Abstract

This paper describes our progress in developing software for performing parallel LU factorization of a large dense matrix on a GPU cluster. Three approaches, with increasing software complexity, are considered: (i) a naive “thunking” approach that links the existing parallel ScaLAPACK software library with cuBLAS through a software emulation layer; (ii) a more intrusive magmaBLAS implementation integrated into the LU solver in the High-Performance Linpack software; and (iii) a left-looking out-of-core algorithm for solving problems that are larger than the available memory on GPU devices. Comparison of the performance gains versus the current ScaLAPACK PZGETRF are provided.

Keywords: parallel dense solver, GPU

1. Introduction

General-purpose computation on Graphics Processing Units (GPGPUs), the use of many-core programmable graphics processors to accelerate computations, is a relatively new programming paradigm for high-performance computing applications. The attraction of using GPGPUs in large supercomputers are threefold: (i) Graphics Processing Units (GPUs) are a readily available commodity, even in consumer grade video cards, and relatively inexpensive; (ii) the GPU’s massively parallel architecture results in many more floating point operations per second (flops) per dollar; and (iii) GPUs are significantly more energy efficient than their CPU counterparts. Only recently, however, have GPUs been integrated into the architecture of large supercomputers. For example, the world’s second fastest supercomputer at the National Supercomputing Center in Tianjin China uses Nvidia GPUs. The forthcoming Cray XK6 Titan Supercomputer at the Oak Ridge Leadership Computing Facility (OLCF) at the Oak Ridge National Laboratory will use Nvidia Fermi GPUs and, when fully installed, is expected to achieve a peak performance in the 10 to 20 petaflops range.
As a result of these emerging hybrid CPU/GPU architectures, many of the underlying numerical algorithms employed by HPC applications must be re-engineered to fully exploit all the available processing power. This paper describes our progress in developing a parallel LU factorization solver for dense matrices that takes advantage of GPU acceleration and is compatible with the PZGETRF LU factorization method in the ScaLAPACK library. Parallel LU factorization is a key computational kernel in the fusion application AORSA2D (All Orders Spectral Algorithm) developed within the Scientific Discovery through Advanced Computing (SciDAC) Numerical Computation of Wave Plasma-Interactions in Multi-dimensional Systems project. AORSA2D models the response of a high temperature plasma to radio-frequency (RF) waves in a tokamak fusion device. The RF, or electromagnetic, waves can be used to drive the current flow, heat the plasma, and control the resulting instabilities in the plasma. AORSA2D requires the solution of a large $N \times N$ system of complex-valued linear equations. For problems of practical interest, the resolution of the mode conversion layer may require $N > 100,000$. Thus the solution of these large, dense, complex-valued linear systems presents a major computational challenge in AORSA2D simulations.

The solution of large, dense, complex-valued linear systems is not unique to fusion applications. Other applications, such as the solution of elliptic partial differential equations by the Boundary Element Method (BEM) and modeling of electromagnetic scattering off aircraft by the Boundary Integral Method (BIM), also require the solution of similar large dense matrices.

The high cost of moving data across the PCI bus between the CPU and the GPU is the primary performance bottleneck on a hybrid CPU/GPU system. For small matrices that fit entirely within the GPU memory footprint, this high cost of data movement may be mitigated by the increased flop rate available on the GPU compared to the CPU. For problems of real practical interest, however, the memory storage required for the system generally exceeds the available GPU memory. An “out-of-core” approach, in which smaller submatrices are transferred to the GPU for computation and then the resulting factors are transferred back, must be explored. Prior serial out-of-core approaches have generally been studied for problems where the matrix size exceeds the available CPU memory and data must be transferred between the CPU and the disk. Similarly, parallel out-of-core linear algebra packages have been shown to achieve high performance in solving large problems.

In this work, we will present similar out-of-core algorithms for LU factorizations between the CPU and the GPU. In Section 2, we will describe several approaches for accelerating LU factorizations with GPUs for both in-core and out-of-core methods. The performance of these GPU-accelerated algorithms are compared to existing methods in in Section 3. Finally, we conclude with suggestions of approaches to adapting existing linear algebra algorithms to these emerging hybrid CPU/GPU architectures in Section 4.

2. Approach

We will base our GPU-acceleration approaches on two widely available software libraries that solve dense linear systems, ScaLAPACK and High-Performance Linpack (HPL). We focus our efforts on these libraries because (i) ScaLAPACK is widely used by many scientific application codes, and (ii) HPL is the comparison benchmark by which the world’s top 500 computers are ranked. We note that, as distributed, HPL solves a system of linear equations, but the LU factorization is not directly recovered. However, with a small modification to keep the global pivot vector, the factorization can easily be recovered.

In the following, we discuss three approaches for accelerating the LU factorizations in ScaLAPACK and HPL, in order of increasing software development complexity. First, we use the original Scalapack software library and replace the Basic Linear Algebra Subroutines (BLAS) library calls with Fortran-callable cuBLAS. Second, we use the Matrix Algebra on GPU and Multi-core Architecture (MAGMA) library to accelerate the BLAS calls in HPL. Finally, we describe a fully out-of-core algorithm that minimizes data movement between the CPU and the GPU during the factorization.

2.1. ScaLAPACK with Fortran-callable cuBLAS

The CUDA Basic Linear Algebra Subroutines (cuBLAS) library is a GPU-accelerated version of the standard BLAS library distributed by NVIDIA. It supports linear algebra operations, such as matrix-matrix and matrix-vector

---

6Web site for Center for Simulation of Wave-Plasma Interactions (CSWPI) is http://www.scidac.gov/fusion/CSWPI.html.
7HPL is available at www.netlib.org/benchmark/hpl/.
8cuBLAS is available at http://developer.nvidia.com/cublas/.
multiplication, for matrix data already present in the GPU device memory. The application developer is generally responsible for the data movement between the CPU and GPU.

A software wrapper layer (also called “thunking”) emulates the standard Fortran BLAS library and, hidden from the application developer, allocates memory on the GPU, copies the data from the CPU to the GPU, performs the BLAS operations using cuBLAS, then copies the results back from the GPU to the CPU, and finally deallocates the GPU memory. The advantage of this approach is that no changes are required in the original application software and only minor changes are required in the linking step of compilation. The software emulation incurs a high overhead in data transfer between the CPU and the GPU, however performance improvements may be achieved for operations on large matrices where the computation time is significantly greater than the time needed for data movement. Ultimately, though, the problem size is constrained by the amount of memory available on the GPU.

2.2. HPL with MagmaBLAS

The MAGMA library includes routines for LU factorization on a GPU, but the library is designed for GPU accelerators attached to a single machine. For practical systems of interest that cannot be stored in the memory of a single GPU, MAGMA is insufficient. Similarly, the Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA) library is currently focused on a multi-threaded parallel linear solver running on a single homogeneous multicore shared-memory machine and is not available for a distributed memory environment.

However, MAGMA provides a limited-functionality BLAS library that is optimized for NVIDIA GPUs. The advantage of the MAGMA library versus Fortran-callable cuBLAS is that an application developer can selectively implement the kernels that are most appropriate for the GPU. For example, matrix-matrix multiplies of sufficient size to mitigate the cost of the data movement would be most appropriate for the GPU, whereas small matrix-vector multiplies may not demonstrate any performance improvement.

In this work, we modified the HPL to use MAGMA for all matrix-matrix multiplies. Note that in this case, no attempt was made to determine whether the size of the matrix was sufficient to mitigate the cost of the data movement.

2.3. An Out-of-Core Factorization Based on ScaLAPACK

Out-of-core factorization methods have been studied in the past in the context of inadequate CPU memory relative to the storage required for the large dense matrix. The ideas developed in these contexts can be applied to this case. In this work, we adapt the “left-looking” out-of-core algorithm for LU factorization described in [7] with a minor change that seeks to minimize the data transfer between the CPU host and the GPU device memory. Central to this algorithm is the necessity for an in-core parallel LU factorization method that operates primarily on the GPU with minimal communication between GPUs. We will describe this in-core method with minimal communication between the GPUs, and then describe the out-of-core method.

2.3.1. In-core Factorization

ScaLAPACK PZGETRF uses a “right-looking” algorithm for LU factorization; we use a similar approach for factorization of a distributed matrix residing in device memory. Since the CPU host does not have direct access to data on the GPU, data on the GPU device must be transferred to temporary buffers on the CPU to be available for remote communication by the Message Passing Interface (MPI) library. This transfer is the primary bottleneck and must be minimized to achieve any performance gain using the GPU.

Consider a block partitioning of a matrix $A$

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix},$$

where $A_{11}$ is a $k \times k$ square matrix. There are several steps for the factorization of matrix $A$. 

---

9 Some multi-GPU functionality is already available with apparent plans for further support in the future.

10 The PLASMA software is available at http://www.icl.cs.utk.edu/plasma/index.html.
1. Assuming the first $k$ columns have already been factored, apply

$$P_1 \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} = \begin{pmatrix} \tilde{A}_{11} \\ \tilde{A}_{21} \end{pmatrix} = \begin{pmatrix} L_{11} \\ L_{21} \end{pmatrix} (U_{11}) ,$$  

where $\tilde{A}_{11} = L_{11} U_{11}, \tilde{A}_{21} = L_{21} U_{11}$, and $P_1$ is the permutation matrix. In our case, $k = \text{MB}$ was chosen to be the matrix block size and the factorization of this narrow panel is performed on the CPU host using ScaLAPACK PZGETRF.

2. Apply the permutation matrix to the unmodified submatrix

$$\begin{pmatrix} \tilde{A}_{12} \\ \tilde{A}_{22} \end{pmatrix} = P_1 \begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix} .$$

This permutation operation was accomplished by copying individual rows from the GPU device to the CPU host and using PBLAS PZSWAP to exchange the rows, and then again copying from the CPU host back to the GPU device.

3. Compute $U_{12}$ by solving the triangular system

$$L_{11} U_{12} = \tilde{A}_{12} , \text{ or } U_{12} \leftarrow L_{11}^{-1} \tilde{A}_{12} .$$

This operation can either be computed by copying data for $A_{12}$ to the CPU host and using PZTRSM in PBLAS, or by broadcasting $L_{11}$ to all processors holding part of $A_{12}$ and computing using cublasZtrsm on the GPU.

4. Perform the rank-$k$ update to $\tilde{A}_{22}$ on the GPU

$$\tilde{A}_{22} \leftarrow \tilde{A}_{22} - L_{21} U_{12} .$$

The majority of the work in the factorization is in the rank-$k$ updating of $\tilde{A}_{22}$. The operation requires broadcasting $L_{21}$ across the processor columns and $U_{12}$ down the processor rows. This rank-$k$ update can then be performed using cublasZgemm on the GPU without further communication.

5. Recursively factor the remaining submatrix

$$P_2 \tilde{A}_{22} = L_{22} U_{22} .$$

6. Finally, apply the $P_2$ permutation to $L_{21}$.

Note that a “left-looking” algorithm results if $k = N - \text{MB}$ ($A_{12}$ is size $\text{MB} \times \text{MB}$) and the first wide column panel is factored in the same recursive manner. A right-looking variant gives good load balancing and higher opportunities for parallelism [11]. Other works [12, 13, 14] have shown that a left-looking variant generates less data transfer compared to the right-looking variant.

2.3.2. Out-of-core Factorization

The out-of-core factorization method is similar to that of the in-core algorithm described in Section 2.3.1. The primary variant is that we assume that the portion of the matrix $A$ belonging to a CPU processor is too large to be fully held in-core to the GPU. Thus, some data movement of the matrix between the CPU and the GPU will be necessary, but must be minimized to achieve good performance.

The out-of-core computation proceeds using two column panels. A wide panel $Y$ (in GPU device memory) accumulates the updates from previously computed factors and a narrow panel $X$ holds the previously computed factors (see Figure 1). The computation steps are essentially similar to the in-core factorization and are described below.

1. Similar to the in-core factorization, application of the permutation matrix $P$ is performed by the CPU host using PZSWAP in PBLAS. The permuted submatrix ($\tilde{A}_{12}$ and $\tilde{A}_{22}$) are copied into panel $Y$ on the GPU device. Parts of the previously computed factors in $L_{11}$ and $L_{21}$ are moved from the CPU into panel $X$ on the GPU.

2. The triangular solve will exceed the memory limitations of the GPU and thus requires copying block rows from panel $Y$ back to the CPU host. The solve is performed using PZTRSM in PBLAS.
3. The rank-$k$ update to the lower part of panel $Y$ is performed as a matrix-matrix multiplication. Similar to the in-core factorization, this requires broadcasting panel $X$ across processor columns and broadcasting part of $U_{12}$ in panel $Y$ down processor rows. Then the rank-$k$ update can be performed using \texttt{cublasZgemm} on the GPU without further communication.

4. After all previous updates are performed, the LU factorization of the lower rectangular part of panel $Y$ ($\tilde{A}_{22}$) is computed using the all in-core algorithm described in Section 2.3.1.

5. The LU factors in panel $Y$ are then copied from the GPU device back to CPU host.

6. The final application of pivoting to the previously computed factors in $L_{21}$ is performed by the CPU host.

The choice of the width of panel $Y$ will affect the performance of the algorithm, but is also governed by the amount of GPU device memory available. If we choose the width of panel $Y$ to be $(N/K)$ columns such that the total size of panel $Y$ is $N \times (N/K)$, or where the entire matrix is $K$ times the width of panel $Y$, the factorization of the first $Y$-panel requires no previous factors. However, subsequent factorization of the $k$-th panel requires the transfer of the previous $k-1$ $Y$-panels. The volume of data transferred is thus $(1 + 2 + \ldots + (K - 1)) \times (N \times (N/K)) = (K - 1)/2 \times N^2$. Thus, the choice of $K$ should be as small as possible to make the width of panel $Y$ as large as possible. This will minimize the amount of data transferred between the GPU and the CPU. However, the width $(N/K)$ will be limited by the amount of device memory available on the GPU.

3. Numerical Results

Numerical experiments were performed on a small CPU/GPU cluster at the OLCF. Fifteen compute nodes were available, each consisting of a dual socket, 6 core AMD 2.6 GHz CPU with 32 GBytes of memory, and two NVidia M2050 GPUs. Each M2050 has 4.3 GBytes of device memory and the transfer rate between the device memory and pinned memory on the CPU is about 5 GBytes/s. Despite having two GPUs per node available, we focused on using only a single GPU per node.

3.1. \textit{ScaLAPACK with Fortran-callable cuBLAS}

Figure 2 shows the performance of ScaLAPACK \texttt{PZGETRF} for factoring a \texttt{complex*16} matrix with $N = 40,000$ on 180 MPI tasks spawned over fifteen nodes (twelve tasks per node, or one task associated with each CPU core). The results suggest that matrix block sizes of $MB = 64$ or $MB = 128$ may be reasonable choices. Choosing a block size that is too small will hamper performance in serial dense matrix operations; however, choosing too large of a block size will reduce the parallelism inherent in the algorithm. For these block sizes, each core attains a performance of over 6 GFlops (or about 72 GFlops per node).

Figure 2 also shows the performance of ScaLAPACK \texttt{PZGETRF} using the Fortran “thunking” wrapper to CUBLAS on a \texttt{complex*16} matrix with $N = 40,000$. This problem is sufficiently small to meet the memory limitations of
the GPU (1.7 GBytes required versus 4.3 GBytes available); however, the performance gains attained by using the GPU do not mitigate the cost of the data movement. The best performance is approximately 22 GFlops per node, significantly less than that attained with a purely CPU implementation.

Figure 2: Comparison of the performance of ScaLAPACK PZGETRF with ScaLAPACK PZGETRF accelerated with Fortran-callable cuBLAS for different block sizes MB for N = 40,000 on a 12 × 15 processor grid.

3.2. HPL with MagmaBLAS

Figure 3 compares the performance of the modified LU factorization in HPL with an accelerated version based on magmaBLAS for a complex*16 matrix with N = 40,000 on a 12 × 15 processor grid. As demonstrated for ScaLAPACK-cuBLAS, the cost of the data movement exceeds the performance improvement attained from the GPU, so the overall performance of the LU factorization is slowed by moving the data to the GPU. However, as shown in Figure 4, in the case of a smaller matrix with N = 25,000 on a 3 × 5 processor grid in which every cluster node executes 1 CPU task and 1 GPU task, the GPU-accelerated version outperforms the CPU-only version by a factor of 6.

Figure 3: Comparison of the performance of HPL LU factorization without and with GPU-acceleration from MagmaBLAS for different block sizes MB for N = 40,000 on a 12 × 15 processor grid.

3.3. Out-of-core Factorization

Table 1 shows the performance data for our complex*16 out-of-core solver for N = 90,000 on 15 nodes. To simplify the alignment and data transfer between the CPU host and the GPU device, the MB block size for the in-core
Figure 4: Comparison of the performance of HPL LU factorization without and with GPU-acceleration from MagmaBLAS for different block sizes MB for \( N = 25,000 \) on a \( 3 \times 5 \) processor grid.

![Comparison chart]

Table 1: Comparison of the performance of ScaLAPACK PZGETRF with an out-of-core factorization method implemented on both the CPU and GPU for \( N = 90,000 \) on 180 MPI processes.

<table>
<thead>
<tr>
<th>MB</th>
<th>processor grid</th>
<th>ScaLAPACK GFLOPs</th>
<th>Out-of-Core GFLOPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>12 \times 15</td>
<td>1103</td>
<td>1813</td>
</tr>
<tr>
<td>128</td>
<td>12 \times 15</td>
<td>1104</td>
<td>1633</td>
</tr>
<tr>
<td>64</td>
<td>15 \times 12</td>
<td>1043</td>
<td>1714</td>
</tr>
<tr>
<td>128</td>
<td>15 \times 12</td>
<td>885</td>
<td>1554</td>
</tr>
</tbody>
</table>

ScaLAPACK matrix is set to be the same block size as the matrix on the GPU. Each MPI task was set to allocate about 256 MBytes of the GPU device memory for panel Y (or each node using about 3 GBytes of the GPU device memory). For simplicity, we have not initially considered explicitly using multiple streams or asynchronous data transfers. The out-of-core factorization achieves approximately 120 GFlops per node with MB = 64 on a 12 \times 15 processor grid for \( N = 90,000 \) versus 72 GFlops per node for the non-accelerated implementation. While the performance speed-up of approximately 50% is not dramatic, the real utility of this algorithm is that it is capable of solving problems that exceed the total GPU device memory.

In comparison, Table 2 shows the performance of LU factorization by the routine magma_zgetrf_gpu of the MAGMA library, as displayed by the MAGMA tester testing_zgetrf_gpu on a single GPU device. The 258 GFlops performance of the MAGMA LU factorization is the upper bound of the performance for the parallel out-of-core solver, since the out-of-core parallel solver must also perform data movement and MPI communication. Moreover, some of the operations such as the factorization of narrow panels are performed using PBLAS and ScaLAPACK on the slower CPUs.

We believe there is still room for performance optimization and tuning, such as individual tuning of matrix block sizes for ScaLAPACK on CPU and GPU devices, exploiting asynchronous data transfer operations, and using lookahead computation of the next panel to reduce the time spent on the critical path for LU factorization. From another perspective, the GPU’s 120 GFlops per node performance is roughly 20 times the performance of a single CPU core. This suggests that pursuing efficient new algorithms implemented on GPUs offers a potential advantage in both cost and power efficiency.

4. Future Work

In summary, we have described an initial development of three approaches for the parallel factorization of dense matrices on a GPU cluster in preparation for hybrid CPU/GPU architectures.
Table 2: Performance of magma_zgetrf_gpu LU factorization on square complex*16 matrices residing in GPU device memory.

<table>
<thead>
<tr>
<th>N</th>
<th>CPU GFlops/s</th>
<th>MAGMA GFlops/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>960</td>
<td>5.3</td>
<td>63.0</td>
</tr>
<tr>
<td>1920</td>
<td>6.7</td>
<td>141.6</td>
</tr>
<tr>
<td>3072</td>
<td>6.9</td>
<td>194.3</td>
</tr>
<tr>
<td>4032</td>
<td>7.6</td>
<td>203.8</td>
</tr>
<tr>
<td>4992</td>
<td>7.7</td>
<td>228.9</td>
</tr>
<tr>
<td>5952</td>
<td>7.7</td>
<td>245.4</td>
</tr>
<tr>
<td>7104</td>
<td>7.8</td>
<td>258.6</td>
</tr>
</tbody>
</table>

1. The naive “thunking” approach of linking with a software emulation layer for BLAS operations requires very little software development effort. However, depending on the problem being solved, little-to-no performance gains may be achieved.

2. Carefully choosing the BLAS operations that will be moved to the GPU and utilizing an optimized GPU BLAS library such as magmaBLAS may yield some performance improvements with a small investment in software development time. However, the cost of the data movement between the CPU and the GPU may exceed any gain if care is not taken.

3. Matrices larger than available GPU device memory can be solved by using a left-looking out-of-core algorithm. This algorithm requires the most investment in software re-engineering, but ultimately may provide the most utility as it is able to solve problems where the system exceeds the available GPU memory.

Future optimizations that may be considered include further separate tuning of matrix block size for ScaLAPACK on CPU and GPU devices, exploiting asynchronous operations in data transfer, and implementing look-ahead computation of the next panel in the out-of-core algorithm to reduce time on the critical path for LU factorization.

5. Acknowledgments

This Research was sponsored by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U. S. Department of Energy. This research used resources of the Center for Computational Sciences at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U. S. Department of Energy under Contract No. DE-AC05-00OR22725.

References


