Single-particle 3D Reconstruction from Cryo-Electron Microscopy Images on GPU

Guangming Tan, Ziyu Guo[*], Mingyu Chen, Dan Meng
Key Laboratory of Computer System and Architecture
Institute of Computing Technology, Chinese Academy of Science
Beijing, China
{tgm,guoziyu,cmy,md}@ncic.ac.cn

ABSTRACT

Single-particle 3D reconstruction from cryo-electron microscopy (cryo-EM) images is a kernel application of biological molecules analysis, as the computational requirement of which is now beyond PetaFlop for a high-resolution 3D structure. In this paper, we quantitatively analyze the workload, computational intensity and memory performance of the application, parallelize it on an emerging multicore architecture GPU-CUDA. Further we apply a percolation technique to decouple computation with memory operations and orchestrate thread-data mapping to reduce the overhead off-chip memory operations. Finally we tested our optimization strategy on a popular open-source package EMAN to GPU-CUDA, which achieves a relative speedup of about 10X to the original CPU-only EMAN. The experimental results also show that the proposed percolation programming greatly improves utilization of memory bandwidth and floating-point units.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance attributes; C.1.4 [**Computer Systems Organization**]: Processor Architectures—*Parallel Architectures*; J.3 [**Computer Application**]: Life and Medical Sciences

General Terms

Performance, Algorithms

Keywords

cryo-EM, GPU, Many-core, Performance tuning

---

[*]Z.Guo makes equal contributions to the work with the first author.

1. INTRODUCTION

Current sequencing methods are producing genomic data at an astounding speed, leaving large amount of research to be done on the structure and functions of the gene expression product, i.e proteins and other biological molecules. The function of a protein molecule is largely determined by its 3D structure, and a number of methods have been developed to provide accurate measurement of such structures, among them are X-ray crystallography, NMR, and electron microscopy. X-ray crystallography provides the highest resolution, and has successfully revealed the structures of a number of molecular. On the other hand, it is difficult for many large molecular compounds and hydrophobic molecules to form the needed crystal. Moreover, the bound forces within lattices will affect the interactions between molecules. Thus the application of this method is limited. NMR is best for measuring water-soluble protein, and small molecules that do not crystallize. However, the major limiting factor is data collection, which often requires several weeks' work on a spectrometer. Cryo-electron micocopy (cryo-EM) [19] had no such limitations and is ideal for large molecules and assemblies in the range of hundreds to thousands of kilo Daltons. Electron microscopy can be applied to various kinds of particles, from protein crystal to non-uniform particles like virus and ribosome. Data collection can be accomplished relatively fast. Samples are embedded in vitreous ice, which preserves the original biological relevant conformation and makes it possible to capture transient states. Therefore, it is suited to process molecules that are problematic with X-ray crystallography. In the past decade, electron microscopy is becoming widely used to address problems that are previously difficult for traditional crystallography.

The basic idea of cryo-EM 3D reconstruction [3] is to build a 3D model through a series of 2D projected images along the projected direction. The principle of this idea is *projection-slice theorem*. In mathematics, the projection-slice theorem in two dimensions states that the Fourier transform of the projection of a two-dimensional function onto a line is equal to a slice through the origin of the two-dimensional Fourier transform of that function which is parallel to the projection line. In cryo-EM technique, each microscopy image is a 2D image projected through different angle for the original body (i.e. protein). A Fourier transformation on these image generates a number of slice. If the number of slice is enough to retrieve information in 3D Fourier space. A inverse Fourier transformation is applied to build the final 3D model.

In fact, the 3D reconstruction based on 2D projection is challenging [17]. First, since amount of bio-molecule in samples has different angles following a random distribution, it is difficult to identify their correct angles. If a molecule's angle is not correctly computed, more iteration steps are needed to keep the final model converged. Second, in order to avoid spoiling the samples, low-dose of electron beam is often used to irradiate. Thus most of the electron microscopy images are of high-noise and low-contrast. Therefore, a large number of electron microscopy images are required to generate the 3D model with high resolution. For example, to get a 3D structure with resolution of less than $10\dot{A}$, the needed number of raw images are usually tens of thousands. On the other hand, as presented in Section 3 the 3D reconstruction algorithm is arithmetic intensive. The reconstruction is an iteration of model refinement. The basic refinement algorithm aligns the raw images with reference model. We estimate its computational requirement defined by floating-point operations for an usual case: the number of iteration $I = 10$, the number of raw images (particles) $N = 10^6$, the number of reference image $S = 10^2$, the pixel of images is $nx \times ny = 10^2 \times 10^2$, the averaged number of floating-point operations of refinement is $f = 10^2$. The number of floating-point for the 3D reconstruction is

$$\#Flop \geq I \times N \times S \times (nx \times ny) \times f \geq 10^{15} = 1PetaFlop \quad (1)$$

Over the past decade, the data amount in electron microscopy increases constantly with the aim to produce images with higher resolution. An experimental/computational technique known as single-particle analysis has been growing rapidly in popularity. Currently the commonly used single-particle software includes EMAN [15],SPIDER [8] and IMIRS [14]. Although they provide convenient ways to perform 3D reconstruction, yet the work is very time-consuming on general CPU due to its high demand on computing capability. In order to cope with these increasing computational efforts, most of these software packages have implemented parallel reconstruction algorithms to be used on various conventional parallel computing systems [15, 8, 2, 22]. Electron microscopy data are typically well suited for parallelization, since both images and processes can be treated independently, e.g. electron micrographs can be backprojected in a common three-dimensional (3D) image independently. The parallelization of these applications therefore results in an almost linear decrease of the computational time, as well as a linear increase in the hardware and power costs.

We are witnessing the emergence of multi/many-core in high performance computing. Since the newest version of graphic processing units (GPU) allow extremely high floating-point arithmetic throughput, it has been widely used to accelerate scientific applications such as general (numerical) computation [10, 5], data mining [11, 21], image processing [13] and so on. Recently, GPU communities have introduced general-purpose programming environment–Nividia CUDA [12] and AMD Streaming [7] to facilitate high productivity of parallel programming on GPUs. Daniel et.al. [4] has implemented another popular 3D reconstruction algorithm on the previous generation of GPU without the support of CUDA. The motivations of this work are the efficient execution of high resolution 3D reconstruction algorithm on a GPU-CUDA architecture and the identification of general performance tuning strategies on GPU-like many-core architecture. In this work we apply a percolation technique to optimize off-chip memory performance for the 3D reconstruction algorithm. The idea of percolation is to decouple computation with memory operation and orchestrate thread-data mapping to reduce the overhead off-chip memory operations. Our proposed algorithm increases utilization of memory bandwidth and improve the floating-point performance greatly.

The rest of this paper is organized as follows. Section 2 describes a typical single-particle 3D reconstruction algorithm. The rationality of porting the reconstruction algorithm to GPU is explained by a detailed workload characterization in section 3. Section 4 focuses on the percolation programming on several time-critical kernels under the CUDA model. The experimental results are reported in section 5. Section 6 concludes this paper.

2. SINGLE-PARTICLE 3D RECONSTRUCTION ALGORITHM

In this section we describe the single-particle 3D reconstruction algorithm, which is implemented in the software package EMAN. Figure 1 shows the flowchart of the cryo-EM image analysis process.
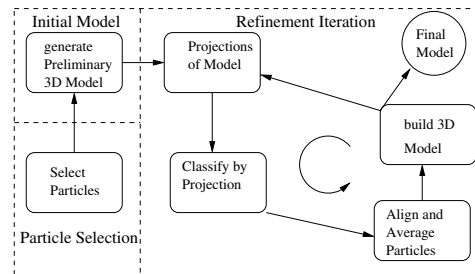

Figure 1: Flowchart of cryo-EM image analysis [15]

Performing a single-particle reconstruction is a three-stage process shown in Figure 1. First, the particles must be selected from scanned raw micrographs or CCD (charge-coupled detector) frames. In most of reconstruction approaches it is called *boxer*, which is usually a semiautomatic selection procedure. Second, because the refinement procedure in single-particle 3D reconstruction algorithm is model-based, the boxed-out particles are used to generate a preliminary 3D model, which are required to allow the refine loop to converge. Finally, the preliminary model is used as the starting point for the main refinement loop. As with other structural techniques like NMR and X-ray crystallography, the experimental data in cryo-EM cannot be directly inverted to generate an optimal 3D model. A roughly preliminary model is iteratively refined against the data. True convergence is achieved when the model remains unchanged for several successive iterations. The refinement loop outlined in the right half of Figure 1 begins by generating a set of N projections with uniformly distributed orientations. Then the particles are classified into different classes by the projections (*particle classification*) and aligned to generate an average model (*class averaging*). Finally, the class average with assigned Euler angles are used to construct a new 3D model for the next round of refinement. Figure 2 illustrates pesudocode of the refinement algorithm. Generally, the particle classification does an excellent job of assigning particles to the correct class. However, in some cases the low

signal-to-noise ratio in the electron micrographs will lead to misassignment of some particles. Fortunately the incorrectly assigned particles will be eliminated by the class averaging procedure. Note that a common procedure of particle clas-

---

SINGLE-PARTICLE 3D RECONSTRUCTION ALGORITHM
1. **while** model T not converged
2. generating M projections of T
   //particle classification
3. **for** each $i \in N$ particles in images
4. **for** $j \in M$ projections
   //rotationally and translationally aligns each particles
   //to projected reference
5. RTFAlign(i, j)
6. **endfor**
7. **endfor**
   //class averaging
8. **for** each particles i with enough similarity value
9. RTFAlign(i, $average_i$)
10. **endfor**
11. InitialAve() // generate an initial class average
12. **for** each each particles i with enough similarity value
13. RTFAlign(i, $average_{initial}$)
14. **endfor**
15. Remove() // remove particles with less similarity
16. build3D() //build 3D model
17. **endwhile**

Figure 2: Outline of single-particle 3D reconstruction algorithm. In some cases line12-15 may be iterated for several times.

ROTATIONAL AND TRANSLATIONAL ALIGNMENT ALGORITHM
INPUT: particle i, horizontal flip i', reference
// step 1. rotational alignment
1.1. **MCF(i)** // autocorrelation function
1.2. **MCF(i')** // flip's autocorrelation function
1.3. **unwrap()** // rectangular coordination to polar one
1.4. **CCFX()** // cross correlation function on x-dimension
1.5. **Rotate()** // rotates with the best rotation angle

//step 2. translational alignment
2.1. **CCF()** //cross correlation function with reference
2.2. **Translate()** //translates image with the maximal CCF

//step 3. score and select the most similarity one
3.1. **dot()** //scores the rotational&translational alignment

---

Figure 3: Outline of RTFAlign algorithm.

sification and class averaging is rotational and translational alignment (RTFAlign in Figure 2), which is the main computational step of single-particle 3D reconstruction. Given a set of reference image (or projection), a particle calculates their rotation and translation vectors, which are used to identify the similarity. Figure 3 describes the algorithm for rotational and translational alignment. The alignment performs various calculations on the two dimensional image, which is stored in a 2D array data structure. The rotational alignment calculates autocorrelation and cross correlation using Fourier transformation. In order to get intensity correlation of the image, the rectangular coordination (x, y) of correlation function is transformed to a polar coordination

(r, θ) as in Equation 2, where nx and ny is the size of each dimension, d and dd stores the particle image in two coordinate system, respectively.

$$(x, y) \Leftarrow f(r, \theta, \frac{n_x}{2}, \frac{n_y}{2})$$
$$t = x - floor(x), \quad u = y - floor(y)$$
$$x' = x + t, \quad y' = y + u$$
$$dd[x', y'] = (1-t) * (1-u) * d[x, y] + t * (1-u) * d[x+1, y]$$
$$+ t * u * d[x+1, y+1] + (1-t) * u * d[x, y+1]$$
$$(2)$$

The cross correlation function needs to be accumulated along x-dimension. Let's denote C_i and C'_i to be the ith row of the original image and the rotated one. After Fourier transformation along the row data, we get $C_i = a_{i1}, a_{i2}, ..., a_{in}$ and $C'_i = a'_{i1}, a'_{i2}, ..., a'_{in}$. The summation along x-dimension is row vector f $(0 < j < \frac{n}{2})$:

$$f_j = \sum_{i=1}^{n} (a_{ij} * a'_{ij} + a_{i(n-j)} * a'_{i(n-j)})$$
$$f_{\frac{n}{2}+j} = \sum_{i=1}^{n} (a_{i(n-j)} * a'_{ij} - a_{ij} * a'_{i(n-j)})$$
$$(3)$$
$$f_0 = \sum_{i=1}^{n} a_{i0} * a'_{i0}$$
$$f_{\frac{n}{2}} = \sum_{i=1}^{n} a_{i \frac{n}{2}} * a'_{i \frac{n}{2}}$$

Then, the peak in correlation function indicating the best alignment angle is used to rotate the original image. The next step is to calculate a shift for a translational alignment. The shift is obtained by cross correlation function with reference image. Assume that the rotational and translational alignment get the a rotation angle θ and translation vector (dx, dy) for particle d, the reference image is r. We do four consecutive operations on the reference: (i) rotate and translate to $r1$; (ii) horizontal flip to $r2$; (iii) rotate with $\theta + \pi$ to $r3$;(iv) horizontal flip to $r4$. The score system computes the dot multiplication between the particle and each transformed reference, then selects one with the highest score.

3. WORKLOAD CHARACTERIZATION

More than 90% execution time of the reconstruction program is consumed by the rotational and translational algorithm (*RTFAlign*). In this section we present a detailed characterization of this procedure, focusing on computation, memory behavior and parallelism. The workload characterization motives our parallel implementation on a many-core processors like GPU. In the experiments, we use three images size in real applications with typical pixels: $96 \times 96, 256 \times 256, 512 \times 512$. Table 1 summarizes the profiling experiments for execution time, arithmetic intensity, branch ratio, working set and cache performance. We present the experimental results for one run of multiple RTFAlign executions in a refine iteration.

3.1 Arithmetic Intensity

Current multi/many-core technology integrates a large number of arithmetic units into one chip. The number of arithmetic operations of an application obviously should be a critical measure of performance on such a multi-core architecture. We use arithmetic intensity - the ratio between

Table 1: Workload characterization for RTFAlign. AI: Arithmetic Intensity. The profiling is collected from a single run of RTFAlign. The time is measured as millisecond.

function	size	time	AI	branch	memory	cache
MCF	96	0.74		20%	228KB	0.3%
	256	20.97	8.5	2.4%	1612KB	1.3%
	512	106.3		1.4%	6437KB	1.2%
CCFX	96	0.10		18%	95KB	0.3%
	256	1.74	4.3	1.5%	714KB	1.2%
	512	7.16		1.2%	2858KB	1.2%
CCF	96	0.35		13.4%	73KB	0.3%
	256	5.12	6.3	1.2%	524KB	1.5%
	512	20.65		1.8%	2097KB	1.3%
unwrap	96	0.11		8.7%	63KB	0.08%
	256	3.75	10.5	8.9%	476KB	0.1%
	512	15.79		17.2%	1905KB	0.1%
Rotate	96	0.12		5.4%	73KB	0.4%
	256	3.71	23.5	5.2%	524KB	0.9%
	512	14.18		12.7%	2097KB	2.4%
Translate	96	0.06		7.7%	73KB	1.1%
	256	1.59	0	7.3%	524KB	30%
	512	6.36		15.2%	2097KB	46%
dot	96	0.01		1.8%	73KB	1.1%
	256	0.28	0.67	1.8%	524KB	2.5%
	512	1.32		3.8%	2097KB	2.7%

arithmetic operations and the number of input and output words required to compute them in a kernel to evaluate how well it is adapted to many-core architecture. The third column in Table 1 is the arithmetic intensity of each kernel in RTFAlign algorithm. All but two kernel (*dot, Translate*) has more than one of arithmetic intensity. The vector dot multiplication's arithmetic intensity is less than 1 because it needs 2 arithmetic operations (one multiplication and one add) and 3 memory operations (two loads and one store). The translation only involves with memory movement. The feature of high arithmetic intensity seems to be well suited to many-core architecture like GPU. However, note that computations with branch instruction may not utilize the many arithmetic units are hindered by the serialization of SIMD pipeline in GPU multithread execution. The fourth column in Table 1 is the ratio of branch instruction execution in each kernel. Although several kernels has more than 10% branch instruction execution, an analysis of source codes indicates that most of the branch executions are out of loops, which means the branch instruction does not results in stall of SIMD on GPU. Besides, through orchestrating data-task mapping, most of branch instruction within a loop could be grouped together into one warp of CUDA.
Observation 1: RTFAlign algorithm is high arithmetic intensive.

3.2 Memory

Another feature of many-core architecture like GPU that they are configured with small shared on-chip memory. We examine detailed memory behavior in several ways. First, we measure the working set. As shown in the 6th column in Table 1 the memory usage of all kernels is proportional to square of pixel. Limited by current cryo-EM technology, the

pixel of most of images is less than 1024×1024. Thus, optimization for small shared on-chip memory could take advantage of the feature of small working set. Second, to quantify locality we sample cache access performance. The last column in Table 1 reports the L1 data cache miss rates on an AMD Barcelona processor. Most of kernel has good spatial locality, which is favorable to share data among threads. Note that *Translate* kernel has high cache miss rate when the pixel size is larger. The translational algorithm needs to translate a 2-D array from the left-bottom to right-top and the memory access strides is proportional to the size of one dimension. Therefore, the access pattern incurs a large number of conflict miss.
Observation 2: The instantaneous working set of RTFAlign algorithm is small and has good spatial locality.

3.3 Parallelism

Looking at the reconstruction algorithm in Figure 2 we could exploit parallelism at two levels. A coarse-grained parallelism is exploited by partitioning N particles into multiple threads, and then each thread performs rotational and translational alignment for its own particles. It obviously leads to a higher instantaneous working set because the threads are performing *RTFAlign* in parallel. Thus, contentions of shared on-chip memory and bandwidth become bottlenecks for scalability of the coarse-grained parallel algorithm. Therefore, an alternative way is to exploit a fine-grained parallelism within the rotational and translational algorithm (See section 4). In the fine-grained parallel algorithm, overhead of thread execution (synchronization, thread creation/switch) often plays an important role in performance. For example, on conventional multi-core architectures a model to implement fine-grained parallelism is OpenMP. Using OpenMP MicroBenchmark Suit 2.0 [1] we measure the thread overhead is usually magnitude of millisecond on a two-way AMD Barcelona 4-cores SMP machine. Note that execution time of several kernels are also magnitude of millisecond, it is difficult to achieve fine scalability on such a conventional multi-core architecture. We write a micro-benchmark to measure the overhead of thread startup and synchronization within a thread block in CUDA thread execution. Table 2 shows that the thread execution overhead is magnitude of microsecond for 9216, 65536, 262144 threads with different configuration.
Observation 3: There is abundant parallelism in RTFAlign algorithm. A fine-grained parallelism is a reasonable way to achieve strong scalability on many-core like GPU.

This workload characterization shows that the single-particle 3D reconstruction algorithm is of high parallelism, high arithmetic intensity and high spatial locality. At first glance, these features are suitable to highly parallel many-core architecture, however, it is not trivially data parallel. For example, strategies to develop a proper data structure for GPU memory management, adopt coalesced memory access and orchestrate data movement with thread mapping need to be carefully designed for high performance.

4. EXPLOITING PARALLELISM ON GPU USING CUDA

This work is based on GeForce 8800, which architecture is introduced in detail in [16]. All data reside in off-chip global memory with the highest latency ($200 \sim 300$ cycles)

blocks * threads	72 * 128	512*128	2048*128	36*256	256*256	1024 * 256	18*512	128*512	512*512
startup	18.6	21.6	35.0	18.7	20.7	27.3	18.9	19.6	24.2
synchronization	18.5	18.9	18.0	18.6	18.3	18.8	19.4	19.5	21.8

at the beginning of execution. Although bandwidth to the off-chip memory is very high at 86.4 GB/s, it can saturate if all threads request access are contiguous elements of memory, which enables hardware to coalesce memory accesses to the same DRAM page (i.e. contiguous 16-word lines). Therefore, optimizations to coalesce accesses and reuse data are necessary to achieve good performance [12, 16]. Considering the memory hierarchy of CUDA model, we generalize its memory to two levels of memory : off-chip global memory with high latency and on-chip shared memory low latency. Inspired by previous works on other multi-threaded many-core architecture [9, 18], we extend a percolation programming model to enable memory coalesce optimization and hide the off-chip memory latency. Note that a notable feature of GPU-CUDA is massive multi-threading to hide latency on a memory hierarchy with high bandwidth. On architectural side the latency-hidden is implemented at instruction level. If we partition the threads into memory threads and computation ones, a more flexible interface is exposed to programmer for scheduling memory and computation operations. A basic idea of percolation programming is to decouple computation with memory access. The percolation strategy for memory coalesce optimization consists of three concurrent pipelining processes:

- *inward percolation:* This step reads data from off-chip global memory to on-chip shared memory. The key insight is to orchestrate mapping between data and threads to avoid uncoalesced access.

- *computation:* After the required data reside shared memory, parallel threads perform real calculation with loaded data.

- *outward percolation:* Finally we write back the results to global memory. Again a thread mapping strategy needs to be selected to improve coalesced access.


```
for (i = 0; i < n ; i += 2) {
c[i] = func(a[i], b[i+1]);
c[i+1] = func(a[i+1], b[i]);
}
        (a) CPU−only version

// n / 2 threads
c[2*tid] = func(a[2*tid],b[2*tid+1]);
c[2*tid+1] = func(a[2*tid+1],b[2*tid]);
        (b) naive GPU version
```

```
__shared _sa[],_sb[],_sc[]
// inward percolation: n threads
_sa[tid] = a[tid]; _sb[tid] = b[tid];

//compute: n / 2 threads
_sc[2*tid] = func(_sa[2*tid], _sb[2*tid+1]);
_sc[2*tid+1] = func(_sa[2*tid+1],_sb[2*tid]);

//outward percolation: n threads
c[tid] = _sc[tid];
        (c) percolation GPU version
```
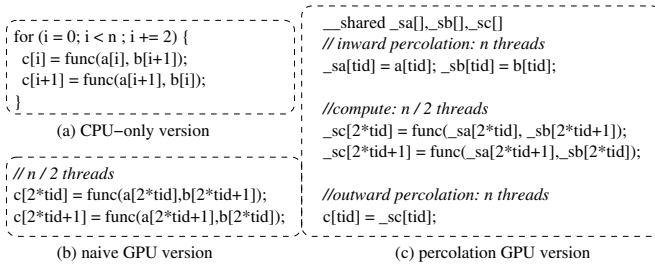
Figure 4: A simple example of percolation programming. The three stages in the percolation GPU code are pipelined.

Figure 4 illustrates a simple example of percolation programming. There are two remarkable advantages of percolation strategy. First, since the off-chip memory access

operations are decoupled with computation, we could develop different optimization strategies to the three pipelined stages, respectively. For example, the thread-data mapping may be adaptive through the different stages. Second, *inward percolation-computation-outward percolation* is actually pipelined and the overhead of off-chip memory access is hidden if the pipelining is full. Besides, if a program exhibits data locality, the computation threads also can re-use the data percolated to on-chip memory. Note that the percolation model requires a synchronization at the end of each pipelining step. The tasks of the three steps may be assigned to different thread blocks in CUDA. Unfortunately there is no mechanism to support synchronization among thread blocks. We employed a global synchronization proposed by Volkov [20]. The idea of percolation is similar to streaming programming style of gather-compute-scatter. The streaming programming uses a DMA mechanism to address issues of data movement between CPU system memory and GPU memory, percolation utilizes the massive multi-threaded hardware units to hide the overhead through GPU memory hierarchy and requires less hardware cost. In this section, based on our proposed new data structure we show how to apply percolation programming to improve coalesced memory access for the alignment algorithm.

4.1 Data Structure

During the rotational and translational alignment several temporary space are allocated to store correlation functions and adjusted images, i.e. rotational or translational image for comparison in dot multiplication. In a C implementation on CPU code we would use dynamic memory management. However, we observe that dynamic memory allocation in CUDA is one order of magnitude slower than that on CPU. In fact, our preliminary implementation using dynamic memory management ($cudaMalloc()$) makes performance on GPU worse for multiple refinement iterations. An algorithmic analysis implies that each RTFAlign follows an similar process, that is, the amount of space for correlation function and adjusted image change little for each turn of RTFAlign execution. As noted in section 3 the working set is small, therefore we propose a new data structure to avoid dynamic GPU memory management.
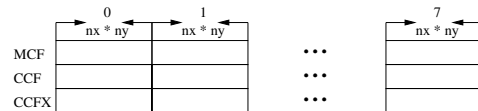

Figure 5: Data structure for RTFAlign's temporary storage. Each region has 8 segments.

Looking at the reconstruction algorithm in Figure 2 we find that RTFAlign is executed in multiple loop iterations. Each iteration processes different particle, however, the size of each temporary storage is determined by the pixel, which

is fixed in the whole algorithm. According to RTFAlign algorithm, the extra temporary memory is mainly used by *MCF*, *CCFX* and *CCFX*. Assume that the pixel is $nx \times ny$, thus we maintain three GPU memory regions, which are physically allocated once time during the first run of RTFAlign. Figure 5 illustrates the data memory on GPU. Each memory region has a pointer to current available segment. At every beginning of consecutive run of RTFAlign we just reset the pointer to start of each memory region. In the following subsections, we present how the proposed fine-grained parallel algorithm performs operations on each segment.

4.2 Rotation and Translation

The kernels *Rotate()*, *Translate()* and *unwrap()* in Figure 3 are responsible for rotation and shift operations on an image. The basic procedures of the three kernels are similar (Eq. 2): for any given point in the new image, calculate its corresponding coordinate in the original image and find the right pixel, then store the points into the new image. Thus the algorithm can be generalize into 2 steps: coordinate computation and interpolation. Their difference in coordinate computation, however, affects their memory access behaviors, respectively. For translation operations, the coordinates of pixels are transformed between two rectangular coordinate system, which leaves the data loads and stores easily coalesced. Both rotation and unwrap operations transform between rectangular and polar coordinate system. Their computation however leads to amount of uncoalesced memory access because of the unaligned coordinate system. Since both operations has the same computational behavior, we only describe optimization strategy to rotation.

For rotation operation parameters determining how much an angle the pixel needs to move around the image center are given in polar coordinate. Therefore, to access the corresponding element in memory, a polar to rectangular coordination transformation has to be performed to calculate the actual index of the element. For a point located in (x, y) with a rotation angle of θ, the old point coordination is $(x\cos\theta + y\sin\theta, -x\sin\theta + y\cos\theta)$. Since transformation result is unlikely to be right on an integer point, we need to perform an interpolation to compute the value of the 2-D function at the given position. Such interpolation involves 4 points around the new point. The result is determined by the point's distance to each of its 4 surrounding integer points, as shown in Figure 6. However, such pattern directly violates the coalesce rules.
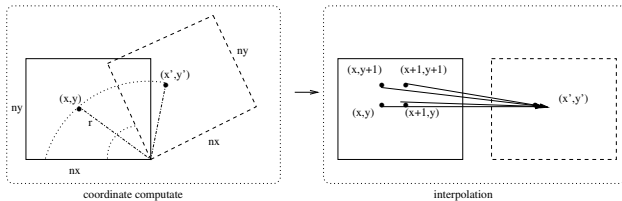
Figure 6: Computation operations in rotation and translation. The rotation angle is θ

Note that the rotation algorithm is iterated with different angle. Our algorithm performs rotations of multiple angles in parallel. For example in Figure 7 the image is concur-
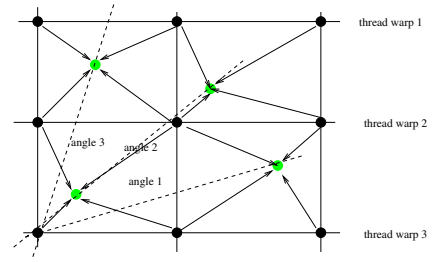
Figure 7: Computation coalesced access and data reuse.

rently rotated from three consecutive angle. With the percolation idea the interpolation operation is split into three-stages: *read data from the original coordinate–compute–write back to the new coordinate*. For read operation the elements in the same row is assigned to the same warp (See Figure 7), thus the words accessed by all threads are easy to arranged into the same segment of size equal to 128 bytes. Then the mapping strategy between data and thread warp for writing back operations is determined by the rotation angle. All elements along the same rotation angle are assigned to the same warp. Therefore, the uncoalesced memory accesses are reduced. On the other hand, such data access pattern clearly demonstrates data locality and reuse. Each iteration reads 4 adjacent points in the image, and the following iteration is expected to reuse between 1 or 2 of the previous loaded points. The exact ratio of data reuse is $|\tan(2*\theta)| * 2 + (1 - |\tan(2*\theta)|) = 1 + |\tan(2*\theta)|$.

4.3 Correlation Function

As shown in top of Figure 8 the calculation of both auto- and cross-correlation functions are composed of three similar steps: i) forward Fourier transformation on two input images. ii) blending operation on the transformed images. iii) backward Fourier transformation on the blent image. Among the three correlation functions, *CCFX* is relatively different in that its operations are on a per-row basis because of 1D real Fourier transformation. In this sub-section we present our parallel algorithm for the blending operation and multiple 1D Fourier transformations.

Assume that the size of an particle is $nx \times ny$, the blending operation reads the particle's data and produces one of the same length. The original implementation needs to read two elements from both input streams and write two elements into result stream, making memory coalescing difficult. Keeping the percolation idea in mind again, we split the blending operation into three explicit stages: *Mapping data from global to shared memory–compute with the loaded data in shared memory–write back result to global memory*. The parallel algorithm performs blending operations rows-by-rows of the image. A trick is to choose different data-thread mapping strategies to avoid uncoalesced access in global memory. The bottom of Figure 8 illustrates the algorithm. During reading/writing data in global memory the number for threads is the same with the number of elements, and each element is consecutively mapped to each thread. The real calculation operations are finished by half number of threads, that is, each thread read/write two elements in shared memory.
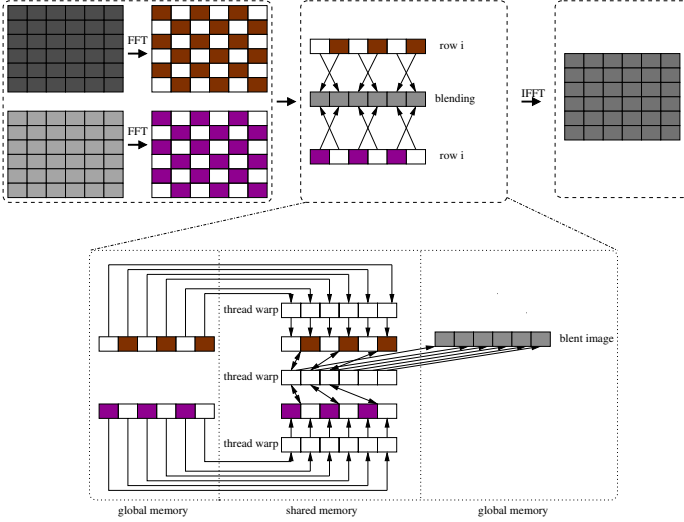
Figure 8: The calculation of correlation function.

Figure 9: Diagram of the overall design of the EMAN package [15].

The algorithm described in Figure 3 needs 1D Fourier transformations for rotational alignment ($CCFX$). Note that the transformation in each row is too small to amortize the CUDA kernel invocation overhead. Fortunately, since rows in the matrixes are stored in a continuous manner, and there is no data dependency between calculation on different rows, we merge different iterations of the blending operation to enable batched Fourier transformation. While the correlation kernel reads elements from both ends of the array towards its center, we keep the load from global memory to shared memory sequential within each thread block. When nx is not multiple of 16, we pad the rows to align the half warp base addresses, enabling maximum coalesced accesses. After such adjustments, batched Fourier transformation demonstrates an approximated 5X speedup against original implementation, and the invocation overhead in correlation kernel is reduced to about 15% of total runtime.

5. IMPLEMENTATION AND EXPERIMENTAL RESULTS

We implemented the proposed GPU algorithm by porting a opensource package EMAN [15], which is a suite of scientific image processing tools aimed primarily at the transmission electron microscopy community. EMAN has a particular focus on performing a task known as single particle reconstruction. Images processed in EMAN are floating point grey scale images. That is, the pixel values in the images are represented as real numbers, not as small integers. EMAN was first released in 1999, and has been under continuous development since. It consists of a C++ library of hundreds of different image/volume processing algorithms with bindings into the popular Python scripting language. We re-wrote the core library with more than ten of thousands lines of C++ codes using CUDA.

5.1 Experimental Setup

The experimental platform includes a GeForce 8800 (G80) and 2.0GHz Opteron processor (peak performance is 8GFlops). The G80 consists of 16 streaming multiprocessors (SMs),
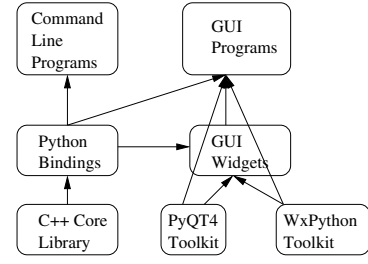
each containing eight streaming processors (SPs) running at 1.35GHz. Its theoretical peak performance is 388.8 GFlops (single-precision floating-point). The off-chip global memory is 1GB and the on-chip shared memory of each SM is 16KB. The latency to off-chip and on-chip memory is more than 200 cycles and 1 cycle, respectively. A maximum of 768 simultaneously active thread contexts is supported per SM, and up to eight thread blocks can be run per SM at one time.

The performance evaluation uses two baseline programs. One is a CPU-only program EMAN (B1) is executed on the Opteron processor for evaluating relative speedup. The implementation of EMAN integrates FFTW [6] as its accelerated library. The compiler option is configured as "-ffast-math -msse2 -O3 -funroll-loops". Another baseline program (B2) is a simple CUDA implementation of the reconstruction algorithm, to which the percolation optimization strategies in section 4 are not applied. Our CUDA implementation uses the vendor's math libraries $cudaFFT$ and $cudaBLAS$ to accelerate Fourier transformation and dot multiplication. The following performance evaluation does not consider the effect of these libraries.

In our experiments, we use three classes of images with different resolution as introduced in section 3. Table 3 summarizes their features.

Table 3: Features of the three images used in our experiments

Name	Resolution in pixels	Number of particles
SMALL	96×96	2560
MEDIUM	256×256	4239
LARGE	512×512	4239

5.2 Performance Evaluation

Currently single-precision floating operations can satisfy most of real experimental requirement (though double-precision is necessary in the near future). The algorithms running on the GPU-CUDA produce identical results as on the CPU for all test sets. In the experiment with *Hepatitis B virus* (referred to as LARGE in the context) Figure 10 shows five steps of 3D reconstruction on GPU-CUDA, which results are exactly the same with that on CPU.

First, we compare our GPU implementation to the CPU-only EMAN program. The performance is measured as a relative speedup, which is calculated by $\frac{CPU\,execution\,time}{GPU\,execution\,time}$.

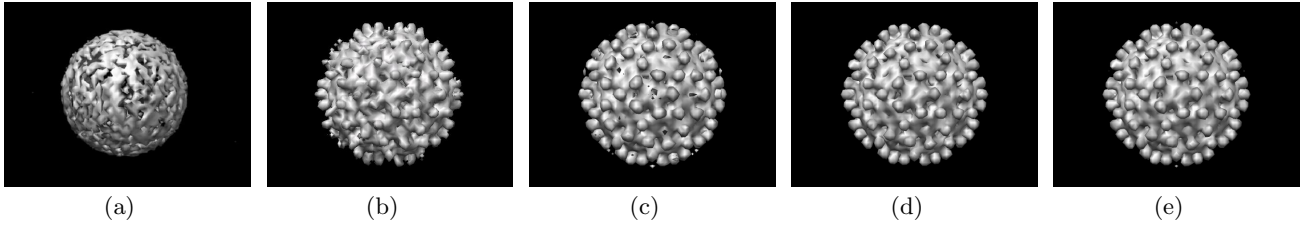(a)   (b)   (c)   (d)   (e)

Figure 10: The first five steps of 3D reconstruction for Hepatitis B virus. It is denoted as LARGE in our experiment presented in this section.

Table 4 summarizes the execution of the whole reconstruction algorithm on both GPU and CPU. As described in Figure 2, the reconstruction is a process of refinement iteration so that the final model is converged, which is controlled by a threshold. In our experiment, the threshold says the number of iterations is 10. In our evaluation, we refer to problem size as the size of pixel. With increasing problem sizes the reconstruction needs more computational capability. On GPU side the proposed parallel program achieves higher speedup for a larger problem size. In the workload characterization analysis presented in Section 2, we observe that the reconstruction algorithm has high arithmetic intensity. GPU provides more capability of arithmetic operations when SIMD operations are fully pipelined. For the case with small problem size the amount of SIMD operations is not enough to fully fill the pipeline. Besides, when the problem size is large, we can startup more threads to hide the latency of memory access. Therefore, it is reasonable to argue that our GPU reconstruction algorithm has fine scalability with the problem size. That means it is a better way to reconstruct a Cryo-electron microscopy image with higher resolution on GPU-like many-core architecture in the future.

Table 4: Comparison of execution time on GPU and CPU

Problem Size	GPU (seconds)	CPU (seconds)	Speedup
SMALL	4965.26	11420.11	2.3
MEDIUM	24963.50	177240.83	7.1
LARGE	85239.14	826819.71	9.7

The kernels outlined in Table 1 occupy more than 90% of execution time. Figure 11 plots their performance improvement measured as speedup to CPU-only program. We observe that each kernel achieves different performance improvement even if some kernels have similar computational behavior. The workload characterization Table 1 tells us that *Rotate* has the highest arithmetic intensity, which leads to the greatest improvement. Although the kernel *Rotate* and *unwrap* have similar computational behavior, *unwrap* only performs coordinate transformation within less than one half of a image, such requirement of computational load leads to its lower speedup than *Rotate*. *CCFX* calculates the cross correlation function by 1D Fourier transformation and needs less computational operations than *MCF/CCF* with 2D Fourier transformation. Therefore, the improvement of *CCFX* is not so good as that of *MCF/CCF*.
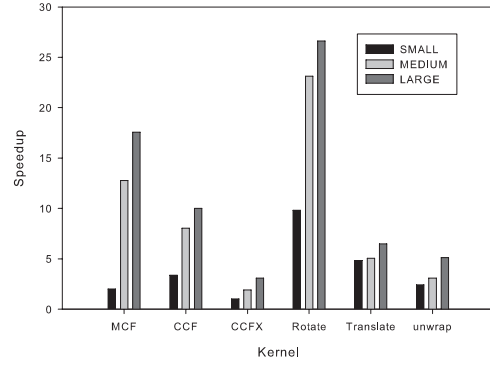

Figure 11: Speedups of the seven kernels for different problem sizes, which are represented by each column bar.

In addition to the effect of arithmetic intensity and computational requirement, the coalesced memory access is another critical performance factor. Figure 12 shows that our proposed parallel algorithm indeed reduces the number of uncoalesced memory accesses (for the sake of space saving, we only present the results for LARGE). In the two 2-D correlation function, our algorithm almost totally eliminates the uncoalesced memory access. However, comparing to correlation function, the memory access patterns of rotation and translation are more irregular. The difference between two coordinate system (rectangular and polar) makes it difficult to achieve both coalesced read and coalesced write at the same time. Therefore, the reduction of uncoalesced memory only appears either read or write.

The optimization to coalesced memory access has a prominent effect on global memory bandwidth. In our percolation strategy, an on-line thread-data mapping is used so that the simultaneous memory access by threads in a warp is coalesced into a single memory transaction. In fact, this pattern reduces the number of accesses to unused memory cells and increases the effective bandwidth. Comparing with the naive baseline program, our percolation implementation explicitly decouples an operation into three pipelining stages. It obviously increases the complexity of programming, but it is worthy because amount of extra uncoalesced memory access are removed. It is clear that the kernels with lots of uncoalesced memory accesses do not take good advantage of the high memory bandwidth G80 provides.
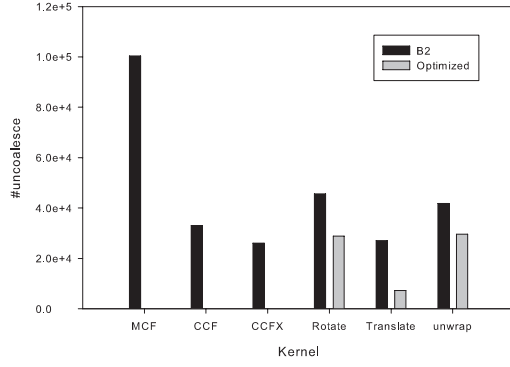
Figure 12: Comparison of the number of uncoalesced memory access. B2: the navie CUDA-based implementation without the proposed percolation optimization. Other figures have the same notation convention.

The difference in bandwidth results in different performance of floating-point operations. As shown in Figure 14 the kernels with high utilization of bandwidth achieves high performance of floating-point units. Comparing to the evaluation of speedup to CPU in Figure 11, we find that the a higher speedup does not means a higher performance of floating-point. For example, *Rotate* has the best speedup, *MCF/CCF* however achieves better floating-point performance because of their high utilization bandwidth.
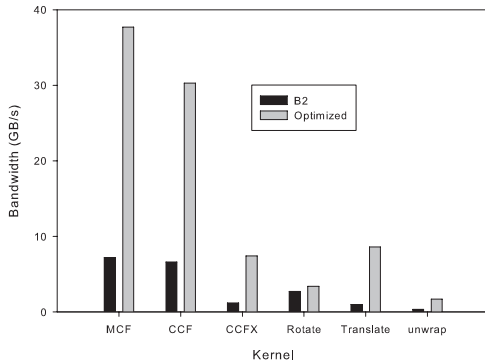

Figure 13: The estimated bandwidth to global memory

5.3 Discussion

This work is the first step to tuning the cyro-EM 3D reconstruction algorithm for Petascale. However, a further important issue is to build more quantitative model that can predict the performance in terms of the workload characteristics and the relevant machine parameters like the number of SIMD threads, the number of memory access, the number of registers, memory access latency and bandwidth for each kernel. We believe that such model would be helpful to build an efficient PetaScale system by co-design of
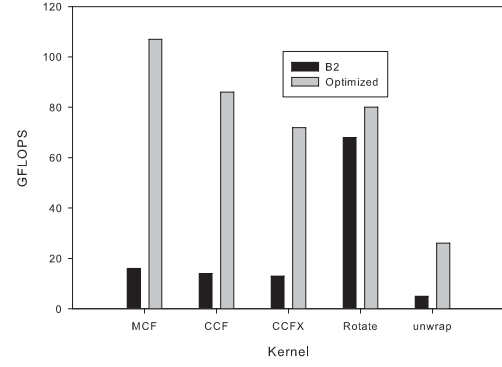

Figure 14: Performance of floating-point operations. *Translate* has no floating-point operations.

architecture and algorithm. Based on the experimental observations, we get following insights for performance tuning and evaluation on GPUs:

- *Good cache locality on CPUs does not mean good memory performance on GPUs.* Table 1 indicates that all kernels of *RTFAlign* has good cache locality when we run the program on a general-purpose CPU. Although the locality also might be exploited on GPUs configured with shared on-chip memory (i.e. data reuse in *Rotation*), uncoalesced memory access caused by parallel threads should be addressed for high performance. Due to the irregular memory access in *Rotate*, the uncoalesced memory access can not be totally avoided so that its bandwidth utilization is relatively low.

- *The proposed percolation technique is effective to address the issue of uncoalesced memory access.* In fact, the key idea to optimize coalesced memory access is orchestrate mapping between thread and data. Therefore, the performance tuning on GPUs for thread-data mapping considers not only load-balance and communication minimization, but coalesced memory accesses. In this work we adopt a percolation technique to decouple an operation into three pipelining stages and online mapping strategies are applied to different stages. For an instance of correlation function, our strategy has eliminated uncoalesced memory accesses for both read and write. It is an important future work to generalize the optimization techniques with more workload applications and different architecture like IBM CELL blade engine.

- *The relative speedup should not be the only performance measurement for an optimization.* Most of previous work on GPUs focused on the measurement of relative speedup to CPUs. Our experiments show that the relative speedup can not indicate whether an optimization technique is really effective on GPUs or not. *Rotate* has the highest arithmetic intensity and achieves the best relative speedup because of the powerfull floating-point capability of GPUs. However, the several kernels of correlation function are more memory-bound. Our proposed optimization strategy for coalesced memory

access improves the performance memory bandwidth and floating-point. In fact, the relative speedup would be miss-understood if the corresponding implementation on CPUs was not carefully considered.

6. CONCLUSIONS

In this paper, we have presented a detailed workload characterization of a single-particle 3D reconstruction algorithm from cryo-EM image and proposed a GPU global memory optimization strategy of percolation, which greatly improves performance of several memory-intensive kernels in the reconstruction algorithm. The final 3D model on GPU has the same quality with that on CPU. The workload characterization of the single-particle 3D reconstruction program demonstrates the rationality of porting it to GPU-like architecture. The workload study indicates that the reconstruction application is high arithmetic intensity, instantaneous working set is small and abundant parallelism. These features are conductive to accelerate the algorithm on GPU, FPGA and other similar many-core architecture. GPU-like many-core architecture provides an alternative way to low power solution, our work demonstrates that it is promising to scale performance for the single-particle 3D reconstruction from cryo-EM on such emerging architectures.

7. ACKNOWLEDGMENTS

8. REFERENCES

[1] J. Bull and D. O'Neill. A microbenchmark suite for openmp 2.0. *SIGARCH Comput. Archit. News*, 29(5):41–48, 2001.

[2] C.O.Sorzano, R. Marabini, J. V. Muriel, J. R. Castro, S.H.Scheres, and J. M. Carazo. Xmipp: a new generation of an open-source image processing package for electron microscopy. *Journal of Structural Biology*, 148:194–204, 2004.

[3] D. DeRosier and A. Klug. A reconstruction of 3-dimensional structure from electron micrographs. *Nature*, 217:130–134, 1968.

[4] D. Diez, H. Mueller, and A. Frangakis. Implementation and performance evaluation of reconstruction algorithms on graphics processors. *Journal of Structural Biology*, 157:288–295, 2007.

[5] Y. Dotsenko, N. Govindaraju, P. Sloan, C. Boyd, and J. Manferdelli. Fast scan algorithms on graphics processors. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 205–213, New York, NY, USA, 2008. ACM.

[6] FFTW. www.fftw.org.

[7] GPU Technology for Accelerated Computing. www.amd.com/stream.

[8] J. Frank, Radermacher, Penczek M., P. Zhu, J., Y. Li, and Ladjadj. Spider and web: processing and visualization of images in 3d electron microscopy and related fields. *Journal of Structural Biology*, 116:190–199, 1996.

[9] G. Gao, K. Likharev, P. Messina, and T. Sterling. Hybrid technology multi-threaded architecture,. In *Proceedings of Frontiers '96: The Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 98–105, 1996.

[10] GPGPU. http://www.gpgpu.org.

[11] S. Guha, S. Krisnan, and S. Venkatasubramanian. Data visualization and mining using the gpu. In *11th ACM International Conference on Knowledge Discovery and Data Mining*, 2005.

[12] NVIDIA CUDA Programming Guide. www.nvidia.com/cuda.

[13] T. Hartley, U. Catalyurek, A. Ruiz, F. Igual, R. Mayo, and M. Ujaldon. Biomedical image analysis on a cooperative cluster of gpus and multicores. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 15–25, New York, NY, USA, 2008. ACM.

[14] Y. Liang, E. Y. Ke, and Z. H. Zhou. Imirs: a high-resolution 3d reconstruction package integrated with a relational image database. *Journal of Structural Biology*, 137:292–304, 2002.

[15] S. Ludtke, P. Baldwin, and W. Chiu. Eman: Semiautomated software for high-resolution single-particle reconstructions. *Journal of Structural Biology*, 128(1):82–97, 1999.

[16] S. Ryoo, C. Rodrigues, S. Baghsorkhi, S. Stone, D. Kirk, and W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM.

[17] S.H. Scheres, M. Valle H. Gao, G.T.Herman, P.P. Eggermont, J. Frank, and J.M.Carazo. Disentangling conformational states of macromolecules in 3d-em through likelihood optimization. *Nature*, 4(1):27–29, 2007.

[18] G. Tan, V.C. Sreedhar, and G. Gao. Just-in-time locality and percolation for optimizing irregular applications on a manycore architecture. In *21st Annual Workshop Languages and Compilers for Parallel Computing (LCPC)*, 2008.

[19] K. Taylor and R. M. Glaeser. Electron diffraction of frozen, hydrated protein crystals. *Science*, 186:1036–1037, 1974.

[20] Vasily Volkov and James W. Demmel. Benchmarking gpus to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[21] W. Wu and P. Heng. A hybrid condensed finite element model with gpu acceleration for interactive 3d soft tissue cutting: Research articles. *Computer Animation and Virtual Worlds*, 15(3):219–227, 2004.

[22] Y. Zheng and P. C. Doerschuk. A parallel software toolkit for statistical 3-d virus reconstructions from cryo electron microscopy images using computer clusters with multi-core shared-memory nodes. In *22rd IEEE International Parallel and Distributed Processing Symposium*, 2008.