A Fast Parallel Matrix Inversion Algorithm Based on Heterogeneous Multicore Architecture

Denggao Yu

Joint work with Shiwen He, Yongming Huang, Guangshi Yu, Lvxi Yang

School of Information Science and Engineering,
Southeast University, Nanjing, China
Email:{220130708}@seu.edu.cn

Content

- Introduction
- Parallel Algorithm for Matrix Inversion
- Implemented on Heterogeneous Multicore Architecture
- Simulation Results
- Conclusion

Introduction

Background

- Necessity to invert large matrix quickly and accurately.
- The Graphics Processor Unit (GPU) is able to provide a low-cost and flexible multicore architecture for high performance computing.

Motivation

- We want to design a fast parallel algorithm for matrix inversion to utilize the computational power of GPU.

Introduction

Existing Work

- [3] and [4] just present the triangular matrix inversion (TMI) on GPU, not the full matrix.
- In [5] and [6], the Gaussian-Jordan and Gaussian elimination algorithms are implemented on GPU.

Our Work

- We firstly designed a fast parallel algorithm for matrix inversion based on Modified Squared Givens Rotations.
- This algorithm was implemented on CUDA to utilize the computational power of GPU.

Parallel Algorithm for Matrix Inversion

It is well known that, inversion of matrix **A** can be performed by firstly decomposing matrix **A** into an upper triangular matrix **R** and a unitary matrix **Q** via using QR decomposition (QRD) [7], namely**,** **A=QR**. And it has been proved that the QRD could be equivalently written as equation (1), then the inversion of matrix **A** could be calculated as $\mathbf{A}^{-1} = \mathbf{U}^{-1}\left(\mathbf{Q}_A \mathbf{D}_U^{-1}\right)^{-1}$.

$$\mathbf{A} = \mathbf{Q}_A \mathbf{D}_U^{-1} \mathbf{U} \qquad (1)$$

- **Relation to the original QRD**

$\mathbf{Q}_A = \mathbf{Q}\mathbf{D}_R$

$\mathbf{D}_R = diag\left(\mathbf{R}\right)$

$\mathbf{D}_\mathbf{U} = \mathbf{D}_\mathbf{R}^2$

U is an upper triangular matrix

function $diag\left(\mathbf{R}\right)$ returns the main diagonal of matrix **R.**

Parallel Algorithm for Matrix Inversion

Step 1: Calculate the upper triangular matrix U

Considering two complex vectors as

$$\begin{bmatrix} \mathbf{a} \\ \mathbf{b} \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & \cdots & a_k & \cdots & a_p \\ b_1 & b_2 & \cdots & b_k & \cdots & b_p \end{bmatrix} \quad (2)$$

Assume that $a_k \neq 0, b_k \neq 0$, the traditional Givens Rotations could be done to eliminate b_k in vector **b** as

$$\begin{cases} c = \left(a_k^* a_k + b_k^* b_k \right)^{1/2} \\ \bar{\mathbf{a}} = c^{-1} \left(a_k^* \mathbf{a} + b_k^* \mathbf{b} \right) \\ \bar{\mathbf{b}} = c^{-1} \left(-b_k \mathbf{a} + a_k \mathbf{b} \right) \end{cases} \quad (3)$$

where $\bar{\mathbf{a}}$ and $\bar{\mathbf{b}}$ are the updated vectors of **a** and **b.**

Parallel Algorithm for Matrix Inversion

To remove the square root operations and divisions involved in equation (3), we firstly translate vectors **a** and **b** to **u** and **v** space respectively.

$$\begin{cases} \mathbf{u} = a_k^* \mathbf{a} \\ \mathbf{v} = \mathbf{b} \end{cases} \quad (4)$$

Then the Givens Rotations equation (3) could be written as

$$\begin{cases} \bar{\mathbf{u}} = \mathbf{u} + v_k^* \mathbf{v} \\ \bar{\mathbf{v}} = \mathbf{v} - \dfrac{v_k}{u_k} \mathbf{u} \end{cases} \quad (5)$$

Then through this transformation, only real division operations are included during the Givens Rotations phase.

Parallel Algorithm for Matrix Inversion

- **Situations when** $u_k = 0$

$$\left.\begin{aligned} \bar{\mathbf{u}} &= \mathbf{v} \\ \bar{\mathbf{v}} &= -\mathbf{u} \end{aligned}\right\} when\ u_k = 0 \qquad (6)$$

Parallel Algorithm for Matrix Inversion

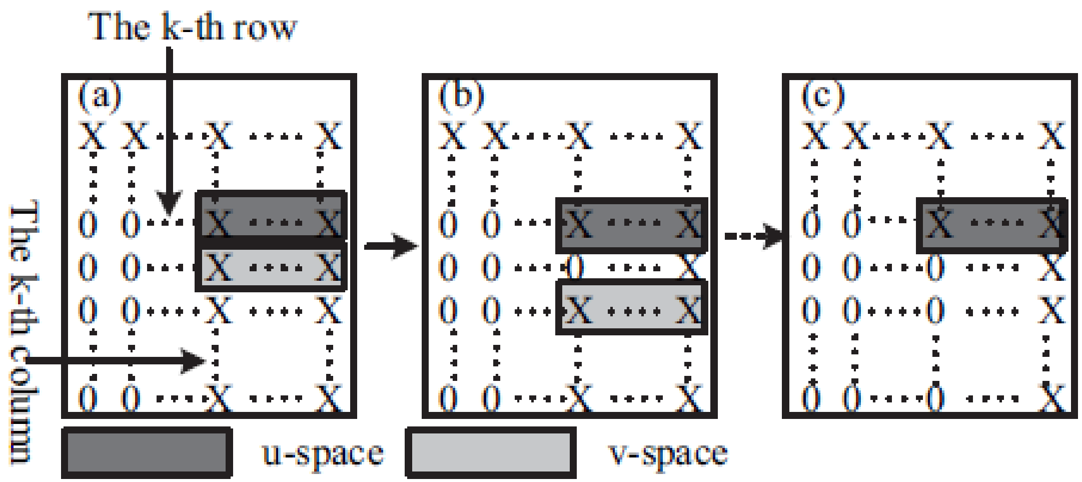


Fig. 1: Elements elimination of the k-th column

Parallel Algorithm for Matrix Inversion

Step 2: Calculate the Inversion Matrix of U

The inversion of the triangular matrix **U** can be easily achieved via the back substitution method [7], i.e.,

$$\mathbf{G}_{ij} = \begin{cases} -\dfrac{1}{\mathbf{U}_{jj}}\left(\displaystyle\sum_{k=i}^{j-1} \mathbf{G}_{ik}\mathbf{U}_{kj}\right) & i < j \\ \dfrac{1}{\mathbf{U}_{jj}} & i = j \\ 0 & i > j \end{cases} \qquad (7)$$

Here $\mathbf{G} = \mathbf{U}^{-1}$.

.

Parallel Algorithm for Matrix Inversion

Step 3: Compute the Inversion Matrix of A

- Recalling equation (1): $\mathbf{A} = \mathbf{Q}_A \mathbf{D}_U^{-1} \mathbf{U} = \mathbf{X}\mathbf{U}$, rewrite it as $\mathbf{U} = \left(\mathbf{Q}_A \mathbf{D}_U^{-1}\right)^{-1} \mathbf{A} = (\mathbf{X})^{-1} \mathbf{A}$. Then we could treat $(\mathbf{X})^{-1}$ as a factor φ . Matrix **U** could be produced from **A** via left multiplied by φ .

- Then $(\mathbf{X})^{-1}$ could be obtained when identity matrix **I** is left multiplied by φ , namely, $(\mathbf{X})^{-1} = (\mathbf{X})^{-1} \mathbf{I}$, which means identity matrix **I** could be rotated in the similar way as matrix **A,** as described in Step 1. After $(\mathbf{X})^{-1}$ is achieved, the matrix inversion could be done as $\mathbf{A}^{-1} = \mathbf{U}^{-1} \left(\mathbf{Q}_A \mathbf{D}_U^{-1}\right)^{-1} = \mathbf{U}^{-1} \mathbf{X}^{-1}$.

Implemented on Heterogeneous Multicore Architecture

Heterogeneous multicore architecture

- A host which is usually a CPU that is used for controlling and processing the serial parts of the algorithm.
- A GPU including a large number of small cores focus on the execution of the parallel parts.
- CUDA is a new hardware and software architecture for parallel computing on.

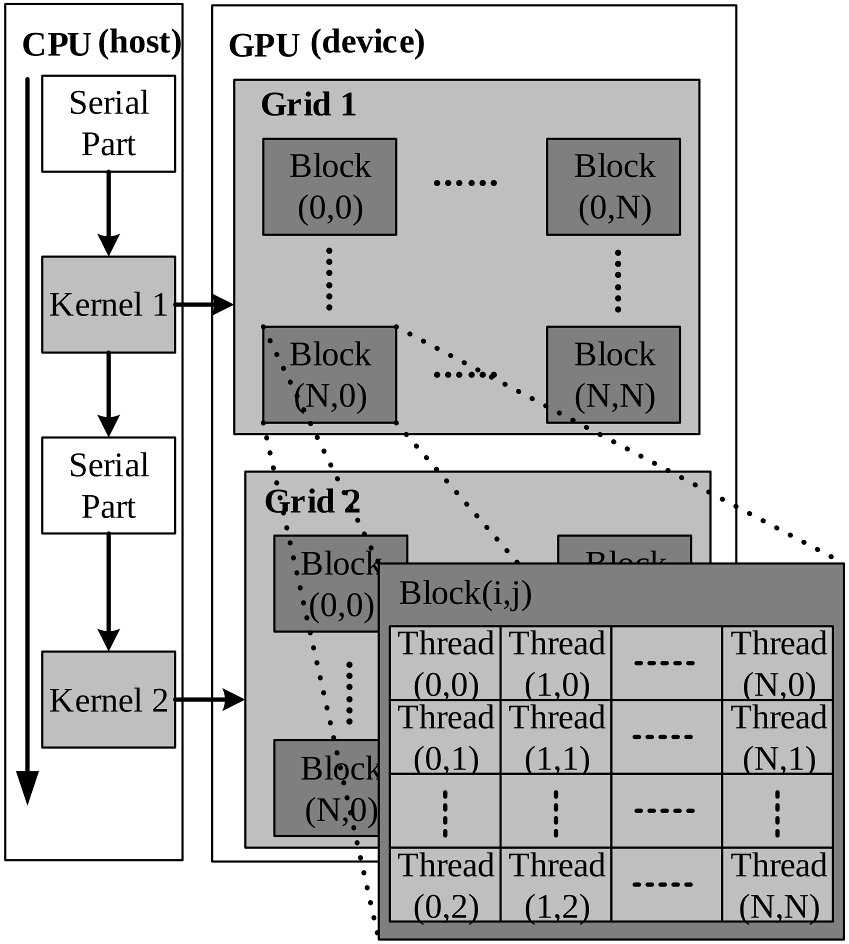


Fig. 2: Heterogeneous Multicore Architecture

Implemented on Heterogeneous Multicore Architecture

Firstly, we create an extension matrix **B=[A | I]**, matrix **A** is the original matrix, matrix **I** is an identity matrix the same dimension as **A**. Then copy matrix **B** from host to device to initialize CUDA.

Step 1: Call *Kernel 1* to obtain upper triangular matrix U and $(\mathbf{Q}_A\mathbf{D}_U^{-1})^{-1}$

- The *Kernel 1* runs on GPU as shown in Fig. 2, which is called by the host. To realize this part in parallel, we aim to create a thread for each element of matrix **B**. Hence we launch *2n* threads for each computation of $\bar{\mathbf{u}}$ *and* $\bar{\mathbf{v}}$. The parallel execution models based on equation (5) is indicated in Fig. 3 and Fig. 4.
- When using equation (6), the parallel models are similar, which is much simpler actually.

Implemented on Heterogeneous Multicore Architecture

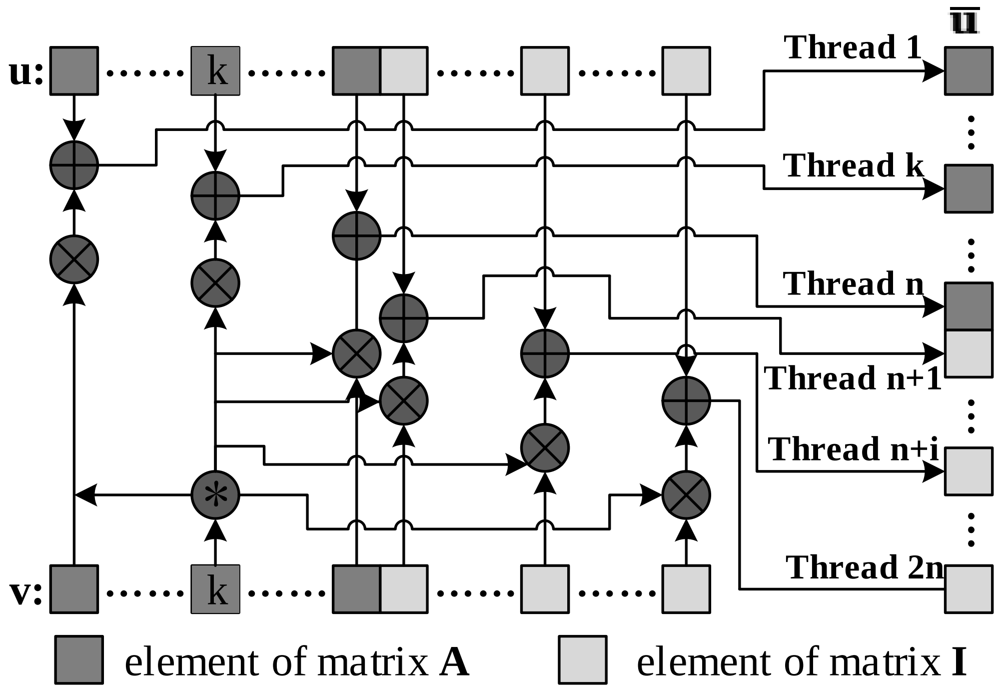


Fig. 3: Parallel execution model while computing $\bar{\mathbf{u}}$

Implemented on Heterogeneous Multicore Architecture

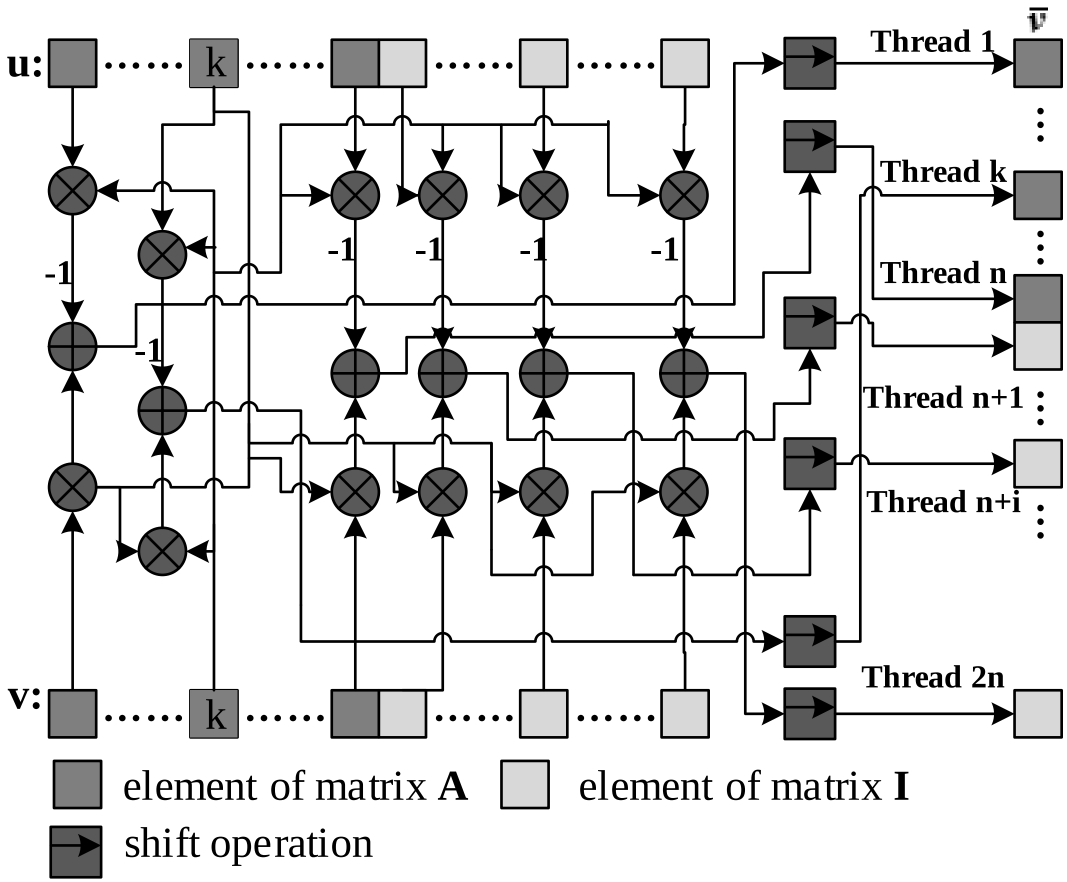


Fig. 4: Parallel execution model while computing $\overline{\mathbf{v}}$

Implemented on Heterogeneous Multicore Architecture

Step 2: Compute U^{-1} on host

- Since the interdependencies between the data preclude the inversion of matrix **U** from being executed in parallel. We compute U^{-1} on host based on the back substitution method as described in equation (7).

Implemented on Heterogeneous Multicore Architecture

Step 3: Call *Kernel 2* to compute matrix multiplication $\mathbf{U}^{-1}(\mathbf{Q}_A\mathbf{D}_U^{-1}$

- Matrix multiplication is very suitable for parallelization. For simplicity, we use matrix **E** and matrix **F** denote $\mathbf{U}^{-1}$ *and* $\left(\mathbf{Q}_A\mathbf{D}_U^{-1}\right)^{-1}$ respectively.

  The parallel execution model of matrix multiplication is shown in Fig. 5.

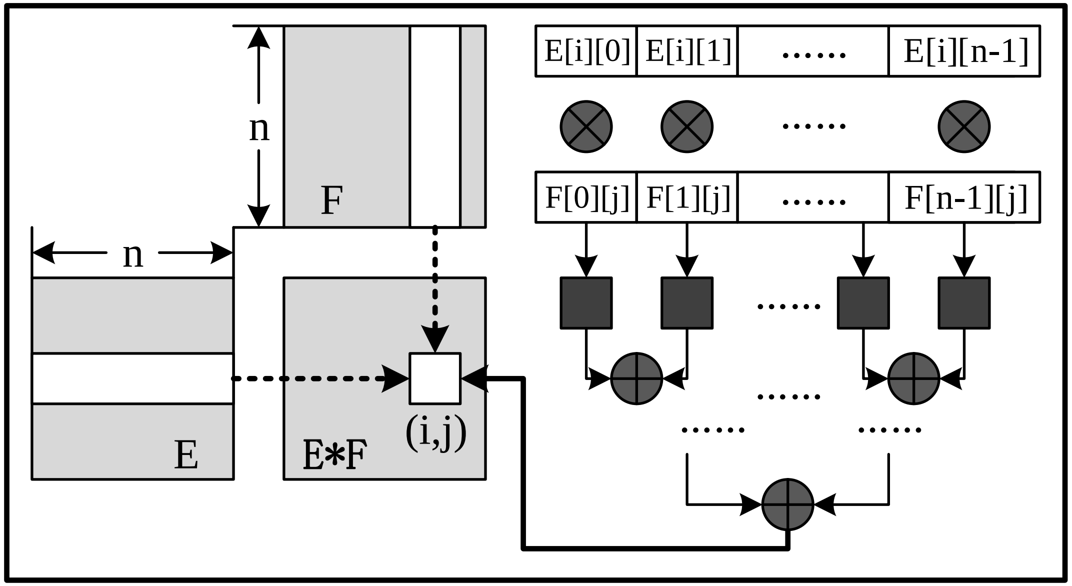


Fig. 5: Parallel execution model for matrix multiplication

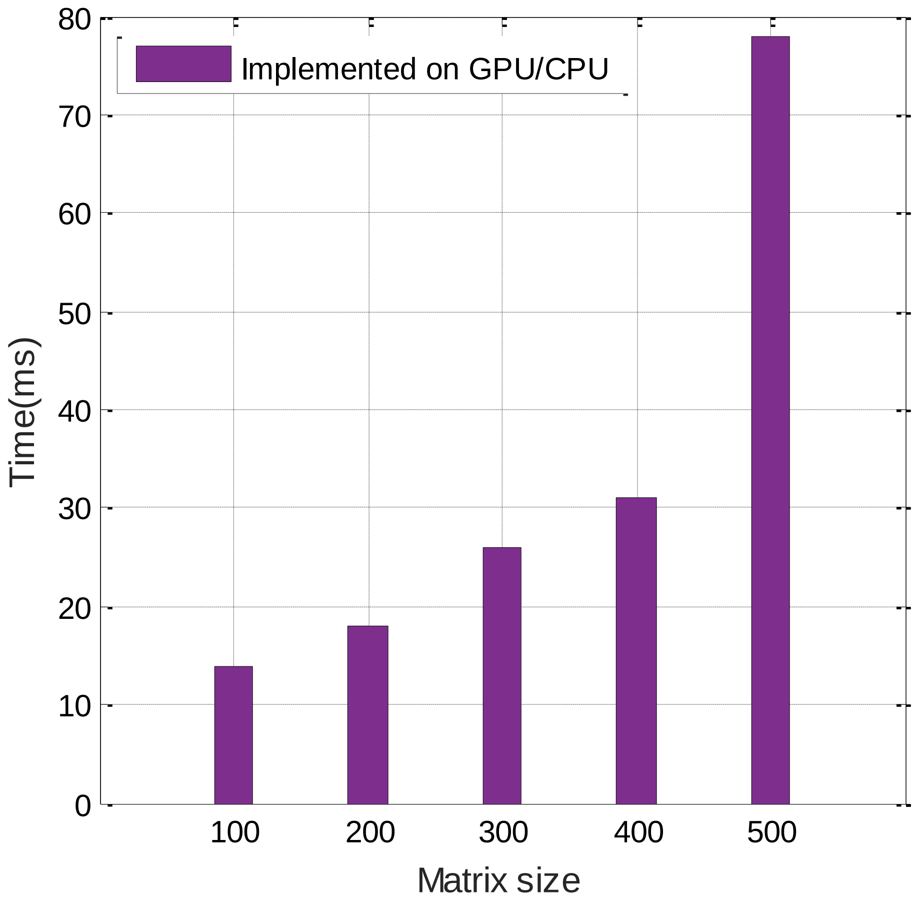
Simulation Results

Our platform consists of an Intel Core i5-3470 four-core CPU and a NVIDIA Geforce GT620 GPU. The concrete parameters of device is shown in TABLE I.

TABLE I Device Parameters

	CPU	**GPU**
Platform	Intel Core i5-3470	NVIDIA Geforce GT620
Number of Cores	4 (only single core was used)	32
Clock Rate	3.2 GHz	1.62GHz
Memory	4GB DDR2 RAM	2G DDR3 memory
System bits	64bits	

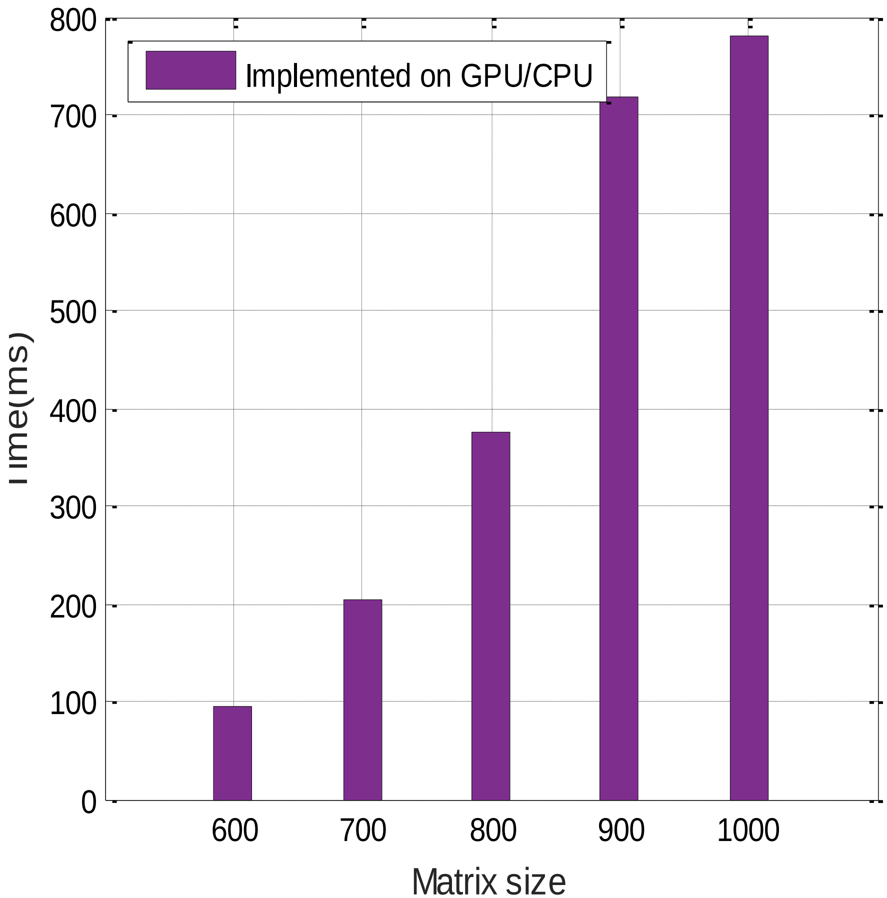
Simulation Results



- The x axis denotes the matrix size from 100×100 to 500 ×500
- The y axis denotes the execution time in milliseconds of the algorithm implemented on CUDA

Fig. 6: Execution times in milliseconds of the algorithm implemented on CUDA

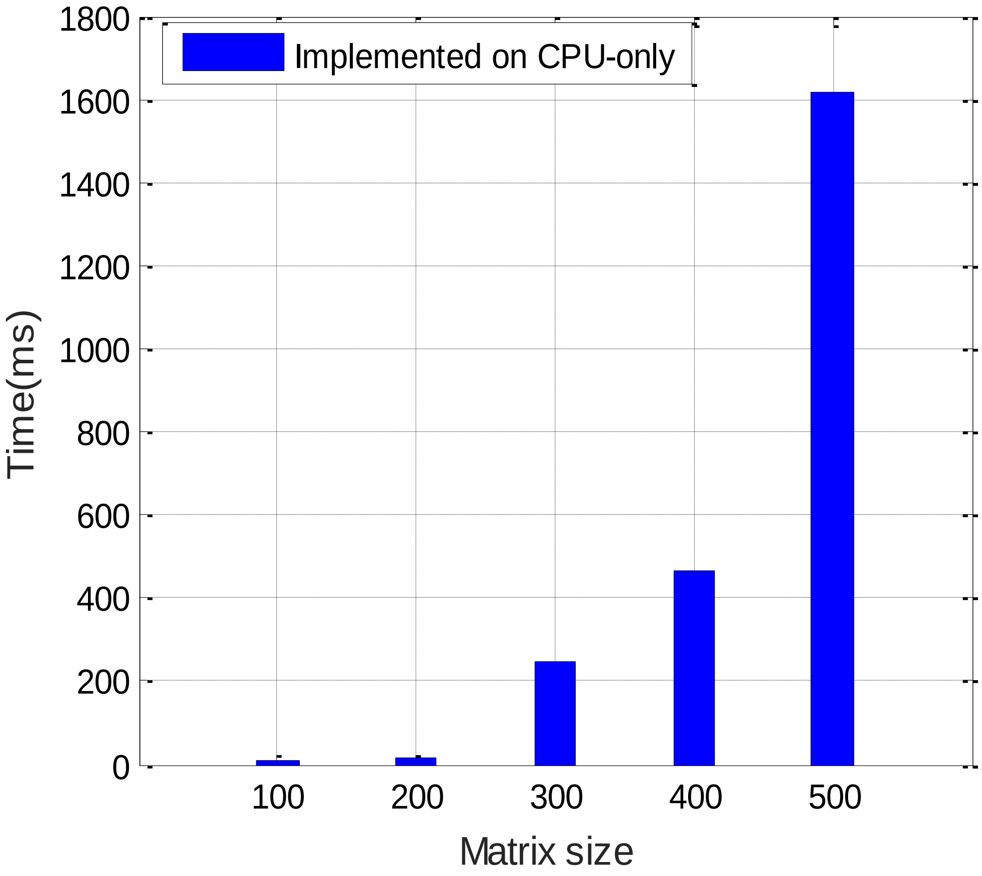
Simulation Results



- The x axis denotes the matrix size from 600×600 to 1000 ×1000
- The y axis denotes the execution time in milliseconds of the algorithm implemented on CUDA

Fig. 7: Execution times in milliseconds of the algorithm implemented on CUDA

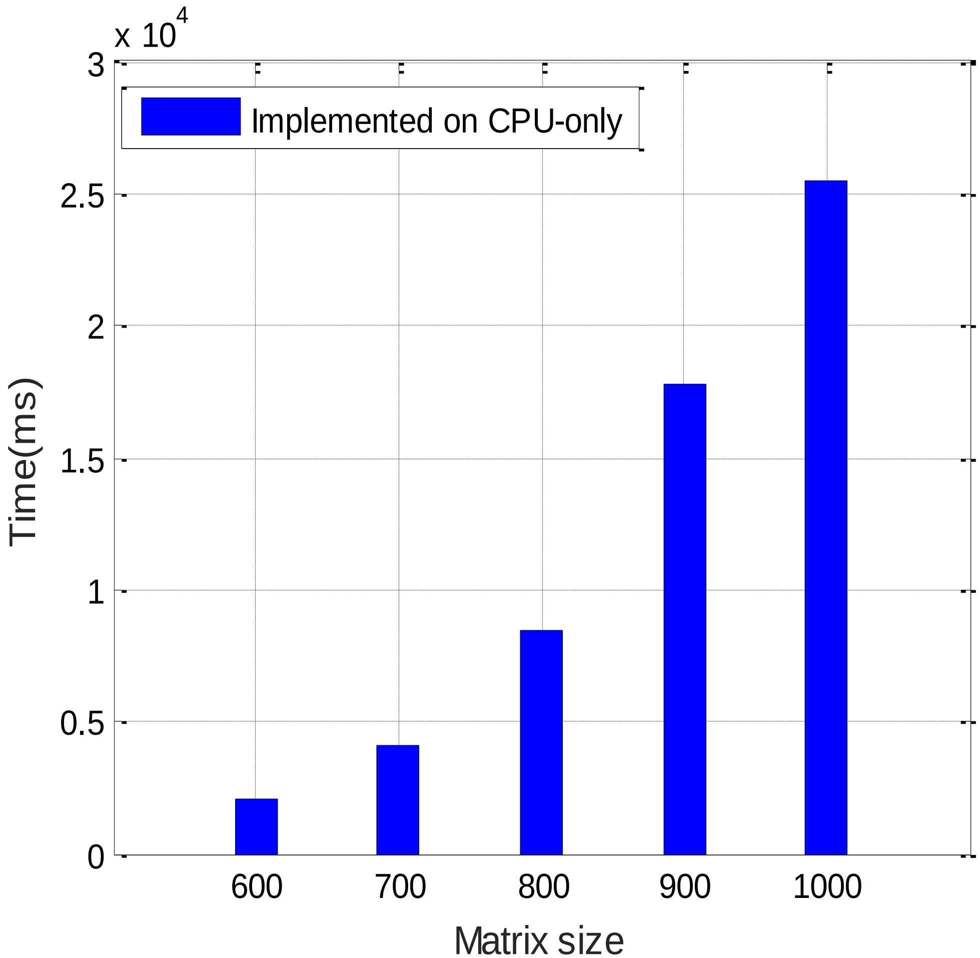
Simulation Results



- The x axis denotes the matrix size from 100×100 to 500 ×500
- The y axis denotes the execution time in milliseconds of the algorithm implemented on CPU-only

Fig. 8: Execution times in milliseconds of the algorithm implemented on CPU-only

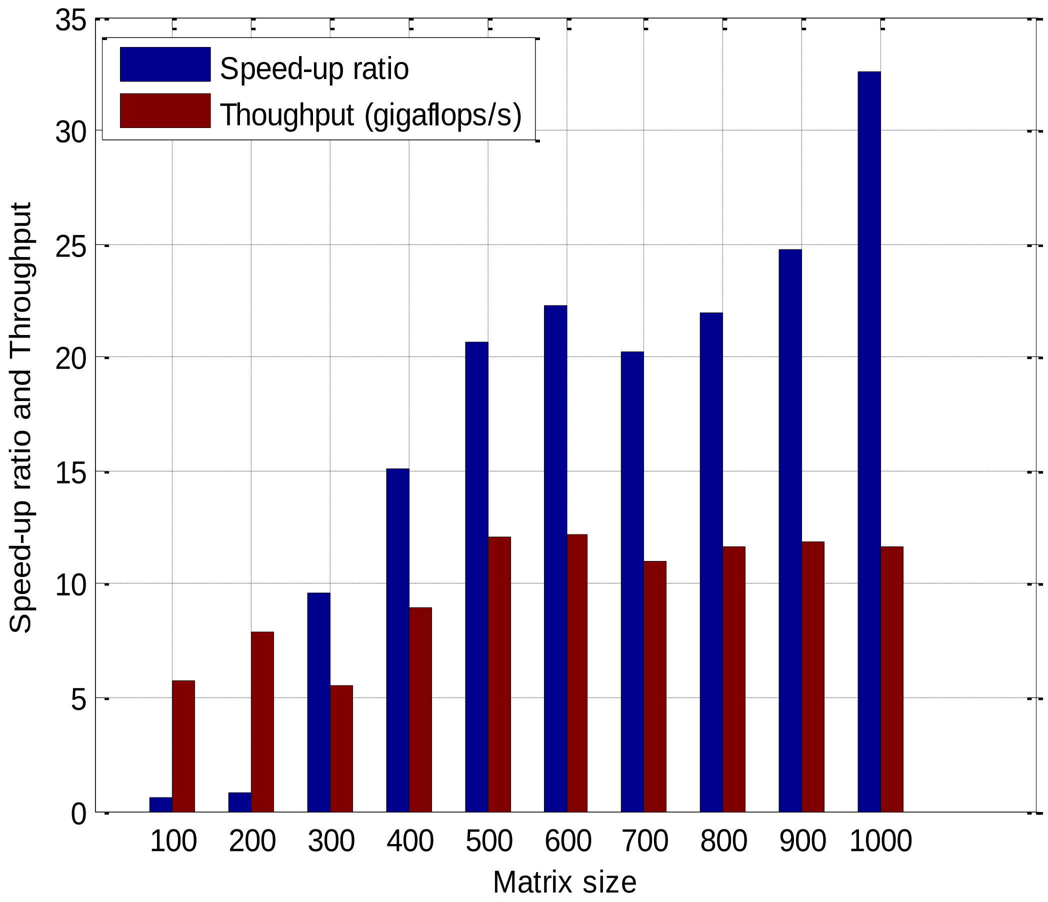
Simulation Results



- The x axis denotes the matrix size from 600×600 to 1000 ×1000
- The y axis denotes the execution time in milliseconds of the algorithm implemented on CPU-only

Fig. 9: Execution times in milliseconds of the algorithm implemented on CPU-only

Simulation Results



- The throughput could be more than 11 gigaflops/s when matrix dimension is larger than 500×500, and run at up to 12.14 gigaflops/s for some configurations.
- The speedup ratio could be 20x for matrix larger than 500×500, and up to around 32.62x for some configurations in our implementation

Fig. 10: Speed-up ratio and throughput of the algorithm implemented on CUDA

Conclusion

• A fast parallel matrix inversion algorithm was designed and implemented on the heterogeneous multicore architecture.

• Parallel execution models were designed called by *Kernel1* and *Kernel2*.

•The throughput could be more than 11 gigaflops/s when matrix dimension is larger than 500×500, and run at up to 12.14 gigaflops/s for some configurations.

•The speedup ratio could be 20x for matrix larger than 500×500, and up to around 32.62x for some configurations in our implementation.

Reference

[1] Sanders, Jason, and Edward Kandrot. CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley Professional, 2010.

[2] Zhang Shu, and Chu Yanli. High performance computing of GPU by CUDA , China Pub. DynoMedia Inc., 2009.

[3] Ries, Florian, et al. "Triangular matrix inversion on graphics processing unit." High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on. IEEE, 2009.

[4] Guerrieri, R., Tommaso De Marco, and F. Ries. "Triangular matrix inversion on heterogeneous multicore systems." IEEE Transactions on Parallel & Distributed Systems 1 (2012): 177-184.

[5] Sharma, Girish, Abhishek Agarwala, and Baidurya Bhattacharya. "A fast parallel Gauss Jordan algorithm for matrix inversion using CUDA." Computers & Structures 128 (2013): 31-37.

Reference

[6] Ezzatti, Pablo, Enrique S. Quintana-Orti, and Alfredo Remon. “High performance matrix inversion on a multi-core platform with several GPUs.” Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on. IEEE, 2011.

[7] Golub, Gene H., and Charles F. Van Loan. Matrix computations. Vol. 3. JHU Press, 2012.

The End

Thanks for your attention!