Accelerating Tridiagonal Matrix Inversion on the GPU

Bemnet Demere, Ebenezer Hormenou and Peter Yoon
Department of Computer Science
Trinity College Hartford CT 06106
{bemnet.demere, ebenezer.hormenou, peter.yoon}@trincoll.edu

Abstract

Inverting a matrix is a more computationally challenging process than solving a linear system. However, in fields such as structural engineering, dynamic systems, and cryptography, computing the inverse of a matrix is inevitable. In this poster, we present an accelerated procedure for computing the inverse of diagonally dominant tridiagonal matrices on the GPU. The algorithm is based on the recursive application of the Sherman-Morrison formula for tridiagonal matrices. The preliminary experimental results on Nvidia Tesla K20c GPUs show that our GPU implementation of the inversion procedure outperforms the conventional CPU-based implementations with a speedup of up to 24x.

Keywords: tridiagonal matrix, inverse matrix, Sherman-Morrison formula, GPU computing, CUDA.

1. Introduction

This poster focuses on an efficient implementation to computing the inverse of diagonally dominant tridiagonal matrices on GPU. We employ the divide and conquer strategy based on a recursive application of the Sherman-Morrison formula [1]. The divide and conquer approach is a widely known method used in matrix computation which has proven to be very efficient to be implemented on various parallel architectures.

The basic steps involve dividing the matrix into submatrices of equal size which are sent to independent threads to compute the inverse. The process is repeated recursively for bigger submatrices until the size of the initial matrix is reached. Our implementation is based on accelerators such as GPUs using CUDA [2], a computing platform developed by Nvidia. Our implementation exploits the highly parallel grid organization of the GPU to simultaneously execute tens of thousands of threads on the matrix.

2. Algorithm for Computing the Inverse

Let *A* be an *n*-by-*n* diagonally dominant tridiagonal matrix, that is,

$$A = \begin{bmatrix} a_1 & b_1 & \dots & 0 & 0 & 0 \\ c_2 & a_2 & b_2 & \dots & 0 & 0 \\ 0 & c_3 & a_3 & b_3 & \dots & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & 0 & c_{n-1} & a_{n-1} & b_{n-1} \\ 0 & 0 & 0 & 0 & c_n & a_n \end{bmatrix}$$

where $b_i \neq 0$; $c_i \neq 0$ and $|a_i| \geq |b_i| + |c_i|$ for $i = 1, 2, \dots, n$. We also assume that $n = 2^q$ for some integer $q > 1$. Our goal is to compute A^{-1}. This is done in three basic steps.

Step 1. Reorganization of the matrix. The main diagonal elements are modified in the following way:

$$a_1^{(q-1)} = a_1$$

$$a_{2i}^{(q-1)} = a_{2i} - c_{2i+1}, i = 1, 2, \dots, 2^{q-1} - 1, \quad (1)$$

$$a_{2i-1}^{(q-1)} = a_{2i-1} - b_{2i-2}, i = 2, 3, \dots, 2^{q-1}, \quad (2)$$

$$a_n^{(q-1)} = a_n$$

Step 2. Inversion of 2-by-2 submatrices. We compute the inverse all *2*-by-*2* submatrices organized in the following way:

$$B_{q-1,i} = \begin{bmatrix} a_{2i+1}^{(q-1)} & b_{2i+1} \\ c_{2(i+1)} & a_{2(i+1)}^{(q-1)} \end{bmatrix} \quad \text{for } i = 0, 1, \dots, 2^{q-1} - 1$$

Then, $B_{q-1,i}^{-1}$ can easily be computed as

$$B_{q-1,i}^{-1} = \frac{1}{\Delta_i} \begin{bmatrix} a_{2(i+1)}^{(q-1)} & -b_{2i+1} \\ -c_{2(i+1)} & a_{2i+1}^{(q-1)} \end{bmatrix} \quad (3)$$

where Δ_i is the determinant of $B_{q-1,i}$

Step 3. Application of Sherman-Morrison Formula. We recursively apply the Sherman-Morrison formula on matrices of size $2^{q-k} \times 2^{q-k}$ for $k = q-2, q-3, \dots, 1, 0$ and $i = 0, 1, \dots, 2^k - 1$. Let

$$B_{k,i} = \begin{bmatrix} B_{k+1,2i}^{-1} & \\ & B_{k+1,2i+1}^{-1} \end{bmatrix} \quad (4)$$

$$B_{k,i}^{-1} = B_{k,i} - \propto_{k,i} B_{k,i} u_{k+1,2i} v_{k+1,2i}^T B_{k,i} \quad (5)$$

where $$\propto_{k,i} = \frac{1}{1 + v_{k+1,2i}^T B_{k,i} u_{k+1,2i}} \quad (6)$$

and

$$u_{k+1,2i} = e_{2^{q-(k+1)}} + e_{2^{q-(k+1)}+1}, \quad (7)$$

$$v_{k+1,2i} = c_{2^{q-(k+1)}(2i+1)+1} e_{2^{q-(k+1)}} + b_{2^{q-(k+1)}(2i+1)} e_{2^{q-(k+1)}+1} \quad (8)$$

where $e_{2^{q-(k+1)}}$ and $e_{2^{q-(k+1)}+1}$ are the $2^{q-(k+1)}$ and $2^{q-(k+1)}$ columns of the $2^{q-k} \times 2^{q-k}$ identity matrix.

Overall, there are *k* steps necessary to compute A^{-1}: Computing the inverse of the 2-by-2 matrices in (3) followed by $k-1$ steps of (5). At each next stage *k*, new submatrices $B_{k,i}$ are constructed by merging two submatrices of the previous stage following (4).

3. GPU Implementation

The structure of the inversion algorithm leads our parallel implementation on the GPU. At each *k*th stage, we need to compute $B_{k,i}^{-1}$ for $i = 0, 1, \dots, 2^k - 1$. CUDA allows us to launch multiple threads in blocks. Threads and blocks can be launched in one, two or three dimension.

In Step 1, we copy the matrix from the host CPU to the GPU, and launch 2^{q-1} GPU threads to reorganize the matrix. The *i*th thread computes $a_{2i}^{(q-1)}$ and $a_{2i-1}^{(q-1)}$. Similarly, we launch 2^{q-1} threads to compute the inverse of 2^{q-1} 2-by-2 matrices in Step 2, and each thread computes the inverse of one submatrix.

Finally, in Step 3, we launch 2^k blocks, each comprised $2^{q-k} \times 2^{q-k}$ threads arranged in two dimension. Each thread is assigned one entry of the matrix, and thus all the matrix operations are performed in parallel. After Stage 3 is completed, we copy A^{-1} back to the host. Figure 2 summarizes the entire process to compute the inverse of an 8-by-8 matrix. At each stage, we distribute the submatrices as described in Figure 1, and $B_{k,i}^{-1}$ is computed as described in Sec 2.

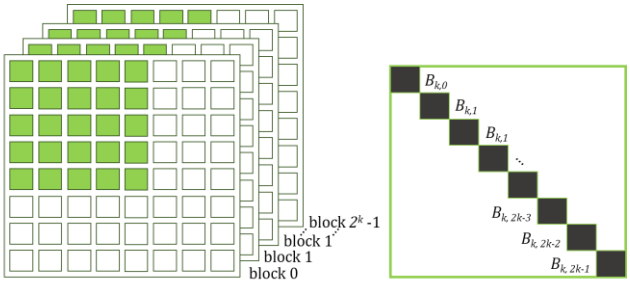

Figure 1. Distribution of blocks for submatrices

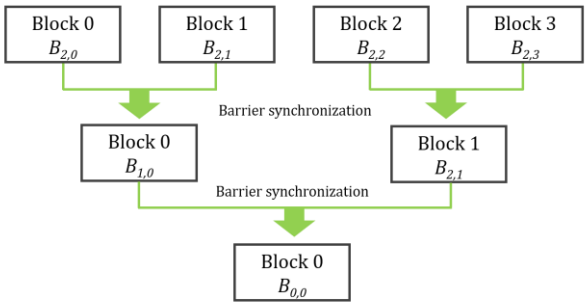

Figure 2. Schematics of the algorithm for an 8-by-8 matrix

4. Experimental Results

All experiments were done on a dual Xeon CPU, each running at 2.0 GHz with 64GB main memory, and an NVidia Tesla K20c GPU with 5GB of global memory. Our experiments show that our implementation outperformed CPU-only approaches, including 16-core CPUs. Figure 3 illustrates the performance of our implementation compared to multi-core CPU implementations. We observe that when the matrix size is sufficiently large, our implementation is at least 24 times faster than the sequential version, and 5 times than the multi-core version.

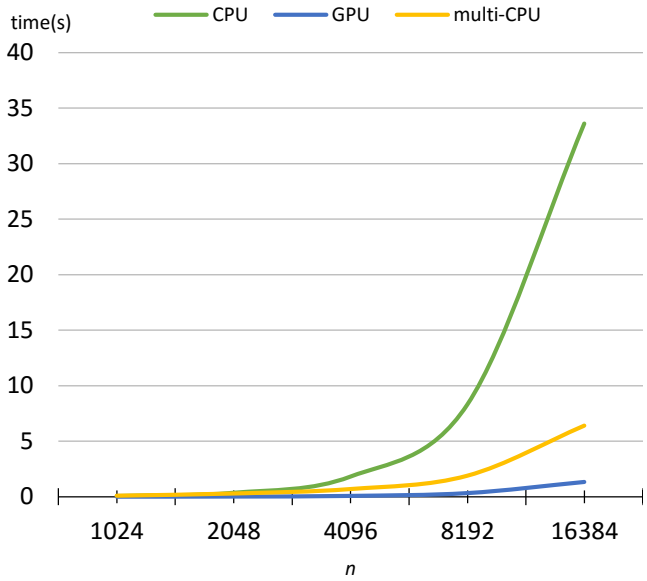

Figure 3. Timing Results

For smaller matrices, the CPU performs as well as the GPU. This is because the cost of data transfer between CPU and GPU outweighs that of the computation. The amount of GPU memory has been a major limiting factor in computing the inverse of large tridiagonal matrices. To this end, we would like to investigate both multi-GPU approach and out-of-core hybrid approach in order to distribute the matrix into a variety of system memories.

5. References

[1] Joan-Josep Climent, Leandro Tortosa, and Antonio Zamora. "A BSP recursive divide and conquer algorithm to compute the inverse of a tridiagonal matrix." *Journal of Parallel and Distributed Computing*, 59:423-444 (1999).

[2] CUDA C programming guide. NVIDIA.