Parallel Huge Matrix Multiplication on a Cluster with GPGPU Accelerators

Seungyo Ryu and Dongseung Kim
Department of Electrical and Computer Engineering
Korea University
Seoul, Republic of Korea
dkimku@korea.ac.kr

***Abstract*— We design a parallel *huge matrix* multiplication algorithm on a cluster of GPU nodes. Since input matrices are too big to accommodate in the memory, the algorithm repeats the loading, computing, storing partial matrix data from/to disk and GPU buffer. The key to achieve the best speedup is not only to use GPU with full performance, but to reduce the overhead in data movement between disk and GPU buffer. We devise an efficient way to lower the latency of supplying the matching pair of the partial matrices to the GPU buffer, and to optimize the data partition, distribution, and disk access using the pipelined way. Experimental results show our algorithm outperforms a generic algorithm, resulting in the computing time reduction by 45%. Also, the scalability of the algorithm enhances with more GPU nodes.**

Keywords*— *matrix multiplication, parallel computing, GPU computing, MPI

I. Introduction

Matrix multiplication belongs to one of the most important computing problems in science and engineering. Recently a number of applications in big data and complex knowledge processing demand great computing power often involved with matrix multiplication. Matrix multiplication has inherent parallelism among the computation of individual elements of resulting matrix C, and the required operations are same, which matches the computing with single-instruction-stream multiple-data-stream (SIMD) operations. Researchers use general purpose GPU (GP-GPU) to exploit such parallelism with low cost to shorten the computing time. Yet, they all deal with *regular matrices* whose size does not exceed the memory or buffer size. Now, we lift the size barrier of input matrices to include *huge matrices*, and develop a fast multiplication method called *external matrix multiplication* on a parallel computing cluster equipped with GPU accelerators.

Since the whole matrices cannot reside in the main memory or GPU buffer memory, they should be divided into submatrices so that at least a pair of them can be loaded in buffer of GPU board for the multiplication. Practically no computation starts until the submatrices are loaded completely in the buffer. It is time consuming process since they are loaded to main memory of CPU, then copied to the buffer of GPGPU. The delay in loading/unloading data is comparable to pure computing time, thus the effect of parallel computation may not be great in the reduction of the execution time. As an example, our test run shows less than a 10-fold speedup using a generic multiplication algorithm with no optimizing effort on a GPGPU (5722 seconds with NVIDIA GeForce GTX 1070, 1920 cores with 1.7GHz) over a single CPU (54142 seconds with Intel i7-7700, quad core, 4.2 GHz clock), although the GPGPU consists of almost two thousand computing cores.

There are many research articles in the literature for efficient matrix multiplication using GPGPU. However, we have not seen the results where the matrix size exceeds the memory or buffer capacity. In early days, Volkov and Demmel [8] implemented GPU matrix multiplication using block matrix multiplication scheme to fully utilize the nature of GPU memory model. Wu and Jaja [9] implemented block matrix multiplication using pipelined GPU computation. Sun and Tong [7] pursued the maximal parallelism in the matrix multiplication with various optimization techniques within the GPU. Li, Ranka, and Sahni [4] implemented Strassen's algorithm on GPU using CUDA for the first time. Lai, Arafat, Elango, and Sadayappan [3] solved the problem with Strassen-Winograd algorithm using CUDA, that works with dynamic peeling. On the other hand, there are recent studies using a new distributed computing environment . Gu et al. [2] tried to reduce the I/O time under Mapreduce environment. DeFlumere and Lastovetsky [1] proposed the matrix partitioning method to minimize computation and communication time for the heterogenous processors.

关于使用GPU高效进行矩阵乘法有许多相关的研究文章

To solve the *huge matrix problem*, we devise a pipelined way of data movement in disk I/O and data transfer between CPU (host) memory and GPU (device) buffer, thus, the computing is

978-1-5386-5555-9/18/$31.00 ©2018 IEEE
DOI 10.1109/IPDPSW.2018.00139

overlapped with data transfer, and an efficient data distribution scheme is developed to reduce the overall communication overhead in supplying matrix data to cluster nodes.

The external matrix multiplication algorithm on a cluster of GPU hardware is described in section II, and an improved algorithm is developed in section III. Experimental results are reported in section IV, and section V concludes the paper.

II. Matrix multiplication using GPU

A. External matrix multiplicaton on GPU

External GPU Matrix Multiplication (EGMM) on a single node uses partial matrices that can fit in buffer space of GPU and main memory of CPU. We are going to divide input matrices A and B into multiple blocks. Let $m \times k$ and $k \times n$ be the sizes of A and B, and they will be partitioned into equal-sized submatrices (blocks) of size $x \times y$ and $y \times z$, respectively. Since the submatrices are smaller than the original matrices, $x \leq m, y \leq k, z \leq n$ should be satisfied. Now, A and B can be written respectively as $(m/x) \times (k/y)$ and $(k/y) \times (n/z)$ submatrix blocks as below. For conveniences, we introduce new parameters $m_b = m/x,\ k_b = k/y,$ and $n_b = n/z$. Refer to the symbols and their meaning in Table I.

$$A = \begin{bmatrix} A_{1,1} & \cdots & A_{1,k_b} \\ \vdots & \ddots & \vdots \\ A_{m_b,1} & \cdots & A_{m_b,k_b} \end{bmatrix}, B = \begin{bmatrix} B_{1,1} & \cdots & B_{1,n_b} \\ \vdots & \ddots & \vdots \\ B_{k_b,1} & \cdots & B_{k_b,n_b} \end{bmatrix},$$

Now, the target matrix $C = A \times B$ is will be $m_b \times n_b$ submatrices, computed with the following relationship:

$$C = \begin{bmatrix} A_{1,1} & \cdots & A_{1,k_b} \\ \vdots & \ddots & \vdots \\ A_{m_b,1} & \cdots & A_{m_b,k_b} \end{bmatrix} \times \begin{bmatrix} B_{1,1} & \cdots & B_{1,n_b} \\ \vdots & \ddots & \vdots \\ B_{k_b,1} & \cdots & B_{k_b,n_b} \end{bmatrix}$$

$$= \begin{bmatrix} \sum_{h=1}^{k_b} A_{1,h} * B_{h,1} & \cdots & \sum_{h=1}^{k_b} A_{1,h} * B_{h,n_b} \\ \vdots & \ddots & \vdots \\ \sum_{h=1}^{k_b} A_{m_b,h} * B_{h,1} & \cdots & \sum_{h=1}^{k_b} A_{m_b,h} * B_{h,n_b} \end{bmatrix}$$

$$= \begin{bmatrix} C_{1,1} & \cdots & C_{1,n_b} \\ \vdots & \ddots & \vdots \\ C_{m_b,1} & \cdots & C_{m_b,n_b} \end{bmatrix}$$

where $C_{i,j} = \sum_{h=1}^{k_b} A_{i,h} * B_{h,j}\ ,\ \ i = 1, \cdots, m_b,\ j = 1, \cdots, n_b$ (1)

Thus, each submatrix $C_{i,j}$ is computed by the cumulative sum of $A_{i,h} * B_{h,j}$, having the size of $x \times z$.

TABLE I. Symbols

Symbol	Size	Meaning
A B C	$m \times k$ $k \times n$ $m \times n$	Input matrix Input matrix Output matrix.
	x, y, z p	$x = m/m_b,\ y = k/k_b,\ z = n/n_b$ The number of computing nodes
$A_{i,j}$	$x \times y$	A is divided into $m_b \times k_b$ submatrices, and each is written as $A_{i,j}$ where $i \in \{1, \cdots, m_b\}$ and $j \in \{1, \cdots, k_b\}$.
$B_{i,j}$	$y \times z$	Similarly, $B_{i,j}$ is made by B partitioned to $k_b \times n_b$ submatrices, $i \in \{1, \cdots, k_b\}, j \in \{1, \cdots, n_b\}$.
$C_{i,j}$	$x \times z$	C is divided into $m_b \times n_b$ submatrices, and each submatrix is written as $C_{i,j}$where $i \in \{1, \cdots, m_b\}$ and $j \in \{1, \cdots, n_b\}$.
$A[i]$	$\frac{m}{p} \times k$	i-th row partitioned matrix of A.
$B[j]$	$k \times \frac{n}{p}$	j-th column partitioned matrix of B. $i, j \in \{1, \cdots, p\}$.
$C[i]$	$\frac{m}{p} \times n$	i-th row partitioned matrix of C where $i \in \{1, \cdots, p\}$.
$C[i][j]$	$\frac{m}{p} \times \frac{n}{p}$	$C[i][j]$ is one block of C when C is partitioned into $p \times p$ blocks, where $i, j \in \{1, \cdots, p\}$.

Now, $C_{i,j}$ is found by computing the partial products iteratively as follows. C_{temp} , initially a zero matrix, is used to store the partial sum. $A_{i,h}$ and $B_{h,j}$ $(h = 1, 2, \cdots, k_b)$ are copied to GPU buffer, then the product $A_{i,h} * B_{h,j}$ is calculated and accumulated to C_{temp}. It is repeated k_b times, then C_{temp} becomes $C_{i,j}$. Finally, it is copied to disk from GPU buffer.

```
Algorithm: EGMM($C, A, B, x, y, z$)
/*External GPU Matrix Multiplication on a single node */
Input: $A, B, x, y, z$.
Output: $C$.
for ( $i = 1; i \leq m_b; i++$)
  for ( $j = 1; j \leq n_b; j++$)
    In GPU, set $C_{temp}$ =0.
    for ($h = 1; h \leq k_b; h++$ )
      Read $A_{i,h}$ and $B_{j,h}$ from disk & move to GPU.
      //GPU matrix multiplication
      $C_{temp} = C_{temp} + A_{i,h} * B_{h,j}$
    Write $C_{temp}$ to disk as $C_{i,j}$.
```

Fig. 1. The sequential algorithm of external GPU matrix multiplication.

B. Parallel external GPU matrix multiplication

To work with multiple GPU nodes, data partitioning, distribution and computing should be well organized and load is properly balanced.

The first parallel algorithm shown in Fig. 2, called *naïve algorithm*, assigns to each node a task to find a set of n/p columns of C matrix. Suppose node 1 is a server (called *root*) and it has the whole matrices of A and B. Each node will be given m/p rows of A and the complete matrix of B by broadcast. Node id is supposed to compute $m/p \times n$ $C[id]$ using $A[id]$ and B. The algorithm is simple in that each node is given equal sized partition of A and the whole matrix of B, which requires no more data further after the distribution.

As workload is evenly distributed to p nodes, computation time of each nodes becomes $1/p$ of the computation time with a single node. However, there occurs some overhead T_{dist} of scattering matrix A and broadcasting the entire matrix B. T_{dist} is estimated as

$$\begin{aligned} T_{dist} &= T_{dr}(A) + T_{scatter}(A) + T_{dw}(A[id]) \\ &\quad + T_{dr}(B) + T_{bcast}(B) + T_{dw}(B) \qquad (2) \end{aligned}$$

In the above equation $T_{dr}(\,)$and $T_{dw}(\,)$ represent the disk read and write times while $T_{scatter}(\,)$ and $T_{bcast}(\,)$ imply the communication times of scatter and broadcast through the network, respectively.

III. Optimizing Parallel Matrix Multiplication

This section describes the pipelined data movement and data partitioning & sharing scheme to improve the performance of the external GPU matrix multiplication.

```
Algorithm: PEGMM($C, A, B, x, y, z, p, id$)
/* Parallel External GPU Matrix Multiplication */
Input: $A, B, x, y, z, p, id$- node id
Output:  a partition of $C$ ($1/p$ of $C$).
  1. /* data distribution step*/
  2. Root scatters $A$ by $m/p$ rows & write as $A[id]$ to disk.
  3. Root broadcasts $B$ & write to disk.
  4. /*product computation step*/
  5. EGMM($C[id]$ , $A[id]$, $B, x, y, z$);
```

Fig. 2. Parallel external GPU matrix multiplication

A. Pipeline I/O scheduler

Parallel external matrix multiplication needs lengthy I/O operations. Matrix data are supplied after reading the disk and distributed to nodes via the network, and data in host (CPU) are copied back and forth in device (GPU) buffer in the computation step. To reduce the waiting time during the disk/network I/O operations, pipelined scheduling is employed as described now.

Partitioned matrix data in the root node are to be transferred to the disk of individual nodes. *Data distribution step* consists of two stages as shown in Fig. 3. Each block of the matrix is loaded and sent one by one. Once a block is read from disk at DR stage, it is sent to its destination through the network in NS stage. In the meanwhile, receiving nodes accept the data from the root at NR, then write to their own disk at DW stage.

Now, the pipelining in *product computation step* is shown in Fig. 4. Both $A_{i,h}$ and $B_{h,j}$ blocks for given i, j meet in the GPU buffer and their product is found. This job iterates sequentially according to the order of the index $h = 1,2, \ldots, k_b$. At DR stage, both $A_{i,h}$ and $B_{h,j}$ blocks are loaded to main memory of the host from the disk. Now, they are copied to GPU memory at M2G stage, then, $C_{temp} = C_{temp} + A_{i,h} * B_{h,j}$ is computed (MM stage). Finally, computed $C_{i,j}$ are written to disk.

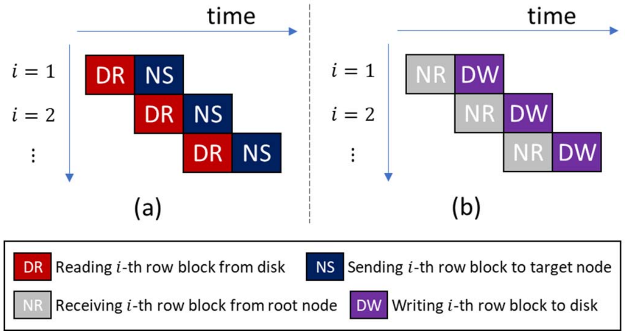

time
$i = 1$
$i = 2$
⋮
DR NS
DR NS
DR NS
(a)
time
$i = 1$
$i = 2$
⋮
NR DW
NR DW
NR DW
(b)
DR Reading i-th row block from disk
NS Sending i-th row block to target node
NR Receiving i-th row block from root node
DW Writing i-th row block to disk

Fig. 3. Data distribution scheduling: (a) pipeline of the root node, (b) pipeline of non-root nodes.

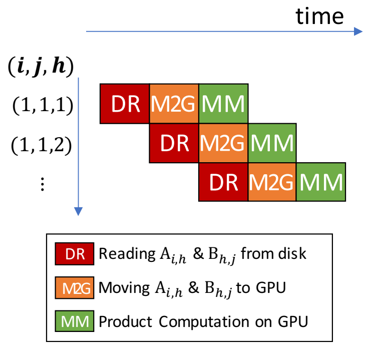

time
$(\boldsymbol{i}, \boldsymbol{j}, \boldsymbol{h})$
(1,1,1)
(1,1,2)
⋮
DR M2G MM
DR M2G MM
DR M2G MM
DR Reading $A_{i,h}$ & $B_{h,j}$ from disk
M2G Moving $A_{i,h}$ & $B_{h,j}$ to GPU
MM Product Computation on GPU

Fig. 4. The pipeline of product computation.

B. Optimizing data distribution

To avoid idling of GPUs before the data arrival in our design, minimum/partial matrix data are distributed first, then the rest data are supplied while the arithmetic computation is in progress. For parallel execution A and B matrices are divided and distributed to each node from the root. Individual nodes receive equal-sized submatrices of A and B, named $A[id]$ and $B[id]$. They are m/p rows of A and n/p contiguous columns of B, respectively. Each node id is assigned to find $m/p \times n\ C[id]$, similar to naïve algorithm. Fig. 5 illustrates this new data partitioning (called Mod_PEGMM) compared to the previous algorithm(PEGMM). A detailed procedure is given below (see also Fig. 6).

Once $B[id]$ is accepted, it is divided and stored into blocks of size $y \times z$ as separate files in the disk. Thus, block data can be accessed fast compared to locating the corresponding block in the matrix B (which is the case of PEGMM). Now, All nodes independently compute the product $C[id][id] = A[id] * B[id]$.

Since the $C[id][id]$ is only $1/p$ of the submatrices to be found by individual nodes, the product computation should iterate with the data exchanges of $B[id]$s, keeping the row partitioned matrix $A[id]$. The submatrices of $B[id]$s are exchanged along with block product computation. Hence, each node id iteratively passes a submatrix of $B[id]$ (i.e. $B[id]_{h,j}$) to its next node r, and node id also receives a block of $B[s]_{h,j}$ from its previous neighbor s, where the source and destination nodes (s, r) are determined as described in Fig. 6. When such $p\ C[id][id]$s are found, the task of getting $C[id]$ is complete, then, it is collected by the root to build the solution C, if needed.

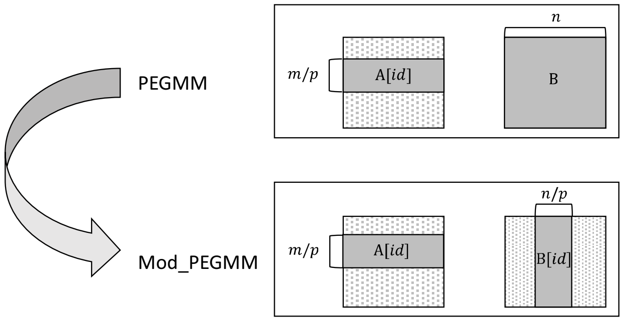


Fig. 5. Two ways of data partitioning for parallel matrix multiplication.

This block partitioning and computing scheme contributes to shorten the execution time. It reduces the data I/O size from the disk resulting in low page fault rate and enabling fast file read, compared to using the whole B matrix in PEGMM. The estimated overhead of data distribution in Mod_PEGMM can be expressed as eq. (3).

$$T_{dist}^{Mod\_PEGMM} = T_{dr}(A) + T_{scatter}(A) + T_{dw}(A[id]) + T_{dr}(B) + T_{scatter}(B) + T_{dw}(B[id]) \quad (3)$$

Since $T_{bcast}(B)$ and $T_{dw}(B)$ in eq. (2) are replaced by $T_{scatter}(B)$ and $T_{dw}(B[id])$, Mod_PEGMM has the lower overhead in distributing matrix B.

```
Algorithm: Mod_PEGMM($C, A, B, x, y, z, p, id$)
/*Modified Parallel External GPU Matrix Multiplication*/
Input: $A, B, x, y, z, p$, $id$ – node id.
Output: a partition of $C$ ($1/p$ of $C$).
/* data distribution step*/
Root scatters $A$ by $m/p$ rows & write as $A[id]$ to disk.
Root scatters $B$ by $n/p$ columns & write as $B[id]$ to disk.
/* product computation step */
for ($q = 1; q \le p; q++$)
  if ($q = 1$)
    /* initial product computation with no data exchange */
    EGMM($C[id][id], A[id], B[id], x, y, z$)
  else
    Determine sender ($s$) and receiver ($r$) ids
    /* $s = (id + p + q)\%p, r = (id + p - q)\%p$ */
    for ($i = 1; i \le m_b/p; i++$)
      for ($j = 1; j \le n_b/p; j++$)
        In GPU, set $C_{temp} = 0$.
        for ($h = 1; h \le k_b; h++$)
          Exchange $B[id]_{h,j}$ & $B[s]_{h,j}$.
          Load $A[id]_{i,h}$ and $B[s]_{h,j}$ to GPU.
          //GPU matrix multiplication
          $C_{temp} = C_{temp} + A[id]_{i,h} * B[s]_{h,j}$
        Write $C_{temp}$ as $C[id][s]_{i,j}$ to disk.
```

Fig. 6. Optimized external matrix multiplication algorithm

IV. Experiment and Discussion

The algorithm is implemented on a Beowulf computer cluster which consists of 8 computing nodes connected via 10G ethernet switch as listed in Table II. Each node is equipped with a quad-core CPU (Intel i7-7700, 4.2GHz) and a GPU accelerator (NVIDIA GeForce GTX 1070). There are 1920 cores in a GPU board and each 128 cores consist a SM (streaming multiprocessor). GPU board has own 8GB on-board buffer and 64KB L1 cache is provided in each SM. Each node/host has 16GB main memory and a 10G ethernet NIC interface. MPICH-3.2 is used for message passing of computer cluster, and OpenMP 3.1 is used for collaboration between pipeline stage. GPU operates under CUDA 8.0 environment.

The matrix multiplication code on GPU was adopted from the work of Volkov and Demmel[8]. Input data A and B are single precision floating point numbers, which are randomly generated in the interval $[0, 1]$. Various sized matrices are used as listed in Table III. Matrix size in Set 1 having 8GB size is chosen such that $m = k = n$, and the whole matrices A, B, C are as small as possible but big enough to be *huge* matrices (bigger than GPU memory capacity). Other sets are generated by doubling $m, n, or\ k$ properly to become bigger matrices. In the experiment, the parameters x, y and z that determine the shape of the block are chosen empirically with the following relationship:

$$x = y = z = \begin{cases} \mathbf{min}(m/p, k, n, 2^{13}), & \text{for PEGMM} \\ \mathbf{min}(m/p, k, n/p, 2^{13}), & \text{for Mod\_PEGMM} \end{cases}$$

TABLE II. COMPUTER CLUSTER CONFIGURATION

Cluster configuration	
No. of nodes	8
Interconnect	10G Ethernet switch
Node configuration	
CPU	Intel i7-7700 (4.2GHz, 4 cores)
RAM	DDR4 16GB
NIC	10Gbps Ethernet NIC
HDD	1TB, SATA3 bus, 7200RPM, 64MB buffer
GPU	NVIDIA GeForce 1070 - Max clock freq. 1.7GHz, - 15 streaming multiprocessors (SM) - 128 CUDA cores per 1 SM (total 1920 cores) - 64KB L1 cache/ SM, 2MB shared L2 cache - GDDR5 8GB memory - PCIe 3.0 ×16 bus

TABLE III. EXPERIMENTAL DATA SET

	$\boldsymbol{A(m \times k)},\ \ \boldsymbol{B(k \times n)}$
Set 1	$2^{15} \times 2^{15},\ 2^{15} \times 2^{15}$
Set 2	$2^{16} \times 2^{15},\ 2^{15} \times 2^{15}$
Set 3	$2^{16} \times 2^{15},\ 2^{15} \times 2^{16}$
Set 4	$2^{16} \times 2^{16},\ 2^{16} \times 2^{16}$
Set 5	$2^{17} \times 2^{16},\ 2^{16} \times 2^{16}$

Experimental results & discussion

Fig. 7 plots the measured data of generic algorithm (PEGMM) and modified algorithms (Mod_PEGMM). The bar graph shows the total execution time including data distribution and computing, whereas the lines plot the parallel speedup with respect to the execution time of $p = 2$. (We omit the measuring the sequential running time of $p = 1$ since it does not contain data distribution step.) Fig. 7 (a) displays the result with the small sized data: Set 1 through Set 3. Mod_PEGMM shows the shorter execution time (by 45% at maximum) than PEGMM. Meanwhile, the parallel speedups of both algorithms are not big enough with respect to the number of computing nodes. It is due to the inclusion of extra data distribution step and the insufficient size of data for efficient parallel computation. Fig. 7 (b), showing the result with big data Set 4 and Set 5, contrasts to the previous experiment of Mod_PEGMM. Due to the division of B, the amount of disk I/O for accessing B shrinks as the number of computing nodes increases. In addition, the relative portion of pure computation over the total execution time increases as matrix grows, thus the scalability improves with more participating nodes in the job. The best speedup of 96.9, with respect to a single node CPU without GPU, is obtained with 8 nodes with GPU accelerator using data Set 4 (32GB data for matrix A and B).

V. Conclusion

A fast algorithm is developed for huge matrix multiplication on computer cluster with GPGPU accelerator. It runs in a pipelined fashion for overlapping I/O operations with computation to reduce the waiting time in data transfer from/to disk, over the network, and loading on the GPU memory. Also, the algorithm includes an efficient matrix distribution scheme to minimize I/O workloads. Compared to the generic algorithm, the proposed method shortens the execution time by 45% and increases the efficiency of parallel computing up to 40%.

If faster data storage like SSD is employed, quicker access to data is expected than the conventional HDD, however, we may need to design a different pipeline scheduling, and also a new partitioning scheme of matrices to further improve the performance.

REFERENCES

[1] A. DeFlumere and A. Lastovetsky, "Searching for the Optimal Data Partitioning Shape for Parallel Matrix Matrix Multiplication on 3 Heterogeneous Processors," Proc. 28th Int'l Symp. Workshops Parallel & Distributed Processing Symposium (IPDPS 14), pp. 17-28.

[2] R. Gu, et al., "Improving Execution Concurrency of Large-scale Matrix Multiplication on Distributed Data-parallel Platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol 28, no. 9. 2017.

[3] P. Lai, H. Arafat, V. Elango, and P. Sadayappan, "Accelerating Strassen-Winograd's matrix multiplication algorithm on GPUs," Proc. 20th Int'l Conf. High Performance Computing (HiPC 13), 2013.

[4] J. Li, S. Ranka, and S. Sahni, "Strassen's matrix multiplication on GPUs," Proc. 17th Int'l Conf. Parallel and Distributed Systems (ICPADS 11), 2011.

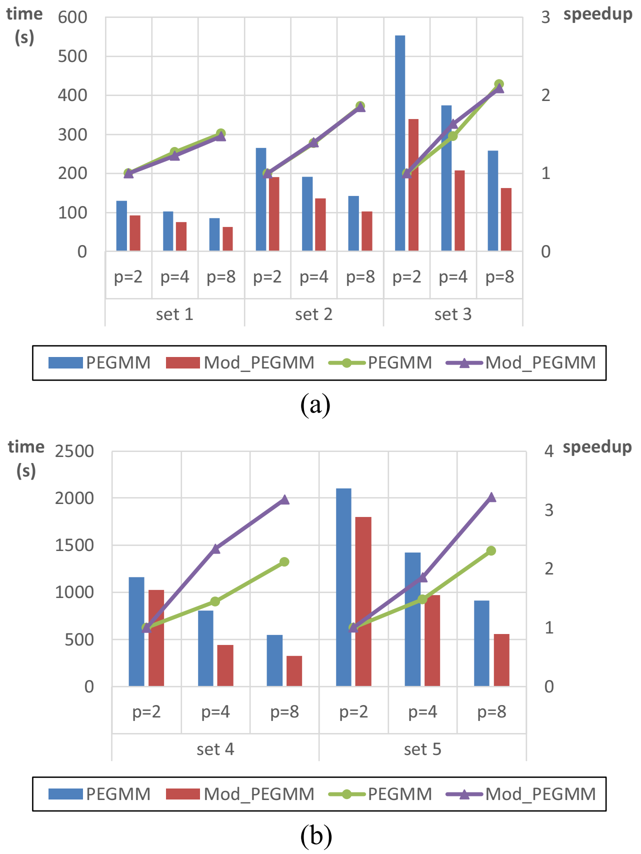
[5] K. Park, "Experimental study of energy efficient computing for sorting and matrix multiplication," master's thesis, Dept. Electrical and Computer Eng., Korea University, Seoul, South Korea, 2013.

[6] D. Rohr, and V. Lindenstruth, "A flexible and portable large-scale DGEMM library for linpack on next-generation multi-GPU systems," Proc. 23rd Euromicro Int'l Conf. Parallel, Distributed and Network-Based Processing (PDP 15), 2015.

[7] Y. Sun, and Y. Tong, "Cuda based fast implementation of very large matrix computation," Proc. 11th Int'l Conf. Parallel and Distributed Computing, Applications and Technologies (PDCAT 10), 2010.

[8] V. Volkov, and J. W. Demmel. "Benchmarking GPUs to tune dense linear algebra," Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC 08), 2008.

[9] J. Wu, and J. Jaja, "Achieving Native GPU Performance for Out-of-Card Large Dense Matrix Multiplication," Parallel Processing Letters, vol. 26, no. 02, 2016.



(a)

(b)

Fig. 7. Comparison of algorithms (a) with data Sets 1, 2, and 3 and (b) with data Set 4 & 5.