one second in reality;
- second, in each time step, it may require several (or more) iterations to achieve convergence if there are nonlinear components in the simulated system;
- third, existing (parallel) methods generally do not achieve any speed-up on small matrices. Most methods even have reverse speed-up reported. Table 1 briefly presents the speedup data from very recent research on GPUs [17], which shows three-time slow down on matrix with \( N < 2,000 \).
### 4. SOLVING LINEAR EQUATIONS

#### 4.1. Algorithm Evaluation

During the simulation, the conductance matrices of pure linear systems remain unchanged, only the right hand sides are updating. In this case, the time cost of LU factorization is irrelevant to solving the equations (Assume they are in the form \(Ax = b\)): calculating for the inverted matrix \(A^{-1}\) only need to be done once, then for the rest, only multiply different \(b\) for different \(x\). Bluebit uses a highly optimized LU factorization algorithm and LU factorization is the best solution for linear, time-invariant systems.

However, pure linear systems are very rare, especially among big systems. In reality, complicated systems are most likely to contain non-linear and time-varying components. That means the advantage of solving equations using LU factorization mentioned in the previous paragraph is most likely lost. Moreover, as we mentioned in the last section that most of the simulation systems are sparse, the inverted matrix is not likely to be sparse but becomes a dense matrix instead, which means:

1. most of the \(O(N^3)\) arithmetic operations devoted to invert the matrix involve zero operands;

2. it is wasteful to reserve storage for unfruitful zero elements;

Considering the fact that all the conductance matrices are very sparse, it might not be the best option to use LU factorization to solve the equations since the inverted matrix might not be sparse and result in much more calculations and storage spaces. We look further into alternative methods, such as CG and GMRES, that can maximally exploit the sparse property of the input matrices.

The conjugate gradient (CG) method is one of the most successful iterative methods for solving large, sparse systems which requires the input system be symmetric positive definite (SPD). Unfortunately, the matrices from our test cases do not generally meet this requirement. However, with a mathematical transformation we can still try solving them by CG. Since \(A^TA\) is SPD for any matrix \(A\), we can transform the problem of solving \(Ax = b\) into solving \(A^TAx = A^Tb\). This approach is called CGNR (Conjugate Gradient on the Normal equations to minimize the Residual) [18].

CNR can handle non-symmetric systems and carry over all the theory for CG. However, the convergence speed of CG/CNR is mostly determined by the condition number\(^2\) of the input system. For the majority of VTB simulator, the condition numbers are very high. For instance, an electronic ship system has a condition number on the order of \(10^{34}\). Applying CGNR to this system will increase the condition number to the magnitude of \(10^{68}\), which makes convergence with CGNR at least challenging if not impossible.

#### 4.2. GMRES

The Generalized Minimal RESidual method (GMRES) was proposed by Youcef Saad and Martin H. Schultz in 1986 [23] as another iterative method for solving nonsymmetric sparse systems. Compared with CGNR, GMRES does not require the extra \(A^T\) transformation, which is a significant advantage here.

Assuming the linear equations can be written as \(Ax = b\), GMRES minimizes, at the \(k\)th iteration, the measure of error over the affine space:

\[
x_0 + K_k
\]

where \(x_0\) is the initial iterate and \(K_k\) is the \(k\)th Krylov subspace

\[
K_k = \text{span}(r_0, Ar_0, ..., A^{k-1}r_0) \text{ for } k \geq 1.
\]

Due to the space limit, we will not discuss mathematic details in GMRES, such as testing the loss of orthogonality and its reorthogonalization, multi-givens transformation and the upper Heisenberg matrix. Those are mathematic routines and can be found in [4]. Let us assume \(k_{max}\) be the maximum iteration number, the pseudo algorithm of GMRES can be described as:

1. \(r = b - Ax, v_1 = r / \|r\|_2, \rho = \|r\|_2, \beta = \rho, k = 0; g = \rho(1, 0, ..., 0)^T \in \mathbb{R}^{k_{max}+1}\)

2. While \(\rho > \varepsilon \|b\|_2\) and \(k < k_{max}\) do
   
   (a) \(k = k + 1\)
   
   (b) \(v_{k+1} = Av_{k+1}\)

   for \(j = 1, ..., k\)

\(^2\)the condition number of \(A\) relative to the norm \(\|\cdot\|\) is \(\|A\| \|A^{-1}\|\) [26]

<table>
<thead>
<tr>
<th>System size</th>
<th>Serial time (second)</th>
<th>Parallel time (second)</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,946</td>
<td>0.0018</td>
<td>0.0060</td>
<td>0.3</td>
</tr>
<tr>
<td>132,000</td>
<td>4.82</td>
<td>0.70</td>
<td>6.8</td>
</tr>
<tr>
<td>1,122,000</td>
<td>436.33</td>
<td>22.10</td>
<td>19.7</td>
</tr>
</tbody>
</table>

Table 1. A very recent result of parallel GMRES implementation
Based on this pseudo code, we can easily calculate the costs: The $k$th GMRES iteration requires a matrix-vector product, $k$ scalar products, and the solution of the Heisenberg least squares problem. The $k$ scalar products require $O(kN)$ floating-point operations and the cost of a solution of the Heisenberg least squares problem, by QR factorization is $O(k^3)$ floating-point operations. Hence the total cost of the $m$ GMRES iterations is $m$ matrix-vector products and $O(m^4 + m^2 N)$ floating-point operations.

From the cost estimation we can infer that the key is to reduce the number of iterations and hence improve the overall performance of GMRES, a technique called preconditioning can be adopted in this situation: Assuming $M$ can satisfy
\[
\| I - MA \|_2 = \rho < 1
\]
Then we replace the system $Ax = b$ with
\[
MAX = Mb
\]
Preconditioning done in this way is called left preconditioning, right preconditioning can be processed in similar way.

If $MA$ has a smaller condition number than $A$, we might expect the relative residual of the preconditioned system to be a better indicator of the relative error than the relative residual of the original system. There are many research about preconditioning methods, some effective preconditioners are based on deep insight into the specific structure of the problem. For instance, in the context of partial differential equations, fast Poisson solvers can be fit in very well as the preconditioner. Common preconditioning techniques, including incomplete factorization preconditioner, polynomial preconditioner, Jacobi preconditioner, etc, can fit well in GMRES in most of the situation.

Obviously, preconditioning is not a free lunch. The extra cost of matrix operation brought by $M$ can be quite considerable in some situation. Using the Matlab implementation of Fast Fourier Transform in [4], the unpreconditioned iteration required roughly 1.2 million floating-point operations while the preconditioned iteration required 0.87 million, the overall speed of the preconditioned code is also about 40% slower than the unpreconditioned one. However, preconditioned iterations are still reported converging more rapidly by many researchers [15][19][22].

Considering the parallelization structure, we choose Incomplete LU factorization (ILU) as our preconditioner since 1) it is one of the most often used preconditioners for nonsymmetric systems, 2) the most expensive calculation in GMRES iteration is matrix-vector calculation, which is similar to what is in ILU. This way we can save some time to design another set of parallel programming model for ILU on top of GMRES. The Preconditioned GMRES (PGMRES) is generally the same with GMRES, except adding a calculation for ILU in step (1) and adding a backward substitution for the ILU transformation in each iteration.

The number of iterations of using ILU as the preconditioner can be significantly smaller than the original GMRES, as the result shown in table 2.

<table>
<thead>
<tr>
<th>System size</th>
<th>Original GMRES</th>
<th>PGMRES</th>
</tr>
</thead>
<tbody>
<tr>
<td>247</td>
<td>289</td>
<td>4</td>
</tr>
<tr>
<td>692</td>
<td>$&gt; 800$</td>
<td>7</td>
</tr>
<tr>
<td>2601</td>
<td>81</td>
<td>3</td>
</tr>
<tr>
<td>5776</td>
<td>126</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 2. iterations to converge for GMRES and PGMRES

This result is from four different practical models of our test bed. It shows that PGMRES using ILU preconditioner converges much more rapidly, hence it is a very promising method for solving these matrices.

5. PARALLELISM MODEL

The previous work we discussed is dedicated on improving overall performance on the serial level. As the recent growth on core performance has focused on on-chip parallelism and multi-cores, new programming models should support successful models of parallelism: task-level parallelism, word-level parallelism and bit-level parallelism. As we introduced before, GPUs offer a desirable platform for high performance parallel applications.

Parallelization of solving linear systems is a very active area of research. However unlike most of the existing research, which is focused on reaching peak hardware calculation ability of a certain type of GPUs or on solving very big sized systems ($N > 100,000$) on expensive computer clusters, we need to focus on two underlying requirements of the system solver:

- solving small and sparse systems.
• solving them repeatedly in very limited real time.

Our focus on these requirements means that the direction of our research differs significantly from the existing research. Considerable effort has been made on algorithm level parallelization. Examples include Andrzej Jordan et. al’s parallel algorithm of CG [2] and Jocelyne’s parallel GMRES [14]. Their works start gaining decent and steady speed up from $N > 1,500$ in average, which still have higher speedup-gain threshold than our target system requires. Wang et. al implemented GMRES on a Tesla GPU [17], and their work also shows 0.3x speed up when $N = 1,946$, 6.8x speed until $N = 132,000$.

We noticed that the major part of calculation of GMRES is matrix-vector product and vector scalar product. So we focus on the fast parallel implementation on GPU of these set of calculations.

5.1. GPU Architecture

In this work we are concerned with programming on a NVIDIA Tesla T10 GPU card. It has 30 multiprocessors with 8 thread processors each, 240 cores in total. The multiprocessors have an SIMD (Single Instruction Multiple Data) architecture. Scalar threads are grouped into SIMD groups called warps. Instructions are synchronously broadcasted by one instruction unit per multiprocessor to scalar cores. NVIDIA Tesla T10 provides extensive programming flexibility such as being able to execute scalar threads with arbitrary memory access patterns and branch behaviors. However this flexibility may cause performance loss if it is not exploited correctly.

Programming with GPUs is difficult: one of the reasons is that GPUs have a rather complex memory structure. The Tesla T10 GPU card has four different types of memories: global memory, local memory, constant memory and texture memory. Each thread has a private local memory. All the threads have access to the same global memory. Constant and texture memory are read-only memory spaces that are available to all threads. Constant memory and texture memory are cached so a read from them costs one memory read from device memory only on cache miss, otherwise it just costs one read from the cache. Global memory and local memory are not cached, so it is very expensive to access those memory spaces. Shared memory is on-chip memory; in Tesla T10, each multiprocessor has 16KB shared memory. In comparison, global or local memory has 400 to 600 clock cycles latency while shared memory has only 1 cycle latency assuming there is no bank conflict. Obviously, utilizing high-speed memories as much as possible can lead to (may be significant) performance improvement.

The GPUs used in this work were designed for PCI Express interface. The time cost of transferring contiguous pieces of data between host and device includes two parts: initial cost and the cost of actual transferring.

$$\text{Time} = \text{initial cost} + \frac{\text{actual byte transferred}}{\text{memory bandwidth}}$$

Since the bandwidth between device memory and the device is much higher than the bandwidth between device memory and host memory, data should be mapped to device memory as much as possible and the transfer between host and GPU should be minimized. Also, batching many small transfers into a big one always performs much better than making each transfer separately because of the initial overhead associated with each transfer.

5.2. Asynchronous Parallelization

Normally we refer to function calls as synchronous calls. Assuming there are two consequent jobs $J_1$ and $J_2$, when $J_1$ is invoked, the programming control will wait until it finishes and then continue on $J_2$. Asynchronous functions, on the other hand, are designed to facilitate concurrent executions between host and device. When host threads call an asynchronous function $J_1$, the programming control returns to the host immediately after the call without waiting to see if the device has completed or not. This way $J_2$ can start at almost the same time with $J_1$ thus achieving a simple structured parallelism. Figure 1 illustrate the control flows.

![Figure 1. Synchronous call and asynchronous call](image-url)

The advantage of utilizing this type of asynchronous parallel between GPU and CPU is that it does not take communication between host and device for granted. If we carefully arrange two independent tasks, both on data and control, to execute this way, we need only incur one cost for parallelization: synchronization of these jobs. The biggest cost of the jobs will be the total time to execute them after synchronization.

Based on PGMRES, we can take advantage of this structure widely. For example: in step (1) PGMRES needs to calculate the initial residual $r = b - Ax$, this involves a matrix-vector
product and has a time complexity of $O(N^2)$. Calculating the preconditioner Incomplete LU factorization (ILU) has similar cost. We can utilize GPU to run the job calculating the initial residual, while the CPU continues calculating ILU. This way we successfully eliminate a very time-consuming part described in [17].

5.3. Low Level BLAS Functions

In section four we analyzed that the cost of GMRES contains two major parts: matrix-vector multiplication and vector scalar product. As of today, many commercial software packages have been developed based on GPU platform such as NVIDIA CUBLAS, NVIDIA CUSPARSE and CULA, etc. These packages provide convenient basic linear algebra routines. However, they do not fit very well in parallelizing our target system.

Let us take one example of calculating the dot product of two vectors on GPU: the NVIDIA CUBLAS library implements this routine as a single function. It takes two vectors as the input and returns their product as the output. This is very easy to use but it leaves considerable room for optimization, especially in reducing memory exchange:

1. two host-to-device memory copies for the two input vectors. When the vectors are not very big, the initial cost of CPU-GPU memory copy can be a big part. This can easily be optimized by combining the two vectors into one and save one initial cost.

2. the CUBLAS dot product function returns a single value, which means another hidden device-to-host memory copy. When we need to calculate many dot products, such as in a loop, but only need to use them after the loop has exited, a big amount of time on this mandatory memory transfer are wasted. We show our optimizations below.

We developed our own special BLAS functions. We will describe the implementation details of a BLAS routine, calculating dot product, as an example:

The calculation of dot product consists of two parts: adding up each pair of elements from each vector and summing the results up. The latter part is a typical reduction problem. Assuming the length of vectors is $N$ and the maximum number of thread is $m$ (Currently NVIDIA CUDA supports 512 threads per block at most), we can replace an $N$ step summation in series with a $(N/m) \times (\log_2(m/2) + 1)$ step summation in parallel. Figure 2 illustrates a simple reduction procedure when we set the maximum thread number to 8.

When $N$ cannot be evenly divided by $m$, we increase $N$ to make it $m$'s integral multiple by adding zero elements to the vectors so that the results would not be affected. After the calculation is done, we have $N/m$ results that need to be added up to get the final result, which can be processed in two ways according to the principle of minimizing the memory transaction between device and host:

- if a result is needed immediately, we copy the $N/m$ results back to CPU and sum them up. Since when $N$ is not very large, CPU can add them up much faster than GPU;
- if multiple dot products need to be calculated, we store all the results in device memory and copy them back to host memory when all the operations are done.

In our implementation, we utilize shared memory for the reduction part described above to achieve better performance. We run CUBLAS routine, our single time dot-product routine and batch dot-product routine on vectors with various sizes for 1,000 times and get the average time cost shown in table 3.

<table>
<thead>
<tr>
<th>vector size</th>
<th>CUBLAS (µs)</th>
<th>Proposed method -single (µs)</th>
<th>Proposed method -batch (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>306.74</td>
<td>36.57</td>
<td>12.47</td>
</tr>
<tr>
<td>2000</td>
<td>307.63</td>
<td>37.46</td>
<td>13.01</td>
</tr>
<tr>
<td>10000</td>
<td>309.75</td>
<td>40.56</td>
<td>16.37</td>
</tr>
<tr>
<td>20000</td>
<td>310.88</td>
<td>44.94</td>
<td>20.75</td>
</tr>
<tr>
<td>50000</td>
<td>320.34</td>
<td>64.3</td>
<td>37.24</td>
</tr>
</tbody>
</table>

Table 3. benchmarks of different dot-product routines

Our single dot-product routine runs 5 – 9 times faster, and the batch dot-product routine is about 10 – 25 times faster than CUBLAS routines in most cases. The key factor to this improvement is because CUBLAS dot-product routine is designed to return a single result, which means a mandatory memory exchange between the GPU and the host; our routines eliminate the unnecessary memory transfer thus minimize the cost of this part. We also notice from table 3 that as
the vector size increases, the ratio of memory exchange cost decreases and the performance difference between CUBLAS and the proposed methods reduces. This proves the importance of correctly utilizing the memory structure in GPU programming paradigm from a different aspect.

Combining the work we have done, we build a hybrid parallelism model for GPU based PGMRES method. We conduct a performance comparison between our own matrix solver and the Bluebit solver that is currently serving in VTB, based on real simulation models. As the table 4 shows.

<table>
<thead>
<tr>
<th>System size</th>
<th>Bluebit (millisecond)</th>
<th>GPU-GMRES (millisecond)</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 100</td>
<td>&lt; 0.1</td>
<td>≈ 1.0</td>
<td>≈ 0.1</td>
</tr>
<tr>
<td>247</td>
<td>5.88</td>
<td>1.2</td>
<td>4.9</td>
</tr>
<tr>
<td>676</td>
<td>14.7</td>
<td>1.73</td>
<td>8.5</td>
</tr>
<tr>
<td>692</td>
<td>16.34</td>
<td>2.46</td>
<td>6.6</td>
</tr>
<tr>
<td>2601</td>
<td>405.84</td>
<td>3.75</td>
<td>108.2</td>
</tr>
<tr>
<td>5776</td>
<td>3225.19</td>
<td>6.97</td>
<td>460.7</td>
</tr>
</tbody>
</table>

Table 4. a performance comparison over solving real VTB systems

Based on the result illustrated in the above table, our GPU-GMRES shows decent speed-up gain for small size systems (200 < N < 1,000), and significant performance improvement for larger ones (1,000 < N < 6,000); as the time cost of Bluebit solver increasing dramatically with N increases, our implementation shows smooth, almost linear time increase: using the Matlab curve fitting tools, the time cost with respect to the matrix size N of the proposed method is \( \approx O(N)^{1.2} \) while Bluebit yields \( \approx O(N)^{2.3} \). However, due to the inevitable parallel overhead, the high-performance CPU based Bluebit packages outperforms our work for extremely small matrices (N < 100).

6. CONCLUSION AND FUTURE WORK

In various industrial fields, scientific and engineering simulation systems help saving great amounts of money, time and energy by building virtual environments and conducting simulated experiments on them. The most significant factor in the overall performance of simulation is the speed of solving system equations generated during run time. Based on a testbed, the VTB, this work investigates thoroughly the simulation process inside the target system and the data characteristics of its conductance matrices to design more optimal algorithms for linear equation solvers. What is more, this work implements a fine-grained parallelism and the stream-programming model utilizing the enormous processing power of NVIDIA many-core GPUs. Comparing to existing methods, our hybrid parallelization programming model begins to show an increase in performance from problem sizes of roughly 1,000 and up. This is key to improving the simulation speed for the sizes of problems our target system is typically used for. Our work mainly focuses on satisfying the special requirement of our testing system’s needs. Nonetheless, our research data and experience might be helpful for other related works in similar applications.

GPU architecture is at the forefront of parallelization technology, however there are some key constraints. For instance, in GPU’s memory model, as described before, the fastest on-chip memory has very limited size (16KB shared memory on each multi-processor on NVIDIA Tesla T10), while the latency of the large global-device memory is significantly higher. And to maintain parallelism, operations on faster memory, such as texture memory or constant memory, are limited to read-only or write-only within a kernel. What is more, GPUs also lack some fundamental computing operands in 64-bit double precision arithmetic, such as 64-bit atomic adding, which is exactly the precision requirement of VTB systems. This situation makes it impossible to implement the entire GMRES algorithm on GPU but have to implement it under a hybrid parallelization instead, thus decrease the overall performance due to the inevitable, expensive CPU-GPU memory exchange. A more detailed and in-depth survey about this area can be found in [28].

The field of GPU computing is maturing. For instance, the newest FERMI structured GPUs provide faster multi-processor for double precision and the size of shared memory is increased by four times. NVIDIA also announced in the next generation of their CUDA structure, kernel-level parallelization can be achieved. We will focus on the latest GPU technologies and incorporate them into our our applications.

7. ACKNOWLEDGMENT

This work was supported through the Office of Naval Research (ONR) grant - N00014-07-1-0686.

REFERENCES


