Investigation on the Use of GPGPU for Fast Sparse Matrix Factorization

Y. Tian*, B. Zhou*, and Y.T. Zhang* and K.W. Chan†

Abstract - Solution for network equations is frequently encountered by power system researchers. With the increasingly larger system size, time consumed network solution is becoming a dominant factor in the overall time cost. One distinct and important feature of the network admittance matrix is that it is highly sparse, which need to be addressed by specialized computation techniques. One technique to accelerate matrix factorization is parallel computation, with which data processing can be divided into different tasks and implemented simultaneously. However, up to now, efficiency of parallel computation algorithm implemented on multi-processor systems is adversely affected by the data communication latency between processors. In this paper, by taking advantage of the parallel computing power of the contemporary Graphic Processing Units (GPU) and designs of sparse technique for matrix factorization implemented on GPU, proposed algorithms are implemented and evaluated on the Computer Unified Device Architecture (CUDA) interface of the NVIDIA GPU. Preliminary results show significant improvement of speed of LU factorization.

Keywords: Sparse matrix factorization, GPU computing, GPGPU, CUDA

1. Introduction

Fast sparse matrix computation is one of the core technologies applied to many power system analysis applications, including load flow, eigenvalue analysis, transient stability analysis, state estimation, etc., which are essential for the operation of modern power systems and involve repeated solution of large sets of linear equations of the general form:

\[ Ax = b \]

where \( A \) is an incidence symmetric sparse matrix, \( b \) is a given independent vector, and \( x \) is an unknown solution vector. With the increasingly larger system size and use of computation intensive analysis applications in operation and control of future intelligent power systems with self-healing capability, the time consumed solution of (1) is becoming a dominant factor in the overall time cost and shall be addressed by specialized computation techniques. Parallel computation is one of such techniques for accelerating sparse matrix computation, with which data processing is divided into different tasks and implemented simultaneously. However, up to now, efficiency of parallel computation algorithm implemented on multi-processor systems is adversely affected by the data communication latency between processors.

In this paper, the parallel computing power of the contemporary Graphic Processing Units (GPU) would be investigated, and customized sparse matrix factorization algorithms would be proposed to take the full advantage of GPU and implemented on the Computer Unified Device Architecture (CUDA) interface of the NVIDIA GPU.

1.1 GPGPU and CUDA

GPGPU stands for General-Purpose computation on Graphics Processing Units (GPU). Nowadays, usage of GPU with powerful floating point calculation capability has been extended beyond the traditional graphics domain and opened a new era [1]. GPU is now used for general purpose in computing intensive and highly parallel applications frequently encountered in areas like scientific computation, engineering, fluid dynamics simulation, medical imaging, financing, etc.

NVIDIA CUDA programming model has been designed to make it easier to develop non-graphical applications on GPUs [2,3]. So far, CUDA is the only C language environment GPU programming interface with good programmability. Currently, CUDA is proved to be the best programming model available, and is generally accepted by the research and development community [4]. Since the

[† Corresponding Author: Department of Electrical Engineering, The Hong Kong Polytechnic University, Hong Kong SAR. (eekwchan@polyu.edu.hk)]

* Department of Electrical Engineering, The Hong Kong Polytechnic University, Hong Kong SAR. (tianye1313@126.com, 09901156@polyu.edu.hk, 06901882r@polyu.edu.hk)

Received: May 1, 2010; Accepted: November 29, 2010
release of CUDA, it has been proved to be efficient in a large number of applications [5-8]. Thus CUDA is chosen in the study on the platform of Visual Studio 2008.

The GPU being used is NVIDIA GeForce GTX260. Each computing unit can perform the 32-bit single precision floating-point arithmetic as well as the 32-bit integer arithmetic [4]. In a GPU, each SM has four different types of on-chip memory, namely, constant memory, texture memory, registers, and shared memory, which all have their own features [9]. For GeForce GTX260, each SM has 16384 32-bit registers and 16KB shared memory, which shall be made full use of.

1.2 Sparse Matrix Computation

A sparse matrix is one that has "very" few non-zero elements [10]. For a common bus admittance matrix with N nodes, if the average branch rate for each node is $a$, the sparsity of this matrix is:

$$\rho = \frac{a+1}{N} \times 100\%$$  \hspace{1cm} (2)

For practical power system, $a$ is usually 3~5, thus for a grid with 500 nodes, the sparsity is approximately 1%.

Considering the solution of the linear problem, Eq. (1), when $A$ is factorized into a lower triangular matrix $L$ and an upper triangular matrix $U$ with ones on the diagonal, Eq. (1) is transformed to:

$$LUx = b$$  \hspace{1cm} (3)

Eq. (3) could then be solved by forward and backward substitution. Still, during the LU factorization, a significant number of multiplications where one or both of the factors are zero may be involved [10].

2. LU Factorization on CUDA

2.1 Sparse Matrix Storage

As described in Section 1.2, it is important to avoid zero storage when packing matrix. Linked list is usually used to store non-zeros. Basically, there are two types of linked list, static linked list and dynamic linked list. Dynamic linked list shall be the most preferable method when it comes to memory saving. And it is quite flexible when the new fill-in is inserted into the format. However, the convenience of locating and indexing fill-ins might be counteracted by time-consuming problems in each memory allocation. Another more important concern is that in the kernel function of CUDA, a pointer can only point to global memory of the GPU. Since global memory access has a long latency of 400~600 clock cycles, static linked list would be adopted.

By introducing static linked list, arrays are used to store parameters of non-zeros. The index of these data indicates their positions in the array. In the study, following arrays are set to pack sparse matrix:

- value: value of non-zero element
- nrow: row number of non-zero element
- ncol: column number of non-zero element
- nir: index of next non-zero element in the same row, equals -1 when this is already the end
- nic: index of next non-zero element in the same column, equals -1 when this is already the end
- fir: index of the first non-zero element in each row
- D: index of each diagonal element

It is assumed that the matrix is diagonally dominant. Defining number of non-zero elements as NUM_NZ, dimension of the matrix as DIM, total space that is to be taken up after all arrays are formed is: 

$$4 * \text{NUM}_\text{NZ} * \text{sizeof} \ (\text{int}) + \text{NUM}_\text{NZ} * \text{sizeof} \ (\text{type}) + 2 * \text{DIM} * \text{sizeof} \ (\text{int}).$$

Here, for the type, it is float for single precision and double for double precision. For the first five arrays, space allocated is determined by existing algorithm which would indicate the number of fill-ins while reordering the matrix.

2.2 Parallel LU Factorization on Kernels

For n*n matrix $A$, the standard LU factorization process is as follows:

$$\begin{align*}
\text{for} \ (p=1; p<n; p++) \\
\quad \text{for} \ (j=p+1; j<=n; j++) \\
\quad \quad a_{pj} &= a_{pj} / a_{pp}; \quad \quad \text{(normalization)} \\
\quad \quad \text{for} \ (i=p+1; i<=n; i++) \\
\quad \quad \quad a_{ij} &= a_{ij} - a_{ip} * a_{pj}; \quad \quad \text{(elimination)} \\
\end{align*}$$

(4)

For a kernel function, a predetermined number of threads are launched to execute the same instructions but work on different data. For any $p$, the normalization and elimination process from the $(p+1)^{th}$ to the $n^{th}$ column needs to be carried out. For any column $j$, $(p+1) \leq j \leq n$, the normalization process just involves $a_{pp}$ and the elimination process just involves $a_{ip}$ and $a_{pj}$. Obviously each column
works independently of any other column and thus can be processed in parallel.

After the matrix is stored in vectors as discussed in Section 2.1, the vectors shall first be transferred to the global memory, and then, the kernel moves the data from global memory to shared memory. One drawback of the shared memory is its small capacity of just approximately 16k Bytes, which would greatly confine the dimension of the matrix that can be factorized on GPU. And for one block, the maximum number of threads is 512, which means that this algorithm can deal with $512 \times 512$ matrix at most. These two confinements can be overcome with partitioning algorithm elaborated in the next section. The pseudo code of the kernel function is listed in Table 1.

Table 1. Single Block LU Factorization Algorithm

| 1. shared memory allocation; |
| 2. transfer data from global memory to shared memory; |
| 3. for (a=0; a< (DIM-1); a++) { |
| 4. allocate threads; |
| 5. each thread: do normalization and elimination for column starting with non-zero element; |
| 6. adjust or fill in the arrays accordingly; } |
| 7. transfer data back to global memory; |

2.3 Parallel LU Factorization of Partitioned Matrix

Network partitioning algorithm was first proposed by Kron in the 1950s [11]. The strategy is to divide a system into smaller ones, solve them independently before an overall solution. For a given power system, the equation based on its admittance matrix is as follows:

$$YU = I$$ (5)

$Y$ is the $N \times N$ admittance matrix, $U$ is the voltage vector and $I$ is the current vector. By tearing certain nodes in the matrix, the original network can be divided into several subsystems. With such nodes as cut nodes, Eq. (5) can be transformed to:

$$
\begin{bmatrix}
Y_{11} & Y_{12} & \cdots & Y_{1n} \\
Y_{21} & Y_{22} & \cdots & Y_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
Y_{n1} & Y_{n2} & \cdots & Y_{nn}
\end{bmatrix}
\begin{bmatrix}
U_1 \\
U_2 \\
\vdots \\
U_n
\end{bmatrix}
= 
\begin{bmatrix}
I_1 \\
I_2 \\
\vdots \\
I_n
\end{bmatrix}
$$ (6)

When $U_i$ is known, node voltage of each subsystem can be calculated as:

$$Y_a U_i = I_i - Y_a U_j, \quad i=1, \cdots, k$$

where $U_i$ is solved by the following:

$$
\begin{bmatrix}
\tilde{Y}_{11} \tilde{U}_1 \\
\tilde{Y}_{21} \\
\vdots \\
\tilde{Y}_{n1}
\end{bmatrix}
= 
\begin{bmatrix}
\tilde{I}_1 \\
\tilde{I}_2 \\
\vdots \\
\tilde{I}_n
\end{bmatrix}
$$

$$
\tilde{Y}_{11} \tilde{U}_1 = \tilde{I}_1 - \sum_{i=1}^{k} Y_a Y_a^{-1} Y_a U_i \\
\tilde{I}_1 = \tilde{I}_1 - \sum_{i=1}^{k} Y_a Y_a^{-1} I_i
$$ (8)

One matrix which is partitioned into four blocks is visualized in the Fig. 1. Blue dots in the figure represent original non-zero elements.

![Fig. 1. Example of Four Blocks Partitioning.](image)

Based on the blocking as shown in Eq. (6), the proposed algorithm divides the matrix into two parts, one part includes the $k$ blocks from $Y_{11}$ to $Y_{k1}$ and is to be transferred to GPU and the other part is what is left in the

Table 2. Multi-block LU Factorization Algorithm

| 1. shared memory allocation; |
| 2. for each matrix block, one thread block is allocated on CUDA, corresponding data are transferred to its shared memory; |
| 3. each thread block does LU factorization independently; |
| 4. transfer data back to global memory; |
Y. Tian, B. Zhou, Y.T. Zhang and K.W. Chan

matrix and is to be processed serially on CPU. The kernel allocates different thread blocks to do the LU factorization for each matrix based on \texttt{blockIdx.x}. The pseudo code of this algorithm is shown in Table 2:

**2.4 Test Results**

The matrix calculation was tested with NVIDIA GeForce GTX260 Extreme+. The GTX260 GPU uses the GT200 architecture. And it has 24 SMs, i.e., 192 1.24GHz processing cores, with 939MB onboard memory. Results from CPU, AMD Athlon 64 x2 3800+ Dual Core Processor with 2.00 GB of RAM, would be adopted as benchmark. Test matrices are all real and symmetric ones. The timing includes time cost for symbolic factorization and numeric factorization. Serial LU factorization is implemented on CPU. And partitioned matrix LU factorization in parallel is implemented on GPU with two blocks.

As is shown in Table 3, GPU results are orders of magnitude slower than CPU. There are several reasons. The outer loop of LU factorization must be serially done because each step needs values from earlier steps. This limits parallelism in the algorithm. As for the inner loop, threads are allocated to operate on each column in parallel. However, for matrices from power system, there are just around 3 non-zeros in each row, which implies only very few of the total threads are utilized. Consequently, better algorithms should be developed to utilize GPU parallel computation power.

**Table 3. Test Result of LU Factorization**

<table>
<thead>
<tr>
<th>Number of nodes</th>
<th>389</th>
<th>1152</th>
<th>1138</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of fill-ins</td>
<td>4389</td>
<td>51825</td>
<td>37174</td>
</tr>
<tr>
<td>CPU timing (ms)</td>
<td>11.21</td>
<td>254.12</td>
<td>123.12</td>
</tr>
<tr>
<td>GPU timing (ms)</td>
<td>48.23</td>
<td>714.53</td>
<td>361.52</td>
</tr>
</tbody>
</table>

**3. Cholesky Factorization on CUDA**

**3.1 Introduction to Cholesky Factorization**

In linear algebra, Cholesky factorization or Cholesky triangle is decomposition of a symmetric, positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose. Cholesky factorization is usually faster than LU factorization. Statement of Cholesky factorization is as follows: If matrix $A$ has real entries and is symmetric (more generally is Hermitian, which would not be considered in this research) and positive definite, then it could be composed as:

$$A = LL^T$$  \hspace{1cm} (9)

$L$ is a lower triangular matrix with strictly positive diagonal entries, and $L^*$ denotes the conjugate transpose of $L$. For its solution, one approach starts from upper left corner of the matrix and calculates row by row; another one also starts from upper left corner but calculates column by column. The latter one is used throughout the implementation.

The column Cholesky factorization proceeds from the first column to the last. And each column being updated receives updates from columns to its left. For the $ith$ column, it could be calculated as follows:

$$
\begin{align*}
\begin{bmatrix}
    t_1 \\
    \vdots \\
    t_i \\
\end{bmatrix} &= \begin{bmatrix}
    a_{ii} \\
    \vdots \\
    a_{in} \\
\end{bmatrix} - \sum_{k<i} \begin{bmatrix}
    l_{ik} \\
    \vdots \\
    l_{in} \\
\end{bmatrix} \\
\begin{bmatrix}
    l_{ii} \\
    \vdots \\
    l_{in} \\
\end{bmatrix} &= \frac{1}{t_i} \begin{bmatrix}
    t_1 \\
    \vdots \\
    t_i \\
\end{bmatrix}
\end{align*}
$$

(10)

The formulation applies to both dense and sparse matrices. As could be noted, $i_{th}$ column receives updates only from effective columns whose first element is nonzero. For sparse matrices, special storage methods need to be designed to find effective updating columns.

**3.2 Symbolic Factorization with Graph Theory**

Graph is often used to represent matrices. By modeling matrices with graphs, a lot of useful characteristics of the graph could be explicitly revealed and more easily utilized. Graph structure of sparse matrix is first established by the adjacency matrix representation, where each row/column is represented by a node. And a linked list is set up which documents nodes connected to it in an increasing order of node number.

One most useful tool in a graph theory approach to matrix factorization is elimination tree, which helps determine structural information of the factorization process. Liu gives a detailed elaboration on it in [12]. In this study, elimination tree is used to determine the row and column structure of matrix in symbolic factorization.

**3.3 Supernode and Numeric Factorization**
A supernodal approach is implemented in the research. By definition, a supernode is a contiguous set of columns whose structure in $L$ consists of a dense triangular block on the diagonal, and an identical set of non-zeros for each column below this dense block. An illustration of supernode is given in Fig. 2.

**Fig. 2. Example of Supernode.**

In Fig. 2, the set {1,2,3,4} of columns forms a supernode. Supernode naturally occurs as the factorization proceeds [13]. Although the column of original matrices might appear quite dissimilar in structure, many of the columns of the Cholesky factor appear identical. This coalescing of column structures occur when factoring any matrix because of the fill-ins produced. One example of factorized matrix from power system is shown in Fig. 3.

Supernodes could be treated as dense matrix for storage and computation, through which efficiency could be greatly improved. As is elaborated in the last section, directly implementing sparse matrix calculation on CUDA would not accelerate the process because CUDA is best suited for large scale and regular data formats, while sparse structure is highly irregular. By forming and executing on supernodes, dense data formats are extracted from sparse matrix to better cater for CUDA architecture. For each supernode, only one copy of column structures needs to be stored.

Supernode is formed by a post-order traversal of the original graph. As has been elaborated in the last section, by storing the same column structure, supernodes could be treated as dense data formats. Referring to (10), it could be seen that updating columns within the same supernode could be first calculated by a dense matrix*vector operation.

**Fig. 4. Illustration of Numeric Factorization.**

In Fig. 4, the process of numeric factorization is illustrated. As could be seen, the red column is the column being updated. The black dots to the left of the red diagonal dot are the row structure, which is predetermined in symbolic factorization. The red dots under the diagonal one are the column structure. There is a supernode which is in the yellow box. Recall (10), the column being updated receives updates from nodes which form its row structure. The supernode is multiplied by the vector in the blue circle which is a dense vector. This computation is done on CUDA.

Two vectors are needed to store value and indexing information for each column. For a supernode, since the column structure for each of its column is the same, only one set of indexing vector is needed. During the factorization process, a vector marker [*] is defined to mark the position of the element being searched for each column so that no repeated search would be done for the next round.

This supernodal approach can be further strengthened by combining it with the partitioned matrix factorization method elaborated in Section 2.3. Since the diagonal blocks and the corresponding off-diagonal blocks are independent of each other and can hence be factorized simultaneously on thread blocks of GPU. Within each block, CUBLAS functions are called to deal with supernodes whenever
necessary.

3.4 Block Supernode Approach

In previous sections, a supernode-column approach is implemented, i.e. the column being updated receives updates from supernodes to its left. An implementation which brings in higher level parallelism is called block supernode approach, by which one supernode as a whole receives updates from elements to its left. This supernode-supernode manner escalates the computation from BLAS Level-2 to BLAS Level-3. Multiplication between dense matrices is a problem that is typically considered to be well suited for CUDA platform.

Dense matrices multiplication based on block supernode approach is illustrated in Fig. 5.

Fig. 5. Supernode-Supernode Update.

There are three supernodes shown in Fig. 5. Supernode on the right hand side is being updated. Contribution of the other two supernodes could be calculated together instead of for each column. Take the supernode on the left hand side for example. Overall contribution can be calculated by multiplying dense matrix extracted from the supernode by the transpose of matrix A. The product is then scattered into the supernode being updated.

3.5 Test Results

With the same condition as in Section 2, the numeric factorization was done on CPU and GPU respectively to compare the timing difference. The result is shown in Table 4.

As could be seen, with small dimensions, GPU results are slower than CPU ones. This is because with low dimensions, size of supernode is small and the advantage of CUBLAS is counteracted by the frequent visits to GPU memory. When the matrix size is big enough, GPU is faster than CPU for numeric factorization.

4. Conclusion

It has been shown that GPU is capable of speeding up sparse matrix factorization. GPU architecture works best for large scale and dense data formats with neat structures. A supernode approach to Cholesky factorization makes such data formats available for parallel computation and thus brings in speedup. Since a larger supernode size implies larger amount of data, more efficiency for parallel factorization would be achieved. Thus, in future research, reordering scheme should be designed, which aims at an optimization between fill-in reduction and supernode size increase. Besides, reordering scheme should be designed which produces better scattering of supernodes. One consideration is to move naturally occurring supernodes from the lower right corner to the left part. Another consideration is to reduce tree height and increase number of branches, so that computation load is balanced among different thread blocks. With the rapid increase in the processing power of modern GPU, its use in accelerating computation intensive power system analysis would be more extensive and important in future intelligent power systems with self-healing capability.

Acknowledgements

This work was supported by the Hong Kong Polytechnic University (Project A-PA9K). Mr. B. Zhou and Mr. Y.T. Zhang would also like to acknowledge the Hong Kong Polytechnic University Research Studentship for their PhD Study.

References

[1] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E.


Ye Tian received BSc degree in Electrical Engineering and its Automation from Tianjin University (PRC) and MSc degree from Hong Kong Polytechnic University. His research interests are GPU computing and image processing.

Yutong Zhang received the BSc degree in Electrical Engineering, from NCEPU University, Beijing, China, in 2005, the MSc degree in Electrical Engineering from the University of Bath (UK), and the PhD degree in Electrical Engineering, from the Hong Kong Polytechnic University. His main research interest is power system stability analysis, assessment and control with large wind power penetration.

Bin Zhou received the BEng degree in Electrical Engineering in 2006 from Zhengzhou University, Zhengzhou, China, and the MEng degree in Electrical Engineering in 2009 from South China University of Technology, Guangzhou, China. He is currently pursuing the PhD degree at the Department of Electrical Engineering in The Hong Kong polytechnic University, Hong Kong SAR. His main research interests include power system control, automation, and optimization.

Ka Wing Chan received his BSc (Hons) and PhD Degrees in Electronic and Electrical Engineering from the University of Bath (UK). He currently is an Assistant Professor in the Department of Electrical Engineering of the Hong Kong Polytechnic University, Hong Kong SAR. His main interest is power system stability analysis and control, power system simulation and security assessment.