Synergia CUDA: GPU-accelerated accelerator modeling package

Q. Lu, J. Amundson
Scientific Computing Division, Fermi National Accelerator Laboratory
P.O.Box 500, Batavia, Illinois 60510, U.S.
E-mail: qlu@fnal.gov

Abstract. Synergia is a parallel, 3-dimensional space-charge particle-in-cell accelerator modeling code. We present our work porting the purely MPI-based version of the code to a hybrid of CPU and GPU computing kernels. The hybrid code uses the CUDA platform in the same framework as the pure MPI solution. We have implemented a lock-free collaborative charge-deposition algorithm for the GPU, as well as other optimizations, including local communication avoidance for GPUs, a customized FFT, and fine-tuned memory access patterns. On a small GPU cluster (up to 4 Tesla C1070 GPUs), our benchmarks exhibit both superior peak performance and better scaling than a CPU cluster with 16 nodes and 128 cores. We also compare the code performance on different GPU architectures, including C1070 Tesla and K20 Kepler.

1. Introduction

Synergia is an open source framework for beam dynamics simulations [1]. It is capable of simulating both single-particle dynamics and multi-particle dynamics where the particle-particle interactions are important, e.g., space charge and wake fields.

A typical bunched particle accelerator beam contains $O(10^{12})$ particles per bunch. Synergia uses the particle-in-cell (PIC) method [3] to simulate the beam dynamics. In the PIC method, a much smaller number of macroparticles are used to simulate the dynamics of the real bunch. A macroparticle is a computational particle that represents many real particles. It is allowed to rescale the number of particles because the Lorentz force depends only on the charge to mass ratio, so a macroparticle will follow the same trajectory as a real particle would. Even with the aggregation of macroparticles, the brute force approach in the simulation of particle-particle interactions will still be computationally expensive for its $O(N^2)$ complexity and the enormous amount of particles in a beam. The method couples the $N$-body integrator to a grid-based Poisson solver resulting in an $O(N \log N)$ estimated computing complexity. The basic steps of a PIC iteration proceeds as follows,

(i) Compute the charge/mass density field on a finite grid points.
(ii) Solve for Poisson equation on the grid.
(iii) Interpolate the force back to the particle positions and integrate the equation of motion.

Synergia has several different Poisson solvers for various boundary conditions. In this work, we focus on its 3D open boundary condition solver, which uses the open Hockney
method. In the following sections, we describe our approaches in parallelizing Synergia on a hybrid architecture, with high-level MPI for distributed memory parallelization and thread-level parallelized computing kernels. We then focus on the CUDA implementation of the computing kernels for running on a cluster of GPUs. Finally, benchmark results and comparisons with pure CPU implementations are presented.

2. Distributed Parallelization of Synergia

Synergia is capable of very large scale beam dynamics simulations. Such jobs can contain from a hundred million to over ten billion macroparticles, and utilize close collaborations from over 100,000 CPU cores or computing devices. It is not nearly feasible to carry out such simulations on a single computer even with acceleration computing devices such as GPUs or Intel MIC. Therefore the initial problem needs to be decomposed into a set of small problems, each individually can then be solved in parallel on a single CPU or a computing device with similar computing powers. MPI is still the most popular and widely supported standard for high performance parallel computing in distributed memory. The Synergia package itself is based on MPI for distributing particles and work loads across the processors and performing the computation collaboratively.

2.1. Parallel Decomposition

As depicted in the introduction, the PIC simulation works in iterations. Each iteration is dependent on the results from the previous iterations. Hence the iterations cannot be parallelized. What we can parallelize is the computations done in an iteration. The straightforward method in parallelizing the iteration is to distribute particles and mesh grid points over processors or MPI tasks. It should be made such that the overhead involved in distributing the work loads remains at minimum, while keeping the distributed work loads on each processor well-balanced to avoid wasting computing powers at “idling”.

Parallel decomposition schemes for the PIC algorithm can be divided into two broad categories [2]:

(i) Direct Eulerian method, where the particle spatial domain is divided into several non-overlapping regions and each processor is assigned on of these sub-domains. Particles are then assigned to processors according to their positions and may migrate between processors’ sub-domains as the system evolves.

(ii) Direct Lagrangian method, where the particles remain affixed to the processors they were initially assigned to, regardless of the system evolution.

Generally speaking, direct Lagrangian method has more balanced work loads, and less communication overheads for not needing to do the particle migration step. It is, however, not suitable for cases when the data associated with the spatial grids is beyond the memory limitation of a single processor.

Simulations for accelerator modeling usually involves very large number of particles, but on a relatively small regular spatial grids ($\sim (64 - 128)^3$). Hence it is suitable to decompose the particles only and keep them affixed to the processors, meanwhile each processor will have a replication of the entire spatial domain. This way, the simulation will have well balanced computations and a simplified communication pattern.

2.2. PIC Method in MPI Parallelization

2.2.1. Charge Deposition

Charge deposition is done locally on each of the processors for the assigned particles. Then a final global reduction is performed to sum up the local charge densities over each and every processor to obtain the global charge density. It is then necessary to
distribute a copy of the global charge density back to every processor. This can be achieved by a \textit{MPI-Allreduce()} collective method at the end of each distributed deposition operation.

2.2.2. \textbf{Field Solver} We use the FFT-based convolution method \cite{3,4} to solve the Poisson equation from the obtained charge density at the grid points $\rho_{i,j,k}$. The field solver involves first finding the Green function at the grid points. For open boundaries, the Green function is obtained by

$$ G(u, v, w) = \frac{1}{\sqrt{u^2 + v^2 + w^2}} \quad (1) $$

The potential at the point $(i, j, k)$ is then

$$ \phi_{i,j,k} = h_x h_y h_z \sum_{i'=0}^{i_{\text{max}}} \sum_{j'=0}^{j_{\text{max}}} \sum_{k'=0}^{k_{\text{max}}} \rho_{i',j',k'} G_{i-i',j-j',k-k'} \quad (2) $$

The convolution can be computed using FFT by appropriate zero-padding of the sequences. Finally, the electric field is determined by the gradient of the potential,

$$ E_{i,j,k} = -\nabla \phi_{i,j,k} \quad (3) $$

For efficiency, we divide the spatial domain into small regions and assign them to MPI processors. Each processor will then be responsible for calculating the Green function, potentials, and electric field of its own region. Shadow regions and data exchanges will be needed for calculating the field. The FFT is done collaboratively among all MPI processes. Finally, an \textit{MPI-Allgather()} is performed to gather the pieces of electric field from all MPI ranks and redistribute the complete field back to every process.

2.2.3. \textbf{Apply Kicks} Once the electric field on the mesh grids are obtained, the last step is pushing particles by integrating the equations of motion. As the fields are only calculated at the grid points, interpolation is needed to approximate the electric fields at the actual location of the particles. We then use the explicit method to update the positions and momenta of each particle assigned to a processor. Since the field data is replicated and available on each of the processors, no further communications are needed in this stage. The work load in this phase is roughly proportional to the number of particles assigned to each processor, therefore we have close to perfect load distribution in this step.

2.3. \textbf{Communication Optimizations} For doing massive parallel simulations, scaling efficiency is critical for the quality of a simulation software package. Because of the nature of the distributed memory parallelism, optimizations which lower the overall ratio of the communication overhead play a key role in improving the scaling efficiency.

Interprocess communications occur in several places in the parallel PIC algorithm described above: \textit{a}) collective reduction and redistribution of the charge density array following local charge depositions; \textit{b}) Fourier transforms, field calculation and distribution in solving the field; and \textit{c}) bunch statistics such as calculating the means and standard deviations of the particle positions and momenta. Profiling shows the extensive amount of data exchange during the field solve portion of the calculation. As a consequence, the peak performance and scaling efficiency are strongly affected, especially when using multiple nodes so that the communication has to go through the network interface. It is possible to reach the point where negative scaling occurs – increasing the number of computing nodes beyond a certain point actually slows down the overall
performance. Limited concurrency is another defect in the current field solver algorithm. The concurrency of the field solver is determined by the size of the spatial grid, which is relatively small in typical accelerator simulations, where the characteristic grid size is $\sim 64 \times 64 \times 64$.

We altered the simulation framework to address the above issues. The new PIC simulator is able to dispatch multiple field solvers simultaneously, with each one capable of solving the field independently. In this way, the fields are calculated redundantly within each sub-group of the entire $\text{MPI COMM WORLD}$, thus confining the data exchanges within the boundary of the processor group. With the emergence of modern multi-core, multi-socket architectures, a single computer node usually contains tens of CPU cores. Meanwhile data exchanges within the same node are much more efficient than going through the network interface. In modern super computers such as the IBM BlueGene series, the computing nodes are organized in hierarchical structures such as a 5-d torus ring in regards of the communication infrastructure, so the computing nodes at the same level naturally have better throughput and lower latency than nodes on different levels. In almost all cases grouping the nodes according to the hierarchical structure for doing the field solver outperforms the original, single-field-solve method. The optimal size and configurations of the sub-group is, of course, very much platform dependent and subject to the actual simulation problem. Using the redundant field solver also breaks the ceiling of the concurrency limitation which the previous algorithm has.

3. Shared Memory Parallelized Computing Kernels

In section 2 we described the high-level parallelization for the PIC method using distributed memory parallelism. The simulation framework based on MPI implementation allows us to utilize an arbitrary number of computing devices with distributed memory to solve the problem collaboratively. Here, by “computing devices” we mean the facility that actually does the computations. Traditionally, a CPU is the only device that is capable of doing general purpose computations. The concept has been expanded since it was first proposed that a graphics processing unit (GPU) can also be used in some forms of general purpose computing, and the first acceleration card along with a toolchain was released by NVIDIA in 2006. Intel has also released products featuring Many Integrated Core (MIC) architecture in the form of an acceleration card. Moreover, the recent multi-core multi-socket CPU architecture has also blurred the boundary of the traditional processors. A set of CPU cores are closely coupled together through a shared memory controller or fast interconnections. One common feature of these computing devices is that they all have multiple computing cores with shared memory access. In this section, we describe how the actual computing kernels are implemented on these devices, specifically, on GPUs with the CUDA programming language.

3.1. Hybrid Parallelization and Heterogeneous Computing

Parallelization is needed in order to take advantages of the multi-core facility presented on most of the common computing devices. As opposed to the high-level MPI parallelization, computing cores within a processor share a global memory space. Thus it is reasonable to treat the computing cores from a sole device together as an entity. Then the entity as a whole will request data from other entities through MPI message passing if the data is not available locally.

This hierarchical or hybrid parallelization setup hides the low-level parallel implementation details from the high-level MPI parallelization. We are allowed to implement the computing kernels using different parallelization libraries or paradigms, such as OpenMP, OpenACC, CUDA, OpenCL, etc., for specific hardware devices. It is also possible to have a mixture of computing kernels, each targeting one specific hardware architecture, to do the heterogeneous computing under the simulation framework.
3.2. Thread Parallelization Using CUDA

In this section we describe our work on implementing the computing kernels of the PIC algorithm for the GPU architecture using CUDA.

3.2.1. Data Structures

Macroparticles in the simulation are represented by their locations and momenta in 3D space. In the simulation, each MPI process is assigned a proportion of particles which will then be transferred to the GPU device memory and will then stay there for the course of the simulation to minimize the host-device data movements. Data associated to the spatial grids such as the charge density array and electric fields at the grid points are stored entirely in the GPU device memory.

3.2.2. Bunch Statistics and Diagnostics

The simulation framework checks the particle bunch statistics regularly to keep track of the macro measurements of all the particles in a bunch, and to make sure that the simulation is in good shape. Typical statistics operations include calculating the average and standard deviations for all particles in a bunch. They are implemented as two step operations: first perform the in-GPU parallel reduction for all particles assigned to a GPU, then perform an MPI reduction to collect the final results from all MPI ranks. Fast GPU reduction algorithms [5] are used to obtain high throughput on the GPU.

3.2.3. Collision-free Shared Memory Charge Deposition

The macroparticles in the PIC simulation are accumulations of individual particles with macro-quantities (number density, charge density, etc.). Particles can be situated anywhere on the continuous domain, but macro-quantities are calculated only on the field mesh points. Using a first order (linear) weighting scheme, a single macroparticle can be found at 8 nearest grid points with different weights. Calculating the accumulated charge density on the field mesh in the shared memory is therefore a collaborative updating scheme that requires proper synchronization and critical region protection.

The traditional method of dealing with updating conflicts uses mutually exclusive locks to make sure that no more than one worker is updating a common memory block at the same time. However, lock operations are extremely expensive. Also, due to the exclusive execution of the protected code, along with the excessive number of particles, the performance of such algorithms can easily degrade to be even worse than plain serial executions.

Given that one macroparticle contributes to 8 nearest mesh points in charge deposition, to avoid the use of any locks we implement the following collision-free algorithm which requires a local synchronization at a thread block level. The algorithm requires a sort of all particles
into their belonging grid cells at the beginning of each iteration. It then renders the grid cells into colors according to their coordinate indexing. We use $i_x$, $i_y$, and $i_z$ to denote the cell indexes on $x$, $y$, and $z$ axis. Cells with odd $i_x$, odd $i_y$, and any $i_z$ are assigned one color, cells with odd $i_x$, even $i_y$, and any $i_z$ a second color, cells with even $i_x$, odd $i_y$, and any $i_z$ a third color, and, finally, cells with even $i_x$, even $i_y$, and any $i_z$ are assigned the fourth color. All the particles belonging to cells with the same color can do a strip-based (cells with the same $i_x$ and $i_y$) deposition without interfering with each other, i.e., can be carried out in parallel without using locks. Depositions in a strip can also be parallelized using a local thread block barrier, as shown in Fig. 2. We use the same number of threads as the number of $z$-axis cells in one thread block. This way each thread is in charge of doing the deposition for one grid cell of particles. In the first step, depositions will be done for four grid points at $i_z$ layer; followed by a thread block barrier then the second depositions for the remaining four grid points at $i_z + 1$ layer.

The sorting step of the collision-free algorithm incurs non-trivial overheads at each iteration. However, since the sorting process only requires to sort particles to their belonging grid cells, it can be parallelized using parallel bucket sort. The concurrency of the bucket sort algorithm is roughly the number of grid cells in the field mesh, which is on the magnitude of millions in typical simulations. It is hence suitable for GPU implementations and yields good scalability at high thread counts.

Moving the raw particle data in the sorting process is another concern in the sorting process. Due to the large set of particle data (~a few gigabytes) assigned to each GPUs, moving the raw data frequently would likely congest the memory bandwidth and impact the overall performance. Using linked lists for holding particles in a grid cell, on the other hand, increases the memory footprint and adds another layer of indirection when accessing particle data. Our algorithm uses a so-called indexed list, which is a mixture of plain array and linked list, for holding particle indexes grouped by grid cells. The data structure requires an indexing array with the size equal to the number of local particles, and an auxiliary array with the size of the total grid cell counts storing the offsets of each cell in the indexing array. The sorting algorithm is then the process of building these two arrays.

3.2.4. Field Solver Computing the Green function is totally independent and takes almost no input data. Therefore it can be parallelized easily using CUDA. The number of points to be calculated is the same as the number of grid points, so the degree of thread concurrency is not an issue here. Calculating the field involves the shadow region that requires data exchanges with the neighboring process. The asynchronous data copy engine is used to maximize throughput by enabling computations while waiting for data. The FFT part uses the CUFFT library from NVIDIA.

3.2.5. Pushing Particles Each GPU now has a copy of the field in its local memory, along with a set of particles that need to be pushed. The operation is again independent with regards to the particles, so each thread will be assigned a small number of particles. These threads are then responsible for updating the particle positions in the phase space accordingly. Having particles sorted by grid cells at the previous charge deposition step provides additional benefits here. As particles located at the same cell can be accessed as a group, access to the global electric field $E_x(i, j, k)$, $E_y(i, j, k)$, and $E_z(i, j, k)$ is reduced by an order of magnitude. It also provides better data locality and cache efficiency when accessing the particle data.

4. Performance Results And Conclusions

The benchmark problem contains close to 21 million macroparticles (20,971,520), and uses a spatial grid of $128 \times 128 \times 128$ for solving the field. We have the benchmark simulation run on different platforms including: a) a single Intel Xeon X5650 CPU; b) an InfiniBand-coupled cluster
with dual Xeon X5650 CPU per node; c) a small Tesla GPU cluster with 1-4 C1060 Tesla cards; and d) a small Kepler GPU cluster with 1-6 K20 Kepler cards. All simulations are done using double precision floating-point numbers.

For the pure CPU version of Synergia, it takes close to 600 seconds for each turn of simulation on a single CPU. The strong scaling on the cluster is not optimal because of the field solver. The best performance we have achieved is about 45 seconds running on 16 nodes and 128 cores. This peak result is outperformed by a single node with four Tesla C1060 GPUs which finishes the turn in only 29 seconds. The code package also exhibits reasonable multi-GPU scaling using a small number of Tesla C1060 GPUs, as shown in Fig. 4 showing the efficiency of the hybrid parallelization of the simulation package. We choose to run the duplicated field solver on each of the GPUs. Therefore, the time for the field solve does not scale with the number of GPUs on the plot. On a single GPU, the time spent on field solver is about an order of magnitude lower than other components. However, when the number of GPU increases the field solver gradually dominates the overall simulation time. Our plan is to improve the field solver with better algorithms so it can be further scaled.

Porting the code to newer generations of GPUs is straightforward. The performance improvements, on the other hand, are significant. A single Kepler K20 GPU is able to outrun a small CPU cluster with 16 nodes and 128 cores by about 40% (45s vs. 26.4s). Using 4× K20 GPUs further brings the simulation time down to about 12 seconds, or about a 50× speedup compared to a single CPU.

**References**

[1] Amundson J, Spentzouris P, Qiang J, and Ryne R 2006 *J. Comp. Phys.* **211** 229-48, [https://cdcvs.fnal.gov/redmine/projects/synergia2](https://cdcvs.fnal.gov/redmine/projects/synergia2).

[2] Joseph R G, Ravunnikutty G, Ranka G, D’Azevedo E, and Klasky S 2011 *Proc. Par. & Dist. Proc. Sym.* p 395-406.

[3] Hockney R W and Eastwood J W 1988 *Computer Simulation using Particles* (Taylor & Francis Group).

[4] Ryne R D 2012 *Proc. Intl. Part. Acutr. Conf.*

[5] Harris M 2007 *NVIDIA Developer Tech.* **2**.