Gauss-Jordan Matrix Inversion Speed-Up using GPUs with the Consideration of Power Consumption

Mahmoud Shirazi
School of Computer Science
Institute for Research in Fundamental Sciences (IPM)
Tehran, Iran
Email: m.shirazi@ipm.ir

Mehdi Kargahi
School of Computer Science
Institute for Research in Fundamental Sciences (IPM)
Tehran, Iran
School of ECE, Faculty of Engineering
University of Tehran, Tehran, Iran
Email: kargahi@ut.ac.ir

Farshad Khunjush
School of Computer Science
Institute for Research in Fundamental Sciences (IPM)
Tehran, Iran
School of Electrical and Computer Engineering
Shiraz University, Shiraz, Iran
Email: khunjush@shirazu.ac.ir

***Abstract*—Matrix inversion is an important requirement for many scientific and practical applications. Parallel architectures like Graphics Processing Units (GPUs) can have noticeable impacts on accelerating such computations, however, at the expense of more power consumption. Both computation speed and power become more challenging when the size of matrices gets larger. This paper proposes an accelerated version of the Gauss-Jordan matrix inversion method using GPUs. The experimental results show that the proposed method is faster than a recent baseline method. As an additional property, the configurations have been made in a manner to consume less power.**

Keywords–Matrix Inversion; GPU; Power Consumption.

I. INTRODUCTION

Matrix invesion is too important in a diverse spectrum of applications including graph theory, computer graphics, cryptography, etc. There are several known methods like Gauss-Jordan [1], Strassen [2], Strassen-Newton [3], Cholesky decomposition [4], and Lower Upper Decomposition [5] for matrix inversion.

As calculating the inverse of a matrix takes $O(n^3)$ (where n is the dimension of the matrix), the usage of many core architectures, like GPU is too effective to reduce the computation time of inversing large matrices. In fact, the highly parallel architecture of GPUs makes them more suitable than general-purpose Central Processing Units (CPUs) for algorithms with large parallelizable blocks of data.

The recursive nature of some matrix inversion algorithms such as Strassen-Newton reduces the effectiveness of parallelization [6]. In recursive algorithms, each iteration needs to wait for the result of previous iteration. In Gauss-Jordan method, each iteration calculates the new values of all elements in the matrix that cause this method to be more appropriate for parallelization. GPUs has demonstrated high computing power in several application fields and we can use GPUs to speedup the Gauss-Jordan matrix inversion method. On the other hand, GPU also produces high power consumption and has been one of the largest power consumers in desktop and supercomputer systems [7].

Girish Sharma et al. [6] proposed a fast GPU-based Parallel Gauss-Jordan method (PGJM). This algorithm calculates the inverse of matrix in two steps using Gauss-Jordan method. Their method uses n and $n \times (n-1)$ threads for steps 1 and 2, respectively. As the matrix size increases, the parallel Gauss-Jordan method will need more threads, blocks, streaming multiprocessors (SMs) and thus more processing cores. In addition, using more processing cores results more power consumption.

Comparing to the method that proposed in [6], here, we improve this method using one step for calculating the inverse of matrix. The improved parallel Gauss-Jordan method (I-PGJM) is faster than PGJM. In fact, using different grid and block dimensions it can affect the power consumption. Thus, We can make some trade-off between the speed-up and power consumption of I-PGJM.

The remainder of this paper is organized as follows. Section II presents a background of Gauss-Jordan and parallel Gauss-Jordan methods, and addresses the highlights to achieve a program that consumes less power. Section III proposes a fast parallel Gauss-Jordan method that has lower power consumption. Experimental results are presented in Section IV. Section V discusses the related work and Section VI concludes the paper.

II. BACKGROUND

This section discusses the Gauss-Jordan method [1], the parallel Gauss-Jordan method [6] and the programming-level techniques for reducing the power consumption.

A. The Gauss-Jordan Method

Gauss-Jordan is an accurate method for matrix inversion that is based on Gauss-Jordan elimination. This method for calculating the inverse of matrix A of size $n \times n$ augment the matrix with the identity matrix (I) of the same size. Thus we have a matrix $C = \{c_{ij}\}$ of size $n \times 2n$ that its left half is A and its right half is I. This method converts the left half of C to identity matrix after some row operations. At the end, the right half of C is equal to A^{-1}.

Converting the left half of C can be broken down into two steps per each column of the left half matrix. The first step converts c_{ii} to 1 by dividing all elements in ith row by c_{ii}. The second step convets all elements in ith column to 0 except

 ISBN: 978-1-61208-416-9

for the ith one. Following these two steps for each column sequentially, the left half of C becomes the unit matrix while the right half becomes the desired inverse of A.

The first step needs to $O(n)$ divisions and the second step needs to $O(n^2)$ computations. Thus, this method takes $O(n^3)$ for calculating the inverse of matrix.

B. The Parallel Gauss-Jordan Method

Girish Sharma et al. [6] proposed the parallel version of Gauss-Jordan method. They implement their method with CUDA on Nvidia GTX 260. Their method used two steps. The first one uses n threads (equal to number of rows or columns in matrix A) that each divides one element of current row ($c_{rr}, ..., c_{r(n+r)}$, where r is the current row index) by c_{ii}. Preventing division by zero exception can be done by checking the value of c_{ii}. If its value was zero then the elements of current row added to elements of kth row such that $c_{kk} \neq 0$. To avoid the extra GPU cycles, resulting from if...else condition, this summation is done prematurely for all elements. The second step uses $n \times (n-1)$ threads to convert the elements of rth columns to zero except for c_{rr}. In this step, they minimize the computations by processing only the first n columns (starting from the rth columns).

This algorithm uses n threads in one block to avoiding division by zero, n threads per one block for step 1 and n threads in n blocks for step 2 that each block handles the computations of one column at a time. Since the maximum number of threads per block in modern GPU are 1024, if the number of columns is greater than 1024, then the matrix splitts vertically.

This algorithm was programmed using CUDA C and was tested against various sizes of the several types of matrices, e.g., identity, sparse, band, random and hollow matrix. The results demonstrate that if the size of the matrix (n) is less than 100, the computation time is linear to the size of the matrix because the maximum number of active threads that can be scheduled at a time is 9216 and for $n = 100$ the number of threads that need for computations at a time is 10100. Thus values around $n = 100$ displaying the quadric nature of the algorithm due to hardware limitation.

In here, we propose a method that uses one step per each column to set c_{rr} to 1 and the remain elements in the rth column to 0. This method that was described in Section III, uses a total number of $n \times (n-1)$ threads and is faster than PGJM.

C. GPU Power-aware Programming

The power consumption of GPUs can be divided into two parts, namely, leakage power and dynamic power [8]. Dynamic power is the power that is consumed by a device when it is actively switching from one state to another. The main concern with leakage power is when the device is in its inactive state. Different components, such as SMs and memories, e.g., local, global, shared, etc. contribute to dynamic power consumption.

Techniques for reducing the GPU power consumption are classified into five categories [8]: dynamic voltage/frequency scaling (DVFS), CPU-GPU workload division-based techniques, architectural techniques, techniques that exploit workload variation to dynamically allocate resources and programming-level techniques.

For reducing the power consumption of Gauss-Jordan method for large matrices, we use programming-level techniques. Yi Yang et al. [9] identify the common code patterns that lead to inefficient use of GPU hardware and increase the power consumption. These code segments that they call it 'bug' are grouped into following categories:

- Global memory data types and access patterns
- The thread block dimensions
- Portability across different GPUs
- The use of constant and texture memory
- Floating-point number computations

Using data types either smaller than 'float' or larger than 'float4' (the data with a size of 4 floats) violates the memory coalescing requirements. The results show that GTX480 delivers much lower bandwidth when the data type is not float, float2, or float4.

In modern GPUs, threads are organized in a thread hierarchy, multiple threads forming a 1D/2D/3D thread block, and thread blocks forming a thread grid. Different size in the thread hierarchy results different performance and power consumption in different GPUs. The experimental results in [9] demonstrate that if the thread ids are used as memory access addresses, the size of the x-dimension in a thread block needs to be at least 32 for GTX480 to maximize the global memory access bandwidth and the optimal configuration to improve data reuse using shared memory is application and data size dependent.

Since different GPUs have different hardware features, the configuration of the program must be altered with different GPUs. However, different hardware features in different GPUs are not necessitating significantly different performance considerations.

Proper use of constant and/or texture memory can achieve high performance and low power consumption due to the on-chip constant and texture caches. Thus, if one method uses constant values, it is better to use constant or texture memory.

If we use a constant float, such as 4.0, the CUDA compiler will treat the constant as a double-precision number that results in less performance and higher power consumption. For fixing this bug if single-precision provides sufficient accuracy, we can use the explicit single-precision floating-point number (4.0f).

We used these hints for reducing the power consumption of I-PGJM. The details of I-PGJM power consumption consideration are shows in Subsection III-B.

III. Improved Parallel Gauss-Jordan Method

In this section, we first propose the improved method for matrix inversion and then consider the power consumption of this method.

A. I-PGJM

As mentioned in Section II, Girish Sharma et al. [6] proposed a two-step method for calculating the inverse of a matrix using Gauss-Jordan method. In this section, we present an improved method that combines these two steps into one and calculates the inverse of a matrix faster than this method.

Suppose we want to calculate the inverse of matrix A of size $n \times n$. The Gauss-Jordan method as mentioned in Section

```
Function FixAll
Input:
    Matrix: Input matrix
    colId: Current column of input matrix
Begin
    Set T to thread id in x dimension
    Set B to block id in x dimension
    Set D to Matrix [colId, colId]
    Set SharedRow[T] to Matrix[colId, T+colId]
    C ← Matrix[B, colId] / D
    Sync all threads
    IF B is equal to colId THEN
        Matrix[B, T+colId] ← Matrix[B, T+colId]/D
    ELSE
        Matrix[B, T+colId] ← Matrix[B, T+colId] – C*SharedRow[T]
End
```

Figure 1. The I-PGJM method for matrix inversion in GPU ($n \leq 1022$).

```
Function FixAll
Input:
    Matrix: Input matrix
    colId: Current column of input matrix
Begin
    T ← blockId.x * blockDim.x + threadId.x
    IF T+colId is greater than Matrix dimension THEN
        Do nothing and return
    Set TIndex to thread id in x dimension
    Set B to block id in y dimension
    Set D to Matrix [colId, colId]
    Set SharedRow[TIndex] to Matrix[colId, T+colId]
    C ← Matrix[B, colId] / D
    Sync all threads
    IF B is equal to colId THEN
        Matrix[B, T+colId] ← Matrix[B, T+colId]/D
    ELSE
        Matrix[B, T+colId] ← Matrix[B, T+colId] – C*SharedRow[TIndex]
End
```

Figure 2. The I-PGJM method for calculating the inverse of large matrices ($n > 1022$) in GPU.

II creates a matrix $C = \{c_{ij}\}$ of size $n \times 2n$ so that its left half is A and its right half is the identity matrix. The right half of matrix C After n iterations that each consists of two steps is the inverse of A.

The first step of iteration r, sets $c_{rj} = \frac{c_{rj}}{c_{rr}}$ where $j = r, ..., n+r$ (Note that there is no need to update the values for $j < r$ and $j > n + r$ [6]) and the second step sets

$$c_{ij} = c_{ij} - c_{ir} \times c_{rj} \tag{1}$$

where $i = 1, ..., n$, $i \neq r$ and $j = r, ..., n + r$. The step 1 and 2 must run sequentially as the step 2 needs the new value of c_{rj} calculated after step 1. To run these two steps in a time, we can define a coefficient $co_{ir} = \frac{c_{ir}}{c_{rr}}$ and rewrite equation 1 as:

$$c_{ij} = c_{ij} - co_{ir} \times c_{rj} \tag{2}$$

where c_{rj} is the original value of rjth element in C. Then, in iteration r we can share the elments of rth row among threads. Thus the elements of matrix C in iteration r are calculated as follows:

$$c_{ij} = \begin{cases} c_{ij} - co_{ir} \times sharedRow_j & i \neq r \\ \frac{c_{ij}}{c_{rr}} & i = r \end{cases} \tag{3}$$

where $j = r, ..., n+r$ and the values of c_{rr} and $sharedRow$ are calculated before updating the values of C using $syncthreads$ in CUDA. Using n threads each in n blocks can calculate the inverse of matrix A. Figure 1 displays the I-PGJM method implemented using CUDA C.

To avoid the division by zero and extra GPU cycles resulting from if...else condition, before the above computations the summation of each row to next row is calculated. Thus, we need $n + 2$ threads in each iteration. As each block calculates the values of one row and the maximum number of threads in each block are 1024, for $n + 2 > 1024$ we need to break down the calculation of each row into several blocks. Thus for $n > 1022$ the grid dimension is $N \times n$ where $N = \lfloor \frac{n}{1024} \rfloor + 1$ and each block has $\lceil \frac{n+2}{N} \rceil$ threads. Figure 2 shows the method for calculating the inverse of large matrices ($n > 1022$).

The total number of threads can be larger than $n+2$. Then, the 'if' condition at the begining of the method in Figure 2 is necessary to avoid accessing the out of range matrix dimension. As each block calculates at most 1024 elements of the current row in the matrix, the size of shared variables can be set to 1024.

Comparing the I-PGJM and PGJM [6], I-PGJM used $(n+2) \times sizeof(float)$ shared data rather than $(3n+2)$ $((n+1) \times sizeof(float)$ in step 1 and $(2n+1) \times sizeof(float)$ in step 2). As mentioned in [6] if the matrix size is greater than 1024 the proposed algorithm will use more blocks for computing the inverse of large matrices. Thus, for large matrices these two methods need to have more blocks.

The experimental results in Section IV show that I-PGJM is faster than the method that proposed by Sharma et al. [6].

B. Power consumption considerations

In the I-PGJM, we try to reduce the use of registers, shared memory and global memory. But increasing the matrix dimension lead to use more blocks for calculating the matrix inversion and hence more power consumption. As mentioned in Subsection II-C, using different data types and access patterns can affect the power consumption. We used different data types for matrix inversion in K20Xm and find that it has similar behaviour as GTX480. Also, a few common data access patterns exist that may cause performance degradation. This performance degradation in K20Xm (that we used in here) is negligible.

Using constant or texture memory can reduce the power consumption. In Gauss-Jordan method, the matrix elements should be updated iteratively. Then, we can not use constant or texture memory.

The grid and block size are important for performance and power consumption. We experiment with different grid and block sizes to achieve less power consumption. Achieving the suitable grid and block dimensions lead to have the fast method that can calculate the inverse of large matrices and has less power consumption.

IV. Experimental Results

The PGJM [6] and I-PGJM were programmed using CUDA C and were executed by Tesla K20Xm GPU. This

 ISBN: 978-1-61208-416-9

GPU has a shared memory size of 48KB and the maximum threads per block is 1024. The block and grid dimensions (x, y, z) are respectively restricted to $(1024, 1024, 64)$ and $(65535, 65535, 65535)$. For comparing the results to the sequential program, we used Intel Core i7 CPU (2.67GHz) with 8192 KB and 6GB of cache and memory size, respectively.

We used random matrices that have different dimensions from 2^1 to 2^{13}. the experimental results are shown in two subsections. The first one demonstrates the results of comparing these methods in term of execution time, and the second one shows the power consumption of proposed method for different grid and block dimensions to get the optimal dimensions for reducing the power consumption.

A. Matrix Inversion Execution Time

In this subsection, we campare the execution time of PGJM, I-PGJM and the sequential Gauss-Jordan method with different matrix dimensions (n). The execution time is calculated using cudaEventRecord function and is the difference in time before passing the data from host to device and after getting the results from device. For $n > 1022$, the grid and block dimensions of proposed method are (N, n) and $(\lceil \frac{n+2}{N} \rceil, 1)$, respectively and for PGJM (for step 2) are $(\frac{n}{1024}, n+2)$ and $(1024, 1)$, respectively. For sequential algorithm, the execution time for $n < 2^5$ is less than one millisecond (see Table I). Figure 3 shows the speedup of I-PGJM with respect to PGJM. The values around 100 for matrix dimension display the quadric nature due to hardware limitation [6]. This behaviour results more speedup for proposed method as matrix dimension increased ($speedup = 9.02$ for $n = 2^{13}$). Thus, I-PGJM is suitable for large matrices. For $n > 1024$ these two methods need to use more blocks, then the speedup slightly reduced for $n = 2048$.

TABLE I. THE EXECUTION TIME (MS) OF I-PGJM, PGJM AND THE SEQUENTIOAL ALGORITHM FOR DIFFERENT MATRIX DIMENSIONS.

n	*I-PGJM*	*PGJM*	*Sequential Algorithm*
2	0.245	0.255	0
2^2	0.273	0.284	0
2^3	0.310	0.353	0
2^4	0.409	0.488	0
2^5	0.575	0.784	0
2^6	0.955	1.412	3
2^7	1.844	3.536	10
2^8	4.761	9.964	120
2^9	21.570	62.314	950
2^{10}	155.691	497.304	7550
2^{11}	1410.690	4209.350	60500
2^{12}	9750.890	44424.500	482619
2^{13}	72648.400	655863.000	3860219

Figure 4 shows the speedup of I-PGJM with respect to the sequential Gauss-Jordan method. The speedup of I-PGJM with respect to the sequential program is around 53 for $n = 2^{13}$. Since the maximum number of threads per block are 1024, for $n > 1024$ the algorithm need to use more blocks and hence the performance drop occurred for $n = 2048$.

B. Power Consumption

The execution time results demonstrate that I-PGJM is more suitable for calculating the inverse of large matrices. As increasing the computation results in more power consumption, the proposed method needs to use a configuration that leads to less power consumption. As mentioned in Subsection II-C and Section III, we used 'float' data type, fewer threads and shared memory to reducing the power consumption. Also, we experiment with different grid and block dimensions to reducing the power consumption.

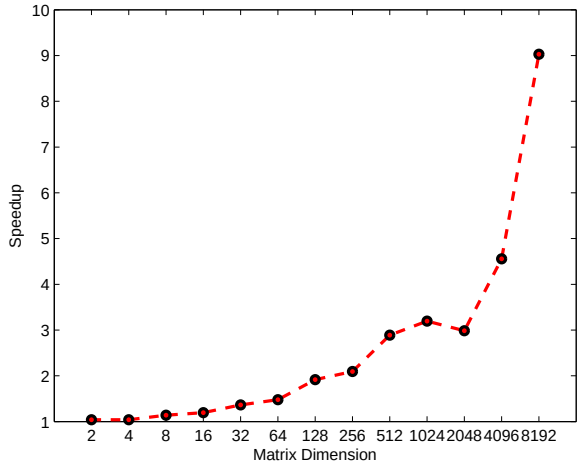

Figure 3. The speedup of the proposed method related to method in [6].

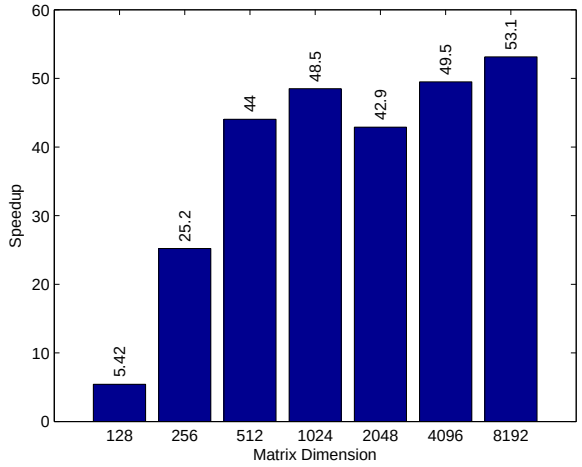

Figure 4. The speedup of I-PGJM with respect to the sequential algorithm.

We measure the run-time power of the proposed method with the NVIDIA Management Library (NVML) [10] by running the proposed method on a thread and NVML on another thread using Pthreads. NVML is a high level utility that called nvidia-smi. NVML can be used to measure power when running the kernel but since nvidia-smi is a high level utility the rate of sampling power usage is very low and unless the kernel is running for a very long time we would not notice the change in power [11]. The nvmlDeviceGetPowerUsage function in the NVML library retrieves the power usage reading for the device, in milliwatts. This is the power draw for the entire board, including GPU, memory, etc. The reading is accurate to within a range of +/- 5 watts error.

In the proposed method, all blocks in one row of a grid

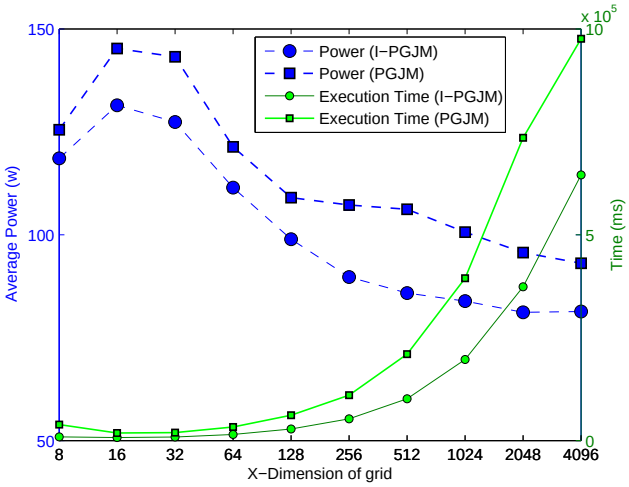

Figure 5. The execution time and power consumption of I-PGJM and PGJM for different grid dimensions. The matrix size is $2^{12} \times 2^{12}$.

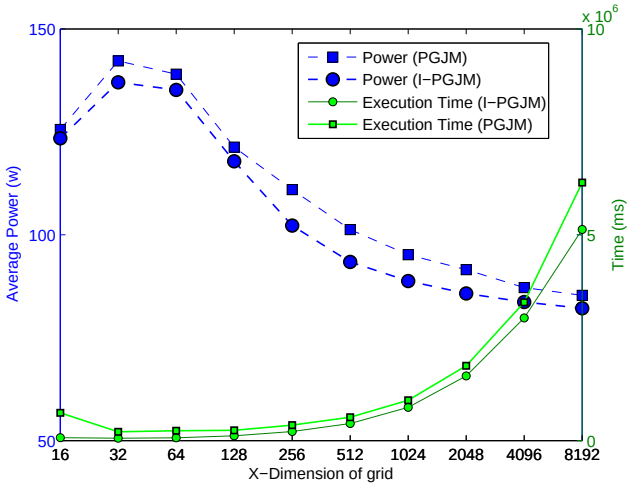

Figure 6. The execution time and power consumption of I-PGJM and PGJM for different grid dimensions. The matrix size is $2^{13} \times 2^{13}$.

compute the elements of one row in matrix and each block has $\lceil \frac{n+2}{N} \rceil$ threads related to a subset of elements in the corresponding row. For constant n, we calculate the execution time and power consumption of the proposed method with different grid and block dimensions. The x-dimension of grid can be changed from N to n. The grid sizes that used for experiments are from 2^i to 2^j where $2^i > N$ and $2^j < n + 2$ for $n > 1022$. The block dimension can get different values such that the total number of threads per block remain constant, i.e., $\lceil \frac{n+2}{N} \rceil$).

Figures 5 and 6 show the execution time and power consumption of I-PGJM and PGJM for calculating the inverse of matrix A of size $n \times n$ for different values of grid dimensions for $n = 2^{12}$ and $n = 2^{13}$, respectively. Increasing the number of blocks in x-dimension of grid results in lower threads in each block (Matrix dimension is constant). Thus, each block used less resources that lead to lower power consumption. On the other hand, the maximum number of blocks that can be scheduled in each SM is limited. Thus, increasing the number of blocks results more execution time.

Figures 7 and 8 show the experimental results for different block dimensions of I-PGJM for calculating the inverse of matrix A of size $n \times n$ for $n = 2^{12}$ and $n = 2^{13}$, respectively. As can be seen in these figures, the difference between maximum and minumum power consumption is negligible. Note that, the power usage reading of NVML is accurate within a range of +/-5 wats error. But, when the x-dimension of blocks is larger than 32, the power consumption increases. Thus, using less than 32 threads in x-dimension of blocks leads to less power consumption. Also, by simply changing the x-dimension of blocks to 32, we have better performance.

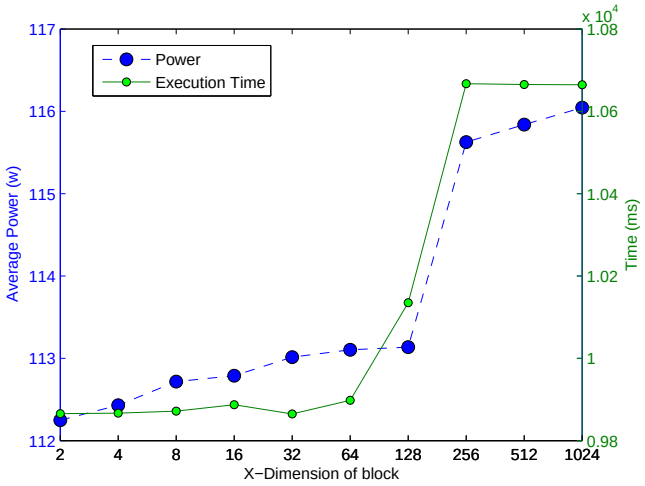

Figure 7. The execution time and power consumption of I-PGJM for different block dimensions. The matrix size is $2^{12} \times 2^{12}$.

V. Related Work

There has been much recent work decreasing the execution time of Gauss-Jordan method for matrix inversion. Some researchers focus on using multiple GPUs [12]. Although splitting the matrix between GPUs and ability to use them is important, in this study, we focus on one GPU and propose a method that has a considerable reduction of the computational time and power consumption.

Peter Benner et al. [13] have studied the inversion of large and dense matrices, on hybrid CPU-GPU platforms, with application to the solution of matrix equations arising in control theory. They have presented several matrix inversion algorithms, based on common matrix factorizations and the GJE method, similar to [12], but for solving equations in control theory.

Kaiqi Yang et al. [14] proposed a parallel matrix inversion algorithm based on Gauss-Jordan elimination with pivoting. This method divides the matrix into some sub-matrices and assign each sub-matrices to one core. Each core updates the sub-matrix using Gauss-Jordan elimination. This method experimented in at most four cores and has communication overhead. Then, this method can not used in many core platforms such as GPU.

The most related work to this paper is the method that is

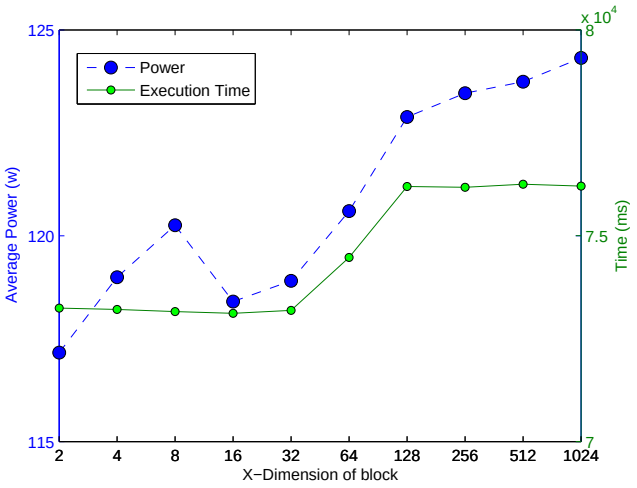

Figure 8. The execution time and power consumption of I-PGJM for different block dimensions The matrix size is $2^{13} \times 2^{13}$.

proposed in [6]. In here, we proposed a method that reduces the computation times of this method. Also with the power consumption hints that described in [8] and [9], we used the configuration that reduces the power consumption of the proposed method.

VI. Conclusion

In this paper, we improved the execution time of Gauss-Jordan matrix inversion method on GPU. In the case of large matrices, as increasing the computations, the power consumption requires more attention. The proposed method used fewer threads and can be configured with different grid and block dimensions. Thus, we can find the configuration that has lower power consumption. The results show that the block dimensions has negligible effect on the power consumption of proposed method and trade-off between performance and power can improve the power consumption of I-PGJM on GPU.

References

[1] S. C. Althoen and R. McLaughlin, "Gauss-Jordan Reduction: A Brief History," Mathematical Association of America, vol. 94, 1987, pp. 130–142.

[2] V. Strassen, "Gaussian elimination is not optimal," Numer Math, vol. 13, 1969, pp. 354 – 356.

[3] D. H. Bailey and H. R. P. Gerguson, "A Strassen-Newton algorithm for high-speed parallelizable matrix inversion," in In Supercomputing '88 Proceedings of the 1988 ACM/IEEE conference on Supercomputing, 1988, pp. 419–424.

[4] A. Burian, J. Takala, and M. Ylinen, "A fixed-point implementation of matrix inversion using Cholesky decomposition." in Proceedings of the 46^{th} international Midwest symposium on circuits and systems. Cairo, 2013, pp. 1431–1433.

[5] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling, Eds., Section 2.3: LU decomposition and its applications, numerical recipes in FORTRAN: the art of scientific computing. New York: Cambridge University Press, 2007.

[6] G. Sharma, A. Agarwala, and B. Bhattacharya, "A fast parallel Gauss Jordan algorithm for matrix inversion using CUDA," Computers and Structures, vol. 128, 2013, pp. 31–37.

[7] G. Wang, Y. Lin, and W. Yi, "Kernel Fusion : an Effective Method for Better Power Efficiency on Multithreaded GPU," in IEEE/ACM International Conference on Green Computing and Communications IEEE/ACM International Conference on Cyber, Physical and Social Computing, 2010, pp. 344 – 350.

[8] S. Mittal and J. S. Vetter, "A Survey of Methods for Analyzing and Improving GPU Energy Efficiency," ACM Computing Surveys, vol. 47, 2014, article No. 19.

[9] Y. Yang, P. Xiang, M. Mantor, and H. Zhou, "Fixing Performance Bugs: An Empirical Study of Open-Source GPGPU Programs," in Proceedings of the 41^{th} International Conference on Parallel Processing (ICPP), 2012, pp. 329 – 339.

[10] "NVIDIA Management Library (NVML)," 2014, URL: https://developer.nvidia.com/nvidia-management-library-nvml [accessed: 2015-05-10].

[11] K. Kasichayanula, D. Terpstra, P. Luszczek, and S. Tomov, "Power Aware Computing on GPUs." in Symposium on Application Accelerators in High Performance Computing (SAAHPC), 2012, pp. 64 – 73.

[12] P. Ezzatti, E. S. Quintana-Ort, and A. Remon, "High Performance Matrix Inversion on a Multi-core Platform with Several GPUs," in Proceedings of the 19^{th} International Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2011, pp. 87 – 93.

[13] P. Benner, P. Ezzatti, E. S. Quintana-Ort, and A. Remon, "Matrix inversion on CPUGPU platforms with applications in control theory," in Concurrency and Computation: Practice and Experience, 2013, pp. 1170 – 1182.

[14] K. Yang, Y. Li, and Y. Xia, "A Parallel Method for Matrix Inversion Based on Gauss-jordan Algorithm," Journal of Computational Information Systems, vol. 14, 2013, pp. 5561 – 5567.

 ISBN: 978-1-61208-416-9