Accelerating Tridiagonal Matrix Inversion on the GPU

Bemnet Demere, Ebenezer Hormenou, and Peter Yoon

Department of Computer Science, Trinity College, Hartford, CT

Introduction

Given an n-by-n diagonally dominant tridiagonal matrix,

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

$b_i \neq 0;\ c_i \neq 0;\ |a_i| \geq |b_i| + |c_i|, i = 1, 2, \dots, n.$

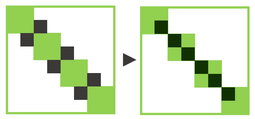
We also assume that $n = 2^q$ for some integer $q > 1$. Our goal is to compute the inverse of A. Inverting a matrix is a more computationally challenging process than solving a linear system. However, in fields such as structural engineering, dynamic systems, and cryptography, computing the inverse of a matrix is inevitable. In structural engineering, for example, one needs to compute the *stiffness matrix* to solve the spring-mass problem.

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. We implement the algorithm on accelerators such as GPUs using CUDA [2], a parallel 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.

Algorithm for Computing the Inverse

1 Rearranging elements

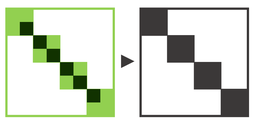
Modify the main diagonal elements: for some $q \geq 1$



$$a_1^{(q-1)} = a_1$$
$$a_{2i}^{(q-1)} = a_{2i} - c_{2i+1}, \quad i = 1, 2, \dots, 2^{q-1} - 1,$$
$$a_{2i-1}^{(q-1)} = a_{2i-1} - b_{2i-2}, \quad i = 2, 3, \dots, 2^{q-1},$$
$$a_n^{(q-1)} = a_n$$

2 Inverting submatrices

Invert each 2-by-2 matrix



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

for $i = 0, 1, \dots, 2^{q-1} - 1$

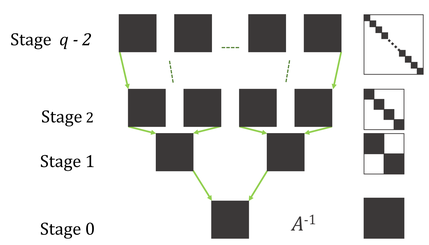
Then, $B_{q-1,i}^{-1}$ can easily be computed as

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

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

3 Application of Sherman-Morrison formula

Apply recursively 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 $B_{k,i} = \begin{bmatrix} B_{k+1,2i}^{-1} & \\ & B_{k+1,2i+1}^{-1} \end{bmatrix}$

then

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

where

$$\propto_{k,i} = \frac{1}{1 + v_{k+1,2i}^T B_{k,i} u_{k+1,2i}}$$

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

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

Here, $e_{2^{q-(k+1)}}$ and $e_{2^{q-(k+1)}+1}$ are the elementary vectors of the identity matrix.

GPU Implementation

1. Matrix A is first copied from the host memory to the GPU memory.
2. 2^{q-1} GPU threads are launched to reorganize the matrix. The i^{th} thread computes $a_{2i}^{(q-1)}$ and $a_{2i-1}^{(q-1)}$.
3. Launch 2^{q-1} threads to compute the inverse of 2^{q-1} 2-by-2 and each thread computes the inverse of one 2-by-2 submatrix.
4. Finally, 2^k blocks are launched, each comprised $(2^{q-k})^2$ threads arranged in two dimension. Each thread is assigned one entry of the matrix, and thus all the matrix operations are performed in parallel (Figure 1).
5. Copy the inverse of A back to the host memory.

```
/* kernel<<< grid_dimension, block_dimension >>> */
reorganize<<<2^(q-1), 1>>>(A, n) /*step 2*/
invert2x2<<<2^(q-1), 1>>>(A, n) /*step 3*/
shermanMorrison(A, n) /*step 4*/
```

```
shermanMorrison(A, n):
  k ← number of stages
  for i ← k to 0
    /* generate vectors u_{k+1,2i} and v_{k+1,2i} */
    uvCompute<<<2^k, 1>>>(...)
    /* compute X_{k+1,2i} ← B_{k,i} u_{k+1,2i} */
    XCompute<<<2^k, (2^(q-k), 2^(q-k))>>>(...)
    /* compute Y_{k+1,2i} ← v^T_{k+1,2i} B_{k,i} */
    YCompute<<<2^k, (2^(q-k), 2^(q-k))>>>(...)
    /* compute ∝_{k,i} */
    alphaCompute<<<2^k, 1>>>
    /* compute B_{k,i} − ∝_{k,i} X_{k+1,2i} Y_{k+1,2i} and update matrix */
    update<<<2^k, (2^(q-k), 2^(q-k))>>>(...)
  end for
```

`XCompute` and `YCompute`

To compute $X_{k+1,2i}$ and $Y_{k+1,2i}$:

- Concatenate all vectors $u_{k+1,2i}$ to form a single vector U_k and compute the product $A_k U_k$ using **`sgemm()`** of the cuBLAS library.
- Concatenate all vectors $v_{k+1,2i}^T$ to form a single vector V_k and compute the product $A_k V_k$ using **`sgemm()`**.
- Note that this can be done without launching 2^k threads.

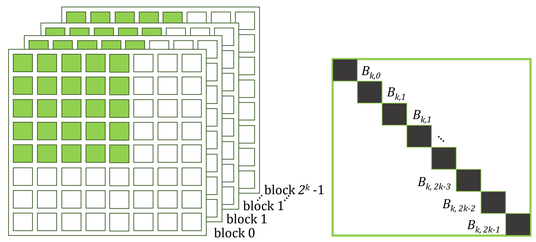


Figure 1. Distribution of blocks for submatrices

`Update`

- Launch 2^k blocks in 3 dimension, as shown in Figure 1, where each block has the size of the submatrix $B_{k,i}$. This ensures that every data point is updated simultaneously.

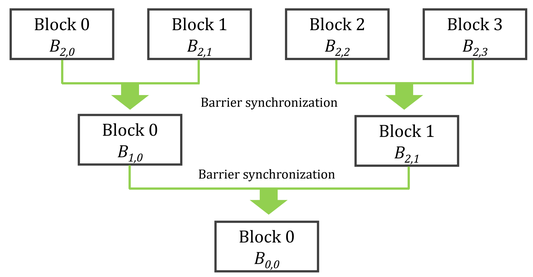


Figure 2. Schematics of the algorithm for an 8-by-8 matrix

Performance

- 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.

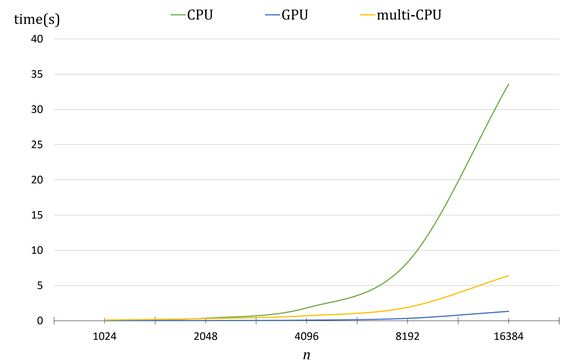


Figure 3. Timing Results

Discussion and Future Work

- Because of the uniform structure of submatrices, workload was well balanced among all threads.
- As expected, applying Sherman Morrison formula took the most time (approximately 70 % of total running time).
- The amount of GPU memory has been a major limiting factor in computing the inverse of large tridiagonal matrices. The largest matrix we were able to invert was 2^{14}–by–2^{14} which is considered to small in many applications.
- We are planning to investigate both multi-GPU approach and out-of-core hybrid approach in order to distribute the matrix to a variety of host and device memories.

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, 2015.
3. cuBLAS library users guide, NVIDIA, 2015.

Acknowledgements

This research was supported by:

- CUDA Teaching Center Program, NVIDIA Research
- Faculty Research Committee and Interdisciplinary Science Program, Trinity College