# Modified Incomplete Cholesky Preconditioned Conjugate Gradient Algorithm on GPU for the 3D Parabolic Equation

Jiaquan Gao, Bo Li, Guixia He

# Modified Incomplete Cholesky Preconditioned Conjugate Gradient Algorithm on GPU for the 3D Parabolic Equation

Jiaquan Gao[1,⋆], Bo Li[1] and Guixia He[2]

[1] College of Computer Science and Technology, Zhejiang University of Technology , Hangzhou 310023, China, gaojq@zjut.edu.cn
[2] Zhijiang College, Zhejiang University of Technology, Hangzhou 310024, China

**Abstract.** In this study, for solving the three-dimensional partial differential equation $u_t = u_{xx} + u_{yy} + u_{zz}$, an efficient parallel method based on the modified incomplete Cholesky preconditioned conjugate gradient algorithm (MICPCGA) on the GPU is presented. In our proposed method, for this case, we overcome the drawbacks that the MIC preconditioner is generally difficult to be parallelized on the GPU due to the forward/backward substitutions, and thus present an efficient parallel implementation method on the GPU. Moreover, a vector kernel for the sparse matrix-vector multiplication, and optimization of vector operations by grouping several vector operations into a single kernel are adopted. Numerical results show that our proposed forward/backward substitutions and MICPCGA on the GPU both can achieve a significant speedup, and compared to an approximate inverse SSOR preconditioned conjugate gradient algorithm (SSORPCGA), our proposed MICPCGA obtains a bigger speedup, and outperforms it in solving the three-dimensional partial differential equation.
**keywords**: conjugate gradient algorithm; modified incomplete Cholesky preconditioner; parabolic equation; GPU

## 1 Introduction

The conjugate gradient (CG) algorithm is one of the best known iterative methods. With a suitable preconditioner, the performance of the CG algorithm can be dramatically improved. The preconditioned conjugate gradient (PCG) algorithm has proven its efficiency and robustness in a wide range of applications. Following the introduction of CUDA (Compute Unified Device Architecture) by NVIDIA in 2007 in rent years [1], Graphic Processing Units (GPUs) have drawn much attentions. Many researchers have attempted to develop the suitable and flexible PCG algorithm for the GPU architecture. Related work can be found in [2–5].

⋆ The research has been supported by the Chinese Natural Science Foundation under grant number 61202049 and the Natural Science Foundation of Zhejiang Province, China under grant number LY12A01027

For a PCG algorithm, the key is how to parallel solve the equation $Mz = r$ ($M$ is the preconditioned matrix) on GPUs when shifting it to the GPU platform. For the modified incomplete Cholesky factorization (MIC) preconditioning, $M = LDL^T$, where $L$ is a lower triangular matrix. As we know, for solving the equation $LDL^Tz = r$, the following two steps are required. First, $LDv = r$ is solved for $v$ by the forward substitution, and then $DL^Tz = Dv$ is solved for $z$ by the backward substitution. It is obviously observed that the forward/backward substitutions are not easy to implement on GPUs if $L$ does not have especial characteristics.

Due to the forward/backward substitutions, the MIC preconditioning is difficult to be parallelized on GPUs. In this study, we present an efficient method for parallelizing the forward/backward substitutions on the GPU, which is the main contribution. The reminder of this paper is organized as follows. In the second section, the problem and the MIC PCG algorithm are described. In the third section, some GPU kernels for the MIC PCG algorithm are proposed. Numerical results are presented in the fourth section. The fifth section contains our conclusions and points to our future research direction.

## 2 Problem and MIC PCG Algorithm

### 2.1 Problem Description

In this study, we consider the following partial differential equation (PDE):

$$\begin{cases} u_t = u_{xx} + u_{yy} + u_{zz}, & (x, y, z, t) \in \Omega \times [0, T], \\ u(x, y, z, 0) = \phi(x, y, z), & (x, y, z) \in \Omega, \\ u(x, y, z, t) = 0, & (x, y, z, t) \in \partial\Omega \times [0, T], \end{cases} \tag{1}$$

where $u$ is the solution, $\phi(x, y, z)$ is a function of variables $x$, $y$ and $z$, $\Omega$ is a regular three-dimensional domain and $\partial\Omega$ denotes the boundary of $\Omega$.

Here we utilize the discrete variational derivative method (DVDM) to discretize the PDE (1). As compared to the finite difference method (FDM), The DVDM can guarantee that the constructed numerical scheme retains the energy dissipation or conservation properties. According to the strategy of the DVDM to construct a dissipative scheme, we define an energy $G(u, u_x, u_y, u_z) = (u_x)^2/2+(u_y)^2/2+(u_z)^2/2$ of the PDE (1) and then obtain the numerical scheme as follows.

$$\begin{aligned} \frac{u^t_{i,j,k} - u^{t-1}_{i,j,k}}{\Delta t} =& \delta^{(2)}_i \left( \frac{u^t_{i,j,k} + u^{t-1}_{i,j,k}}{2} \right) + \delta^{(2)}_j \left( \frac{u^t_{i,j,k} + u^{t-1}_{i,j,k}}{2} \right) \\ &+ \delta^{(2)}_k \left( \frac{u^t_{i,j,k} + u^{t-1}_{i,j,k}}{2} \right), \end{aligned} \tag{2}$$

where

$$\delta_i^{(2)} u_{i,j,k} = \frac{u_{i+1,j,k} - 2u_{i,j,k} + u_{i-1,j,k}}{(\Delta x)^2},$$
$$\delta_j^{(2)} u_{i,j,k} = \frac{u_{i,j+1,k} - 2u_{i,j,k} + u_{i,j-1,k}}{(\Delta y)^2},$$
$$\delta_k^{(2)} u_{i,j,k} = \frac{u_{i,j,k+1} - 2u_{i,j,k} + u_{i,j,k-1}}{(\Delta z)^2}.$$

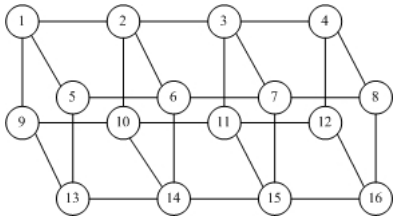


**Fig. 1.** A $4 \times 2 \times 2$ three-dimensional grid

Assume that a $4 \times 2 \times 2$ three-dimensional grid, shown in Fig. 1, is defined, the scheme (2) can be written as the following linear system for a certain time $t \in [0, T]$.

$$Au = b, \tag{3}$$

where the coefficient matrix A is denoted as follows:

$$\begin{bmatrix}
* & * & & * & & & * & & & & & & & & & \\
* & * & * & & * & & & * & & & & & & & & \\
& * & * & * & & * & & & * & & & & & & & \\
& & * & * & & & * & & & * & & & & & & \\
* & & & * & * & & & * & & & * & & & & & \\
& * & & * & * & * & & & * & & & * & & & & \\
& & * & & * & * & * & & & * & & & * & & & \\
& & & * & & * & * & & & & * & & & * & & \\
* & & & & * & * & & * & & & * & & & & & \\
& * & & & & * & * & * & & & & * & & & & \\
& & * & & & & * & * & * & & & & * & & & \\
& & & * & & & & * & * & & & & & * & & \\
& & & & * & & & * & & * & * & & & & & \\
& & & & & * & & & * & & * & * & & & & \\
& & & & & & * & & & * & & * & * & * & & \\
& & & & & & & * & & & * & & * & * &
\end{bmatrix},$$

a sparse, symmetric, positive definite, seven-diagonal, diagonally dominant $16 \times 16$ matrix of regular structure, $b$ is a vector of size 16, and $u$ is an unknown solution vector of size 16. Therefore, it can be obviously seen that the key is to solve the linear system (3) for solving the 3D PDE (1).

### 2.2 MIC PCG Algorithm

For solving the symmetric linear system similar to (3), the PCG algorithm has proven its efficiency and robustness in a wide range of applications. Its detailed procedure is shown in Algorithm 1.

**Algorithm 1:** PCG algorithm for the linear system $Au = b$
Input: $A, b$; Output: $u$

01 $k = 0$
02 $u = [1, 1, \cdots, 1]^T, r = b - Au, Mz = r, p = z, \rho_0 = z^T r$
03 Repeat
04 $q = Ap, \alpha = \rho_0 / p^T q$
05 $u = u + \alpha p$
06 $r = r - \alpha q$
07 $Mz = r$
08 $\rho_1 = z^T r, \beta = \rho_1 / \rho_0, \rho_0 = \rho_1$
08 $p = r + \beta p$
09 $k = k + 1$
10 Until ($k < maxiter$ and $r^T r > tol$)

For the above linear system (3), the preconditioned matrix $M = LDL^T$, where $L$ is a lower triangular matrix with nonzero values along the main diagonal and at off-diagonal locations only where $A$ has nonzero entries. Generally, the forward/backward substitutions both need $n$ ($n$ is the element number of the solution vector) steps to finish calculations of elements. However, we observe that the forward/backward substitutions for this case both can be finished in 6 steps instead of 16 steps according to the progressions shown in Fig. 2.

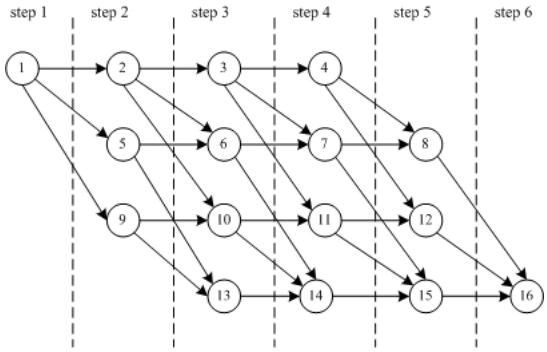


**Fig. 2.** Progression of the forward substitution

The computational grid in Fig. 2 is the same as in Fig. 1, and arrows represent dependencies between elements of the grid. The calculation of an element in each step does not only depend on its adjacent row and column neighbors, but also depends on its adjacent layer neighbors. For example, the element 14 is dependent to its adjacent row and column elements 10 and 13 and its adjacent layer element 6. However, not all elements have these neighbor elements and the number of neighbors varies with the location of the element. Moreover, for any element, its dependent neighbors must be calculated earlier than it in order to

start its calculation. Moreover, the elements of each step are independent each other and can be concurrently calculated. All these things are helpful for shifting the forward/backward substitutions to the GPU platform.

## 3 GPU Kernels and Optimization

The primary kernels used in the implementation of Algorithm 1 are explained in this section. The kernels represented here perform mathematical operations used in Algorithm 1. Following Algorithm 1, the basic operations are sparse matrix-vector multiplications, vector operations and the forward/backward substitutions.

### 3.1 Sparse Matrix-Vector Multiplication

Here the sparse matrix $A$ is stored with the compressed sparse row (CSR) format: (1) the array $a$ contains all the nonzero elements of $A$; (2) the array $colidx$ contains column indices of these nonzero elements; and (3) entries of array $rowstr$ point to the first element of subsequent row of $A$ in arrays $a$ and $colidx$.

Sparse matrix-vector multiplications (SpMVs) represent the dominant cost in the PCG algorithm for solving large-scale linear systems. Fortunately, the SpMV for the CSR format is easy to be parallelized on the GPU. A straightforward CUDA implementation, which is referred to as the scalar kernel, uses one thread per row. However, its performance suffers from several drawbacks. The most significant among these problems is that threads with a warp (a bunch of 32 CUDA threads) can not access the arrays $a$ and $colidx$ in a coalesced manner. In [6], An alternative to the scalar kernel, called the vector kernel, assigns one warp to each matrix row. For the vector kernel, it accesses the arrays $a$ and $colidx$ contiguously, and therefore overcomes the principal deficiency of the scalar kernel.

Thus, here we will utilize the vector kernel to compute the SpMV on the GPU, and refer the readers interested in the detailed GPU implementation of the vector kernel to the literature [6].

### 3.2 Vector Operations

As we can see in Algorithm 1, the vector operations are the vector copy, the scalar vector product, the *saxpy* operation and the inner product of vector. In order to optimize these operations, we try to group several operations into a single kernel. For example, the *saxpy* operation as well as the vector copy are grouped in the same kernel. On the other hand, we perform the inner products needed for the computation of $\alpha$ and the inner products involved in the $\beta$ computation in single kernels.

### 3.3 Forward/Backward Substitution

In this section, we will exhibit our proposed method of the forward/backward substitutions on the GPU. In order to better show our proposed method, the $4 \times 2 \times 2$ grid in Fig. 1 are extended to a $64 \times 48 \times 3$ grid.

Since the implementation technique of the forward substitution is also suitable for the backward substitution for this case, here we only discuss the forward substitution. Assume that each thread block is assigned with $x \times y$ solution elements, the number of required thread blocks can be calculated by the following formula:

$$N^{tb} = \left[\frac{NROW}{x}\right] \times \left[\frac{NCOL}{y}\right] \times NLAY. \tag{4}$$

By Eq.(4), 36 thread blocks are required if each thread block is assigned with $16 \times 16$ ($x = 16$ and $y = 16$) solution elements for the $64 \times 48 \times 3$ three-dimensional grid. A way of possible grouping of 36 thread blocks on the GPU is illustrated in Fig. 3.

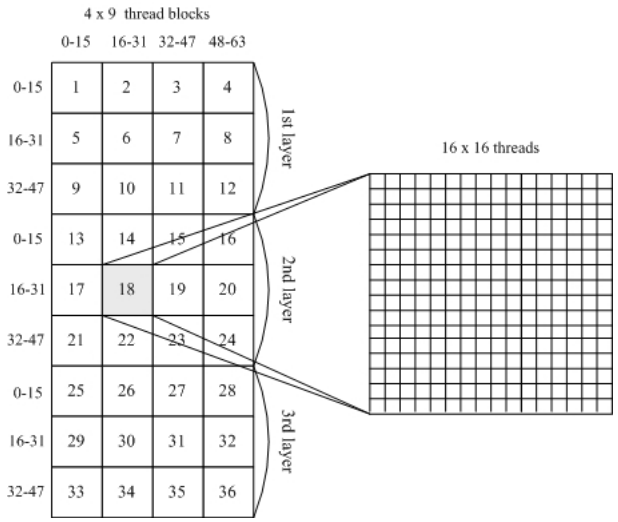


**Fig. 3.** $4 \times 9$ sized grid of GPU thread block

According to the progression of the forward substitution in Fig. 2, it is obviously found that the amount of data must be almost transferred between two adjacent thread blocks from the same layer and from the neighboring layers. For example, consider calculations of elements in thread block 18. To start its execution, on one hand, the thread block 18 must wait for transmission from its adjacent thread blocks 14 and 17 to which it is independent in the same layer; on the other hand, it also waits for transmission from its adjacent thread block

6 on which it depends in the neighboring layer. The total amount of data to be received from the same layer is $(x + y)$. However, the total amount of data to be received from the neighboring layers is $(x \times y)$. Since the data exchange between thread blocks must be performed on the GPU global memory, the transmissions will be the primary source of the overall communication latency.

Here in order to decrease the amount of exchanged data between thread blocks from the neighboring laryers, we utilize the share memory to store the values calculated by a thread block, and the thread blocks which are located in different layers and whose position are the same are mapped to a thread block. For example, thread block 1 in the first layer, thread block 13 in the second layer and thread block 25 in the third layer can be mapped to a thread block in Fig. 3. If so, the data exchange between thread blocks of the adjacent layers will be no longer required because they are located in the same thread block. Thus, the amount of exchanged data required is to decrease from $(x \times y + x + y)$ to $(x + y)$ when the elements in a thread block are calculated. For example, when calculating elements in thread block 18 in Fig. 3, only the data transmissions from its adjacent thread blocks 14 and 17 in the same layer are required, and thus the total amount of exchanged data is 32 rather than 288. At the same time, the number of required thread blocks is also reduced and can be calculated as

$$N^{tbnew} = \left[\frac{NROW}{x}\right] \times \left[\frac{NCOL}{y}\right]. \tag{5}$$

Therefore, the number of required thread blocks for this case is decreased from 36 to 12. Figure 4 summarizes the progression of the forward substitution with 12 active thread blocks.

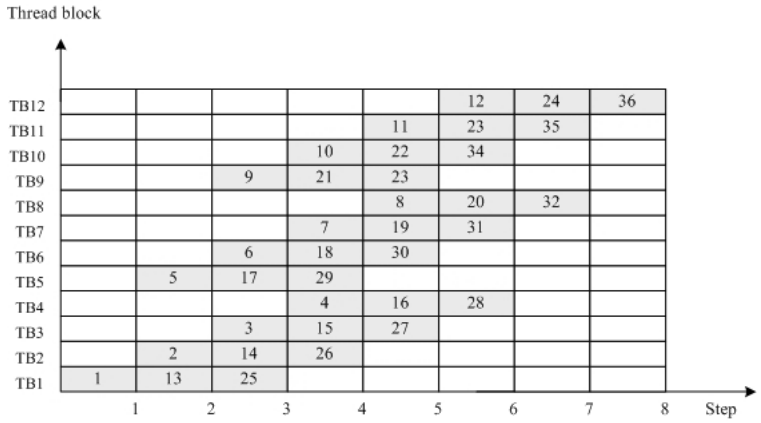


**Fig. 4.** Progression of the forward substitution with 12 active thread blocks

Furthermore, in order to decrease the frequency of reading data from the GPU global memory, we allocate a two-dimensional $(x+1)\times(y+1)$ array in shard

memory instead of $(x \times y)$ to store data for a $x \times y$ sized thread block. The extra row and column are allocated for dependent data from neighbor thread blocks in the same layer. Moreover, to manage the dependencies between adjacent thread blocks, we store a matrix in the GPU global memory where each thread block can check the status of adjacent thread blocks to which it is dependent. A thread block waits for a spinlock until the dependent thread blocks finish calculations and write results to the global memory. Then it reads these results and continues with its calculations.

## 4 Numerical Results

All experiments in this section are conducted on the environment which is based on GNU/Linux Ubuntu v10.04.1 with an Intel Xeon Quad-Core 2.66 GHz, 12GB RAM (CPU) and NVIDIA Tesla C2050, 448 CUDA Cores, 6GB RAM (GPU).

Firstly, the following seven three-dimensional grid models, $800 \times 800 \times 30$, $480 \times 480 \times 80$, $480 \times 480 \times 320$, $128 \times 128 \times 480$, $512 \times 512 \times 256$, $128 \times 128 \times 1440$, $1440 \times 1440 \times 80$, are chosen to test the performance of our proposed forward substitution (GPUFBS). We respectively add up the computational time of GPUFBS and the CPU implementation of the forward/backward substitutions and then show them in Table 1. The time unit is microsecond denoted by $ms$.

**Table 1.** Speedups of GPUFBS

| Grid model | CPU ($ms$) | GPU ($ms$) | Speedup |
|---|---|---|---|
| $800 \times 800 \times 30$ | 1020 | 125 | 8.16 |
| $480 \times 480 \times 80$ | 1058 | 101 | 10.47 |
| $480 \times 480 \times 320$ | 4784 | 278 | 17.20 |
| $128 \times 128 \times 480$ | 458 | 39 | 11.74 |
| $512 \times 512 \times 256$ | 3958 | 256 | 15.38 |
| $128 \times 128 \times 1440$ | 1463 | 67 | 21.83 |
| $1440 \times 1440 \times 80$ | 9560 | 893 | 10.70 |

From Table 1, we can see that GPUFBS achieves a significant speedup for all seven three-dimensional grid models due to high utilization of GPU. As observed from Fig. 4, the thread blocks have high concurrency and the concurrency of thread blocks is improved as the number of layers increases.

By observing $480 \times 480 \times 80$ and $480 \times 480 \times 320$ grid models, it can be found that when the column and row sizes are given, GPUFBS with high number of layers has a bigger speedup than that of with low number of layers. Similarly, it can also seen that for a given number of layers, GPUFBS with a big size of $column \times row$ achieves a higher speedup than that with a small size of $column \times row$ by comparing the $480 \times 480 \times 80$ grid model with the $1440 \times 1440 \times 80$ model. Furthermore, as observed from the $128 \times 128 \times 1440$ and $128 \times 128 \times 480$ grid

models, for the same size of $column \times row$ ($128 \times 128$), the speedup obtained by GPUFBS with 1440 layers is nearly 2 times of the speedup obtained by GPUFBS with 480 layers. However, we compare the $480 \times 480 \times 80$ grid model with the $1440 \times 1440 \times 80$ grid model and find that for the same number of layers 80, the speedup obtained by GPUFBS with $1440 \times 1440$ ($column \times row$) is only 1.02 times of the speedup obtained by GPUFBS with $480 \times 480$ ($column \times row$). Therefore, we can conclude that the number of layers and the column and row sizes both have an impact on the performance of GPUFBS, but the number of layers for GPUFBS has a larger impact than the column and row sizes.

Secondly, we test the validity of our proposed MIC PCG method on GPU (MICPCGA) for solving the 3D PDE comparing with the PCG algorithm on the GPU (SSORPCGA) suggested in [3]. Let $\Omega = \{(x, y, z)|0 \leq x \leq 1, 0 \leq y \leq 1, 0 \leq z \leq 1\}$, $\phi(x, y, z) = 3 \sin \pi x \sin \pi y \sin \pi z$ for the PDE (1), and define the error (ER) $= \max |u_{i,j,k} - \tilde{u}_{i,j,k}|$, where $u_{i,j,k}$ and $\tilde{u}_{i,j,k}$ respectively denote the exact solution and the approximate solution at the point $(x_i, y_j, z_k)$. For this case, the exact solution is $u(x, y, z, t) = 3e^{-3\pi^2 t} \sin \pi x \sin \pi y \sin \pi z$, and the following five grid models, $100 \times 100 \times 100$ (M21), $250 \times 250 \times 100$ (M22), $250 \times 250 \times 250$ (M23), $100 \times 100 \times 250$ (M24), $100 \times 100 \times 1000$ (M25), are chosen. In the following, taking $x = 0.5, y = 0.5 (z = 0.1, 0.3, 0.5, 0.7, 0.9)$ for example, Table 2 shows the errors of MICPCGA and SSORPCGA. The speedups of MICPCGA and SSORPCGA are summed in Table 3.

**Table 2.** Errors of MICPCGA and SSORPCGA ($t = 0.1$)

| Method | 0.1 | 0.3 | 0.5 | 0.7 | 0.9 |
|---|---|---|---|---|---|
| Exact solution($10^{-1}$) | 0.4799 | 1.2565 | 1.5531 | 1.2565 | 0.4799 |
| ER MICPCGA M21($10^{-5}$) | 1.1236 | 1.2342 | 1.4347 | 1.2358 | 1.2362 |
| ER MICPCGA M22($10^{-5}$) | 1.1192 | 1.1331 | 1.3019 | 1.1376 | 1.1164 |
| ER MICPCGA M23 ($10^{-5}$) | 0.5786 | 0.8941 | 1.1893 | 0.8937 | 0.5734 |
| ER MICPCGA M24 ($10^{-5}$) | 1.0304 | 1.1343 | 1.2184 | 1.1342 | 1.0328 |
| ER MICPCGA M25 ($10^{-5}$) | 1.0122 | 1.1121 | 1.1834 | 1.0101 | 1.0152 |
| ER SSORPCGA M21 ($10^{-5}$) | 5.5632 | 5.9822 | 6.2876 | 5.9896 | 5.5617 |
| ER SSORPCGA M22 ($10^{-5}$) | 4.9127 | 5.3939 | 5.9918 | 5.3908 | 4.9103 |
| ER SSORPCGA M23 ($10^{-5}$) | 2.3243 | 2.4232 | 3.1967 | 2.4234 | 2.3253 |
| ER SSORPCGA M24 ($10^{-5}$) | 5.1342 | 5.1458 | 5.7633 | 5.1442 | 5.1354 |
| ER SSORPCGA M25 ($10^{-5}$) | 4.7482 | 4.8294 | 5.2304 | 4.8261 | 4.7476 |

From Table 3, we can see that for models M21 and M22, MICPCGA has a comparable speedup with SSORPCGA, and for models M23, M24 and M25, MICPCGA obtains a bigger speedup than SSORPCGA. Furthermore, it can be found that MICPCGA can obtain better approximate solutions than SSOR-PCGA for all cases. Therefore, we can conclude that MICPCGA outperforms SSORPCGA for this case.

**Table 3.** Speedups of MICPCGA and SSORPCGA

| Method and Model | CPU ($s$) | GPU ($s$) | Speedup |
|---|---|---|---|
| MICPCGA M21 | 3.71 | 0.40 | 9.27 |
| MICPCGA M22 | 23.91 | 2.43 | 9.83 |
| MICPCGA M23 | 63.58 | 5.21 | 12.20 |
| MICPCGA M24 | 9.62 | 0.93 | 10.34 |
| MICPCGA M25 | 39.36 | 3.51 | 11.21 |
| SSORPCGA M21 | 3.92 | 0.42 | 9.33 |
| SSORPCGA M22 | 25.40 | 2.60 | 9.76 |
| SSORPCGA M23 | 65.19 | 7.02 | 9.28 |
| SSORPCGA M24 | 10.13 | 1.03 | 9.83 |
| SSORPCGA M25 | 41.80 | 4.24 | 9.85 |

## 5 Conclusion

In this study, we present an efficient parallel method for the forward/backward substitutions on the GPU, and thus propose a MIC PCG algorithm on the GPU. Numerical results show that our proposed MIC PCG algorithm has a good behavior, and outperforms the PCG algorithm suggested by Helfenstein and Koko.

For GPUFBS, its efficiency has been validated in this study. Next, we will extend the constructing idea of GPUFBS to other PCG algorithms with the following preconditioners, the incomplete-LU factorization (ILU), the modified incomplete-LU factorization (MILU) and their variants, and furthermore do research for the three-dimensional partial differential equation.

## References


1. NVIDIA Corporation: Cuda programming guide 2.3. Technical Report, NVIDIA (2009)
2. Buatois L., Caumon G.: Concurrent number cruncher: a GPU implementation of a general sparse linear solver. Int. J. Parallel Emergent Distrib. Syst., 24(3), 205-223 (2009)
3. Helfenstein R., Koko J.: Parallel preconditioned conjugate gradient algorithm on GPU. J. Comput. Appl. Math., 236(15), 3584-3590 (2012)
4. Gravvanis G.A., Filelis-Papadopoulos C.K., Giannoutakis K.M.: Solving finite difference linear systems on GPUs: CUDA parallel explicit preconditioned biconjugate conjugate gradient type methods. J. Supercomput., 61(3), 590-604 (2012)
5. Galiano V., Migallón H., Migallón V.: GPU-based parallel algorithms for sparse nonlinear systems. J. Parallel Distrib. Comput., 72(9), 1098-1105 (2012)
6. Bell N., Garland M.: Efficient sparse matrix-vector multiplication on CUDA. Technique report, NVIDIA (2008)