Memory-Efficient and Stabilizing Management System and Parallel Methods for RELION Using CUDA and MPI

Jingrong Zhang[1,2], Zihao Wang[1,2], Yu Chen[1,2], Zhiyong Liu[1], and Fa Zhang[1](✉)

[1] High Performance Computer Research Center, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China
{zhangjingrong,wangzihao,chenyu,zyliu,zhangfa}@ict.ac.cn
[2] University of Chinese Academy of Sciences, Beijing, China

Abstract. In cryo-electron microscopy, RELION has been proven to be a powerful tool for high-resolution reconstruction and has quickly gained its popularity. However, as the data processed in cryoEM is large and the algorithm of RELION is computation-intensive, the refinement procedure of RELION appears quite time-consuming and memory-demanding. These two problems have become major bottlenecks for its usage. Even though there have been efforts on paralleling RELION, the global memory size still may not meet its requirement. Also as by now there is no automatic memory management system on GPU (Graphics Processing Unit), the fragmentation will increase with iteration. Eventually, it would crash the program. In our work, we designed a memory-efficient and stabilizing management system to guarantee the robustness of our program and the efficiency of GPU memory usage. To reduce the memory usage, we developed a novel RELION 2.0 data structure. Also, we proposed a weight calculation parallel algorithm to speedup the calculation. Experiments show that the memory system can avoid memory fragmentation and we can achieve better speedup ratio compared with RELION 2.0.

Keywords: cryoEM · RELION · CUDA · Performance tuning

1 Introduction

Single particle cryo-electron microscopy (CryoEM) uses images of randomly-oriented particles to reconstruct 3D density map [1]. Over the past decades, cryoEM is increasingly becoming a mainstream technology for studying the architecture of biological macromolecules [2]. RELION (REgularised LIkelihood OptimisatioN) [3,4] significantly increased possible resolution of reconstruction algorithm by adopting expectation maximization (EM) algorithm. By now, many cryoEM structures were determined to a resolution better than 5$\mathring{A}$ using RELION [2,5–7].

However, expectation maximization (EM) algorithms [8] need to integrate over the hidden variables in the expectation step, which requires large amounts of computing time [9] and memory. Taking the EM algorithm used in RELION as example, calculating one dataset usually takes days of time. In 3D refinement stage, the test data used in the tutorial of RELION [10] with 6,496 β-galactosidase $128 * 128$ particles. It costs more than 2 GB memory per process and 40 CPU hours, even though this dataset is really small.

To accelerate RELION, we must accelerate the calculation of EM algorithm. In RELION, the calculation of weights takes a major part of the processing time. However, even though RELION 2.0 has been implemented on GPU, multi-threads are launched. The calculation of weights still take up a large part of calculation time. In this step, a lot of global memory accessing occurrs. To solve this problem we developed a weight calculation parallel algorithm. By making use of the shared memory on GPU, we can dramatically reduce the time of global memory accessing thus shortening its processing time.

Like many other softwares used in cryoEM, in order to cope with the time-consuming problem, RELION adopts parallel technology on Graphics Processor Units (GPUs) to speedup the processing procedure [11]. However, the stability and performance of GPU relies heavily on its hardware characteristics. For memory-intensive applications like RELION, memory-related code may dramatically influence the performance. Also there is not a sophisticated memory management system to handle the fragmentation problem. Fragmentation is generated during the randomly allocating and de-allocating operation. Fragments divide the successive memory into several pieces of memory. When the program try to allocate a large piece of memory, even though the total available memory is sufficient, the allocation can still fall because of the fragmentation. However this severe problem has not caught enough attention. By now, there still is no stable memory management strategy introduced in RELION parallelization. For example, we use NVIDIA K20c to process one dataset EMPIAR-10028 with $360 * 360$ particle size, "Out of Memory" error occurs during later iterations. Facing this condition, we developed a stabilizing memory manage system to ensure the robustness of program, especially for large dataset. In this system, we avoid fragmentation based on the characteristics of iterative methods. Also our method can be applied to other iterative methods.

As the total memory consumption of RELION exceeds the memory of GPU. Reducing memory consumption is of great importance. Weight array is the most memory demanding data structure. We analysed the data structure of weight array carefully and find out that we can reduce the memory requirements of weight array by redesign its structure delicately.

In summarize, in this work, we designed a memory-efficient and stabilizing management system to guarantee the robustness of our program and the efficiency of GPU memory usage. Then to reduce the memory usage, we developed a novel data structure. Also, we developed a weight calculation parallel algorithm to speedup the calculation.

Our paper is organized as follows: in Sect. 2, we introduce the related work; in Sect. 3, we introduce our stabilizing management system; in Sect. 4 a memory-efficient data structure is shown; in Sect. 5, we present the weight calculation parallel algorithm. The results of experiment are presented in Sect. 6. After that we conclude the paper.

2 Related Work

In this section we will introduce the basic algorithm flow of RELION and its accelerating work.

RELION adopts a modified adaptive expectation maximization (EM) algorithm based on the work of Tagare et al. [8]. Instead of assigning the best orientation for each particle, RELION generates a probability distribution over all orientations for each particle. We use $\Gamma_{i\phi}^{(n)}$ to indicate the posterior probability of orientation assignment ϕ for the ith image in the nth iteration ($i \in [0, I]$). In particular, the orientations $\phi \in [0, \Phi]$ is in a five dimension domain, comprising 3 rotations and 2 translations, i.e. $\Phi = (rRot, rTilt, rPsi, tX, tY)$.

The expectation-maximization algorithm iteratively optimizes the structure through a two-step procedure. The first step is "Expectation", in this step, computer-generated projections (here we also call them re-projections) of the structure are compared with the real particle images, resulting in relative similarity information about the relevant orientations of the images. The second step is "Maximization", the images are combined with the prior information into a smooth, 3D reconstruction. Meanwhile, the power of the noise and the signal in the data are updated.

In step "Expectation", RELION implements a modified version of the adaptive expectation-maximization algorithm. It owns two times finer procedure. For each particle image i, in the first pass ($oversampling = 0$), $\Gamma_i^{(n)}$ is evaluated over the entire domain using a relatively coarsely sampled grid. RELION selects the sub-domain of all orientations ϕ corresponding to the highest values of $\Gamma_{i\phi}^{(n)}$ that sum to 99.9% of the total probability mass on the coarse grid. Then, in the second pass ($oversampling = 1$), $\Gamma_i^{(n)}$ is evaluated only over the selected sub-domain using a finer grid which owns 32 times more sampling points than the coarse sampling grid [4].

Based on the analysis of GeRelion [12] and Relion 2.0 [11], the most time-consuming part is the step "Expection". And the most memory demanding data structure is weight array Γ.

GeRelion [12] is a GPU-accelerated version of RELION 1.4. Based on the high parallel ability of GPU, GeRelion adopts a four-level parallel model. Also to reduce the memory consumption, the sparse array is compacted to continuous vector and an aux vector was introduced to store the indexes of the weight array, which also will cause memory consumption. When the weight array is not sparse, this structure will take up more memory than original structure. Based on this consideration, we developed a new data structure which uses less memory.

RELION 2.0 [11] also accelerates the expectation-maximization algorithm on GPU. To reduce the memory consumption, it used single precision instead of double. Meanwhile, as the texture memory owns cache, RELION 2.0 stores the re-projection image on texture memory to accelerate the memory accessing. However, as the size of re-projection increase, one projection image will not fit in the texture memory cache. This will introduce frequently data swap in cache. The accelerating performance will be significantly weaken. Meanwhile, the calculation of weight array takes up majority time of program, in which accessing re-projection takes up a big part of the time. So we designed a different accelerate strategy to speedup this part.

3 Stack-Based Stabilizing Management System

RELION involved a lot of internal variables to perform its calculation. For many of these variables, their memory requirement may differ with sampling and particles, which means these variables require frequently and repeatedly allocation and de-allocation. Without memory management system on GPU, repeatedly allocation and de-allocation may introduce memory fragmentations. In later stage of iterations, these fragments can cause very serious consequence, i.e. application crash.

As mentioned above, there is no suitable memory management system by now. RELION 2.0 adopted linked list to collect memory fragments. The linked list can merge adjacent fragments. But it still can not prevent the generating of fragments.

Essentially EM algorithm is one of those iterative methods. Within each iteration, there is a loop for processing each particle. When processing one particle, there are variables allocating and de-allocating during the calculation for each orientation. So the lifetime of different variables can be classified into four types, global scope, iteration scope, during processing one particle (job scope) and during processing one orientation. As the lifetime is quite regular, we can control them using characteristic. The corresponding operation is shown Fig. 1. At the beginning of the whole program, we first allocate variables with global scope. While at the beginning of each iteration, we allocate memory for variables of iteration scope. Then, we allocate variable used in each job of processing one particle. After one job completed, the variables of job scope are released. At the end of one iteration, the variables of iteration scope are released. And at the end of program, variables of global scope are released. As the variables allocated last will alway be freed first. Based on this rule, we can build a stack based memory management system thoroughly to avoid memory fragments.

4 Memory-Efficient Data Structure

As mentioned above, the memory requirements of weight array are quite large, especially for the fine sampling pass. And, as mentioned, the global memory of

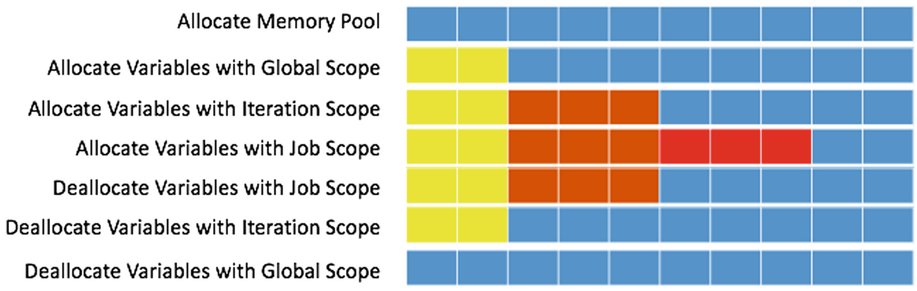
Allocate Memory Pool
Allocate Variables with Global Scope
Allocate Variables with Iteration Scope
Allocate Variables with Job Scope
Deallocate Variables with Job Scope
Deallocate Variables with Iteration Scope
Deallocate Variables with Global Scope

Fig. 1. Memory consumption on host.

GPU is quite limited. So more efficient data structure to store the weight array is of great importance.

By analyzing the data structure of weight array carefully, we find out that we can reduce the memory requirements of weight array by redesigning its structure delicately. In the first pass, all possible assignments are calculated. But before the second pass, RELION selects assignments with high probability. In the second pass, only assignments with high probability will be considered with fine sampling steps. Because the adoption of fine sampling, the size of weight array in the second pass is 32 times (the multiplication of 8 orientations and 4 translations) as large as that in the first pass. While because the selecting step, most of the assignments are neglected, only a few elements in weight array are non-zero.

Obviously, the storage of neglected assignments wastes a lot of memory. In our compressed structure, we can only store the non-zero value and their corresponding subscript. We observed that the assignments in first pass correspond to 32 assignments in the second pass. The weight of these 32 assignments must be stored in weight array successively. This trait can be used to reduce the size of aux index array. Instead of storing the column subscript of each non-zero element, only the first subscript of the 32 assignments will be stored.

The conversion of compressed data structure and original structure is shown in Fig. 2. The upper side of this figure shows the original data structure. The upper left array indicates the weight array, the upper right array is its corresponding array, this array indicates which assignments should be used in the second pass.

The bottom side of this figure shows our compressed data structure. The array in the middle stores all the non-zero elements. Below this array, we show the subscript. For each group of successive 32 elements, we only store the subscript of the first assignment. The array at the bottom stores the information about the number of non-zero elements for each particle. For convenient data accessing, we use the start point of each particle to show this information. For each particle, the number elements is N, the non-zero elements in matrix is n. For each particle, $size\_original$, $size\_conpressed$ and $size_new$shows the number of bits cost in

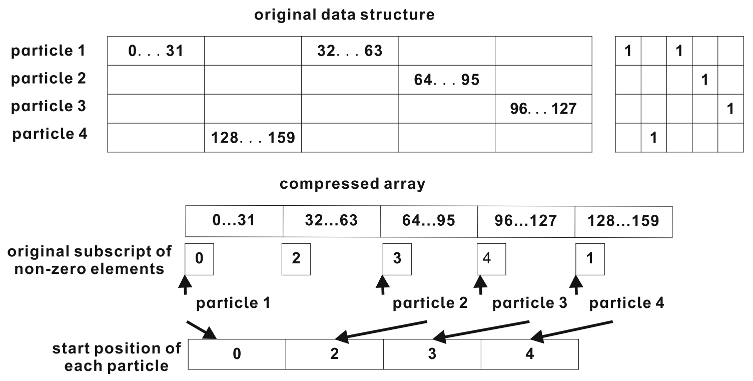

Fig. 2. Memory-efficient data structure.

original version of RELION, general compressed method and our method. We can easily know, when $n > 0.67N$, the memory consumption of general compressed method will exceed original version. However the memory consumption of our method won't exceed original version unless $n > 0.98N$.

$$\begin{aligned}&size\_original = N * (32 * 8)\\&size\_conpressed = n * (32 * (8 + 4))\\&size\_new = n * (32 * 8 + 4)\end{aligned} \tag{4.1}$$

5 Weight Calculation Parallel Algorithm

In step Get Squared Differences (GSD), it calculates the actual squared difference term of the gaussian probability function. Its main part can be written in Formula 5.1:

$$\varGamma_{i\phi}^{(n)} = \frac{1}{2}\sum_{k=1}^{N}((Re(p_{\Phi k}) - Re(X_{ik}))^2 + (Im(p_{\Phi k}) - Im(X_{ik}))^2) \tag{5.1}$$

where $p_{\phi k}$ is the *kth* component of the projection at orientation assignment ϕ. X_{ik} is the *kth* component of the particle at the *ith* image.

Different from later iterations, in the first iteration, this step calculates the normalized cross-correlation coefficient. Although the formulas of these two conditions are different. Their computational complexity is of the same order. As for each particle, we need to calculate the weight for each assignment. This means each projection image should compare with each particle image, accordingly. Its calculation can be formalized as following:

$$\begin{aligned}
\begin{bmatrix} \Gamma_{i_1\phi_1} & \Gamma_{i_1\phi_2} & \dots & \Gamma_{i_1\phi_m} \\ \Gamma_{i_2\phi_1} & \Gamma_{i_2\phi_2} & \dots & \Gamma_{i_2\phi_m} \\ & \dots & & \\ \Gamma_{i_k\phi_1} & \Gamma_{i_k\phi_2} & \dots & \Gamma_{i_k\phi_m} \end{bmatrix} &= \begin{bmatrix} \sum\limits_{k=1}^{N} f_{i_1,\phi_1,k} & \dots & \sum\limits_{k=1}^{N} f_{i_1,\phi_m,k} \\ \sum\limits_{k=1}^{N} f_{i_2,\phi_1,k} & \dots & \sum\limits_{k=1}^{N} f_{i_2,\phi_m,k} \\ & \dots & \\ \sum\limits_{k=1}^{N} f_{i_u,\phi_1,k} & \dots & \sum\limits_{k=1}^{N} f_{i_u,\phi_m,k} \end{bmatrix} \\
&= \begin{bmatrix} \sum\limits_{k=1}^{n_1} f_{i_1,\phi_1,k} & \dots & \sum\limits_{k=1}^{n_1} f_{i_1,\phi_m,k} \\ \sum\limits_{k=1}^{n_1} f_{i_2,\phi_1,k} & \dots & \sum\limits_{k=1}^{n_1} f_{i_2,\phi_m,k} \\ & \dots & \\ \sum\limits_{k=1}^{n_1} f_{i_u,\phi_1,k} & \dots & \sum\limits_{k=1}^{n_1} f_{i_u,\phi_m,k} \end{bmatrix} + \dots + \begin{bmatrix} \sum\limits_{k=n_c}^{N} f_{i_1,\phi_1,k} & \dots & \sum\limits_{k=n_c}^{N} f_{i_1,\phi_m,k} \\ \sum\limits_{k=n_c}^{N} f_{i_2,\phi_1,k} & \dots & \sum\limits_{k=n_c}^{N} f_{i_2,\phi_m,k} \\ & \dots & \\ \sum\limits_{k=n_c}^{N} f_{i_u,\phi_1,k} & \dots & \sum\limits_{k=n_c}^{N} f_{i_u,\phi_m,k} \end{bmatrix} \\
&= \begin{bmatrix} \Gamma_{i_1\phi_1}(1,n_1) & \dots & \Gamma_{i_1\phi_m}(1,n_1) \\ \Gamma_{i_2\phi_1}(1,n_1) & \dots & \Gamma_{i_2\phi_m}(1,n_1) \\ \Gamma_{i_k\phi_1}(1,n_1) & \dots & \Gamma_{i_k\phi_m}(1,n_1) \end{bmatrix} + \dots + \begin{bmatrix} \Gamma_{i_1\phi_1}(n_c,N) & \dots & \Gamma_{i_1\phi_m}(n_c,N) \\ \Gamma_{i_2\phi_1}(n_c,N) & \dots & \Gamma_{i_2\phi_m}(n_c,N) \\ \Gamma_{i_k\phi_1}(n_c,N) & \dots & \Gamma_{i_k\phi_m}(n_c,N) \end{bmatrix}
\end{aligned} \tag{5.2}$$

In formula 5.2, the first line shows the common way of calculation. The calculation of weight $\Gamma_{i\phi}$ is independent of $\Gamma_{i'\phi'}$. Considering the structure of GPU card, accessing time to global memory can be 100 times greater than shared memory [13]. So reducing the global memory accessing can dramatically improve the time performance of applications. Here $\Gamma_{i_1\phi_1}(n,n') = \sum\limits_{k=n}^{n'} f_{i_1,\phi_1,k}$ means the partial results of $\Gamma_{i_1\phi_1}$. The second and third lines of formula 5.2 show that we can calculate part of weights first and sum these intermediate results to get the final results. Generally speaking, instead of calculating $\Gamma_{i\phi}$ independently, we calculate a group of $\Gamma_{i\phi}(n,n')$ together. First, threads load a portion of projection $p_\phi(n,n')$ and particle $X_i(n,n')$ into shared memory, where we can access them much more quickly. Then, the partial results of $\Gamma_{i_1\phi_1}$ can be calculated using shared memory.

As shared memory are available for all threads of one block, we can reuse the data and reduce the data accessing time. Take a $32 * 32$ block as an example, the global memory accessing time can be reduced by 32 times. As step "Get Squared Differences (GSD)" take majority time in expectation step, reducing global memory accessing can efficiently reduce the processing time of this step.

6 Experiments

6.1 Environment and Test Dataset

Generally, we adopts two dataset to perform our experiment. First, we use the dataset used in the tutorial of RELION, owning 6,496 β-galactosidase particles (EMPIAR-10017 [10]). The size of each particle is $128 * 128$. Its experiments are carried out on a machine running the Ubuntu operating system 64-bit with an Intel(R) Xeon(R) CPU E5630 at 2.53 GHz. The GPU card is NVIDIA Tesla K20c, with 2496 stream processors and 5 GB global memory.

Another dataset is "plasmodium falciparum 80S ribosome bound to the anti-protozoan drug emetine", which owns 100,000 particles with pixels size 360 * 360 (EMPIAR-10028 [2,5]). Its experiments are carried out on a machine running the Ubuntu operating system 14.04, 64-bit with an Intel(R) Xeon(R) CPU E5-2680 v3 at 2.50 GHz. The GPU card is NVIDIA GeForce GTX Titan X, with 3072 stream processers and 12288 MB global memory. In our test, we use 5 MPI processes (one thread for each process), each slave process owns one GPU.

6.2 Performance of Stack-Based Memory Management System

In this section, we show the detailed working process of stack-based memory management system. To do so, we record the variation of used memory when running the dataset EMPIAR-10028. As shown in Fig. 3, the top pointer indicates that memory before this pointer is used memory. Memory with larger address is continuous free memory. In this method, the fragment are avoided. In this figure, we can see that the memory operation indeed shows the characteristic of iterative algorithm. But as marked with red rectangle, the memory operation owns randomness. Also the randomness doesn't influent the working process of stack-based memory.

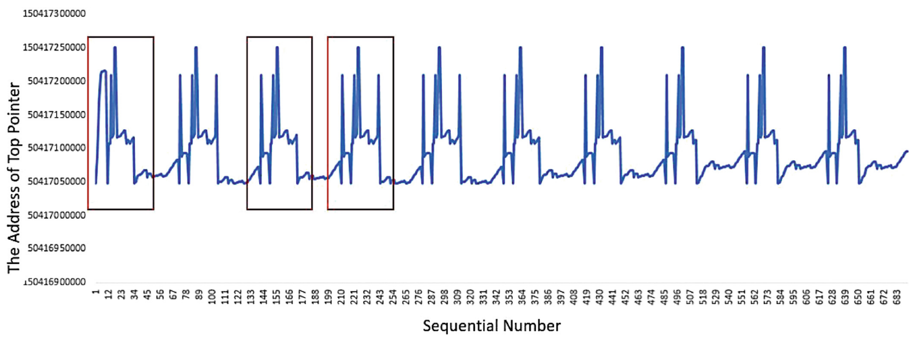

Fig. 3. The top pointer of used memory (Color figure online)

6.3 Memory Consumption Optimization

We modified the data structure of array $\Gamma^{(n)}$, which saves a lot of memory. For dataset EMPIAR-10017, we show the memory consumption of double precision version. As discussed in RELION 2.0 the single precision version doesn't adversely affect results. We still keeps the double version for users demanding more accurate calculation. As the size of single-precision floating-point number is half of double ones, the memory of single precision is just half of the double version.

We record the max memory consumption of array $\Gamma^{(n)}$ for one particle to show the performance. As shown in Fig. 4. The most memory demanding array appears in the middle of the iterations. In these iterations, our structure significantly reduce the memory requirements. The upper bound of the size of array $\Gamma^{(n)}$ is mainly the sampling number in first pass in both float mode and double mode. The memory consumption of array $\Gamma^{(n)}$ is generally 1/32 of the old version.

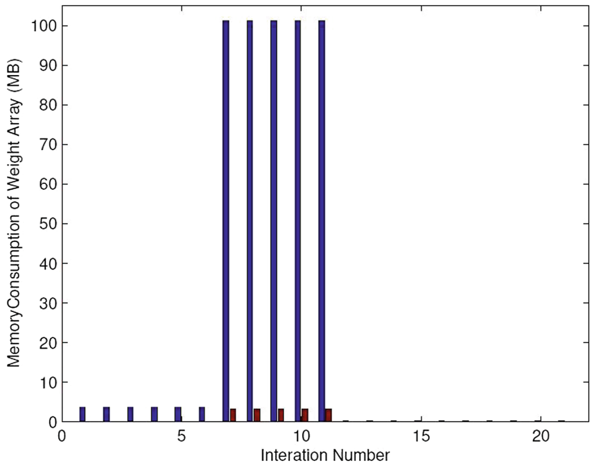

Fig. 4. Maximum memory consumption of array "$\Gamma^{(n)}$" for one particle for β-galactosidase in double mode.

6.4 The Speedup Ratio of Weight Calculation Parallel Algorithm

In this section, we test the performance of our weight calculation parallel algorithm. Different from the texture memory, the performance of our method is not restricted by the total size of re-projection. We compared our method with RELION 2.0, both methods are GPU parallelized. The speedup ratio shows the improvement by our method. We tested the methods on dataset EMPIAR-10028. In Fig. 5, the horizontal axis indicates the iteration number. As the size and number of re-projection image may vary during iterating. We use different iterations to indicate different processing size of data. As we can see, the performance of two methods is similar. However, in iterations of larger dataset, the advantage of reusing shared memory shows up.

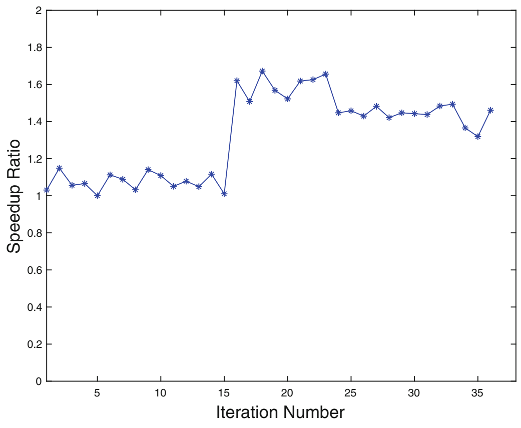

Fig. 5. Speedup ratio of weight calculation parallel algorithm.

6.5 The General Speedup Ratio

After analyzing the speedup ratio, we summarize the total speedup ratio for each iteration. As our program is based on the MPI version of RELION, we don't modify the master-slave MPI parallel model. In this experiment, we compare the CUDA+MPI version with MPI version using the same number of processes and threads. Different from the MPI version, each slave process owns one GPU. From Fig. 6, we can see that the double precision version can reach 80x speedup ratio for dataset EMPIAR-10028 in the most time-consuming iterations.

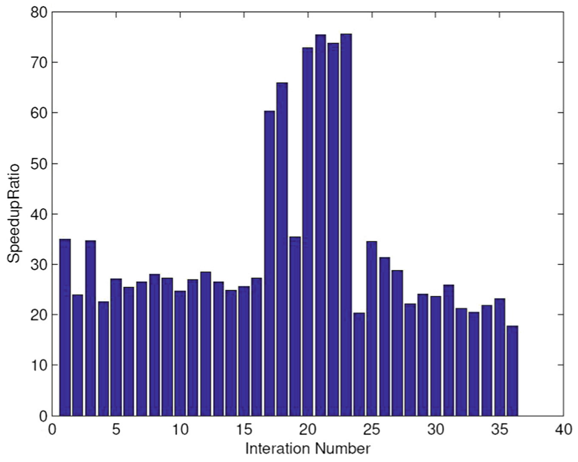

Fig. 6. The general speedup ratio.

To compare the performance of RELION 2.0 and our method, we used dataset EMPIAR-10017 with single precision version. We compare them with the original MPI version and record the speedup ratio. Figure 7 shows the speedup ratio respectively. Generally speaking, our version owns better speedup ratio. Especially in the most time consuming iteration, our version can reach 105x speedup ratio, while the speedup ratio of RELION 2.0 is 96x.

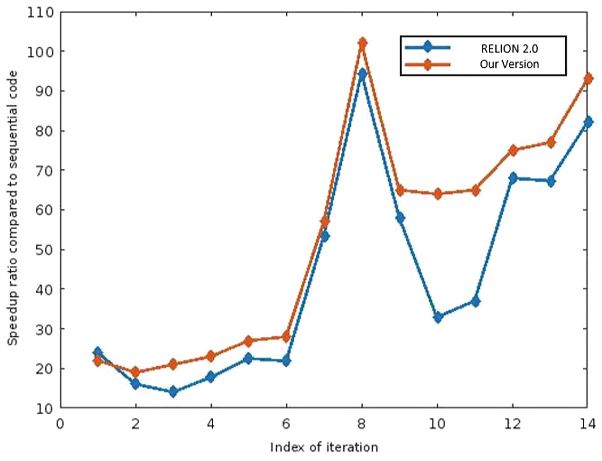

Fig. 7. Compare the performance of RELION 2.0 with our method.

7 Conclusion

In this work, we proposed stack-based memory management system, which can ensure the program to proceed without "Out of Memory" error and enable it to process dataset with large particle size. Then we introduced our compressed data structure which will dramatically reduce the memory consumption. After that, we developed the weight calculation parallel algorithm using shared memory. Its performance won't be affected by the size of re-projection. The results of experiment show that the memory system can avoid memory fragments. And we have achieved better speedup ratio compared to RELION 2.0.

Acknowledgments. This research is supported by the National Key Research and Development Program of China (2017YFA0504702), the NSFC projects Grant No. U1611263, U1611261, 61472397, 61502455, 61672493 and Special Program for Applied Research on Super Computation of the NSFC-Guangdong Joint Fund (the second phase).

References

1. Li, X., Grigorieff, N., Cheng, Y.: GPU-enabled FREALIGN: accelerating single particle 3D reconstruction and refinement in fourier space on graphics processors. J. Struct. Biol. **172**(3), 407–412 (2010)
2. Bai, X., McMullan, G., Scheres, S.H.: How cryo-EM is revolutionizing structural biology. Trends Biochem. Sci. **40**(1), 49–57 (2015)

3. Scheres, S.H.: A Bayesian view on cryo-EM structure determination. J. Mol. Biol. **415**(2), 406–418 (2012)
4. Scheres, S.H.: RELION: implementation of a Bayesian approach to cryo-EM structure determination. J. Struct. Biol. **180**(3), 519–530 (2012)
5. Wong, W., Bai, X., Brown, A., Fernandez, I.S., Hanssen, E., Condron, M., Tan, Y.H., Baum, J., Scheres, S.H.: Cryo-EM structure of the plasmodium falciparum 80S ribosome bound to the anti-protozoan drug emetine. Elife **3**, e03080 (2014)
6. Amunts, A., Brown, A., Bai, X., Llácer, J.L., Hussain, T., Emsley, P., Long, F., Murshudov, G., Scheres, S.H., Ramakrishnan, V.: Structure of the yeast mitochondrial large ribosomal subunit. Science **343**(6178), 1485–1489 (2014)
7. Liao, M., Cao, E., Julius, D., Cheng, Y.: Structure of the TRPV1 ion channel determined by electron cryo-microscopy. Nature **504**(7478), 107–112 (2013)
8. Tagare, H.D., Barthel, A., Sigworth, F.J.: An adaptive expectation-maximization algorithm with GPU implementation for electron cryomicroscopy. J. Struct. Biol. **171**(3), 256–265 (2010)
9. Sigworth, F.J., Doerschuk, P.C., Carazo, J., Scheres, S.H.W.: Maximum-likelihood methods in cryo-EM. Part i: theoretical basis and overview of existing approaches. Methods Enzymol. **482**, 263 (2010)
10. Scheres, S.H.: Single-particle processing in RELION-1.3 (2014)
11. Kimanius, D., Forsberg, B.O., Scheres, S.H., Lindahl, E.: Accelerated cryo-EM structure determination with parallelisation using GPUs in RELION-2. eLife **5**, e18722 (2016)
12. Su, H., Wen, W., Du, X., Lu, X., Liao, M., Li, D.: Gerelion: GPU-enhanced parallel implementation of single particle cryo-EM image processing. bioRxiv 075887 (2016)
13. Corporation N.: CUDA in C best practices guide. NVIDIA Corporation (2016)