# Symplectic multi-particle tracking on GPUs

Zhicong Liu[a,b,c], Ji Qiang[b,*]

[a] *Key Laboratory of Particle Acceleration Physics and Technology, Institute of High Energy Physics, Chinese Academy of Sciences, Beijing 100049, China*
[b] *Lawrence Berkeley National Laboratory, Berkeley, California 94720, USA*
[c] *University of Chinese Academy of Sciences, Beijing 100049, China*

## Abstract

A symplectic multi-particle tracking model is implemented on the Graphic Processing Units (GPUs) using the Compute Unified Device Architecture (CUDA) language. The symplectic tracking model can preserve phase space structure and reduce non-physical effects in long term simulation, which is important for beam property evaluation in particle accelerators. Though this model is computationally expensive, it is very suitable for parallelization and can be accelerated significantly by using GPUs. In this paper, we optimized the implementation of the symplectic tracking model on both single GPU and multiple GPUs. Using a single GPU processor, the code achieves a factor of 2-10 speedup for a range of problem sizes compared with the time on a single state-of-the-art Central Processing Unit (CPU) node with similar power consumption and semiconductor technology. It also shows good scalability on a multi-GPU cluster at Oak Ridge Leadership Computing Facility. In an application to beam dynamics simulation, the GPU implementation helps save more than a factor of two total computing time in comparison to the CPU implementation.

*Keywords:* Particle accelerator, Symplectic, Multi-particle tracking, GPU
*PACS:* 29.20.-c

*Corresponding author
*Email addresses:* `zhicongliu@lbl.gov` (Zhicong Liu), `jqiang@lbl.gov` (Ji Qiang)

## 1. Introduction

Numerical simulation plays an important role in beam physics study and the design of high intensity particle accelerators, where the space charge effects from Coulomb interactions of charged particles dominate. Most simulation codes in the accelerator community use the particle-in-cell (PIC) method as the self-consistent space charge solver [1, 2, 3, 4, 5, 6, 7, 8, 9]. The PIC method is an efficient algorithm to include self-consistent space charge effects in the simulation. In this algorithm, at each step, particles are deposited onto a computational grid to obtain charge density distribution on the grid. The Poisson equation is solved on the grid to yield space-charge fields. These fields are then interpolated from the grid back to the locations of particles to advance particles' momenta. Using this method, the computational complexity is reduced from $N_p^2$ of the direct particle to particle method to $\alpha N_p + \beta N_{cells} \log N_{cells}$. Here, $N_p$ is the number of macroparticles used in the simulation, $N_{cells}$ is the number of numerical grid cells, $\alpha$ is a constant depending on the scheme of deposition and interpolation, and $\beta$ is a constant associated with the numerical method used to solve the Poisson equation. Here, we have assumed that an efficient numerical method (e.g. FFT) is used to solve the Poisson equation on the grid.

In accelerator beam dynamics simulation, for a multi-particle Hamiltonian system, an important constraint is the symplectic condition. If the symplectic condition is not satisfied, some non-physical effects resulting from numerical algorithms would be introduced into the simulation, and eventually disturb the results of beam dynamic study [10]. For widely used momentum conserved PIC model, this condition is nevertheless violated. Symplectic integrators without including self-consistent space-charge effects were constructed for single particle Hamiltonian systems [10, 11, 12]. Recently, a fully symplectic multi-particle tracking model including space-charge effects was introduced and proved to be effective in serving as symplectic Poisson solver in long-term simulation [13]. This model uses a gridless spectral method to calculate the space charge fields. Here, the gridless model refers to a method that the self-consistent space-charge fields are computed directly from the ensemble of particles instead of the density distribution on a computational grid. It can effectively reduce the emittance growth associated with numerical grid heating compared with the PIC algorithm.

The gridless particle tracking model was studied for a periodic system in plasma physics with the advantage of avoiding the numerical grid heating error in the PIC model [14, 15, 16]. These gridless finite-size particle plasma models are either electrostatic or gyrokinetic and are not particle tracking models typically employed in beam physics study. The gridless particle tracking model has not been used to study space-charge effects in high intensity beams through a particle accelerator (non-periodic system) until the recent study for symplectic multi-particle tracking.

The gridless method is slower than the PIC method on serial computer. The computational complexity of this model is $\alpha N_p N_{modes}$, where $N_{modes}$ is the number of modes used to solve the Poisson equation. If one uses $16 \times 16 \times 16$ modes in the simulation, it would cost a few hundred times more computing time than the PIC model. Fortunately, this model has a very regular data structure. It can be parallelized using a particle-decomposition method with perfect load balance (This was also observed in reference [15]). Moreover, there is only one global communication at each step for the space-charge fields calculation. This makes it very suitable for massive parallelization, especially on GPUs.

The GPU, which was originally developed for computer graphics and video game, now becomes a general-purpose computing processor. In contrast to the CPU, one GPU contains several hundreds or even thousands of cores, as shown in Fig. 1. Figure 2 shows the architecture of a common house-use GPU processor, GeForce GTX 1060 [17, 18]. It is composed of a number of Streaming Multiprocessors (SM) blocks, and each SM block contains 32 CUDA cores. It uses high-bandwidth bus ($\sim$200Gb/s) to connect the on-chip memory with the computing cores and is optimized for simultaneous parallel computation, particularly for single instruction multiple data (SIMD) operations [19]. Manufacturers of GPUs have approached general-purpose computation with their own application program interfaces (APIs). The CUDA language is a parallel computing platform and programming model for GPUs developed by NVIDIA [20]. It enables a fast implementation of numerical models on GPUs and dramatically increases computing performance by harnessing the computing power of GPUs.

The PIC model has been implemented on GPUs in previous studies [21, 22, 23, 24, 25]. For example, a significant improvement of computing performance by a factor of 40 on a GPU with respect to a single CPU core computer was reported in [24]. To implement the PIC code on a GPU, one has to deal with the data locality during the deposition/interpolation stage, and the global communication during the solution of the Poisson equation. To the best of our knowledge, the implementation of the gridless model on GPUs has not been reported before. Using the CUDA language, the gridless symplectic multi-particle tracking code can be sped up significantly on GPUs. When running on a single GTX 1060 GPU, it achieves a speedup of a factor of $2-10$ for a range of problem sizes compared with the computing time on a single CPU node using 32 cores with vector processing turned on. Also, the speedup increases almost linearly with the number of GPUs for some problem sizes on a multi-GPU cluster.

In this paper, after the introduction, the symplectic multi-particle tracking model is reviewed in Section 2. Then, we present the code structure and its GPU implementation in Section 3 and performance test of the tracking code in Section 4. After that, an application example using this implementation is presented in Section 5. Finally, conclusions are drawn in Section 6.

## 2. Symplectic multi-particle tracking model

In beam dynamics simulation, a transfer map $m_i$ is symplectic if and only if its Jacobian matrix $M_i$ satisfies the following condition [26, 27]:

$$M_i^T J M_i = J \tag{1}$$

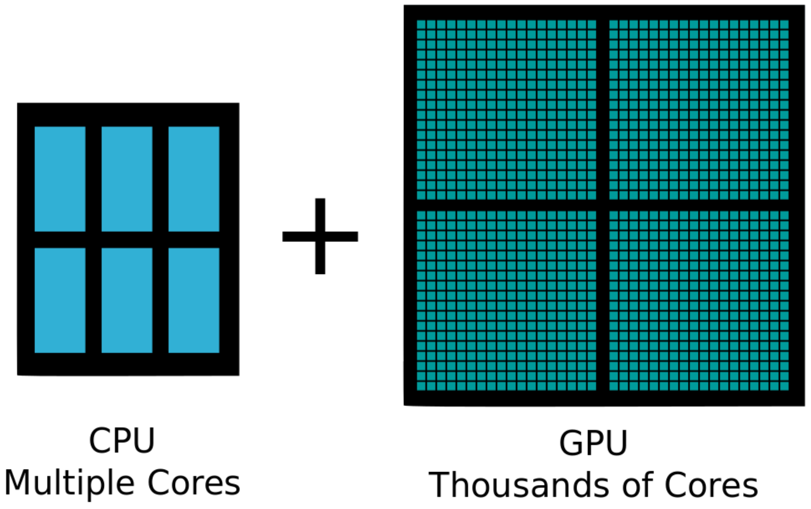


Figure 1: A schematic plot of a GPU vs a CPU.

where $J$ is a $6N \times 6N$ matrix defined as:

$$J = \begin{pmatrix} 0 & I \\ -I & 0 \end{pmatrix} \tag{2}$$

and $I$ is the $3N \times 3N$ identity matrix.

For a multi-particle system including space charge Coulomb interactions, an approximate Hamiltonian of the system can be written as:

$$H = H_1 + H_2 \tag{3}$$

where:

$$H_1 = \sum_i p_i^2/2 + \sum_i q\psi(r_i) \tag{4}$$

$$H_2 = \frac{1}{2}\sum_i \sum_j q\varphi(r_i, r_j) \tag{5}$$

The $H_1$ includes contributions from external fields, and $H_2$ includes those from space charge effects. With transfer maps $m_1$ and $m_2$ derived from $H_1$ and $H_2$, a second order integrator $m(\tau)$ can be written as [11]:

$$m(\tau) = m_1(\tau/2)\, m_2(\tau)\, m_1(\tau/2) \tag{6}$$

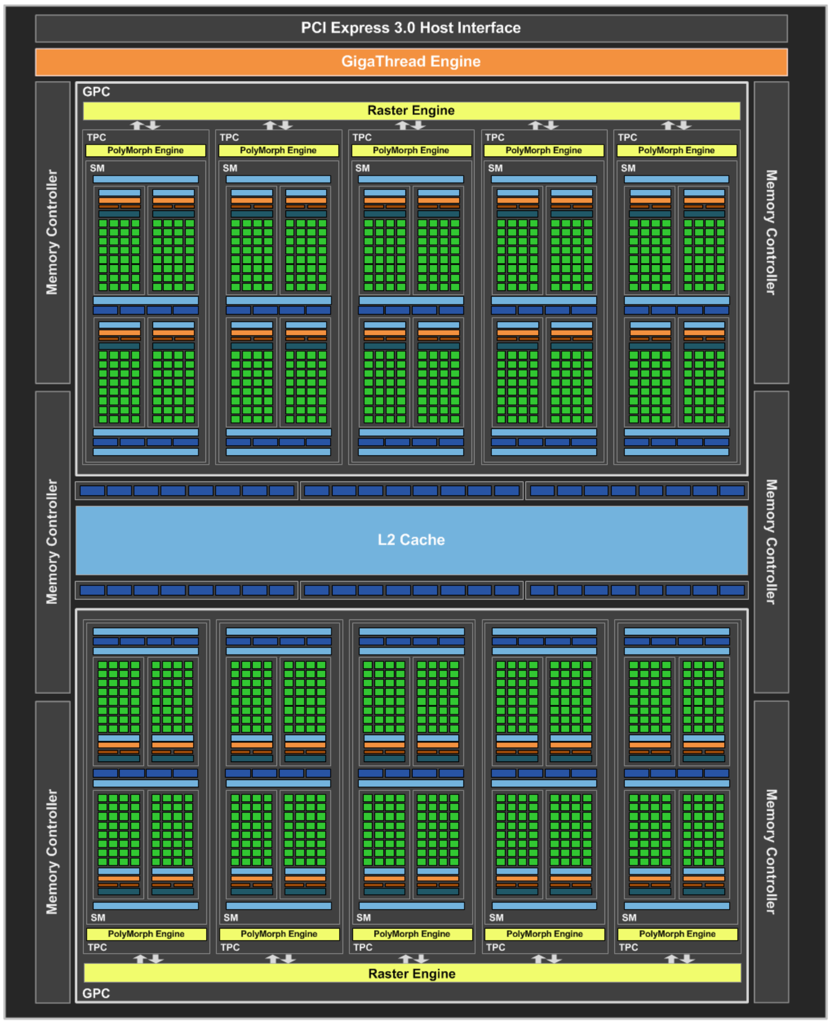


Figure 2: The architecture of the GeForce GTX 1060 GPU processor [17, 18].

If both $m_1$ and $m_2$ are symplectic, the integrator $m$ would be symplectic. The symplectic transfer map, $m_1$, can be obtained by using the single particle magnetic optics method in most accelerator elements, while the transfer map $m_2$ can be written as:

$$\begin{aligned} r_i(\tau) &= r_i(0) & (7)\\ p_i(\tau) &= p_i(0) - \frac{\partial H_2(r)}{\partial r_i}\tau & (8) \end{aligned}$$

And its Jacobian matrix is

$$M_2 = \begin{pmatrix} I & 0 \\ L & I \end{pmatrix} \tag{9}$$

where

$$L_{ij} = \frac{\partial p_i(\tau)}{\partial r_j} = -\frac{\partial^2 H_2(r)}{\partial r_i \partial r_j}\tau \tag{10}$$

is a symmetric matrix, so that the $M_2$ satisfies the symplectic condition.

In a 3D bunched beam, the $H_2$ can be written as:

$$H_2 = \kappa\gamma_0 \sum_i \sum_j \varphi(r_i, r_j) \tag{11}$$

where $\kappa = q/(lmC^2\gamma_0^2\beta_0)$, $l = C/\omega$ is the scaling length, $\beta_0 = v_0/C$, and the $\varphi$ is the space-charge Coulomb interaction potential which can be obtained from the solution of the Poisson equation. The above Hamiltonian includes both the electric potential and the longitudinal magnetic vector potential. The electric potential in the beam frame can be obtained from the solution of the Poisson equation:

$$\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} + \frac{\partial^2\phi}{\partial z^2} = -\frac{\rho}{\varepsilon_0} \tag{12}$$

with approximated boundary conditions:

$$\begin{aligned} &\phi(x=0,y,z) = 0, \quad \phi(x=a,y,z) = 0\\ &\phi(x,y=0,z) = 0, \quad \phi(x,y=b,z) = 0\\ &\phi(x,y,z=0) = 0, \quad \phi(x,y,z=c) = 0 \end{aligned} \tag{13}$$

where $a$, $b$ and $c$ are the length of boundary in each direction respectively and $c$ is large enough so that the potential goes to zero at this boundary.

The potential $\phi$ and the density $\rho$ can be expanded as:

$$\rho(x,y,z) = \sum_{l=1}^{N_l}\sum_{m=1}^{N_m}\sum_{n=1}^{N_n}\rho^{lmn}\sin(\alpha_l x)\sin(\beta_m y)\sin(\gamma_n z) \tag{14}$$

$$\phi(x,y,z) = \sum_{l=1}^{N_l}\sum_{m=1}^{N_m}\sum_{n=1}^{N_n}\phi^{lmn}\sin(\alpha_l x)\sin(\beta_m y)\sin(\gamma_n z) \tag{15}$$

where

$$\rho^{lmn} = \frac{8}{abc}\int_0^a\int_0^b\int_0^b \rho(x,y,z)\sin(\alpha_l x)\sin(\beta_m y)\sin(\gamma_n z)dxdydz \tag{16}$$

$$\phi^{lmn} = \frac{8}{abc}\int_0^a\int_0^b\int_0^b \phi(x,y,z)\sin(\alpha_l x)\sin(\beta_m y)\sin(\gamma_n z)dxdydz \tag{17}$$

and

$$\alpha_l = \frac{l\pi}{a}, \beta_m = \frac{m\pi}{b}, \gamma_n = \frac{n\pi}{c} \tag{18}$$

Substituting the above expansions into the Poisson equation and making use of the orthonormal condition of the sine functions, we obtain

$$\phi^{lmn} = \frac{\rho^{lmn}}{\varepsilon_0(\alpha_l^2+\beta_m^2+\gamma_n^2)} \tag{19}$$

In the multi-particle tracking, the charge density $\rho(x,y,z)$ can be represented as:

$$\rho(x,y,z) = \sum_{j=1}^{N_p} w\delta(x-x_j)\delta(y-y_j)\delta(z-z_j) \tag{20}$$

where $w$ is the charge weight of each individual particle and $\delta$ is the Dirac function.

Using the above equations, we obtain the electric potential as:

$$\phi(x,y,z) = \frac{1}{\varepsilon_0}\frac{8}{abc}w\times$$

$$\sum_{j=1}^{N_p}\sum_{l=1}^{N_l}\sum_{m=1}^{N_m}\sum_{n=1}^{N_n}\frac{\sin(\alpha_l x_j)\sin(\beta_m y_j)\sin(\gamma_n z_j)\sin(\alpha_l x)\sin(\beta_m y)\sin(\gamma_n z)}{(\alpha_l^2+\beta_m^2+\gamma_n^2)} \tag{21}$$

From the above electric potential, the interaction potential $\varphi$ between particles $i$ and $j$ can be written as:

$$\begin{aligned}\varphi(x_i, y_i, z_i, x_j, y_j, z_j) &= \frac{1}{\epsilon_0}\frac{8}{abc} w \sum_{l=1}^{N_l}\sum_{m=1}^{N_m}\sum_{n=1}^{N_n} \frac{1}{\alpha_l^2+\beta_m^2+\gamma_n^2} \sin(\alpha_l x_j)\sin(\beta_m y_j)\sin(\gamma_n z_j) \\ &\quad \times \sin(\alpha_l x_i)\sin(\beta_m y_i)\sin(\gamma_n z_i) \end{aligned} \tag{22}$$

Then, the Hamiltonian $H_2$ corresponding to the space-charge interactions can be expressed as:

$$\begin{aligned} H_2 = {} & \frac{1}{2\varepsilon_0}\frac{8}{abc} w\kappa\gamma_0 \\ & \times \sum_{i=1}^{N_p}\sum_{j=1}^{N_p}\sum_{l=1}^{N_l}\sum_{m=1}^{N_m}\sum_{n=1}^{N_n}\left[\frac{\sin(\alpha_l x_j)\sin(\beta_m y_j)\sin(\gamma_n z_j)}{(\alpha_l^2+\beta_m^2+\gamma_n^2)}\right. \\ & \qquad \left. \times \sin(\alpha_l x_i)\sin(\beta_m y_i)\sin(\gamma_n z_i)\right] \end{aligned} \tag{23}$$

Finally, we obtain the transfer map $m_2$ for the space-charge kick in the $x$ direction as:

$$\begin{cases} x_i(\tau) = x_i(0) \\ p_{xi}(\tau) = p_{xi}(0) - \tau\dfrac{1}{\varepsilon_0}\dfrac{8}{abc} w\kappa\gamma_0 \\ \quad \times \displaystyle\sum_{j=1}^{N_p}\sum_{l=1}^{N_l}\sum_{m=1}^{N_m}\sum_{n=1}^{N_n}\left[\frac{\alpha_l\sin(\alpha_l x_j)\sin(\beta_m y_j)\sin(\gamma_n z_j)}{(\alpha_l^2+\beta_m^2+\gamma_n^2)}\right. \\ \qquad \left. \times \cos(\alpha_l x_i)\sin(\beta_m y_i)\sin(\gamma_n z_i)\right] \end{cases} \tag{24}$$

where momentum $p_{xi}$ is normalized by $mC$, and the maps in $y$ and $z$ directions are similar, and can be found in [13].

## 3. Code optimization on GPU

In most accelerator elements, particles are advanced by a number of steps in the simulation. Using the second-order symplectic integrator described in the above section, at each step, a particle is first pushed using the external transfer map for half step, then kicked by the space charge transfer map for one step, and then pushed by external transfer map for another half step. The space charge kicker can consume more than 90% of the total computing time for some problem size. Porting the CPU code to the GPU, we optimized the implementation to make it more suitable on the GPU architecture and to improve the computing efficiency. The code computes the space charge fields in three subroutines: subroutine one computes the trigonometric function, subroutine two computes the $\Phi^{lmn}$, and subroutine three computes the space-charge fields and pushes the particle. Each subroutine consists of one or two kernels, and the optimization strategy for each kernel is different. Details about optimization strategies are discussed in the following subsections. All real number numerical operations use double precision accuracy in the GPU implementation.

## *3.1. Calculation of trigonometric function*

The computing of a trigonometric function is computational expensive since it involves many floating point operations. In order to save computational cost, it is important to minimize the number of trigonometric function calculations inside the code. In this study, we defined three temporary variables for each particle to calculate the trigonometric functions as:

$$\begin{aligned} S_j^l &= sin(\alpha_l x_j), \quad & C_j^l &= cos(\alpha_l x_j) \\ S_j^m &= sin(\alpha_m x_j), \quad & C_j^m &= cos(\alpha_m x_j) \\ S_j^n &= sin(\alpha_n x_j), \quad & C_j^n &= cos(\alpha_n x_j) \end{aligned} \tag{25}$$

where $j$ is the index of a particle, and $l$, $m$ and $n$ are the indices of a spectral mode in three directions.

Then, the transfer map $m_2$ (equation 24) in one direction is rewritten as:

$$p_{xi}(\tau) = p_{xi}(0) - \tau \frac{1}{\varepsilon_0} \frac{8}{abc} \omega \kappa \gamma_0 \sum_{j=1}^{N_j} \sum_{l=1}^{N_l} \sum_{m=1}^{N_m} \sum_{n=1}^{N_n} \frac{\alpha_l S_j^l S_j^m S_j^n C_i^l S_i^m S_i^n}{(\alpha_l^2 + \beta_m^2 + \gamma_n^2)} \tag{26}$$

Compared with the equation 24, the new transfer map saves a lot of computational costs by avoiding computing these trigonometric functions inside four loops. In this subroutine, each particle takes one thread. Its structure is relatively simple and does not require significant change from the CPU code.

This subroutine takes only about 2% of the total computing time consumed by the space charge kicker. However, it generates $2 \times (N_l + N_m + N_n) \times N_j$ data, which uses most space of the global memory and limits the problem size that can be solved. This is an example of saving computing time at the cost of memory usage. On the GTX 1060 GPU with 6 GB global memory, including the inevitable memory fragmentation, the code can handle about one million particles with $64 * 64 * 64$ modes.

The speed of this subroutine is limited by the global memory accessing bandwidth. To further improve this part, the layout of particle data is modified to form structure of array (SoA) instead of array of structure (AoS), in order to get coalesced memory access. The particle data on the CPU side is allocated with page-locked memory, to achieve a faster data copy speed between the CPU and the GPU.

### *3.2. Calculation of* $\Phi^{lmn}$

In the transfer map Eq. 26, the summation of index $j$ is for every particle, and the sequence of summation can be switched to save computational cost. Using a three-dimensional temporary variable $\Phi^{lmn}$:

$$\Phi^{lmn} \equiv \sum_{j=1}^{N_j} S_j^l S_j^m S_j^n, \tag{27}$$

if we compute the $\Phi^{lmn}$ for $N_l \times N_m \times N_n$ modes using all particles first and store this three dimensional variable, the transfer map Eq. 26 can be rewritten as:

$$p_{xi}(\tau) = p_{xi}(0) - \tau \frac{1}{\varepsilon_0} \frac{8}{abc} w\kappa\gamma_0 \sum_{l=1}^{N_l} \sum_{m=1}^{N_m} \sum_{n=1}^{N_n} \frac{\Phi^{lmn} \alpha_l C_i^l S_i^m S_i^n}{(\alpha_l^2 + \beta_m^2 + \gamma_n^2)} \tag{28}$$

In this way, the computational complexity is reduced from $O(N_p^2 * N_{modes})$ to $O(N_p * N_{modes})$, which makes the symplectic particle tracking model feasible.

The purpose of this subroutine is to get the $\Phi^{lmn}$ for each mode, so it is natural for every thread to take a mode. However, in a typical simulation, one uses only $16 \times 16 \times 16$ modes, the number of threads is too small for a GPU with many cores (1280 cores on the GTX 1060). To achieve better load balancing, the CUDA stream technique is used to attain higher concurrency. We divide particles into several groups and introduce a temporary variable $\Phi^{lmn}_{temp,i}$ as:

$$\Phi^{lmn}_{temp,i} \equiv \sum_{j=Nstart_i}^{Nend_i} S^l_j S^m_j S^n_j \tag{29}$$

Then, the total $\Phi^{lmn}$ is obtained by the summation:

$$\Phi^{lmn} = \sum_i \Phi^{lmn}_{temp,i} \tag{30}$$

The speed of this subroutine is also limited by the memory bandwidth. In the implementation, before we calculate the $\Phi^{lmn}$, a transpose of the $S_j$ is performed first in order to make use of coalesced reading.

### *3.3. Calculation of particle pushing*

Take $x$ direction as an example, the change of momentum can be rewritten as:

$$\Delta p_{xi} \equiv \tau \frac{1}{\varepsilon_0} \frac{8}{abc} \omega \kappa \gamma_0 \sum_{l=1}^{N_l} \sum_{m=1}^{N_m} \sum_{n=1}^{N_n} \frac{\Phi^{lmn} \alpha_l C^l_i S^m_i S^n_i}{(\alpha_l^2 + \beta_m^2 + \gamma_n^2)} \tag{31}$$

The $\Delta p_{xi}$ is obtained using $\Phi^{lmn}$, $S_i$ and $C_i$ which we had discussed in the above two subsections. The transfer map 28 can be rewritten in a concise form:

$$p_{xi}(\tau) = p_{xi}(0) - \Delta p_{xi} \tag{32}$$

Limited by the number of registers, the calculation of $\Delta p_{xi}$ and the particle pushing are executed separately in three directions in order to achieve high GPU occupancy. Because the innermost loop will access $\Phi^{lmn}$ in different sequence, a transpose for $\Phi^{lmn}$ is necessary before calling this subroutine to achieve coalesced reading.

In the subroutine of each direction, each thread takes one particle. Limited by the size of the constant memory and the size of the shared memory, the subroutine three has two branches: one is for the case that mode number is less than $20 \times 20 \times 20$, and the other is for the case that mode number is greater than $20 \times 20 \times 20$.

### *3.3.1. Branch 1: number of modes* $<= 20 \times 20 \times 20$

When the mode number is less than $20 \times 20 \times 20$, the constant memory is used to store $\Phi^{lmn}$. Constant memory is a special memory on GPU optimized for broadcasting. It is fast when multiple threads access the same address at the same time. Hence, it is suitable to hold $\Phi^{lmn}$, which would be read by every thread. It is the constant memory size that determines the threshold mode numbers in the two branches. The total amount of constant memory in a common GPU card is 65536 bytes, which can only hold 8192 double-precision floating numbers. This corresponds to about $20 \times 20 \times 20$ modes.

In this branch, the kernel in each direction uses shared memory to store $S_i$ and $C_i$ in the innermost loop. The on-chip shared memory is small, only 64 KB per Streaming Multiprocessor block. It is much faster to access this shared memory than the global memory. Limited by the size of the shared memory, the GPU occupancy is only 25%. This is still useful since the share memory latency is much lower (roughly 100 times) than the uncached global memory latency [19].

One test was done to evaluate the speed of using the global memory rather than the shared memory. In this test, the global memory accessing latency is hidden by carefully arranging the memory and by letting all threads in a warp access the sequent memory address. Here, a warp is the minimum number of threads that execute the same instruction at the same time (usually 16 or 32 on a GPU). The GPU occupancy can reach near 100% because it is not limited by shared memory size any more. However, the time spent in this global memory test is nearly as twice as that using the shared memory since the global memory accessing is slow and frequent.

### *3.3.2. Branch 2: number of modes* $> 20 \times 20 \times 20$

When the mode number is greater than $20 \times 20 \times 20$, both the shared memory and the constant memory limit the speed of this kernel. The straightforward way to obtain $\Delta p_{xi}$ and to push particle in the above branch will not work.

Due to the constant memory size limitation, the $\Phi^{lmn}$ will be stored in the global memory instead of the constant memory in this branch. Due to the use of coalesced reading by multiple threads to access the same address, the speed of using the global memory is only 10% slower than that using the constant memory.

Limited by the size of the shared memory, we separate the calculation of $\Delta p_{xi}$ and the pushing particle. In the calculation of $\Delta p_{xi}$, the modes are divided into several groups to meet the limitation of the shared memory size. This is similar to that in section 3.2. Each particle takes several threads and each thread handles corresponding modes and obtains temporary variable $\Delta p_{xi}^{temp,j}$:

$$\Delta p_{xi}^{temp,j} \equiv \tau \frac{1}{\varepsilon_0} \frac{8}{abc} \omega \kappa \gamma_0 \sum_{l=1}^{N_l} \sum_{m=1}^{N_m} \sum_{n=Nstart_j}^{Nend_j} \frac{\Phi^{lmn} \alpha_l C_i^l S_i^m S_i^n}{(\alpha_l^2 + \beta_m^2 + \gamma_n^2)} \tag{33}$$

Then the $\Delta p_{xi}^{temp,j}$ is summed up to push the particle using the Eq. 32.

$$\Delta p_{xi} = \sum_j \Delta p_{xi}^{temp,j} \tag{34}$$

There are trade-offs in this way. By dividing the number of modes into multiple segments, more memory is needed to store the temporary variable $\Delta p_{xi}^{temp,j}$. The additional memory usage is proportional to both the number of particles and the number of modes. This results in reduction of the maximum allowable number of particles by about 20% with the given memory size.

## 4. Performance test on GPUs

We have done two tests to measure the efficiency and scalability of the symplectic multi-particle tracking model on GPUs. The first one is to run the code on a common home-use GPU card and the efficiency is compared with that running on a single state-of-the-art CPU node with 32 cores (Intel Xeon $Phi^{TM}$ 7250) at the National Energy Research Scientific Computing Center (NERSC) [28]. The second test is to run the code on a GPU cluster, Titan, a hybrid-architecture supercomputer located at the Oak Ridge Leadership Computing Facility (OLCF), to show the speedup scaling with the number of GPUs.

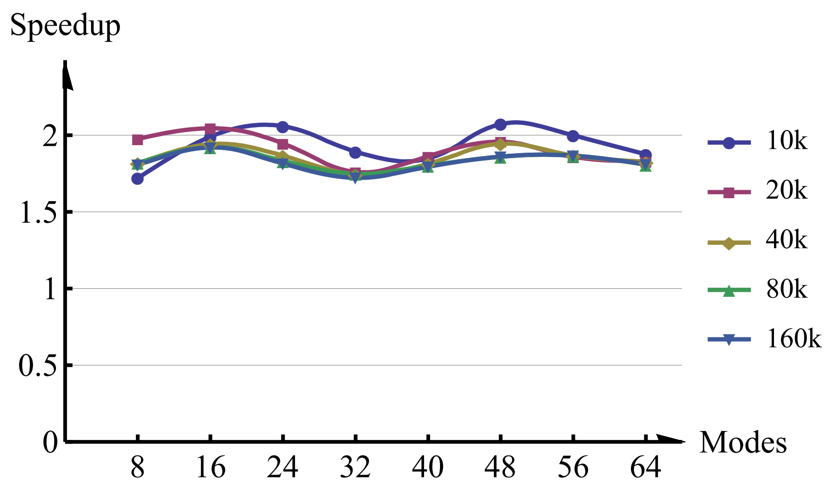
## *4.1. Single GPU Speedup*

We tested the performance of the symplectic tracking code implemented using the CUDA language on a single GPU processor. The single GPU is GeForce GTX 1060 with 6 GB global memory (Pascal architecture), and the implementation uses CUDA version 8.0. The power consumption for this GPU is 130 W and the semiconductor technology used in this processor is based on 16-nanometer manufacturing technology. The speedup is calculated by using the CPU runtime divided by the GPU runtime. The CPU code with the same optimization of the trigonometric function and the potential calculation as discussed in the preceding section was run on a node (Intel Xeon $Phi^{TM}$ processor 7250) using 32 cores with vector processing turned on. The power consumption of the Intel Xeon Phi processor node is 215 W and the semiconductor technology used in the processor is based on 14-nanometer manufacturing technology. The Intel Xeon Phi processor node consists of 68 cores. Here, we used only about half of the number of cores so that the power consumptions in both the GPU computing and the CPU computing are about the same.

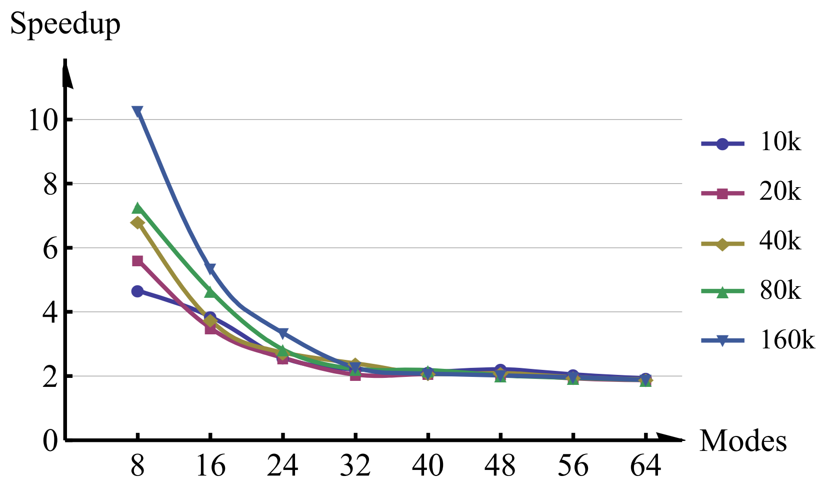
In the measurement of computing time, two comparisons are made separately for space charge kicker only and the entire code. The space charge kicker includes the time of copying data from the CPU side to the GPU side, getting the space charge fields, kicking particles, and copying data back to the CPU side, while the entire code includes functions, besides the space charge kicker, such as initialization, external element field transfer map kicker, coordinate transformation, parameter input, and diagnostic output.

As a performance comparison, we calculated the speedup by comparing with the time on a CPU node (Intel Xeon Phi with 32 cores and vector processing turned on) with similar power consumption and semiconductor technology.

Figure 3a shows the speedup of the space charge kicker with different problem sizes. The speedup of the GPU implementation for the space charge kicker is about a factor of two and does not change significantly with different problem sizes. This is due to the improvement of computing efficiency on both the GPU processor and the CPU node with larger problem size. Figure 3b shows the



(a) Speed up of the space charge kicker on a single GPU compared with a CPU node.



(b) Speed up of the entire code on a single GPU.

Figure 3: Speedup versus mode number on a single GPU card compared with a CPU node with different number of macroparticles in the simulation.

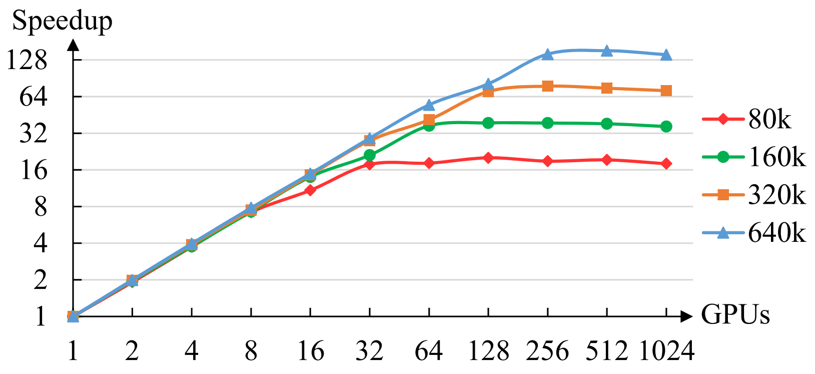
speedup of the total computing time with different problem sizes. When the number of spectral modes is relatively small (e.g. $8 \times 8 \times 8$), the speedup can reach beyond 10. This is due to the fact that for this problem size, the time spent in the space charge kicker is small. The computing time spent in the other part of code becomes dominant. This part of code (e.g. external transfer map) makes better use of the large number of cores in the GPU and has higher speedup in comparison to the CPU implementation. When the number of spectral modes increases, the time spent in the space charge kicker becomes dominant and the speedup approaches a factor of two in comparison to the CPU implementation.

In the above speedup measurement, on the single CPU node, the vector processing capability was turned on during the simulation. Without using the vector processing, the GPU speedup can be doubled for larger problem sizes.

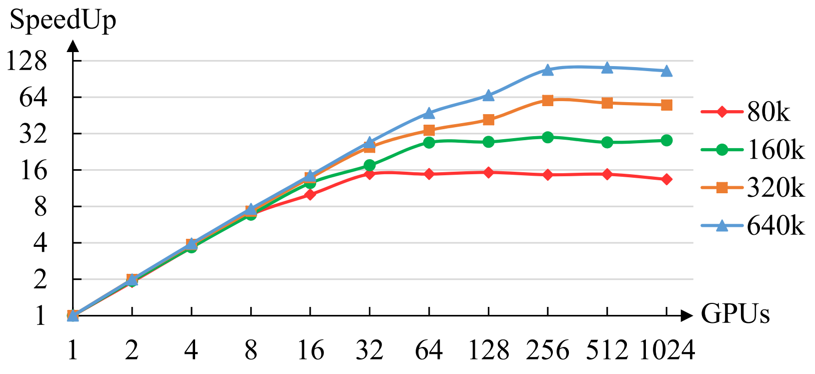
### *4.2. GPU cluster speedup*

A strong scaling test of the symplectic GPU code was done on a multi-GPU cluster to check how this model performs with an increasing number of GPUs. With advanced 16-core AMD Opteron CPUs and one NVIDIA K20x Kepler GPU in each node, Titan is one of the most powerful supercomputers in the US [29]. In the Titan cluster, each computing node contains one GPU. The data exchange among GPUs is done through copying the data from the GPU to the CPU within the node, communicating using the Message Passing Interface(MPI) among CPUs, then copying it back from the CPU to the GPU. In this scaling test, we used $16\times16\times16$ modes, which is a typical configuration in real simulation. Figure 4 shows the speedup of the symplectic code running on multiple nodes compared with that running on a single node for different problem sizes.

As shown in Fig. 4a, the speedup of space charge kicker increases almost linearly (up to 16 GPUs) with the number of GPUs at the beginning and then gradually reaches a limit around 256 GPUs. The linear increase of speedup at the beginning is due to small amount of data exchange, i.e. communication among different nodes, which is a great advantage of the gridless symplectic



(a) speedup of the space charge kicker.



(b) speedup of the entire code.

Figure 4: Speedup of the symplectic multi-particle tracking code on Titan GPU cluster.

tracking model and depends only on the mode number and is independent of the particle number. On the other hand, the maximum speedup it can achieve is also limited by the particle number, and the linear range will extend with the use of more particles, which means more computational workloads. Taking the example of $160,000$ particles on 64 GPUs (the green dot-marked line in Fig. 4a), it reaches a maximum speedup of about 40. With each GPU containing 2688 cores, we used total $172,032$ cores, which is even more than the particle number. As a result, the speedup becomes saturated after 64 GPUs. However, with the increase of the number of particles, the maximum number of GPUs that can be effectively used also increases. With $640,000$ particles in the simulation, the speedup saturates around 128 on 256 GPUs.

Figure 4b shows the speedup of the entire code, including the transfer map, coordinates conversion, and diagnostic output. Those functions listed above are also parallelized, but it's more difficult to achieve a high parallel efficiency due to the intrinsic small computational workload. The speedup of the entire code decreases slightly compared with that of only the space charge kicker.

The above speedup measurements show that the performance of the GPU implementation varies with the problem size. In the beam dynamics simulation, the choice of the problem size such as the number of modes and the number of particles depends on specific beam physics application. In a typical application, $16 \times 16 \times 16$ modes and $160,000$ particles would be a reasonable choice. In some other applications involving complex phase space distribution, more spectral modes and particles would be needed in order to have sufficient accuracy.

## 5. Application to beam dynamics simulation

Using the GPU symplectic multi-particle tracking code, we carried out beam dynamics simulations through a periodic focusing channel with different currents. In the simulation, we set the phase advance per turn with 0 current to be 2.398. Here, each turn of lattice consists of 10 identical cells. Each cell is 1 meter long with two transversely linear focusing elements, two longitudinally

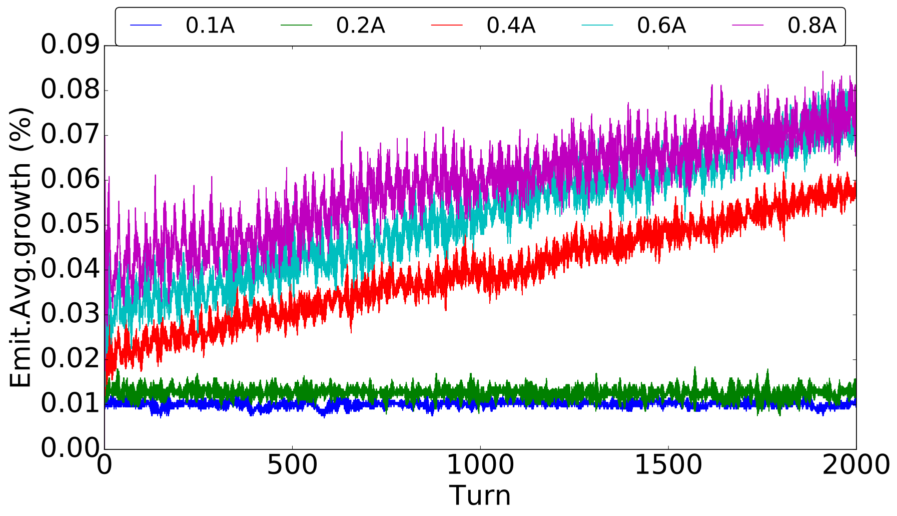


Figure 5: Emittance evolution with different currents.

linear focusing elements, and four drifts. With increasing beam current, the tune will be depressed and moves across the third order resonance line of 2.333 with a 0.6A beam current. There is a thin lens sextupole at the end of each turn to excite the resonance.

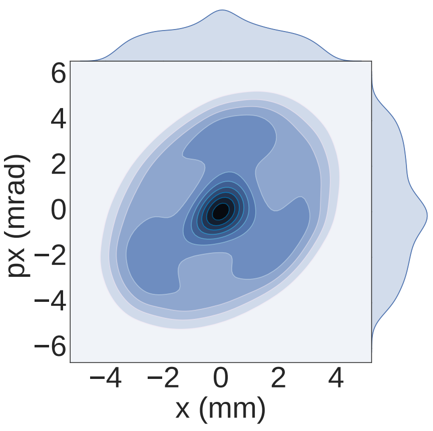
Figure 5 shows the emittance growth with different currents. The simulations were done using $16 \times 16 \times 16$ modes and $160,000$ particles. The emittance (a measure of beam property) stays almost constant at 0.1A and 0.2A beam currents, where the depressed tunes are away from 2.333. However, it keeps growing at 0.4A, 0.6A, and 0.8A currents, where the depressed tunes approach and go below 2.333. This emittance growth is due to the space charge enhanced third-order nonlinear resonance.

The Poincaré maps of phase space coordinates of a few particles near the third-order resonance is plotted in figure 6. In these contour plots, darker color means larger particle density. Different plots denote the particles starting from different initial positions. Caused by the third-order resonance, the Poincaré map is distorted and shaped into a triangle with three islands around it. The particles affected by the resonance would gradually move outward towards large amplitude. Eventually, these particles would become part of beam halo and get lost.

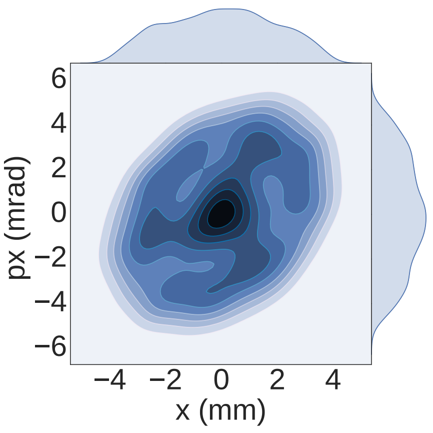
The comparison between the above symplectic gridless particle model (with CPU implementation) and the traditional momentum conserved particle-in-cell tracking model was reported in reference [13]. For the above application, we also ran the simulation using the Intel Xeon Phi CPU node with 32 cores at NERSC. Using the above GPU implementation reduces the total computational time by more than a factor of two.

## 6. Conclusions

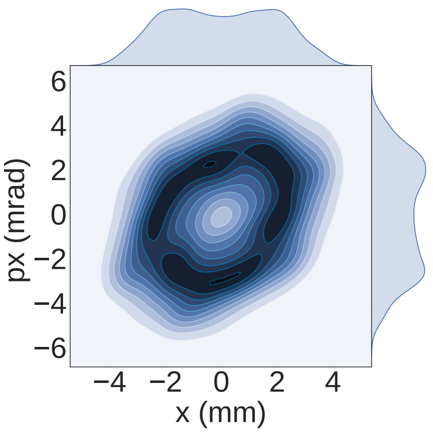
A gridless symplectic multi-particle tracking model was implemented on GPUs using the CUDA language. The gridless spectral tracking algorithm has the advantage to satisfy the symplectic condition and effectively reduce the numerical noise driven emittance growth. On a single GPU processor, using a common home-use GPU, GTX 1060, we achieved a factor of $2-10$ speedup for a range of problem sizes with respect to the computing time from a similar power consumption and semiconductor technology CPU node. This symplectic model also shows good scalability on a multi-GPU cluster. Several beam dynamics simulations were done using the GPU implementation with different currents through a periodic focusing channel. No emittance growth is seen when the depressed tune is far away from the third-order resonance, while it keeps growing when the tune approaches the resonance. Using the GPU implementation discussed in this paper helps save the total computational time by more than a factor of two for this application in comparison to the time using the CPU implementation with similar power consumption. In the future study, we will continue to extend this code and to compare the parallel efficiency of this code on different architectures. We would also like to compare this gridless symplectic model with the PIC model on multiple GPUs.



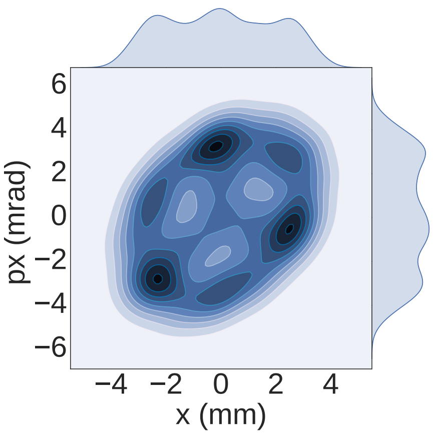
(a)



(b)



(c)



(d)

Figure 6: Poincare maps of the normalized phase space coordinates for four particles starting with different initial conditions.

## 7. Acknowledgments

This work was supported by the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. One of the authors, Zhicong Liu, would also like to extend his thanks for the financial support from the China Scholarship Council (CSC, File No. 201604910876) and the Institute of High Energy Physics, Beijing, China. We have used computing resources at the Oak Ridge Leadership Computing Facility (OLCF) and at the National Energy Research Scientific Computing Center (NERSC).

## References

[1] C. K. Birdsall and A. B. Langdon, Plasma Physics via Computer Simulation, Taylor Francis, New York, 2005.

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

[3] A. Friedman, D. P. Grote and I. Haber, Phys. Fluids **B 4**, 2203 (1992).

[4] J. Qiang, R. D. Ryne, S. Habib, V. Decyk, J. Comput. Phys. **163**, 434 (2000).

[5] J. Qiang, M. A. Furman, R. D. Ryne, Journal of Computational Physics 198 (1) (2004) 278.

[6] J. Qiang, S. Lidia, R. D. Ryne, and C. Limborg-Deprey, Phys. Rev. ST Accel. Beams **9**, 044204, 2006.

[7] J. Amundson, P. Spentzouris, J.Qiang and R. Ryne, J. Comp. Phys. vol. 211, 229 (2006).

[8] D. Uriot, N. Pichoff, Tracewin, CEA Saclay, June.

[9] Y. K. Batygin, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment 539 (3) (2005) p.455.

[10] P. Channell and C. Scovel, Nonlinearity 3 (2) (1990) 231.

[11] E. Forest and R. D. Ruth, Physica D **43**, p. 105, 1990.

[12] H. Yoshida, Physics Letters A 150 (5-7) (1990) 262.

[13] J. Qiang, Phys. Rev. Accel. Beams 20 (2017) 014203.

[14] V.K. Decyk, Description of Spectral Particle-in-Cell Codes from the UPIC Framework, http://picksc.idre.ucla.edu/publications/publications-reportsand- notes/, (accessed: 30.07.15).

[15] G. Vlad, S. Briguglio, G. Fogaccia, B. Di Martino, Comp. Phys. Comm. **134**, 58 (2001).

[16] C.-K. Huang, Y. Zeng, Y. Wang, M.D. Meyers, S. Yi, B.J. Albright, Comp. Phys. Comm. **207**, p. 123 (2016).

[17] http://wccftech.com/nvidia-geforce-gtx-1060-final-specifications/.

[18] https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1060/.

[19] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, J. C. Phillips, in Proceedings of the IEEE 96 (5), (2008) 879.

[20] Nvidia, CUDA C Programming guide (2010).

[21] G. Stantchev, W. Dorland, N. Gumerov, J. Parallel Distrib. Comput. 68, p. 1339, (2008).

[22] H. Burau, R. Widera, W. Hnig, G. Juckeland, A. Debus, T. Kluge, U. Schramm, T. E. Cowan, R. Sauerbrey, and M. Bussmann, IEEE Trans. Plasma Sci. 38, 2831, (2010).

[23] V. K. Decyk, T. V. Singh, Comp. Phys. Comm. **182**, p. 641 (2011).

[24] X. Pang and L. Rybarcyk, Comp. Phys. Comm. **185**, p. 744 (2014).

[25] F. Hariri, T.M. Tran, A. Jocksch, E. Lanti, J. Progsch, P. Messmer, S. Brunner, C. Gheller, L. Villard, Comp. Phys. Comm. **207**, p. 69 (2016).

[26] S.-Y. Lee, Accelerator physics, World Scientific, Publishing Co Inc, 2004.

[27] A. W. Chao, K. H. Mess, M. Tigner, F. Zimmermann, Handbook of accelerator physics and engineering, World scientific, 2013.

[28] http://www.nersc.gov/users/computational-systems/cori/configuration/.

[29] https://www.olcf.ornl.gov/computing-resources/titan-cray-xk7/.