Journal of Software Engineering and Applications, 2019, 12, 321-338
https://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

Implementation of a Particle Accelerator Beam Dynamics Code on Multi-Node GPUs

Zhicong Liu[1,2], Ji Qiang[1*]

[1]Lawrence Berkeley National Laboratory, Berkeley, CA, USA
[2]Key Laboratory of Particle Acceleration Physics and Technology, Institute of High Energy Physics, Chinese Academy of Sciences, Beijing, China
Email: *jqiang@lbl.gov

How to cite this paper: Liu, Z.C. and Qiang, J. (2019) Implementation of a Particle Accelerator Beam Dynamics Code on Multi-Node GPUs. *Journal of Software Engineering and Applications*, **12**, 321-338.
https://doi.org/10.4236/jsea.2019.129020

Received: February 27, 2019
Accepted: September 1, 2019
Published: September 4, 2019

Abstract

Particle accelerators play an important role in a wide range of scientific discoveries and industrial applications. The self-consistent multi-particle simulation based on the particle-in-cell (PIC) method has been used to study charged particle beam dynamics inside those accelerators. However, the PIC simulation is time-consuming and needs to use modern parallel computers for high-resolution applications. In this paper, we implemented a parallel beam dynamics PIC code on multi-node hybrid architecture computers with multiple Graphics Processing Units (GPUs). We used two methods to parallelize the PIC code on multiple GPUs and observed that the replication method is a better choice for moderate problem size and current computer hardware while the domain decomposition method might be a better choice for large problem size and more advanced computer hardware that allows direct communications among multiple GPUs. Using the multi-node hybrid architectures at Oak Ridge Leadership Computing Facility (OLCF), the optimized GPU PIC code achieves a reasonable parallel performance and scales up to 64 GPUs with 16 million particles.

Keywords

Particle Accelerator, Particle-In-Cell, GPU, Parallel Beam Dynamics Simulation

1. Introduction

The modern particle accelerator as one of the most important inventions in 20th century provides an important tool in scientific discovery and industrial application. For example, large accelerators are used in high energy physics and nuclear physics to study the fundamental structure and property of matter, to

discover new fundamental particles, and to understand the origin of the universe. Particle accelerators are also used to generate high brightness x-ray radiation and high-intensity neutron flux for research in material science, biology, chemistry, physics, and others. In industry, particle accelerators are used for radiotherapy, ion implantation, and other applications.

Inside the particle accelerator, a train of charged particle beams are produced, confined, and accelerated to a range of energies (from MeV to TeV) for different applications. In the particle accelerator design and operation, one major area (beam dynamics) is to study the dynamic behavior of those charged particles inside the accelerator in order to minimize the loss of charged particles onto the pipe wall causing radioactivity, and to maximize the brightness of the beam to achieve best performance in high energy colliders and x-ray radiation light sources. To study the charged particle beam dynamics self-consistently, the particle-in-cell (PIC) method has been used in the particle accelerator community [1]-[8]. In this method, at each step, particles are deposited onto a computational grid to obtain charge density distribution in spatial domain. Then, the Poisson equation is solved on the grid in the moving beam frame to attain electric fields due to the Coulomb interaction of charged particles on the grid. These fields (also called space-charge fields) are then interpolated from the grid back to the particles and transformed to the laboratory frame following the relativistic Lorentz transform. The space-charge fields together with the external fields from the particle accelerator devices are used to advance particles. This step is repeated many times until the beam moves out of the accelerator or the maximum computing time is reached. The PIC method for beam dynamics simulation is usually computationally expensive since it tracks a large number of macroparticles (more than millions) and has to solve the Poisson equation self-consistently at each step. A number of parallel PIC beam dynamics codes using Message Passing Interface (MPI) were developed in the accelerator community for high intensity/high brightness beam simulations [2] [3] [4] [5] [6].

The pure MPI based parallel beam dynamics code is useful on parallel multi-processor computers. However, these massive parallel computers can be expensive. Meanwhile, the Graphics Processing Unit (GPU), which was originally developed for computer graphics and video game, now becomes a general-purpose computer processor and cost-effective for high-performance computing [9] [10] [11]. Moreover, one GPU contains several hundreds or even thousands of computing cores. For example, a single Nvidia GTX GPU consists of several Streaming Multiprocessor (SM), and each SM contains many computing cores. It uses high-bandwidth bus (~200 Gb/s) connecting the memory on chip to the computing cores and is optimized for simultaneous parallel calculations, particularly for single instruction multiple data (SIMD) operations [12]. Manufacturers of GPU have approaches to general-purpose computation with their own application program interfaces (API). The Compute

Unified Device Architecture (CUDA) is a parallel computing platform and programming model for GPUs developed by the NVIDIA [13] [14]. It enables a fast implementation of numerical models on a GPU and dramatically increases computing performance by harnessing the computing power of the GPU.

A number of PIC codes (especially in plasma physics community) were implemented on GPUs in previous studies and significant improvement of computing performance was reported in [15]-[27]. Most of those studies focused on the performance optimization of the PIC code on a single GPU. However, as the size of problem increases (e.g. with the number of simulation particles >10 millions), the memory of a single GPU (typically a few GB) can no longer store the problem for simulation, multiple GPUs are needed. Meanwhile, some large-scale high-performance computers such as Titan and Summit at Oak Ridge Leadership Computing Facility (OCLF) [28] [29] have multi-node hybrid architecture where each node contains one or multiple GPUs. In previous studies, multiple GPUs were used for electromagnetic plasma PICs [16] [19] [20]. To the best of our knowledge, there was no report on the implementation of a parallel particle accelerator beam dynamics PIC code on multiple GPU nodes. In this paper, the MPI based parallel beam dynamics PIC code, ImpactT [6], was implemented and optimized using the CUDA parallel computing platform on both a single GPU and multi-node GPU architectures. Using a single GTX 1060 GPU, the code speeds up by more than 40 times compared with that running on an AMD Opteron 6134 CPU core. This is about twice faster than the original MPI version running on the 64-core AMD CPU computer. Besides the techniques used for single GPU optimization, we also tested two parallel strategies for multi-GPU performance optimization.

The organization of the paper is as follows, after the Introduction, the PIC particle tracking model and the race condition on the GPU of the hybrid architecture computer are reviewed in Section 2. Then, we present the PIC code structure and its GPU optimization, especially the parallel depositor without conflict, in Section 3. After that, the performance of the PIC code on a single GPU and two multi-node GPUs is presented in Section 4. Finally, conclusions are drawn in Section 5.

2. Multi-Particle Beam Dynamics PIC Model

Inside particle accelerators, the charged particles evolve subject to the following equations:

$$\frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}t} = \frac{\boldsymbol{p}c}{\gamma} \tag{1}$$

$$\frac{\mathrm{d}\boldsymbol{p}}{\mathrm{d}t} = q\left(\frac{\boldsymbol{E}}{mc} + \frac{1}{m\gamma}\boldsymbol{p}\times\boldsymbol{B}\right) \tag{2}$$

where $\boldsymbol{r} = (x, y, z)$ denotes the particle spatial coordinates, $\boldsymbol{p} = \left(p_x/mc, p_y/mc, p_z/mc\right)$ the particle normalized mechanic momentum, m the particle rest mass, q the particle charge, c the speed of light in vacuum, γ

the relativistic factor defined by $\sqrt{1+\boldsymbol{p}\cdot\boldsymbol{p}}$, t the time, $\boldsymbol{E}(x,y,z,t)$ the electric field, and $\boldsymbol{B}(x,y,z,t)$ the magnetic field. Here, the electric and the magnetic fields include both the space-charge fields from the solution of the Poisson equation and the external fields.

The solution of the Poisson equation can be written as:

$$\phi(x,y,z)=\frac{1}{4\pi\epsilon_0}\int G(x,x',y,y',z,z')\rho(x',y',z')\mathrm{d}x'\mathrm{d}y'\mathrm{d}z' \tag{3}$$

where G is Green's function of the Poisson equation, ρ is the charge density distribution function. For the charged particle beam inside the accelerator, the pipe aperture size is normally much larger than the size of the beam. In this case, an open boundary condition can be assumed for the solution of the Green's function in the above equation. Here, the Green function is given by:

$$G(x,x',y,y',z,z')=\frac{1}{\sqrt{(x-x')^2+(y-y')^2+(z-z')^2}} \tag{4}$$

Now consider a simulation of an open system where the computational domain containing the particles has a range of $(0,L_x)$, $(0,L_y)$ and $(0,L_z)$, and where each dimension is discretized using N_x, N_y and N_z point, from Equaiton (3), the electric potentials on the grid can be approximated as:

$$\phi(x_i,y_j,z_k)=\frac{h_x h_y h_z}{4\pi\epsilon_0}\sum_{i'=1}^{N_x}\sum_{j'=1}^{N_y}\sum_{k'=1}^{N_z}G(x_i-x_{i'},y_j-y_{j'},z_k-z_{k'})\rho(x_{i'},y_{j'},z_{k'}) \tag{5}$$

where $x_i=(i-1)h_x$, $y_j=(j-1)h_y$, and $z_k=(k-1)h_z$. The direct numerical summation of the above equation for all grid points can be very expensive and the computational cost scales as N^2, where $N=N_xN_yN_z$ is the total number of grid points. Fortunately, this summation can be replaced by the summation in a periodic doubled computational domain. In this periodic doubled computational domain, the original Green's function in the negative domain, *i.e.* $G(-r)$, is mapped to the extended domain following the periodic condition. The charge density in the extended domain is set to zero. In this periodic system with a new periodic Green's function and charge density, the summation can be done efficiently using the Fast Fourier Transform (FFT) method whose computational cost scales as $O(N\log(N))$. This new summation yields exactly the same values as the original summation inside the original domain [30].

Using the above mathematical equations, a schematic diagram of a single step of the PIC model in the beam dynamics simulation is shown in **Figure 1**. First, the charged particles are deposited onto the mesh grid to obtain charge density distribution on the grid. Next, the field on the grid is obtained by solving the Poisson equation using the above FFT based convolution method and interpolated back to individual particle location. Finally, the particles are pushed using the electric and magnetic fields including both the self-consistent space-charge fields and the external fields by solving Equations (1) and (2) using a numerical integrator. This loop repeats for many times until the stopping criterion is reached.

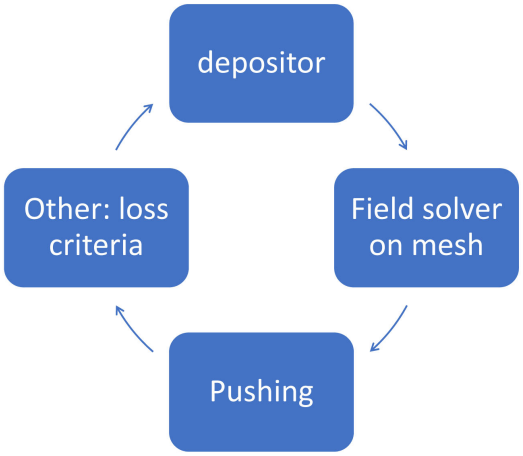

Figure 1. A single step of the PIC model.

To implement the above beam dynamics PIC model onto the multi-node hybrid architecture, the race condition arises and may lead to wrong results during the deposition stage due to the use of multiple threads in the GPU. The race condition occurs when two or more threads access shared data in a memory and try to write in it simultaneously [31] [32]. Usually in a multi-thread scheduling system, threads can be scheduled at any sequence, so coders cannot determine the order in which the threads would attempt to access the shared data. Therefore, the results are dependent on the thread scheduling algorithm, *i.e.* both threads are "racing" to access and change the data. To avoid the race condition, it's necessary to sort the particle with respect to the grid before the deposition by dividing the grid into smaller tiles, as shown in **Figure 2**. Each tile is associated with a thread and each tile contains a number of grid points. Assuming N is the number of tiles, we declare N arrays corresponding to N tiles and assign the particle data into each array. At each step, the particles are sorted into different tiles after the particle advance. In this way, each thread handles particles in the corresponding tile without the race condition. A flow chart of the PIC algorithm including the reordering is shown in **Figure 3**. In the following section, we will describe the components which are different from the original PIC algorithm, as marked yellow in **Figure 3**.

3. Implementation on Multiple GPUs

The implementation of the particle accelerator beam dynamics simulation code on GPUs is discussed in this section. The particles are distributed among multiple GPUs uniformly (in the replication method) or based on their spatial positions (in the domain-decomposition method) [33]. With the particles on each GPU, we will reorder them into individual tile to avoid the race condition. Then, those particles are deposited onto a computational grid to obtain the charge density distribution on the grid. Next, the Poisson equation is solved on the grid to attain the space-charge fields. Finally, those fields together with the external fields are used to push the particles in phase space.

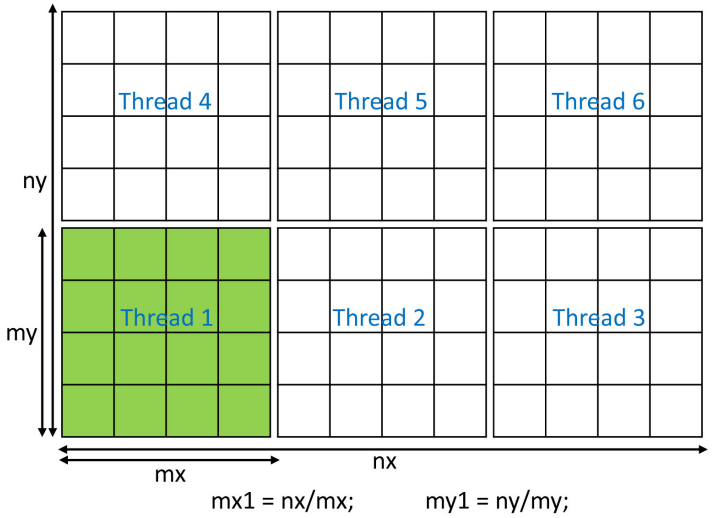

Figure 2. A schematic plot of tiles with computational grids.

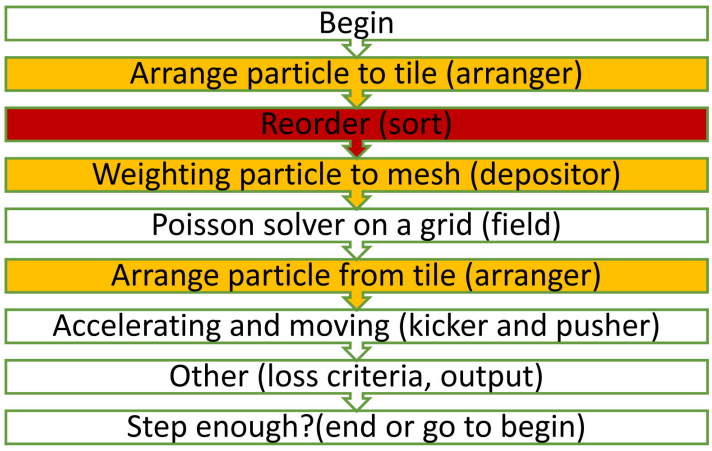

Figure 3. A flow of the PIC chart with the particle reordering.

3.1. Reorder

The implementation of pusher, kicker, and depositor on multiple GPUs were done by distributing the total number of particles among the GPUs. However, the particles must be reordered (*i.e.* sorted) at each time step before the particle deposition to avoid the race condition, which is not very straightforward since it is highly irregular and hard to execute in parallel. Here we use a buffer array as a temporary storage.

Firstly, the arrays nhole and ndirec are declared to handle the indices and the number of particles that would leave the current tile to each direction, as shown by the orange arrows in **Figure 4**. The nhole is preallocated at a given size, which determines the maximum number of particles leaving these current tiles. The size is determined by the available GPU memory size. If the number of particles leaving a tile exceeds the maximum number, an exception would rise and the code would stop. In this case, the user should use a smaller number of particles or run the code on a GPU with larger memory size.

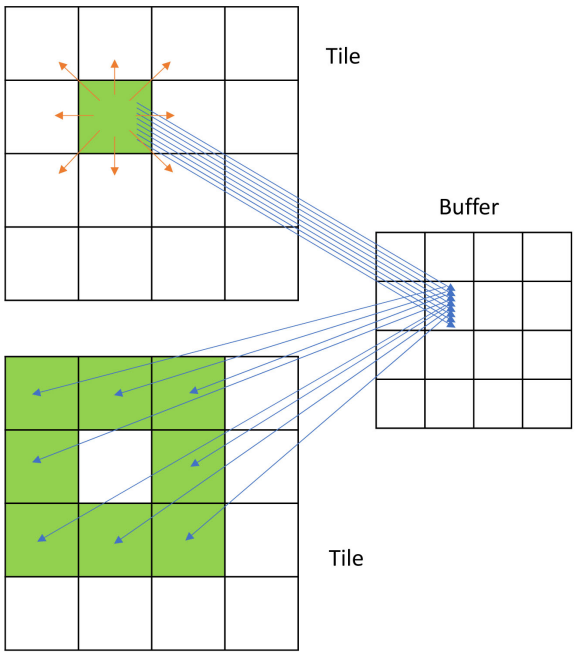

Figure 4. A schematic plot of particle reordering for different tiles.

Secondly, the particles leaving a tile are copied into an ordered global buffer: pbuff. Same as nhole, the size of pbuff is also determined by the maximum number of particles that would leave the current tile. With a running sum to the ndirec, we can know the memory address where we would put the particles to, so the particles going to the same direction are stored contiguously.

Thirdly, for each tile, we could know how many particles would move in and where they are located in pbuff by the nhole and ndirec of the neighbor tiles. If there is a particle that leaves this tile, the hole left would be filled by the incoming particle first. After all holes are filled, the new incoming particle is put to the end of the particle array. If there are still holes left after including all incoming particles, some particles at the bottom of the array are moved upward to fill in the holes to ensure that the particles in this tile always occupy a contiguous memory.

The procedures can be summarized as follows:

- Step 1: Write the indices of particles leaving a tile and their direction to nhole and ndirec.
- Step 2: Particles leaving a tile are copied into an ordered global buffer: pbuff.
- Step 3: According to nhole and ndirec, the buffer data is copied back into particle array.

With those procedures, there would be no race condition because each thread only handles its own tile and buffer.

3.2. Depositor

After the particles reordering, the memory locations of the particles in the same tile are arranged contiguously. In this way, each thread can handle the particles

in a tile without thread conflict to obtain the local density distribution rhoTile. Then, the global density distribution rho is attained by combining all local rhoTiles, as shown in Figure 5.

When using multiple GPUs, we have two options. One is to have different GPUs handle different spatial subdomains and communicate before and after deposition, which is called domain decomposition method. The other one is to let all GPUs deposit the particles onto the entire domain and performs a communication afterwards, which we call data replication method.

In the following, an example using 4 GPUs is shown to compare these two methods with the assumption that the total number of grid points is $64\times64\times64$, and the number of tiles is $16\times16\times16$ so that each tile contains $4\times4\times4$ grid points.

3.2.1. Domain Decomposition Method

In the domain decomposition method, each GPU only needs to process the corresponding domain. Now that the number of tiles is $16\times16\times16$, the domain size for each GPU would be $4\times16\times16$ tiles when running on 4 GPUs. However, this method requires prior sorting of the particles with respect to the subdomains to ensure that the particle data is located in the memory of correct GPU, thus additional communication and computation is necessary. The procedure is as follows:

1) Move particles among different GPUs.

(a) Pick particles. Each thread handles a tile, so we have $4\times16\times16=1024$ threads. It is less than the core number on a GPU, and we are unable to fully utilize the GPU.

(b) Communication among GPUs

i) Copy from GPU memory to host node memory. The total amount of data to be copied is $4\times16\times16\times(nGPU-1)\times nPtcMax$, in which $nPtcMax$ is the max number of particles to be transferred to another GPU.

ii) Communication through MPI send/receive.

iii) Copy from host memory to GPU memory. The total amount of data to be copied is same as above.

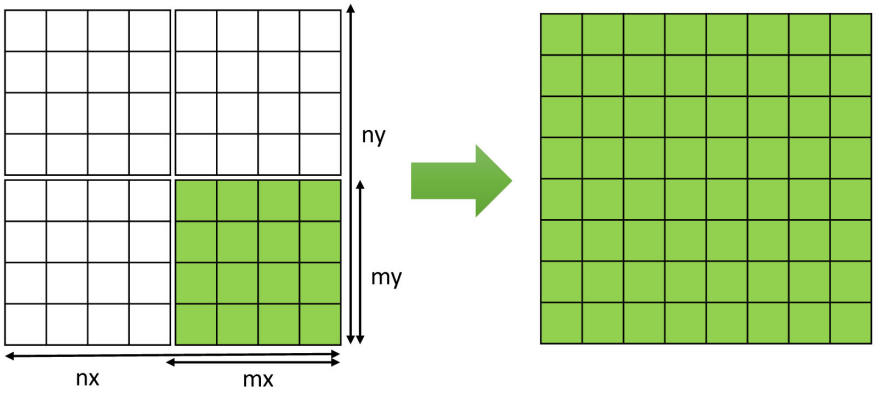

Figure 5. A schematic plot of deposition and combination.

2) Reorder particles inside the GPU, as shown in Section 3.1.

3) Deposit inside the GPU. It should be noted that the number of particles on each GPUs is different, so it may take a longer time to deposit.

4) Gather the particle density among GPUs.

a) Copy $16\times64\times64$ grid points from GPU memory to host memory.

b) Communicate $64\times64\times64$ grid points through MPI Allgether.

c) Copy $16\times64\times64$ grid points from host memory to GPU memory.

3.2.2. Data Replication Method

In the replication method, all GPUs contain the same number of particles and do the same work. Compared with domain decomposition method, it eliminates the need to exchange particle data among GPUs. The process is as follows, each corresponding to that from the domain decomposition method above.

1) No sorting among different GPUs.?

2) Reorder particles inside the GPU, with $16\times16\times16=4096$ tiles per GPU.

3) Deposit inside the GPU. Since the first step is to sort the particles in the GPU, the number of particles on each GPU is the same, the number of grid points is $64\times64\times64$.

4) Gather the particle density among GPUs.

a) Copy $64\times64\times64$ grid points from GPU memory to host memory.

b) Communicate $64\times64\times64$ grid points through MPI AllReduce.

c) Copy $64\times64\times64$ grid points from host memory to GPU memory.

Comparing two methods, the domain decomposition method has extra communications in the first step, which results in a smaller amount of computational workload in the following steps. However, it might not be worthy costing communication to get less computation since the scalability of the PIC code is mainly limited by the memory bandwidth and the communication speed, not to mention that in the domain decomposition method the GPUs cannot be fully utilized for a typical problem size. So, in the code and the following performance test, the replication method is chosen for the depositor.

3.3. Poisson Solver

After depositing the particles onto the grid, the next step is to solve the Poisson equation on the grid. The main part of the Poisson solver is the FFT. In the GPU implementation, we used NVIDIA's CUDA Fast Fourier Transform Library (cuFFT) [34] to do this. Similar to the depositor, there are two ways to execute the Poisson solver on multi-GPUs. One is the domain decomposition method, which refers to the PIC program on the CPU and uses different processors to handle different spatial subdomains; The other is the replication method to directly make all GPUs do the same work. Because the Poisson solver is a critical and time-consuming part of the entire code, we have implemented and compared both methods.

The advantage of domain decomposition method is that by using multi-GPUs to process different spatial subdomains, each GPU will have less computation

work load and thereby the speed of the program will be improved. The drawback is that domain decomposition method requires communication among different GPUs. Currently, the GPUs cannot directly exchange information among each other, especially between different nodes. This has to be carried out through the host(CPU), which means that the communication requires three steps: copy data from the GPU to the host(CPU), communicate among CPUs, and copy data back from the CPU to the GPU.

Assuming that the grid points in the directions of X, Y and Z are N_x, N_y and N_z respectively, when performing the FFT in the X direction, the array length of the transform would be N_x, and the number of transformations would be $N_y \times N_z$. If we use 4 GPUs, the GPU 1 needs to process data as $\mathrm{rho}[N_x]\left[0 \to \frac{N_y}{4}\right][N_z]$. Similarly, the data for GPU 2, 3, and 4 would be $\mathrm{rho}[N_x]\left[\frac{N_y}{4} \to 2\frac{N_y}{4}\right][N_z]$, $\mathrm{rho}[N_x]\left[2\frac{N_y}{4} \to 3\frac{N_y}{4}\right][N_z]$, and $\mathrm{rho}[N_x]\left[3\frac{N_y}{4} \to N_y\right][N_z]$. Each GPU only needs $\frac{N_y}{4} \times N_z$ transforms. Ideally, it would take only a quarter of the time to run on 4 GPUs compared with that on a single GPU. However, after the Fourier transform in X direction, additional data moving is required for the Y-direction operation. Currently, the data on each GPU is $\mathrm{rho}[N_x]\left[(n-1)\frac{N_y}{4} \to (n)\frac{N_y}{4}\right][N_z]$, but the data needed for the Fourier transform in Y direction is $\mathrm{rho}\left[(n-1)\frac{N_x}{4} \to (n)\frac{N_x}{4}\right][N_y][N_z]$. Data transposing and exchanging among GPUs would be necessary. Since the GPUs cannot communicate with each other directly, we need to copy the data from the GPU back to the CPU memory and communicate on the CPU side, which will takes extra time. So the efficiency of the domain decomposition method in comparison to the replication method will depend on the difference between the extra data moving time and the reduced computation time. More detailed performance comparison will be presented in the performance study Section 4.

3.4. Particle Pushing

As the particles are put into different tiles after the particle reordering and deposition, we have two strategies to parallelize the particle pushing. One is to parallelize by tiles just like the depositor, while the other one is to arrange particles data back to a compact format and push in a typical parallel mover. A test was done and showed that pushing particles by tiles results in a load imbalance in the situation where some tiles contain much more particles than others. So, despite additional time of copying data, we arrange the particles data back to the compact format. The particle pushing can be summarized as: Step 1, arrange particles back to compact format array dev_ray [*N*] [6]; Step 2, push and kick particles; Step 3, arrange particles to tile format array dev_ray_tile for next

reordering and deposition.

4. Performance Tests

The performance of the beam dynamics GPU code on hybrid computer architectures was tested on a single GPU, a multi-noide GPU cluster Titan, and a GPU cluster SummitDev [35] [36]. Titan is a multi-node hybrid architecture supercomputer located at Oak Ridge National Laboratory (ORNL). It has 18,688 nodes each containing a 16-core AMD Opteron 6274 CPU with 32 GB of memory and an NVIDIA Tesla K20X GPU with 6 GB of memory [28]. Each Titan GPU contains 2688 CUDA cores at 732 MHz. The SummitDev system is an early access system of ORNL's next supercomputer Summit [29]. Each SummitDev node has 2 IBM POWER8 CPUs and 4 NVIDIA Tesla P100 GPUs. Each GPU contains 3584 cores and 16 GB memory. Before the performance study of the entire code on those GPUs, we first tested the performance of the Poisson solver which is usually the most time-consuming part of the code on Titan.

4.1. Performance Test of the Poisson Solver

We first tested the time spent on solving the Poisson equation on Titan using the domain decomposition method with $64\times64\times64$ grid points, as shown in Figure 6. The blue line is the total time, and the different columns represent the time spent on different parts of the Poisson solver. The total time scales reasonably well with an increasing number of GPUs, and reaches the minimum with 32 GPUs, after which the time for transpose and communication becomes dominant. The time needed for copying data between CPU and GPU is reduced almost linearly with the number of GPUs, while the time for communication among the CPU nodes decreases up to 32 GPUs but begins to increase after the communication becomes dominant. Looking into the detail, we can see that the computation time only takes a very small fraction of the total time, while the time for copying data between CPU and GPU and the communication among nodes dominates the total time.

We then tested this parallel strategy using larger problem size. Figure 7 shows the solver time as a function of GPUs with $128\times128\times128$ grid points. It is seen

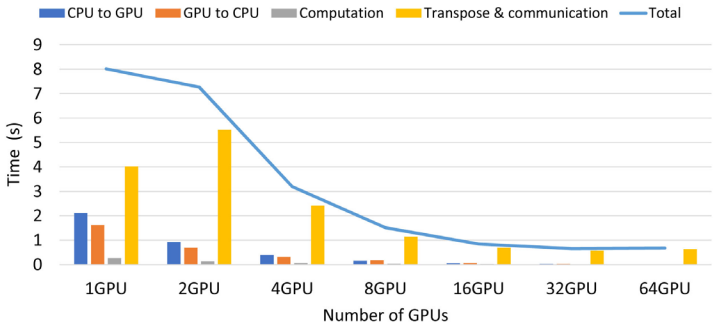

Figure 6. The scalability of the Poisson solver under domain decomposition method using $64\times64\times64$ grid points.

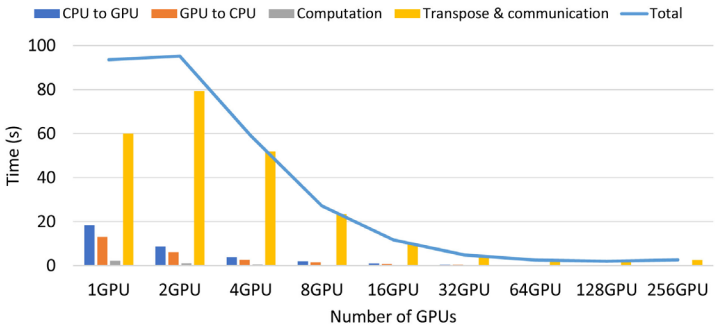

Figure 7. The scalability of the Poisson solver under domain decomposition method using $128\times128\times128$ grid points.

that in this case, the minimum time is reached with 128 GPUs since the amount of computation becomes larger. However, using the domain decomposition method, the minimum computing time of the solver in both $64\times64\times64$ and $128\times128\times128$ cases is still larger than the computing time on a single GPU without data copying and communication, which will be the time using the replication method. This minimum total computing time is mainly limited by the memory bandwidth between the CPU and the GPU and the communication speed among the CPU nodes. So, this parallelization strategy would not be very useful until the system has large enough memory bandwidth to copy data. It is expected the next generation GPU from NVIDIA would allow direct copy technology, which can directly communicate among multiple GPUs and will reduce the data copying time significantly. In that case, it would be more efficient to use the domain decomposition parallel strategy in the Poisson Solver. At present in the following performance study, the replication method is used to let all GPUs run the same Poisson solver.

4.2. Performance Study on a Single GPU

The performance of the GPU beam dynamics PIC code is first tested on a single NVIDIA GeForce GTX 1060 GPU with 6GB memory size. As a comparison, we also run the CPU code on an AMD Opteron(TM) Processor 6376 with 2.3 GHz clock speed. The speedup is calculated by the CPU runtime divided by the GPU runtime. In this performance test, the grid number is $64\times64\times64$ while the particle number varies from 16 thousand to 1.6 million.

Figure 8 shows the speedup as a function of the number of particles using the single GPU. For small problem size, the speed up of the entire PIC code is over 50 and decreases to about 30 as the number of particles increases to 1.6 million. There is a large variation in the speedup of individual function of the code. The speedup of some functions, such as depositor, pusher&kicker, and output, increases with the increase of the particle number. However, the speedup of the Poisson solver, colored as orange in Figure 8, is about 64 and is independent of the change of the particle number. The light blue and dark blue columns are the speedups of the depositor and the diagnostic output of the charged particle beam

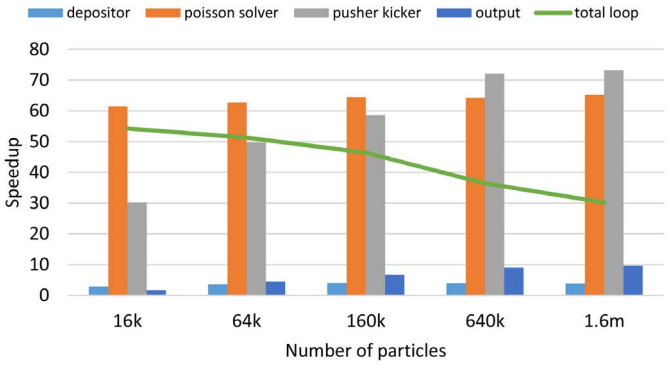

Figure 8. Speedup of the beam dynamics GPU PIC Code on a single GPU versus the number of particles.

information. The GPU's depositor also includes the particle reordering operation. Because of the irregularity of reordering, the speedup of depositor is relatively low. The diagnostic output also contains the calculation of the statistics of beam distribution. The reason for low speedup is due to the limit of output bandwidth. The relatively low speedups of the depositor and the output reduce the speedup of the entire code. The speedup of the entire code decreases when the particle number becomes larger. The reason is that the time consumed by the depositor, which has a lower speedup, dominates when the particle number becomes larger, as shown in Figure 9.

4.3. Performance Study on Multi-Node GPUs

After testing on a single GPU, we ran performance test of the GPU PIC code on the multi-node Titan GPUs. Figure 10 shows the results with 1.6 million particles. The total computing time decreases with the increase of the GPU number, and reaches the minimum with 32 GPUs. This is because the time consumed by pusher&kicker and depositor become dominant in the large particle number case as seen in Figure 9. Those functions scale well on multiple GPUs as the number of particles on each GPU becomes less, the amount of computation decreases too.

We further tested the performance of the GPU PIC code using a larger number of particles, 16 million particles. The total computing time as a function of the number of GPUs is shown in Figure 11. It is seen that the scalability of the code improves and the minimum computing time reaches 64 GPUs in this test. In the example above, we could not run the test on 1 or 2 GPUs due to the limit of the GPU memory size. Unlike CPU memory, which can be easily extended, the GPU memory is fixed in a given GPU model. Ideally, for a GPU with memory size of 6 GB, the maximum particle number is about 80 million. Here, each particle has 9 attributes and each is stored as a double precision number. However, it is not practical to attain this number in the real simulation since multiple copies of the particle array are used in the code. This is also affected by the fragment of the GPU memory. Besides the computing efficiency, the limit of memory size is another reason why we need to use multiple GPUs.

Similar to the scaling study on Titan, we also carried out a scaling study of the GPU PIC code on SummitDev, a more advanced GPU-accelerated early user test supercomputer at OCLF. In this computer, the direct communication among multiple GPUs are not enabled yet. Figure 12 shows the total computing time as a function of the number of GPUs with 1.6 million particles. The total time decreases with the increase of GPUs, and reaches the minimum with 16 GPUs. Compared with the same problem size running on the Titan, it takes 50% less time due to the improved hardware capability. Figure 13 shows the results with 16 million particles. Limited by the GPU memory size, the code cannot run on one GPU. In this case, the total computing time monotonically decreases due to the availability of a larger amount of computation.

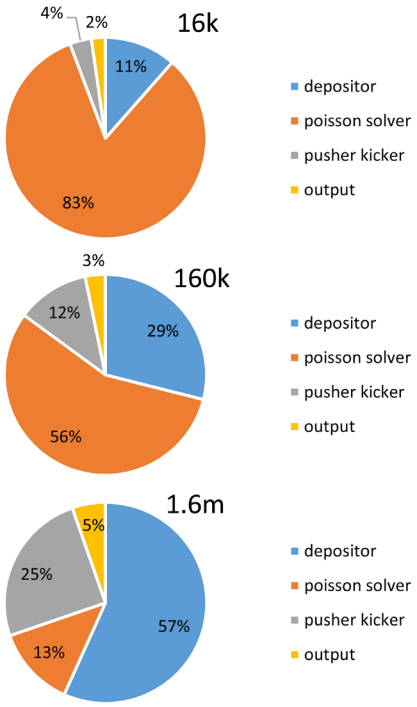

Figure 9. The percentage of time taken by each part of the program with different number of particles.

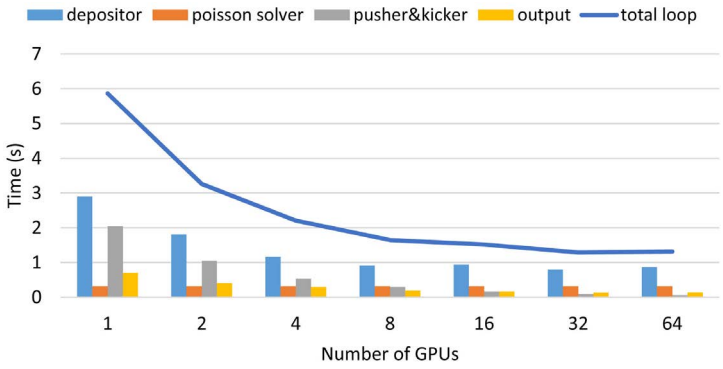

Figure 10. The scalability of the PIC code using $64\times64\times64$ grid points and $1.6M$ particles on Titan.

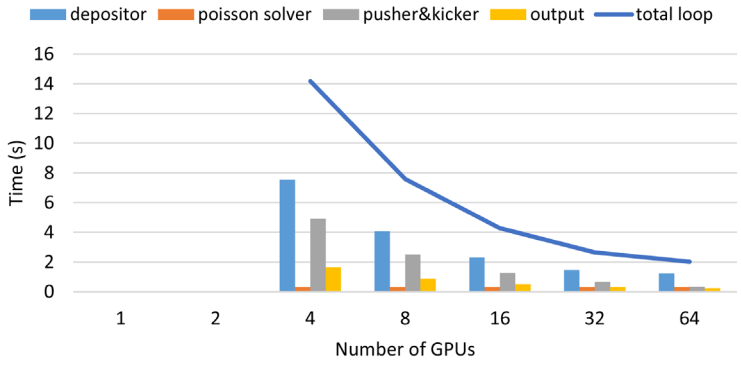

Figure 11. The scalability of the PIC code using $64\times64\times64$ grid points and $16M$ particles on Titan.

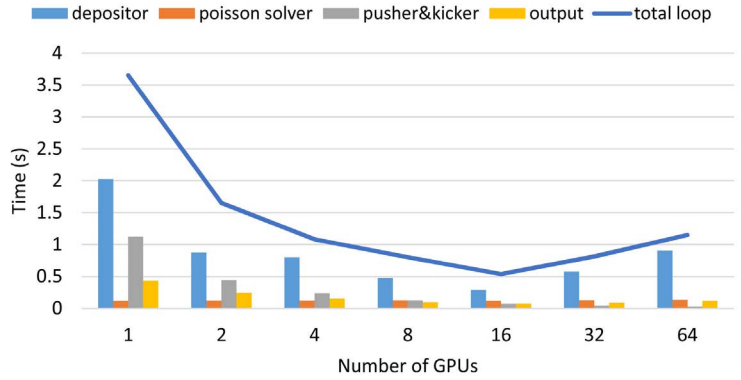

Figure 12. The scalability of the PIC code using $64\times64\times64$ grid points and $1.6M$ particles on SummitDev.

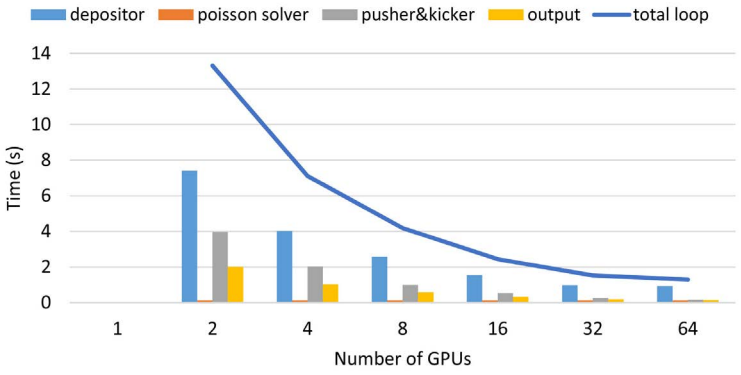

Figure 13. The scalability of the PIC code using $64\times64\times64$ grid points and $16M$ particles on SummitDev.

5. Conclusion

A multi-particle parallel beam dynamics simulation code based on the PIC method was implemented and optimized on hybrid multi-node GPU architectures using the CUDA parallel computing platform. The GPU code structure and the parallel strategy were discussed to avoid race condition and to achieve better performance. On a single GPU card, we achieved a maximum speedup of more than 50 compared with a single CPU core. The GPU PIC code

also shows reasonably good scalability (up to 64 GPUs) on multi-node GPU clusters Titan and SummitDev when the particle number is moderate. This scalability will further improve with the use of a large number of particles (>100 million), which is needed in some high-resolution accelerator beam dynamics applications.

Acknowledgements

We would like to thank Dr. Hongzhan Shan for helpful discussions. This work was supported by the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. One of the author, Zhicong Liu, would like to extend his thanks for the financial support from China Scholarship Council (CSC, File No. 201604910876). This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Birdsall, C.K. (1991) Particle-in-Cell Charged-Particle Simulations, Plus Monte Carlo Collisions with Neutral Atoms, PIC-MCC. *IEEE Transactions on Plasma Science*, **19**, 65-85. https://doi.org/10.1109/27.106800

[2] Friedman, A., Grote, D.P. and Haber, I. (1992) Three-Dimensional Particle Simulation of Heavy-Ion Fusion Beams. *Physics of Plasmas*, **4**, 2203-2210. https://doi.org/10.1063/1.860024

[3] Qiang, J., Ryne, R.D., Habib, S. and Decyk, V. (2000) An Object-Oriented Parallel Particle-in-Cell Code for Beam Dynamics Simulation in Linear Accelerators. *Journal of Computational Physics*, **163**, 434-451. https://doi.org/10.1006/jcph.2000.6570 http://www.sciencedirect.com/science/article/pii/S0021999100965707

[4] Qiang, J., Furman, M.A. and Ryne, R.D. (2004) A Parallel Particle-in-Cell Model for Beam: Beam Interaction in High Energy Ring Colliders. *Journal of Computational Physics*, **198**, 278-294. https://doi.org/10.1016/j.jcp.2004.01.008

[5] Amundson, J., Spentzouris, P., Qiang, J. and Ryne, R. (2006) Synergia: An Accelerator Modeling Tool with 3-D Space Charge. *Journal of Computational Physics*, **211**, 229-248. http://www.sciencedirect.com/science/article/pii/S0021999105002718 https://doi.org/10.1016/j.jcp.2005.05.024

[6] Qiang, J., Lidia, S., Ryne, R.D. and Limborg-Deprey, C. (2006) A Three-Dimensional Quasi-Static Model for High Brightness Beam Dynamics Simulation. *Physical Review Accelerators and Beams*, **9**, Article ID: 044204. https://doi.org/10.1103/PhysRevSTAB.9.044204

[7] Uriot, D. and Pichoff, N. (2015) Tracewin, CEA Saclay.

[8] Batygin, Y.K. (2005) Particle-in-Cell Code BEAMPATH for Beam Dynamics Simulations in Linear Accelerators and Beamlines. *Nuclear Instruments and Methods in Physics Research Section A*: *Accelerators*, *Spectrometers*, *Detectors and Associated Equipment*, **539**, 455-489. https://doi.org/10.1016/j.nima.2004.10.029

[9] Pharr, M. and Fernando, R. (2005) GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. Addison-Wesley Professional, Boston, MA.

[10] Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E. and Purcell, T.J. (2007) A Survey of General-Purpose Computation on Graphics Hardware. Computer Graphics Forum, **26**, 80-113. https://doi.org/10.1111/j.1467-8659.2007.01012.x

[11] Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E. and Phillips, J.C. (2008) GPU Computing. *Proceedings of the IEEE*, **96**, 879-899. https://doi.org/10.1109/JPROC.2008.917757

[12] Geforce (n.d.) Geforce GTX 1060 Graphics Cards from Nvidia Geforce https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1060/

[13] Nvidia, C. (2010) Programming Guide.

[14] Sanders, J. and Kandrot, E. (2010) CUDA by Example: An Introduction to General-Purpose GPU Programming. Portable Documents, Addison-Wesley Professional, Boston, MA.

[15] Stantchev, G., Dorland, W. and Gumerov, N. (2008) Fast Parallel Particle-to-Grid Interpolation for Plasma PIC Simulations on the GPU. *Journal of Parallel and Distributed Computing*, **68**, 1339-1349. https://doi.org/10.1016/j.jpdc.2008.05.009

[16] Burau, H., Widera, R., Honig, W., Juckeland, G., Debus, A., Kluge, T., Schramm, U., Cowan, T.E., Sauerbrey, R. and Bussmann, M. (2010) PIConGPU: A Fully Relativistic Particle-in-Cell Code for a GPU Cluster. *IEEE Transactions on Plasma Science*, **38**, 2831-2839. https://doi.org/10.1109/TPS.2010.2064310

[17] Decyk, V.K. and Singh, T.V. (2011) Adaptable Particle-in-Cell Algorithms for Graphical Processing Units. *Computer Physics Communications*, **182**, 641-648. https://doi.org/10.1016/j.cpc.2010.11.009

[18] Joseph, R.G., Ravunnikutty, G., Ranka, S., D'Azevedo, E. and Klasky, S. (2011) Efficient GPU Implementation for Particle in Cell Algorithm. 2011 *IEEE International Parallel & Distributed Processing Symposium*, Anchorage, AK, 16-20 May 2011, 395-406. https://doi.org/10.1109/IPDPS.2011.46

[19] Rossi, F., Londrillo, P., Sgattoni, A., Sinigardi, S. and Turchetti, G. (2012) Towards Robust Algorithms for Current Deposition and Dynamic Load-Balancing in a GPU Particle in Cell Code. *AIP Conference Proceedings*, **1507**, 184-192. https://doi.org/10.1063/1.4773692

[20] Bastrakov, S., Donchenko, R., Gonoskov, A., Efimenko, E., Malyshev, A., Meyerov, I. and Surmin, I. (2012) Particle-in-Cell Plasma Simulation on Heterogeneous Cluster Systems. *Journal of Computational Science*, **3**, 474-479. https://doi.org/10.1016/j.jocs.2012.08.012

[21] Chen, G., Chacon, L. and Barnes, D.C. (2012) An Efficient Mixed-Precision, Hybrid CPU-GPU Implementation of a Nonlinearly Implicit One-Dimensional Particle-in-Cell Algorithm. *Journal of Computational Physics*, **231**, 5374-5388. https://doi.org/10.1016/j.jcp.2012.04.040

[22] Azevedo, E.F.D., *et al.* (2013) Hybrid MPI/OPENMP/GPU Parallelization of XGC1 Fusion Simulation Code. *The International Conference for High-Performance Computing, Networking, Storage and Analysis*, Denver, CO, 17-22 November.

[23] Ibrahim, K.Z., *et al.* (2013) Analysis and Optimization of Gyrokinetic Toroidal Simulations on Homogenous and Heterogenous Platforms. *The International Journal of High Performance Computing Applications*, **27**, 454-473. https://doi.org/10.1177/1094342013492446

[24] Decyk, V.K. and Singh, T.V. (2014) Particle-in-Cell Algorithms for Emerging Computer Architectures. *Computer Physics Communications*, **185**, 708-719. https://doi.org/10.1016/j.cpc.2013.10.013

[25] Pang, X. and Rybarcyk, L. (2014) GPU Accelerated Online Multi-Particle Beam Dynamics Simulator for Ion Linear Particle Accelerators. *Computer Physics Communications*, **185**, 744-753. https://doi.org/10.1016/j.cpc.2013.10.033

[26] Shah, H., Kamaria, S., Markandeya, R., Shah, M. and Chaudhury, B. (2017) A novel Implementation of 2D3V Particle-in-Cell (PIC) Algorithm for Kepler GPU Architecture. *Proceedings of* 2017 *IEEE* 24*th International Conference on High-Performance Computing* (*HiPC*), **1**, 378-387.

[27] Fatemi, S., Poppe, A.R., Delory, G.T. and Farrell, W.M. (2017) AMITIS: A 3D GPU-Based Hybrid-PIC Model for Space and Plasma Physics. *Journal of Physics: Conference Series*, **837**, Article ID: 012017. https://doi.org/10.1088/1742-6596/837/1/012017

[28] Titan (2019) Titan Advancing the Era of Accelerated Computing. https://www.olcf.ornl.gov/olcf-resources/compute-systems/titan/

[29] Summit (2019) Introducing Summit. https://www.olcf.ornl.gov/summit/

[30] Hockney, R.W. and Eastwood, J.W. (1988) Computer Simulation Using Particles. Adam Hilger, New York.

[31] Netzer, R.H. and Miller, B.P. (1992) What Are Race Conditions?: Some Issues and Formalizations. *ACM Letters on Programming Languages and Systems* (*LOPLAS*), **1**, 74-88. https://doi.org/10.1145/130616.130623

[32] Quinn, M.J. (2004) Parallel Programming in C with MPI and Open MP. McGraw-Hill Inc., New York.

[33] Qiang, J. and Li, X. (2010) Particle-Field Decomposition and Domain Decomposition in Parallel Particle-in-Cell Beam Dynamics Simulation. *Computer Physics Communications*, **181**, 2024-2034. https://doi.org/10.1016/j.cpc.2010.08.021

[34] Nvidia, C. (2010) Cufft Library.

[35] Tiwari, D., Gupta, S., Gallarno, G., Rogers, J. and Maxwell, D. (2015) Reliability Lessons Learned from GPU Experience with the Titan Supercomputer at Oak Ridge Leadership Computing Facility. In: *Proceedings of the International Conference for High-Performance Computing, Networking, Storage and Analysis*, ACM, New York, 38. https://doi.org/10.1145/2807591.2807666

[36] Wells, J., Bland, B., Nichols, J., Hack, J., Foertter, F., Hagen, G., Maier, T., Ashfaq, M., Messer, B. and Parete-Koon, S. (2016) Announcing Supercomputer Summit. Technical Report, ORNL (Oak Ridge National Laboratory), Oak Ridge, TN.