Proceedings of the IASTED International Conference
Parallel and Distributed Computing and Systems (PDCS 2012)
November 12 - 14, 2012 Las Vegas, USA

AN EFFICIENT OUT-OF-CORE IMPLEMENTATION OF BLOCK CHOLESKY DECOMPOSITION ON A MULTI-GPU SYSTEM

Lin Cheng
Department of Engineering
Trinity College
300 Summit Street
Hartford, CT 06106 USA
Lin.Cheng@trincoll.edu

Hyunsu Cho, Peter Yoon, and Jiajia Zhao
Department of Computer Science
Trinity College
300 Summit Street
Hartford, CT 06106 USA
{Hyunsu.Cho, Peter.Yoon, Jiajia.Zhao}@trincoll.edu

ABSTRACT
The Cholesky decomposition is one of the most efficient preconditioners to iterative schemes for solving linear systems such as the conjugate gradient method. However, we are often faced with situations where a linear system exceeds the capacity of existing memory. In this paper we present an efficient out-of-core implementation of the block Cholesky decomposition on a multi-GPU system, which will be able to handle linear systems of arbitrary size. Our implementation exploits in a streamlined fashion three core memory systems: GPU memory, CPU host memory, and virtual memory on the disk. We also demonstrate that incorporating memory traffic reduction, efficient data allocation and task overlapping is critical in optimizing performance. Our experiment shows that our implementation outperforms a multi-core CPU version by at least a factor of 30 for large matrices. We have also successfully applied our work to image segmentation.

KEY WORDS
Cholesky decomposition, general-purpose GPU computing, image segmentation

1 Introduction

Cholesky decomposition is one of the most widely used matrix factorization methods to solve linear systems of the form $Ax = b$, where A is symmetric positive definite. The Cholesky decomposition is known to be numerically stable because of its diagonal dominance and requires fewer operations than other direct methods [1]. This type of system arises in a number of applications areas including medical imaging, radars, and sonars.

Our main contribution of this paper is to develop an efficient out-of-core implementation of Cholesky decomposition using multiple graphics processing units (GPUs), when the given linear system exceeds the amount of available memory space. General-purpose GPU computing has been proven effective in accelerating scientific computations in recent years [5, 8]. It allows a high-level of data parallelism which is suitable for computing matrix factorizations such as Cholesky decomposition.

In our approach, we seek to minimize overhead associated with data transfer among different memory spaces, including device memory, host memory, and virtual memory space on the disk. To this end, we use a block version of outer-product formulation of Cholesky decomposition [3]. Using this form, each block can be operated independently of one another. We implemented our algorithm using CUDA, a parallel computing platform for the GPUs developed by Nvidia [7]. We also incorporated block operations of CUBLAS library, a linear algebra package highly optimized for CUDA [6].

In our multi-GPU implementation, each GPU is assigned only part of the matrix it needs to process. In terms of mapping, we evenly distribute computations on multiple GPUs. As a result, our out-of-core algorithm requires a significant amount of inter-GPU communication, and CPU must coordinate all the necessary communications between devices. We discuss our central strategies to reduce the amount and communication cost of inter-device memory traffic. To enhance the performance even further, we employ techniques such as task overlapping, look-ahead strategy, and peer-to-peer communication.

This paper is organized as follows: in the following section we discuss the implementation details of block Cholesky decomposition on a multi-GPU environment. Section 3 demonstrates how our work exploits the virtual memory on the disk. In Section 4 we present the performance results of our implementation. Finally, we conclude in Section 5.

2 Parallel Implementation

We first introduce the basic concept of GPU computing followed by a brief description of a block Cholesky decomposition algorithm. We also present a detailed parallel implementation on a multi-GPU system.

2.1 GPU Computing and CUDA

For our parallel implementation of block Cholesky algorithm, we seek to exploit a massive computing power of GPUs. Though originally designed for graphics-intensive applications, GPUs can be highly optimized for massively

DOI: 10.2316/P.2012.789-010

parallel numerical applications. GPUs comprise a large number of small computing cores and offer an excellent environment for fine-grain parallelism, where overall execution throughput is more important than the speed of individual cores [5].

During the past several years, the popularity of GPUs in general-purpose computing in addition to the graphics applications has grown significantly mainly because of their cost efficiency. CUDA played an integral role in general-purpose programming using GPUs. Unlike other graphical APIs such as OpenGL® and Direct3D®, CUDA offers a highly flexible interface that supports a variety of computations [5].

CPUs and main system memory are often referred as the *host* while GPUs are referred as *devices*. A *kernel* is a routine that executes on a device. On the CUDA platform, we add qualifiers to an ordinary C function to define a kernel. Those kernels designated as `__global__` can be invoked only by the host, while those qualified by `__device__` can be called only by a kernel [8]. In other words, devices begin their execution paths at a `__global__` kernel.

To effectively exploit data parallelism, CUDA uses two units of tasks called *blocks* and *threads*. A GPU device assigns a copy of kernel code to each block. GPU blocks share identical kernel instructions but maintain a distinct list of local variables. Each block is split into threads, which execute concurrently. But unlike blocks within a device, threads within a block may share variables. Also, a device kernel can easily synchronize threads within a block, whereas only the host can synchronize blocks within a device.

A multi-GPU system requires additional attention. First of all, GPU devices maintain separate lists of kernel invocations. Each kernel invocation applies to one device only. One effective way to coordinate kernel launches on multiple devices is to use host threads. By using multiple host threads, we can concurrently generate kernel requests to all devices. We use the POSIX threads standard to create host threads.

In addition, unlike CPU cores, GPU devices do not share their memory. Unless the problem at hand contains no data dependency, we need devices to communicate with one another. A conventional method is to copy the data from the sender device to the host and then to the recipient device. To make this data transfer more efficient, CUDA offers a new way of inter-device communication with higher throughput: *peer-to-peer* memory copy. Unlike the conventional approach, which involves two memory transfers, the new approach involves only one. The data is not transferred back to the host but directly between devices. One important limitation is that peer-to-peer memory copy is not always feasible. In many motherboards, PCI Express channels are divided into groups of two. Peer-to-peer copy is currently infeasible across separate groups. In such cases, it is necessary to perform memory transfer via the host.

2.2 Block Cholesky Decomposition

Suppose A is an $n \times n$ symmetric positive definite matrix. The Cholesky factor G of A is the upper triangular matrix such that

$$A = G^T G$$

In this paper we consider a block version of the outer-form Cholesky decomposition algorithm because it is suitable for parallel implementation. We describe the algorithm to compute Cholesky decomposition in three phases:

Algorithm 1 Block Cholesky: Outer Product Form [1]

Given an $n \times n$ symmetric positive definite matrix A, we divide A into a $q \times q$ grid of square submatrices of identical dimensions. Let $A_{(i,j)}$ be the (i,j)th submatrix in A. (Note that we use a 0-based indexing.) Also, let R_k be the set of submatrices $A_{(k,k)}, A_{(k,k+1)}, \cdots, A_{(k,q-1)}$ and h_k be the set of submatrices $A_{(k,k+1)}, A_{(k,k+2)}, \cdots, A_{(k,q-1)}$. Let $\tilde{A}_k$ be the union of $R_{k+1}, R_{k+2}, \cdots, R_{q-1}$ (cf. Figure 1). Finally, let $chol(B)$ denote the non-block version of Cholesky decomposition on symmetric positive definite matrix B. The following algorithm computes the Cholesky factor of A:

for $k = 0$ **to** $q - 1$ **do**
  /* Phase I */
  $A_{(k,k)} \leftarrow chol(A_{(k,k)})$

  /* Phase II */
  for $c \leftarrow k + 1$ **to** $q - 1$ **do**
    $A_{(k,c)} \leftarrow A_{(k,k)}^{-T} A_{(k,c)}$
  end for

  /* Phase III */
  for $r \leftarrow k + 1$ **to** $q - 1$ **do**
    for $c \leftarrow r$ **to** $q - 1$ **do**
      $A_{(r,c)} \leftarrow A_{(r,c)} - A_{(k,r)}^{T} A_{(k,c)}$
    end for
  end for
end for

We make the following observations from the algorithm:

- Each phase must wait for the previous phase to complete. Any two phases of one iteration cannot be overlapped.
- Phase I might prove to be a computational bottleneck. Since the data involved in Phase I is comparatively small (only as large as one submatrix), Phase I cannot gain as much speedup from data parallelism as Phase II and Phase III.
- Since the submatrix size is relatively small in practice (e.g., 128×128), it is advantageous to carry out Phase I in one device only; otherwise, communication overhead might outweigh any benefits gained from using

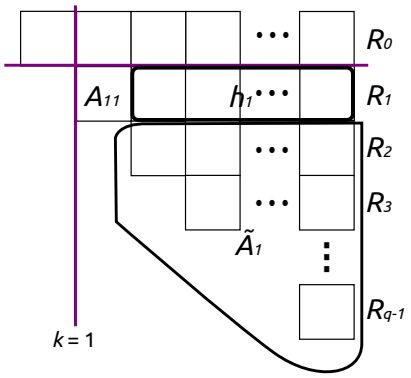

Figure 1: A grid of submatrices (iteration $k = 1$ shown)

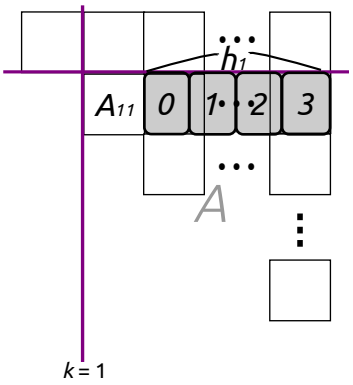

Figure 2: Allocating equal portions of h_k to devices (iteration $k = 1$ shown)

multiple devices. On the other hand, Phases II and III involve significant enough data that it is overall beneficial to allocate those data on multiple devices.

- As mentioned earlier, GPU threads can be easily synchronized within a device kernel. Our implementation of $chol(A_{(k,k)})$ divides the loops into unit tasks which in turn are assigned to multiple GPU threads.
- We do not compute $A_{(k,k)}^{-T}$ in Phase II in practice since calculating the matrix inverse is costly and numerically unstable. Instead, we solve the equivalent linear system $A_{(k,k)}^{T} X = A_{(k,c)}$ (where X is a set of unknown columns) and assign the solution X back to $A_{(k,c)}$. For this we use the `cublasDtrsm` function in the CUBLAS library.
- Phase III mainly involves matrix-matrix multiplication and addition. CUBLAS provides a Level 3 function named `cublasDgemm` for operations in the form $C \leftarrow \alpha AB + \beta C$ [6].

2.3 Data Allocation

In Phase II, devices receive approximately equal portions of h_k in the current iteration k (cf. Figure 2). We make sure that entries in each column stay together, as required by the linear solver.

In Phase III, each of $R_i (k+1 \leq i \leq q-1)$ in $\tilde{A}_k$ is allocated to one device in an alternating fashion (cf. Figure 3). Our approach allocates a roughly equal amount of data to each device. Also, we make sure that entries in each submatrix stay together, as required by the matrix multiplication function.

2.4 Data Communication

Since each device does not have the full picture of the matrix A, we must develop an efficient plan to coordinate the communication among devices. First of all, each device is assigned a fixed set of R_i's (cf. Figure 3) and keep modifying different portions of it in different iterations. In other words, after we copy certain R_i's in A to each device at the very first iteration, they stay on that device throughout the program execution. As a result, throughout each iteration k, devices can send back only the portion that requires no more modification, namely, the modified R_k. The rest of the data on each device stays and waits to be further modified by the same device.

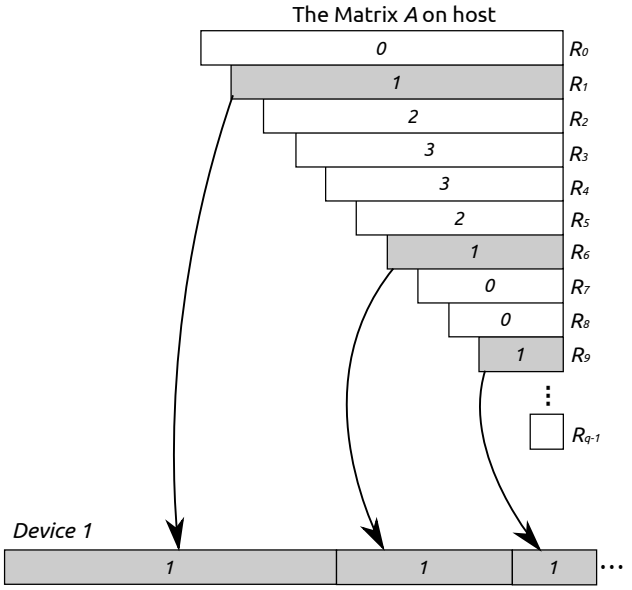

Figure 3: Allocating submatrices to devices (device 1 shown)

The transition from Phase II to Phase III can be carried out with minimal amount of communication between the host and devices. Suppose we are in iteration k. As we continue to the next iteration $k+1$, the device in charge of R_{k+1} copies and stores certain portions of R_{k+1} (cf. Figure 4) to be $A_{(k+1,k+1)}$ and h_{k+1}. This can be done efficiently using peer-to-peer memory copy implemented in CUDA. Inter-device communication through peer-to-peer chancel achieves a throughput twice as much as one through the host.

When we move from Phase II to Phase III, we assemble the portions of h_k scattered among different devices

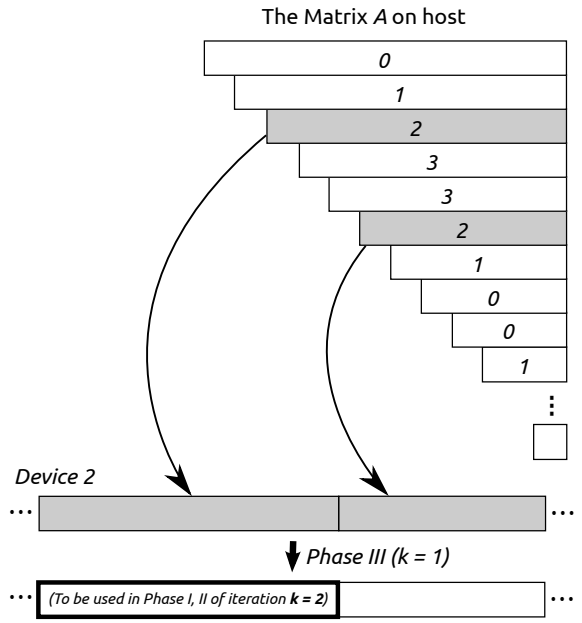

Figure 4: Compile $A_{(k+1,k+1)}$ and h_{k+1} at iteration k

into a complete h_k by using peer-to-peer memory copy. Each device broadcasts its portion of h_k to all the other devices (cf. Figure 5). As a result, every device has a complete picture of h_k, to be used in Phase III. Also, at the end of each iteration k, the device in charge of R_{k+1} extracts $A_{(k+1,k+1)}$ and broadcasts it to all the other devices. This submatrix is then used in the next iteration $k+1$.

2.5 Task Overlapping

Our implementation also makes use of task overlapping between kernel engines and copy engines, a feature supported by Tesla® models [8]. We allocate the tasks to two separate streams which can operate in parallel. Stream 0, for example, takes care of kernel invocations, while Stream 1 performs memory transfers. This resolves one particular bottleneck at Phase I: all other computations must wait for its completion. Phase I of iteration $k+1$ takes place while device 0 broadcasts h_k to other devices (cf. Figure 6). Note that the operations at the top of the figure are carried out only once by Device 0 at the first iteration.

2.6 Synchronization and Coordination

Our implementation is based on a hybrid computational model, which involves both CPUs and GPUs. CPUs coordinate the communication between GPUs, while GPUs perform the actual computations. A common technique is to spawn multiple CPU threads and assign them GPU devices [8]. Hence, to synchronize all the GPU devices, we must also synchronize the CPU threads. We use mutexes and condition variables to place a barrier between Phase II and Phase III as we need to make sure that all devices finish broadcasting h_k to one another. Without such barrier,

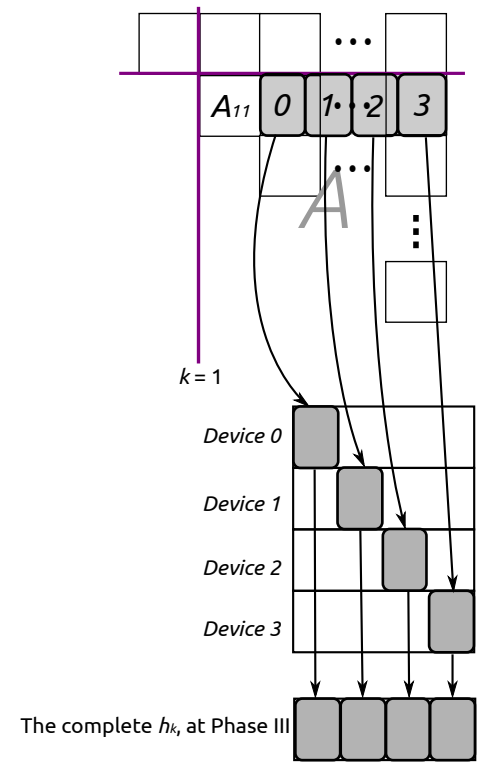

Figure 5: Assemble and broadcast h_k through peer-to-peer copy

the data would be exposed to unpredictable factors such as thread scheduling, thereby increasing the chance of corrupting the data. The POSIX threads standard provides for an API for creating and managing mutexes and condition variables.

Note that tasks allocated to the same stream are queued and executed one by one. Thus, barriers are only necessary to synchronize across multiple streams. Since Phases I and II are assigned to the same stream (Steam 0), there is an implicit barrier between them.

3 Extension to Disk I/O

We extend our work to handle the cases where the matrix is too large to fit into both the host and device memory. Thus, the matrix must be stored in a secondary storage. We streamline computation over three layers of memory: GPU memory, host memory, and virtual memory on the disk.

We first divide the matrix in submatrices, each of which can be loaded to the host memory as necessary. At each phase, we load several submatrices into the host memory, modify them on the devices, and write them back to the disk. We adapt the algorithm in the following way: Each submatrix should be large enough to reduce latency involved in file I/O but small enough so that several of them can be loaded into one GPU memory. An additional consideration should be given to the size of the host memory. Since the host memory on most systems is larger

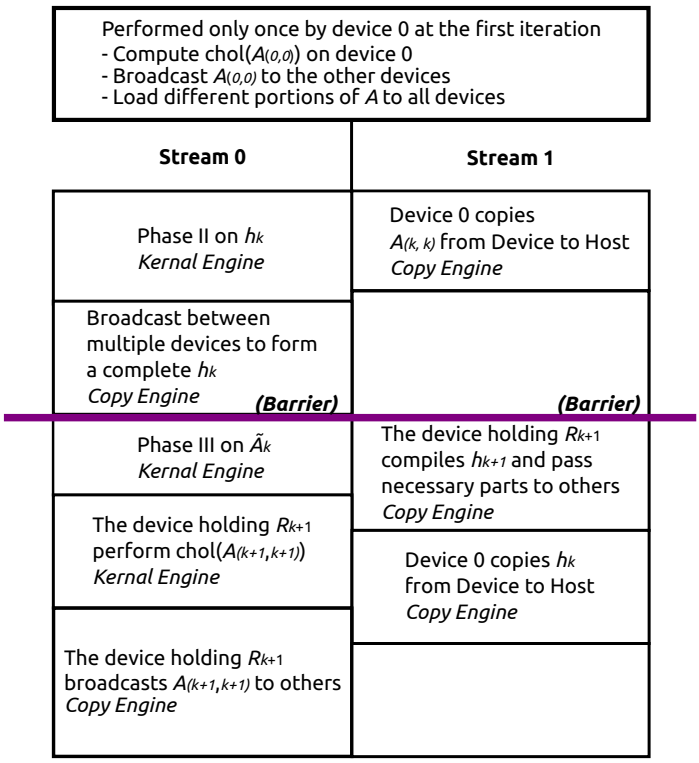

Figure 6: An overview on task overlapping

than the GPU memory, in order to reduce memory latency, one should read from the disk more submatrices than GPU memory can hold at one time.

Suppose the system matrix A is stored in the virtual memory on the disk. As in Algorithm 1, A is divided into a grid of submatrices (e.g. 4096×4096). Algorithm 2 computes the Cholesky decomposition of A. To reduce latency for disk access, we develop a more regular access pattern by reading and writing in large chunks. Specifically, we employ the following techniques:

- *Prefetching*: load from the disk as many submatrices as allowed by the host memory.
- *Delayed write*: write to the disk only when the program finishes modifying all the submatrices that were loaded to the host memory.

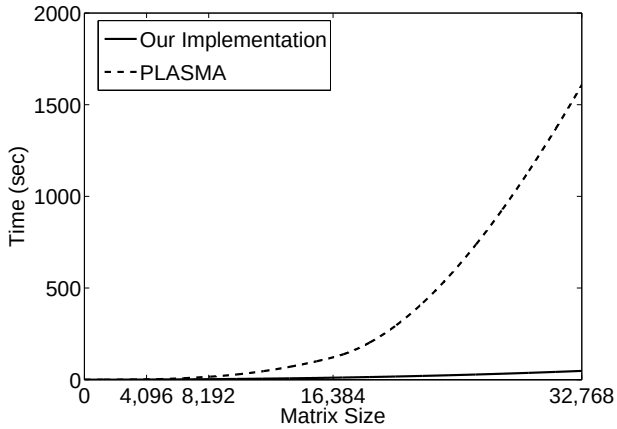

Figure 7: Run time of our program and PLASMA

4 Experimental Results

Our testing platform comprised two dual 2.4 GHz Intel® Xeon® quad-core CPUs with a total of 16GB of main memory, four Tesla C2050 graphics cards with 3GB memory, and CUDA 4.2 runtime system running on the Linux operating system. Our experiments used double-precision floating point arithmetics. In addition, all the implementations under testing have been compiled from the sources on our testing platform.

We first compared our implementation with the corresponding routine of PLASMA [2], a parallel linear algebra library for scalable multi-core architectures. Our implementation outperformed PLASMA by a factor of 33 for the largest case tried (cf. Figure 7). We also observed the performance of our implementation comparable to that of MAGMA [2], a matrix algebra library on GPUs and multi-core architectures. For instance, for a 32768×32768 input, our implementation showed an average throughput of 244.1 GFLOPs/sec while MAGMA performs at 242 GFLOPs/sec. We chose a block size of 128×128 because MAGMA and PLASMA are optimized for this block size. Experiments also show that this true for our implementation as well. But the optimal block size may differ in other

Algorithm 2 Block Cholesky with Disk I/O

Let $chol(B)$ denote the Cholesky decomposition of matrix B using Algorithm 1. Other notations remain the same.

for $k = 0$ **to** $q - 1$ **do**
  /* Phase I */
  Load $A_{(k,k)}$ from the disk to the host
  All devices partake in $A_{(k,k)} \leftarrow chol(A_{(k,k)})$
  Write back $A_{(k,k)}$ to the disk.

  /* Phase II */
  Broadcast $A_{(k,k)}$ to all devices.
  for for each submatrix $A_{(k,c)}$ in h_k **do**
    Load from the disk submatrices of h_k
    Allocate the submatrices to devices to compute:
    $A_{(k,c)} \leftarrow A_{(k,k)}^{-T} A_{(k,c)}$
    Write back the modified submatrices to the disk.
  end for

  /* Phase III */
  for each R_r in $\tilde{A}_k$ **do**
    for each submatrix $A_{(r,c)}$ in R_r **do**
      Load from the disk as many submatrices of R_r as possible.
      Load the necessary portions of h_k.
      $A_{(r,c)} \leftarrow A_{(r,c)} - A_{(k,r)}^{T} A_{(k,c)}$
    end for
  end for
end for

environments.

One of the major advantages of our implementation is that it is scalable to very large systems. It is limited only by the amount of virtual memory space on the disk. For instance, under the current testing environment, our program successfully computed the Cholesky factor of a 65536×65536 matrix in 886 seconds. Note that the matrix occupies 32 GB of memory, exceeding the capacity of the host memory. For this case, both MAGMA and PLASMA failed to compute the decomposition due to memory problem.

We also applied our parallel implementation to an image segmentation problem. Based on the work of Grady [4], we considered the random walk segmentation algorithm, a semi-automatic framework that takes user defined seeds as input and segments regions of interest in an image, such as a tumor. The major computational cost in performing the random walk segmentation lies in solving a large sparse linear system. A preconditioned conjugate gradient method is an attractive choice among many well-known methods. We first implemented a generalized version of the conjugate gradient method to work with images of arbitrary dimensions. To compute the preconditioner, we implemented incomplete Cholesky factorization algorithm presented by Golub and Van Loan [3]. Our implementation correctly segmented a region of interest (cf. Figures 8 and 9). Note that Figure 9 involves an 87195×87195 linear system that occupies 31.6 GB of memory.

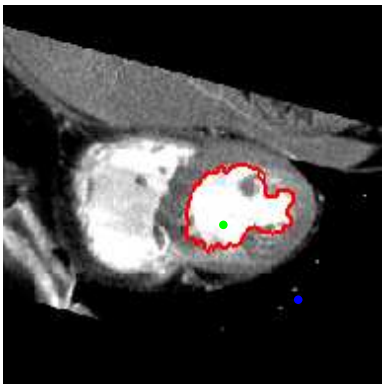

Figure 8: The segmented image of an axial CT scan [4] (192×192). The resulting system (36862×36862) required 10.1 GB of memory.

Figure 9: The segmented image of a crow (341×256). The original picture was used under permission of Department of Environmental Science, Trinity College.

5 Conclusion

We presented an efficient out-of-core implementation of Cholesky factorization on a multi-GPU system. Our implementation handles not only the linear systems which exceed the capacity of GPU memory, but those systems which require more than the available main memory. We have also carefully chosen the block size to minimize memory and disk latency when host-to-device, device-to-device, and host-to-disk communications become inevitable. The numerical experiments show a significant speedup of our implementation over a multi-core CPU implementation.

In addition, our parallel implementation was successfully used as a preconditioner to accelerate a conjugate-gradient based procedure for image segmentation problem.

References

[1] D. S. Watkins, *Fundamentals of Matrix Computations* (New York: John Wiley & Sons, Inc., 1991).

[2] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects, *Journal of Physics: Conference Series* Vol. 180, 2009.

[3] G. H. Golub and C. F. Van Loan, *Matrix Computations* (Baltimore: The Johns Hopkins University Press, 1996).

[4] L. Grady, Random Walks for Image Segmentation, *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28(11), 2006, 1768-1783.

[5] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach* (Burlington, MA : Morgan Kaufmann Publishers, 2010).

[6] Nvidia. "CUDA Toolkit 4.2 CUBLAS Library," 2012.

[7] Nvidia. "Nvidia CUDA C Programming Guide," 2012.

[8] J. Sanders and E. Kandrot, *CUDA By Example: an introduction to general-purpose GPU programming* (Upper Saddle River, NJ: Addison-Wesley, 2011).