2012 IEEE 18th International Conference on Parallel and Distributed Systems

Accelerating Volkov's Hybrid Implementation of Cholesky Factorization on a Fermi GPU

Shih-Chieh Wei
Department of Information Management
Tamkang University
Tamsui, Taiwan 25137
seke@mail.im.tku.edu.tw

Bormin Huang
Space Science and Engineering Center
University of Wisconsin-Madison
Madison, WI 53706, USA
bormin@ssec.wisc.edu

***Abstract*—In linear algebra, Cholesky factorization is useful in solving a system of equations with a symmetric positive definite coefficient matrix. Cholesky factorization is roughly twice as fast relative to LU factorization which applies to general matrices. In recent years, with advances in technology, a Fermi GPU card can accommodate hundreds of cores compared to the small number of 8 or 16 cores on CPU. Therefore a trend is seen to use the graphics card as a general purpose graphics processing unit (GPGPU) for parallel computation. In this work, Volkov's hybrid implementation of Cholesky factorization is evaluated on the new Fermi GPU with others and then some improvement strategies were proposed. After experiments, compared to the CPU version using Intel Math Kernel Library (MKL), our proposed GPU improvement strategy can achieve a speedup of 3.85x on Cholesky factorization of a square matrix of dimension 10,000.**

***Keywords**- Cholesky factorization, general purpose graphics processing unit, parallel computing*

I. INTRODUCTION

In linear algebra, to solve a system of linear equations, LU factorization is often used on a general coefficient matrix. When the coefficient matrix satisfies the condition of a symmetric positive definite matrix, Cholesky factorization can be used which is roughly twice as efficient as LU factorization [1]. In linear least squares problem, Cholesky factorization is useful in solving the normal equation to minimize the sum of differences between the two sides of the original fitting equation. Cholesky factorization also finds other applications in non-linear optimization, Monte Carlo simulation, and Kalman filters.

In recent years graphics processing units (GPUs) with hundreds of computing cores have become affordable for scientific computation. Currently, an NVIDIA high-end Tesla C2050 GPU card implementing the Fermi architecture has 448 computing cores. It delivers a theoretical peak performance of 1.03 TFLOPs in single precision. The combined features of general-purpose supercomputing, high parallelism, high memory bandwidth (144 GB/s), low cost, and compact size make a GPU-based desktop computer an appealing alternative to a massively parallel system made up of commodity CPUs. Increasing programmability of commodity GPUs allows their usage as General Purpose computation on GPUs (GPGPUs), which have recently attracted great attention for scientific applications [2][3].

This work aims to find an implementation of Cholesky factorization with high performance on the recent Fermi GPU. First, several open source GPU-based Cholesky factorization programs on the Internet were located, analyzed, and evaluated. We found that Volkov's implementation [10] performs the best. Then several strategies for performance improvement were proposed and tested on Volkov's implementation. Compared to the CPU version using Intel Math Kernel Library (MKL) [4], our proposed GPU improvement strategy on Volkov's implementation can achieve a speedup of 3.85x for Cholesky factorization of a square matrix of dimension 10,000.

The rest of this paper will be arranged as follows. Section II describes Cholesky factorization, Volkov's hybrid implementation and our improvement strategies. Section III shows the experimental results on these GPU implementations. Section IV summarizes the paper.

II. THE CHOLESKY FACTORIZATION AND ITS GPU IMPLEMENTATION

A. THE CHOLESKY FACTORIZATION

In the following, we assume that all matrices contain only real elements. Given a symmetric positive definite matrix $\boldsymbol{A}$, the Cholesky factorization can factorize a lower triangular matrix $\boldsymbol{L}$ such that $\boldsymbol{A}=\boldsymbol{L}\,\boldsymbol{L}^{\mathrm{T}}$. As an example, the Cholesky factorization of a 4x4 matrix $\boldsymbol{A}$ can be expressed as follows.

*Send correspondence to: bormin@ssec.wisc.edu

1521-9097/12 $26.00 © 2012 IEEE
DOI 10.1109/ICPADS.2012.147

$$A=\begin{pmatrix} A_{11} & A_{21} & A_{31} & A_{41} \\ A_{21} & A_{22} & A_{32} & A_{42} \\ A_{31} & A_{32} & A_{33} & A_{43} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{pmatrix}$$

$$=L\,L^{\mathrm{T}}=\begin{pmatrix} L_{11} & 0 & 0 & 0 \\ L_{21} & L_{22} & 0 & 0 \\ L_{31} & L_{32} & L_{33} & 0 \\ L_{41} & L_{42} & L_{43} & L_{44} \end{pmatrix}\begin{pmatrix} L_{11} & L_{21} & L_{31} & L_{41} \\ 0 & L_{22} & L_{32} & L_{42} \\ 0 & 0 & L_{33} & L_{43} \\ 0 & 0 & 0 & L_{44} \end{pmatrix}$$

$$=\begin{pmatrix} L_{11}^2 & L_{21}L_{11} & L_{31}L_{11} & L_{41}L_{11} \\ L_{21}L_{11} & L_{21}^2+L_{22}^2 & L_{31}L_{21}+L_{32}L_{22} & L_{41}L_{21}+L_{42}L_{22} \\ L_{31}L_{11} & L_{31}L_{21}+L_{32}L_{22} & L_{31}^2+L_{32}^2+L_{33}^2 & L_{41}L_{31}+L_{42}L_{32}+L_{43}L_{33} \\ L_{41}L_{11} & L_{41}L_{21}+L_{42}L_{22} & L_{41}L_{31}+L_{42}L_{32}+L_{43}L_{33} & L_{41}^2+L_{42}^2+L_{43}^2+L_{44}^2 \end{pmatrix} \quad (1)$$

From the above, we can obtain the following formula for the elements of $\boldsymbol{L}$ [1]. As $\boldsymbol{L}$ is lower triangular, only elements below ($i>j$) and on the main diagonal ($i=i$) need to be computed.

$$L_{i,i}=\sqrt{A_{i,i}-\sum_{k=1}^{i-1}L_{i,k}^2}\ , \quad (2)$$

$$L_{i,j}=\frac{1}{L_{j,j}}\left(A_{i,j}-\sum_{k=1}^{i-1}L_{i,k}L_{j,k}\right)\ \text{for } i>j. \quad (3)$$

It can be seen that each element $L_{i,j}$ only depends on the elements on the left and above. Therefore the elements can be computed from left to right and top to bottom in columns or rows. When the two dimensional matrices are stored in column major order which is our case here, column-wise computation is often used to exploit the local access pattern in a cache.

B. GPU IMPLEMENTATION OF THE CHOLESKY FACTORZATION

Traditionally, LAPACK (Linear Algebra PACKage) is the major source of library for doing various matrix factorizations which include LU factorization on general matrices and Cholesky factorization on symmetric positive definite matrices. In LAPACK, POTRF (symmetric POsitive definite matrix TRiangular Factorization) is the function for Cholesky factorization. For POTRF, as the input and output is a symmetric matrix, only the specified upper or lower triangular matrix will be used. In the following, we will assume that only the lower triangular matrix is used. For CPU with multiple cores, the Intel Math Kernel Library (MKL) has the state-of-the-art multithreaded implementation of LAPACK. Therefore MKL POTRF is often used as the baseline for comparison with other parallel implementations using GPU.

We have found several available GPU implementations of Cholesky factorization from Internet. Among them, CULA POTRF [7] is a commercial implementation without source code. We will mainly use it for performance comparison. For those implementations with source code, we have evaluated the POTRF functions from Bouchaert [9], Volkov [10] and MAGMA [8]. The result shows that Volkov's version performs the best. Therefore we will analyze Volkov's implementation below and propose some strategies for further improvement.

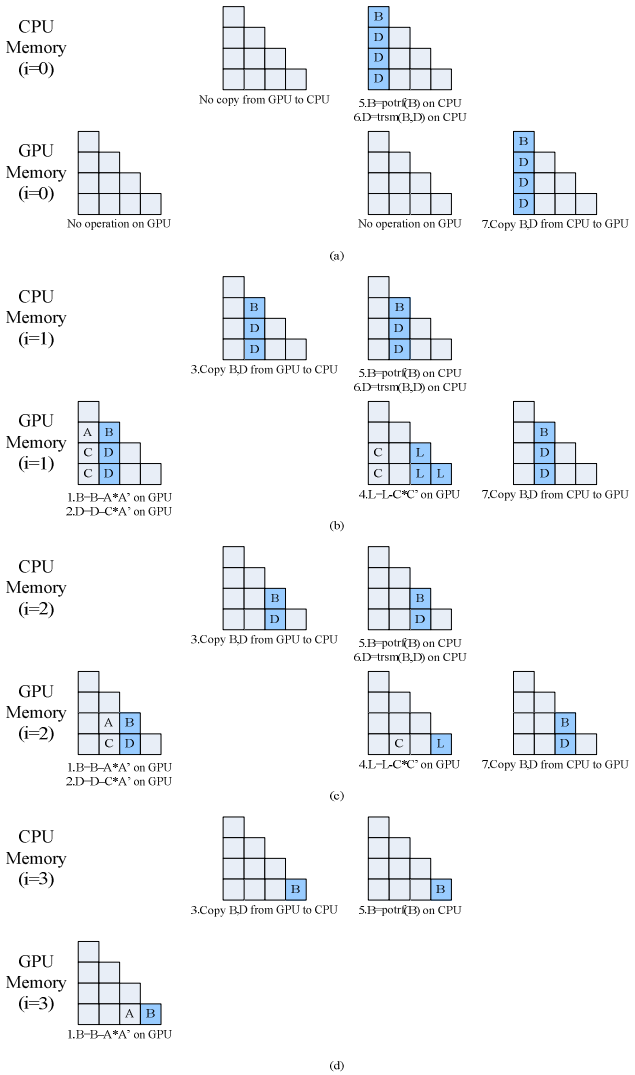

Figure 1. A matrix of 4x4 panels is used to illustrate Volkov's hybrid implementation of blocked Cholesky factorization on CPU and GPU.

Volkov's implementation is a right-looking variant of blocked Cholesky factorization as adopted by LAPACK [11]. For hybrid implementation using both CPU and GPU, there are different ways of workload balancing in addition to the panel factorization on CPU [12][13]. Fig. 1 illustrates Volkov's implementation using a sample matrix of 4x4 panels. Based on the hardware in use, Volkov fixed the panel size to 64 x 64 elements. As input and output, the given matrix is initially on CPU and copied to GPU for computation when necessary. In Fig. 1, four stages (i=0~3) are executed with each stage responsible for one panel column (denoted in gray color) from left to right. Except the first and the last stages where certain steps are skipped, all interim stages will execute the following 7 steps.

(1) Compute B=B-A A^T by the SYRK (SYmmetric Rank K update) function on GPU.
(2) Compute D=D-C A^T by the GEMM (GEneral Matrix Multiplication) function on GPU.
(3) Copy B,D from GPU to CPU.
(4) Compute L=L-C C^T by the SYRK function on GPU.
(5) Compute B=potrf(B) by the MKL POTRF function on CPU.
(6) Compute D=trsm(B,D) by the MKL TRSM (TRiangular Solve Multiple right hand sides) function on CPU.
(7) Copy B,D from CPU to GPU.

By considering the elements in eq.(1) as panels, it is easy to see how each panel is computed following the steps of stages in Fig. 1. The square root operation on an element corresponds to the Cholesky factorization operation on a panel. From Fig. 1, it can be seen that there is an overlap of CPU and GPU computation in steps 4,5 and 6 of each interim stage. If these two computations can take about the same time without much waiting before the copy synchronization, the compute power of both resources will be best utilized. Note that the pinned memory technique has been used in Volkov's implementation for faster copy process.

Based on Volkov's implementation, our improvement strategies consist of three folds. First, while Volkov used the MKL POTRF and MKL TRSM functions on CPU, he used his own developed BLAS (Basic Linear Algebra Subprogram) functions such as SYRK and GEMM on GPU. As the CUDA environment upgrades its CUBLAS functions [6], we found that using current CUBLAS SYRK and CUBLAS GEMM is faster now. Second, in the hybrid blocked version of Cholesky factorization, the size of the panel will affect the compute speed of each panel column and thus the balance of the CPU and GPU overlap in time. Based on the hardware in use, Volkov fixed the panel size to 64 elements. Based on our C2050 hardware which has 448 cores, we tune the panel size up to 384 for faster speed. Third, Volkov's code assumes the initial matrix on CPU which requires much memory copy for GPU computation. We developed a version that assumes the initial matrix on GPU. Only data needed for CPU computation has to be copied to and fro. We have wrapped up the revised code in a library form for easy use.

III. RESULT

To evaluate the GPU accelerated performance of the various implementations of Cholesky factorization, we run the test on random matrices of sizes 1000 to 10000 in 1000 steps. To generate symmetric positive definite matrices, a random zero-mean signal matrix $\boldsymbol{X}$ is first produced and then its covariance matrix $\boldsymbol{A}$ which fulfils the symmetric positive definite property is computed by $\boldsymbol{A} = \boldsymbol{X}\boldsymbol{X}^T$. For all experiments, only results for single precision floating numbers are reported.

TABLE I. SPECIFICATION OF THE NVIDIA TESLA C2050 GPU CARD.

Number of Streaming Processor Cores	448
Frequency of Processor Cores	1.15GHz
Total Dedicated Memory	3 GB GDDR5
Memory Speed	1.5 GHz
Memory Interface	384-bit
Memory Bandwidth	144 GB/sec

Our implementation and testing are based on an environment with a quad-core 2.4 GHz Intel Xeon E5620 CPU and an NVIDIA Tesla C2050 1.15 GHz GPU. With the Intel Hyper-Threading Technology, a maximum of 8 threads are available for execution. The specification of the NVIDIA Tesla C2050 GPU are shown in Table I. NVIDIA Tesla C2050 consists of 14 multiprocessors. Each multiprocessor has 32 thread processors. Each thread processor inside a multiprocessor runs synchronously. Thus, all 32 thread processors execute the same instruction at the same time. Threads are organized into three level hierarchies. The highest level is a grid, which consists of thread blocks. A thread block is a three dimensional array of threads. A current generation of NVIDIA's GPUs group threads in groups of 32 threads called warps. A multiprocessor issues the same instruction to the all threads in a warp. When threads take divergent paths multiple passes are required to complete the warp execution. Separate multiprocessors run asynchronously. Each Tesla C2050 GPU has 3 GB device memory, whose access bandwidth is higher than CPU's access to DRAM memory.

The test environment runs CentOS 5.7 with Kernel 2.6.18. For CUDA programming on GPU, CUDA Driver 4.0 [5] is used together with the CUDA-based linear algebra library of CULA Dense Package R13a [7]. Intel Math Kernel Library (MKL) 10.3.2.137 [4] is mainly used for full Cholesky factorization as well as blocked Cholesky factorization for the diagonal panels on CPU.

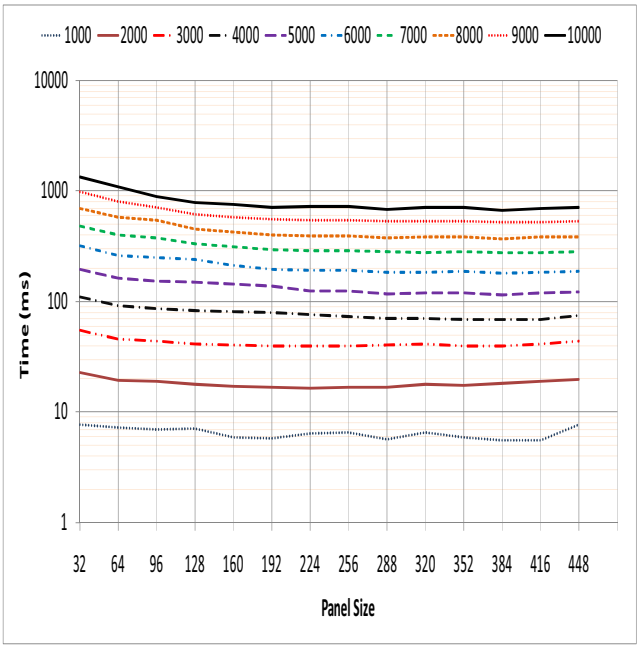

Figure 2. The running time of Volkov's implementation of Cholesky factorization (POTRF) on Fermi GPU for panel sizes 32 to 448 in 32 steps on a matrix of orders 1000 to 10000 in 1000 steps.

Fig. 2 shows our tuning process in finding a good panel size for Cholesky factorization of matrices with different orders. Considering the warp size of 32 in current GPU scheduling, a panel size in multiples of 32 up to the maximum 448 cores available in GPU is tested. It can be seen that for a matrix order larger than 3000, a panel size of 384 can best exploit the GPU hardware.

Table II shows the running time of the various implementations of Cholesky factorization which include MKL, CULA, Volkov, Bouchaert, and MAGMA. Among them only MKL runs purely on CPU. Others use GPU in combination with CPU. When using CPU, as many as 8 threads may be used for parallel computation on our quad-core test environment. Except MAGMA whose initial matrix is on GPU, all other implementations assume initial matrix on host and require transfer to GPU for computation. The running time shown is the average of 100 runs.

To show the improvement in our modified implementation, Volkov_N64 denotes the original Volkov implementation with a panel size of 64 and no use of CUBLAS functions; Volkov_N64_CU denotes the Volkov implementation using CUBLAS function; Volkov_N384 denotes the Volkov implementation with a panel size of 384 and no use of CUBLAS; and Volkov_N384_CU denotes our final modified Volkov implementation with a panel size of 384 and use of current CUBLAS functions. From Table II, it can be seen that as the dimension of the square matrix grows, the gain of using GPU over pure CPU increases. Among those using GPU, our final modified version Volkov_N384_CU runs the fastest, followed by original Volkov, CULA, and Bouchaert in order.

TABLE II. THE RUNNING TIME (S) OF VARIOUS IMPLEMENTATIONS OF CHOLESKY FACTORIZATION (POTRF) ON GPU AND THE MKL IMPLEMENTATION ON CPU.

Time (s)	MKL	CULA	Volkov N64	Volkov N64 CU	Volkov N384	Volkov N384 CU	Bouchaert	MAGMA
1000	0.004	0.011	0.006	0.006	0.005	0.005	0.020	0.008
2000	0.026	0.037	0.019	0.019	0.019	0.019	0.074	0.024
3000	0.075	0.081	0.045	0.045	0.040	0.039	0.161	0.056
4000	0.172	0.145	0.091	0.092	0.078	0.069	0.280	0.082
5000	0.330	0.247	0.164	0.164	0.136	0.113	0.436	0.185
6000	0.570	0.371	0.262	0.263	0.219	0.183	0.622	0.304
7000	0.890	0.540	0.404	0.403	0.335	0.274	0.842	0.465
8000	1.320	0.725	0.578	0.577	0.477	0.372	1.100	0.396
9000	1.860	0.982	0.810	0.810	0.670	0.526	1.446	0.952
10000	2.540	1.240	1.098	1.097	0.908	0.674	1.834	1.286

Fig. 3 shows the speedup of the various implementations of Cholesky factorization relative to MKL. A similar trend can be seen that as the dimension of the matrix grows, the speedup of all GPU-based implementations shows more gains. However it can be found that at dimension 1000, all GPU-based implementations are actually slower than MKL on CPU. The benefit of using GPU shows only at dimension 2000 upwards for our modified version Volkov_N384_CU, at dimension 3000 upwards for CULA and at dimension 7000 upwards for Bouchaert. From Fig. 3, there is peculiarity of performance boost with MAGMA at dimensions 4000 and 8000. This might be traced to the particular MAGMA GEMM implementation on Fermi GPU. Also, Volkov_N64_CU almost overlaps with original Volkov_N64 while Volkov_N384_CU is much better than Volkov_N384. This shows that CUBLAS works better with large panel size instead of small ones. Among the GPU-based implementations, our modified version

Volkov_N384_CU shows a speedup that scales up best with dimensions.

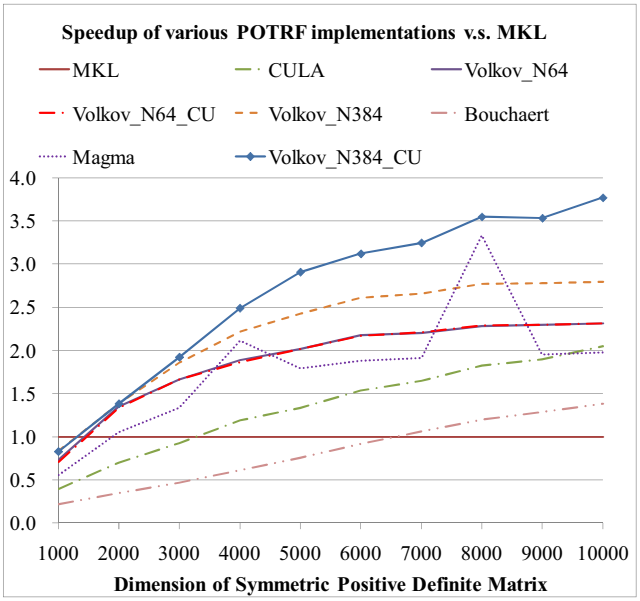

Figure 3. The speedup of various implementations of Choleksy factorization (POTRF) on GPU relative to the state-of-the-art MKL implementaiton on CPU.

IV. SUMMARY

Cholesky factorization is an important tool in solving systems of equations with symmetric positive definite coefficient matrices. With advances of technology, more GPU-based implementations have shown up to accelerate the factorization speed. In this work, we evaluated several available GPU implementations of Cholesky factorization on the recent Fermi GPU and found that Volkov's hybrid implementation using both CPU and GPU is the most competitive one. Then we analyzed Volkov's implementation and proposed several improvement strategies. The experiments show that our modified version can achieve a speedup of 3.85x relative to the state-of-the-art MKL implementation. As the speedup is related to the specific hardware in use, these improvement strategies might be applied to other platforms and even factorizations for optimized performance on a specific environment.

REFERENCES

[1] G. H. Golub and C.F. Van Loan, Matrix Computations, John Hopkins University Press, 1996.

[2] D. B. Kirk and W.-m. W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Morgan Kaufmann Publishers, 2010.

[3] J. Sanders and E. Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley, 2011.

[4] Intel, Intel Math Kernel Library Reference Manual, MKL 10.3, 2011.

[5] NVIDIA, NVIDIA Cuda Reference Manual, Version 4.0, 2012.

[6] NVIDIA, Cuda CUBLAS Library, Version 4.0, 2010.

[7] EM Photonics, Cula Reference Manual, Release R13, 2011.

[8] MAGMA, Matrix Algebra on GPU and Multicore Architectures Software, available at http://icl.cs.utk.edu/magma /software/.

[9] R. R. Bouckaert, Matrix inverse with CUDA and Cublas, available at http://www.cs.waikato.ac.nz/~remco/

[10] V. Volkov and J. W. Demmel, LU, QR and Cholesky factorizations using vector capabilities of GPU, Technical Report No. UCB/EECS-2008-49, University of California at Berkeley, 2008.

[11] J. Dongarra, I. S. Duff, D. C. Sorensen, H. A. van der Vorst, Numerical Linear Algebra for High-Performance Computers, SIAM, 1998.

[12] M. Baboulin, J. Dongara, and S. Tomov, Some Issues in Dense Linear Algebra for Multicore and Special Purpose Architectures, Technical Report UT-CS-08-200, University of Tennessee, May 6, 2008.

[13] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Orti, Solving Dense Linear Systems on Graphics Processors, Technical Report ICC 02-02-2008, Universidad Jaime I, February 2008.