Parallelization of the Particle-in-cell-Code PATRIC with GPU-Programming

Jutta Fitzek

GSI Helmholtzzentrum für Schwerionenforschung
Planckstraße 1
64291 Darmstadt
j.fitzek@gsi.de

Abstract: The Particle-in-cell (PIC) code PATRIC (Particle Tracking Code) is used at the GSI Helmholtz Center for Heavy Ion Reasearch to simulate particles in circular particle accelerators. Parallelization of PIC codes is an open research field and solutions depend very much on the specific problem. The possibilities and limits of GPU integration are being evaluated. General GPU aspects and problems arising from collective particle effects are put into focus with an emphasis on code maintainability and reuse of existing modules. The studies have been performed using NVIDIA®'s Tesla C2075 GPU. This contribution summarizes the findings.

1 Introduction

Computer simulations play an important role in physics research to complement or replace experiments. This contribution focuses on simulations for circular particle accelerators at the GSI Helmholtz Center for Heavy Ion Reasearch, Darmstadt, Germany. Particle accelerators are used in physics to investigate the structure of matter. In this particular application, simulations are used to study the impact of parameter variations on the particle motion that are not easily measurable. The particle motion is defined by the accelerator layout as well as interactions between the particles, and is evaluated over space and time. The simulations are very computationally intensive. Message Passing Interface (MPI) has been successfully used in parts since 2001, but still long running simulations take hours.

Graphics processing units (GPUs) allow for mass-execution of algorithms on large amounts of data and are therefore more frequently used for parallelization. The beam physics department at GSI decided thus to evaluate the use of GPUs in their existing simulations. The test system contains a 2.67 GHz Intel Xeon X5650 processor and NVIDIA's Tesla C2075 GPU that is programmed using CUDA C. The preference for NVIDIA is mainly motivated by freely available libraries such as cuFFT and cuBLAS. In the studies, the possibilities and limits of the GPU integration are investigated. Several modifications to the present algorithms are discussed and evaluated. The result are maintainable parallelized algorithms that allow for up to six-times faster simulations and will be the basis for future developments.

The remainder of this article is structured as follows: Section 2 introduces the general parallelization aspects of GPUs, Section 3 describes the simulation methods, Section 4 and 5 present the realized modifications and discuss the findings, Section 6 concludes.

2 Parallelization with GPUs

GPUs are massively parallel accelerators originally introduced for graphic processing, but nowadays also used for general purpose computing. In contrast to CPUs, that execute different programs sequentially, GPUs execute one program with hundreds of parallel execution units. To the developer, the GPU is represented through a logical abstraction layer, NVIDIA's Compute Unified Device Architecture (CUDA). The PC is referred to as *host*, the graphics card as *device*. A *kernel* describes the procedure for a single execution unit on the GPU and is executed in many *threads* that each have their own index to access the data. Threads grouped in *blocks* are executed together and have a shared memory. Blocks are structured in *grids*.

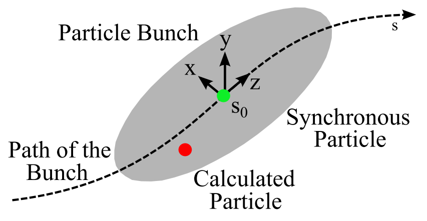
The Tesla C2075 GPU consists of 14 multiprocessors with 32 cores each [NVI09, p. 7]. At runtime, blocks get assigned to multiprocessors according to their resource usage [KH10, p. 84]. Blocks are independent to ensure scalability [Far11, p. 86]. Out of one block, warps of 32 threads get executed in parallel. Several warps are active in a time-sliced way [KH10, p. 88]. The GPU has different memory types: slow global memory for data exchange with the host system, fast shared memory for threads within a block, and very fast registers per thread. Optimal memory usage is essential for improving the speedup.

GPU algorithms can be analyzed treating each thread as a logical processor in the shared memory model [SK10, p. 66]. Threads in a warp act like a vector computer of the 1970s and fit in the SIMD category [Far11, p. 88] in the classification of Flynn [Fly72]. For the whole GPU, the author follows the view of [KH10] who suggests SPMD (single program, multiple data) known from MPI, where autonomous processors execute the same program on different data, which fits to independent blocks on the GPU. Since the exact block execution is not known here, only single warps are analyzed using the Parallel Random Access Machine (PRAM) model [JáJ92, p. 9 ff.] which is based on the well-known RAM model [AHU74, p. 5 ff.]. Further statements regarding the whole GPU can be derived, but must not resemble real measurements.

3 Simulation of the Particle Motion

The simulation model used is shortly described as basis. In accelerators, charged particles are guided, accelerated and interact with each other through electromagnetic forces. This Lorentz force can be written as $\vec{F}=q\cdot(\vec{v}\times\vec{B}+\vec{E})$, with q being the charge and $\vec{v}$ the velocity of the particle, and $\vec{B}$ and $\vec{E}$ the magnetic and electric fields surrounding the particle. The magnetic field bends and focuses the particles transversally. The electric field accelerates the particles longitudinally and bundles the beam into bunches [Wil05, pp. 3-4].

In simulation, particles or larger macro particles are described relative to the synchronous (ideal) particle s_0 (see Fig. 1) using a vector with x, x' as horizontal, y, y' as vertical positional and directional deviation, z as longitudinal positional deviation and v als momentum deviation [Wil05, p. 76 ff.]. x, y, v are measured in mm, x', y' in mrad, v in per-mil. While particles move through the accelerator, elements like magnets act on them. To track the particles in the simulation, their movement is realized as matrix-vector multiplication with magnets being represented as transport matrices.



$$\bar{p}' = M \cdot \bar{p} = \begin{pmatrix} M_{11} & M_{12} & 0 & 0 & 0 & M_{16} \\ M_{21} & M_{22} & 0 & 0 & 0 & M_{26} \\ 0 & 0 & M_{33} & M_{34} & 0 & 0 \\ 0 & 0 & M_{43} & M_{44} & 0 & 0 \\ M_{51} & M_{52} & 0 & 0 & 1 & M_{56} \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ x' \\ y \\ y' \\ z \\ v \end{pmatrix}$$

Figure 1: Simulation of the particle motion

To simulate the particle interaction, the forces between the particles are discretized on a grid and then the grid acts back on the particles. Doing so drastically reduces the complexity compared to a full n-body simulation with $\mathcal{O}(n^2)$. This technique is called Particle-in-cell (PIC) method and is used since the 1950s for plasma simulations, see [BL05, p. 3]. The calculation cycle for one discrete time step is shown in Fig. 2. It starts on the right side with a given particle distribution. Based on the particle density, in the first step the charge (ρ) and current distribution (J) are interpolated on the grid. In the second step (field solver) the electric and magnetic space charge field on the grid is calculated, which can be done e. g. with a forward and backward FFT. The electromagnetic field gets integrated and results in an electrostatic potential [Rei08, p. 173]. In the third step, forces are derived from the potential, that diffract or accelerate the particles [Rei08, p. 164]. In the last step (particle pusher) new particle coordinates are calculated based on these forces.

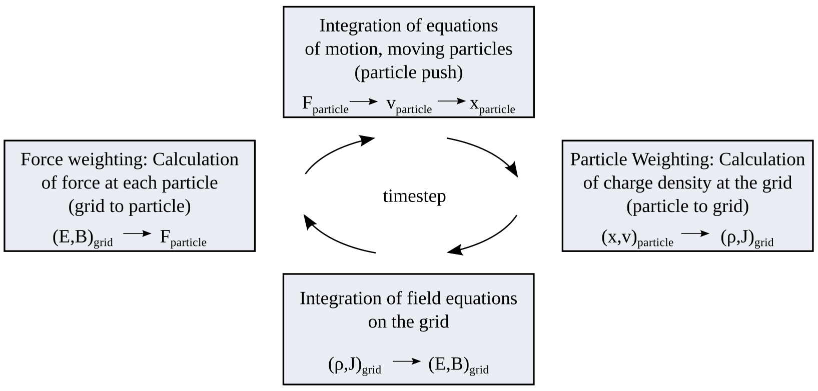

Integration of equations of motion, moving particles (particle push)
$F_{particle} \rightarrow v_{particle} \rightarrow x_{particle}$
Particle Weighting: Calculation of charge density at the grid (particle to grid)
$(x,v)_{particle} \rightarrow (\rho,J)_{grid}$
Integration of field equations on the grid
$(\rho,J)_{grid} \rightarrow (E,B)_{grid}$
Force weighting: Calculation of force at each particle (grid to particle)
$(E,B)_{grid} \rightarrow F_{particle}$
timestep

Figure 2: PIC calculation cycle (see [BL05])

4 Particle Tracking and General GPU Aspects

The existing simulation code PATRIC (Particle TrackIng Code) [BFK06] is a C++ program developed by the beam physics department and is specific to the research emphasis at GSI. Here, a simplified version was used that focuses on particle tracking. Since the particle transport steps in PATRIC (matrix-vector multiplications) take 64% of the time, this step was ported to the GPU. Particle tracking is memory bound: the compute to global memory access (CGMA) ratio is only 3 instead of 30 for fully exploiting the GPU [KH10, p. 97]. Therefore memory usage was analyzed. For the up to a few hundred transport matrices, the faster constant memory is too small, so global memory is used with the keyword `const` to benefit from caching. In measurements, the difference to constant memory was negligible. Particle data is also kept in global memory due to its size. To have neighboring threads access neighboring data, the array of structures (AoS) was converted to a structure of arrays (SoA) [Far11, p. 6], with e. g. all 'x' coordinates in one array.

The hypothesis, whether it is beneficial to calculate a single coordinate or a whole particle per thread, was tested through measurements using a simplified linear optics with 16 transport matrices and 128 turns with $100,000$ particles. One particle per thread with its more favorable CGMA ratio was faster, resulting in a speedup of 1.18 compared to the CPU version, see Tab. 1. The versions in brackets highlight noteworthy aspects: host-device synchronization has hardly any impact, most of the GPU time is spent on calculations as expected, but – most importantly – data copy takes about 27% of the execution time.

For the number of threads per block NVIDIAs optimal block size calculator [NVI13b] suggests several possibilities. In measurements, 64 up to 256 threads showed marginal differences below 1.5%, above 256 threads the execution time slightly went up. So the number of 256 threads per block was chosen, following also a recommendation by [Far11].

The next goal was keeping particles between transport steps on the GPU to avoid data transfer, as suggested by [NVI13a]. Methods were added for data copy and for dealing with lost particles (additional `boolean` array). To investigate the behavior with varying problem size, typical number of particles up to $1,000,000$ were simulated. For the CPU version, linear scaling is expected, since the central loops over all particles are of complexity $\mathcal{O}(n)$. This is reflected well by the measurements, see Fig. 3. The complexity of the

Table 1: Single transport step on the GPU: speedup of 1.18 with one particle per thread

Version	CPU time	GPU time	Sum
CPU: Original version	19.04 s	—	19.04 s
GPU: Transport step, Thread: Coordinate	15.21 s	2.74 s	17.95 s
(Version w/o synchronization)	15.26 s	2.73 s	17.99 s
(Version w/o calculation)	15.16 s	0.43 s	15.59 s
GPU: Transport step, Thread: Particle	15.30 s	0.82 s	16.12 s
(Version w/o data copy)	10.83 s	0.99 s	11.82 s

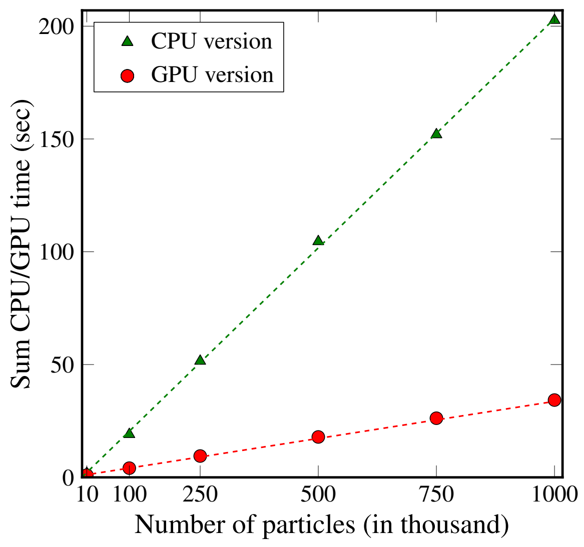


Figure 3: Particle tracking on the GPU, speedup of 6

GPU version is reduced to $\mathcal{O}(\frac{n}{p})$, but data copy imposes additional costs. A positive linear scaling can be also observed for the GPU version, which shows a six-times speedup and is faster above $8,150$ particles. Concluding, using the GPU for particle tracking leads to a good speedup, if the particles are kept on the GPU. However, a comparable measurement with a GPU with 2 multiprocessors instead of 14 was even slower than the CPU version, demonstrating that the number of parallel execution units is of course the main factor.

Several questions were addressed using the parallelized version. First, the impact of particle loss was analyzed. With a maximum thread divergence at 50% particle loss, the GPU time only increased by 1.5%, because each second thread indeed does nothing instead of calculations. Thus particle loss can be neglected and no re-sorting of particles is necessary.

Second, floating point arithmetic was examined. Double precision performance has been a weak point of GPUs in the past. Single precision proved less than 5% faster. Thus double precision can be kept, as it is necessary for long running simulations.

Lastly, the focus was put on output of intermediate results. While output every 5 turns (90 transport steps) results in $10.6s$ for the CPU version and $2.6s$ for the GPU version, output after each turn already takes $19s$ and $4s$ whereas output after each transport step leads to $445s$ and $92s$. Thus output should be limited. To reduce the time for data copy, it was tested to overlap the transport step on the GPU with data copy to the host using streams. Particle arrays were duplicated and pointer switching was used. Measurements showed only 5% performance gain. Due to the dominating memory access, not much can be overlapped and streams do not pay off here. Intermediate results contain calculated beam parameters, that are used to observe the beam quality and intensity. Instead of transferring the full particle data back to the host, those calculations can be done di-

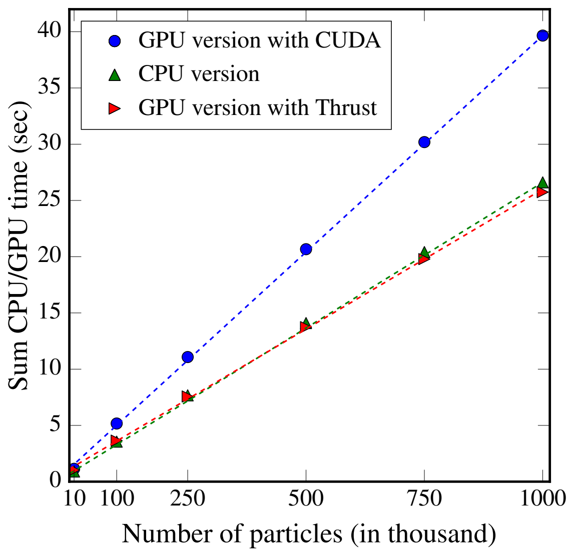


Figure 4: Beam emittance: CUDA vs. Thrust vs. CPU, Thrust slightly faster

rectly on the GPU. As example, the beam emittance was chosen. The rms-emittance (root mean square emittance, 1σ-divergence) as a measure for beam quality describes the (preferably small) geometric bundling of the beam around the optimal orbit, i. e. beam width multiplied by divergence, in mm $\times$ mrad. For the horizontal plane it is defined as: $\varepsilon_x = \sqrt{\langle x^2\rangle\langle x'^2\rangle - \langle xx'\rangle^2}$ [Rei08, p. 321]. The CPU version was compared to two GPU versions using CUDA and Thrust. For the CUDA version, data was summed block-wise using reverse binary reduction and sequential addressing with an atomic update at the end. Although the complexity per block is reduced from $\mathcal{O}(n)$ to $\mathcal{O}(\log n)$, it is slower, see Fig. 4. The version using the Thrust library [NVI13c] with `thrust:reduce` is significantly faster and comparable to the CPU version, since Thrust is highly optimzed. It allows for quick GPU integration, but as the API is limited, the author would not recommend it for realizing complex algorithms. A noteworthy result is that beam parameter calculations are possible on the GPU and in the discussed case Thrust is preferable. But since reduction operations are very expensive on the GPU with hardly any speedup, in general these calculations should be kept on the CPU despite the extra copy steps. This is also favorable with respect to many CPU methods already being in existence that do not need to be ported.

5 Collective Effects

The existing simulation code LOBO (Longitudinal Beam Dynamics Simulations Code) [BFH00] is also developed by the beam physics department and focuses on longitudinal effects which are realized as one dimensional simulation. The represented elements are the radio frequency cavities with their longitudinal forces. The program e. g. simulates the particle capture into particle bunches. The simulation of collective effects between the particles allows to study if the beam becomes unstable. More about the simulations in LOBO can be found in [ABF12]. Here, collective effects are put into focus for parallelization.

LOBO realizes the PIC algorithm described in Section 3. The particles are represented as vector with z als longitudinal positional deviation and dp als momentum deviation, the grids are one dimensional. Every step of the PIC cycle is implemented as separate method. In the original program, 85% of the time is spent on interpolating the particles on the grids, 4% on moving the particles and only 1% on calculating the space charge fields. Since for the latter efficient FFT algorithms exist both for the CPU and GPU, the focus was put on the interpolation steps.

For parallelizing the PIC algorithm using a GPU, two main approaches exist (e. g. [A$^+$12]). In the first approach, the grid information is updated from each particle, thus as many memory accesses are necessary as number of particles. Those memory accesses have to be synchronized, which is the main effort of this approach. In the second approach, the grid information is calculated per grid point using sorted particles. Here, only as many memory accesses are necessary as number of grid points. In this approach the main effort lies in sorting the particles. Both approaches were implemented and evaluated.

For the first approach, as many threads as particles were used. It is necessary to reset the grid information and perform a global synchronization before updating the grid, which was realized with a separate kernel. The concurrent memory accesses were done with atomic updates. For the CPU version, again a linear behavior is expected. For the GPU version the atomic updates of the grid data are expected to be resource-intense. As shown in Fig. 5 the

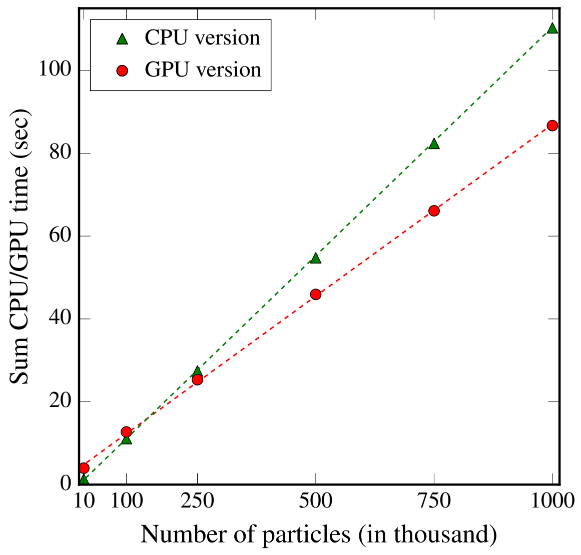


Figure 5: Collective effects: Interpolation step with atomic updates, speedup of 1.19

GPU version has a small speedup of 1.19 and is faster above $170,000$ particles. A more detailed look at the runtime performed with $250,000$ particles shows, that nearly 70% of the GPU time is spent for interpolating the particles on the grids. Compared to 12% for the particle push it is obvious, that the concurrent write access is indeed the bottleneck.

To identify, how the grid size effects the runtime, different measurements with typical grid sizes were performed. For the CPU version a linear correlation between grid sizes and runtime can be observed. The same exists for larger grid sizes on the GPU. That the runtime does not decrease any further below 256 grid points can be attributed to the relatively high number of concurrent write accesses for smaller grids.

Table 2: Collective effects: Comparison of grid sizes

Version / Grid Size	**256**	**512**	**1024**
CPU version	$13.49\ s$	$27.44\ s$	$56.74\ s$
GPU version	$22.38\ s$	$24.81\ s$	$53.16\ s$

The implementation with atomic updates offers only a small benefit compared to the CPU version and depends strongly on the particle numbers and grid sizes. The GPU version can only be recommended beyond $170,000$ particles and 512 grid points.

The second version, where particles are sorted after each particle push according to their mapping to grid points allows to treat the grid points separately. An additional array keeps the necessary mapping information for each particle. As many threads as grid points are needed. Each thread updates the left and right grid point and works with all particles in between. Additional arrays keep the information per thread about the start index, stop index and number of particles it has to interpolate. Update of grid points is done with $2 \cdot k$ atomic updates, where k is the number of grid points. Much less update operations are needed, since the number of grid points is typically 1-2 orders of magnitude smaller than the number of particles. Sorting of the particles is the major effort here and was done using the Thrust functions `thrust::sort_by_key` and `thrust::lower_bound`.

Unfortunately this implementation proved to be eight-times slower than the CPU version. This can be explained with several aspects: the number of total threads is quite low, and also no measures were taken for load-balancing between the threads. This prevented an optimal utilization of the GPU. It is no surprise that sorting is indeed very resource-intense. Much more effort would be needed to optimize the sorting before further following this approach. Additionally, measures should be taken to subdivide the particles and to use more threads for a better GPU saturation, but this implies more synchronization points.

The implemented version for the GPU with pre-sorting was too slow and would need much more adaptation to become more comparable with the CPU-based version. The version with atomic updates already showed a slight benefit compared to the CPU version. Also atomic updates are better supported with each graphics card generation, and it is expected, that they will become faster in the future. Thus it is recommended to use this version as basis for future developments.

6 Summary and Outlook

The possibilities for parallelizing the existing particle simulations PATRIC and LOBO with GPU programming have been studied. Based on a simplified version of PATRIC, a six-times speedup could be achieved for the particle tracking with one particle per thread and while keeping the particles as long as possible on the GPU. This good acceleration is possible, since each particle can be treated independently.

The effect of outputting intermediate results was measured and it was found that it should be limited to a minimum. Overlapping with streams showed no performance gain, since the ratio of calculations to memory accesses is very small. Beam parameters were calculated on the GPU for which the highly optimized Thrust library should be preferred over self-written code. But since no performance gain was observed, the beam parameter calculations can be kept on the CPU to benefit from the reuse of existing routines.

Thread divergence due to particle loss and single vs. double precision floating point performance were evaluated and resulted in only 1.5% and 5% performance loss. Therefore, no further measures have to be taken. Especially, the double precision code can be kept.

Based on a version of LOBO, possibilities for parallelizing collective effects have been investigated. The step of the PIC calculation cycle that interpolates particles on the grid is hard to parallelize with the competing write accesses on the grid as bottleneck. Two approaches followed in other research have been implemented and compared. The version with atomic updates showed a slight benefit compared to the CPU version and will be followed further. The version with pre-sorting of particles was too slow, since the possibilities of the GPU could not be exploited well.

Concluding, the usage of GPUs for particle simulations can be recommended. For well parallelizable problems, a good performance gain can be expected. Tracking was realized with a six-times speedup. For calculations over all particles or the synchronization of many threads, the use of the GPU is limited. But with even a small speedup of 1.19 for collective effects, simulations that combine both aspects can also profit from GPU integration.

While this study concentrated on the GPU, hybrid environments become popular also for PIC algorithms [Dec15]. With the findings it can be suggested for mixed MPI/GPU code that each node should at least calculate $10,000$ particles for tracking and $200,000$ for collective effects. Since the typical number of particles here is up to 1 million, the effort of developing and maintaining hybrid code has to be carefully weighted against any speedups.

In general, GPUs offer good opportunities for parallelization without the need for special parallel computers. Since programming APIs are not yet unified, adaptation effort might arise in the future. For smaller projects, the author suggests to develop not too specialized but instead more algorithm-focused, maintainable code. The tendency of bigger projects to develop specific libraries for GPU integration is no option here. However, libraries could be used, if they were publicly available. Future generic APIs that hide even hybrid parallel architectures and allow for less hardware-oriented programming would be most preferable, as suggested in [B$^+$12]. For the GPU hardware, the trend towards further integration with the CPU will remove the need for data copying. With data copy being a major bottleneck, particle simulations can benefit from that in the future.

References

[A$^+$12] K. Amyx et al., Accelerating Particle-Tracking Based Beam Dynamics Simulations with GPUs, In *Proc. of the GPU Technology Conference (GTC) 2012 - Poster*, San Jose, CA, USA, 2012. Tech-X Corporation, Argonne National Laboratory.

[ABF12] S. Appel and O. Boine-Frankenheim, Microbunch dynamics and multistream instability in a heavy-ion synchrotron, *Phys. Rev. ST Accel. Beams*, 15:054201, May 2012.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, Reading, MA, USA, 1974.

[B$^+$12] R. Buchty et al., A Survey on Hardware-aware and Heterogeneous Computing on Multicore Processors and Accelerators, *Concurr. Comput. : Pract. Exper.*, 24(7):663–675, May 2012.

[BFH00] O. Boine-Frankenheim and I. Hofmann, Vlasov simulation of the microwave instability in space charge dominated coasting ion beams, *Physical Review Special Topics - Accelerators and Beams*, 3(10):104202, October 2000.

[BFK06] O. Boine-Frankenheim and V. Kornilov, Implementation and Validation of Space Charge and Impedance Kicks in the Code Patric for Studies of Transverse Coherent Instabilities in the Fair Rings, In *Proc. of the International Computational Accelerator Physics Conference (ICAP) 2006*, pages 267–270, Chamonix, France, 2006.

[BL05] C. K. Birdsall and A. B. Langdon, *Plasma Physics via Computer Simulation*, Taylor and Francis Group, New York, NY, USA, 2005.

[Dec15] V. K. Decyk, Skeleton Particle-in-Cell Codes on Emerging Computer Architectures, *Computing in Science Engineering*, 17(2):47–52, Mar 2015.

[Far11] R. Farber, *CUDA Application Design and Development*, Morgan Kaufmann, Amsterdam, Netherlands, 2011.

[Fly72] M. Flynn, Some Computer Organizations and Their Effectiveness, *Computers, IEEE Transactions on*, C-21(9):948–960, Sept 1972.

[JáJ92] J. JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley, Reading, MA, USA, 1992.

[KH10] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach (Applications of GPU Computing Series)*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010.

[NVI09] NVIDIA Corp., Fermi Compute Architecture Whitepaper, 2009.

[NVI13a] NVIDIA Corp., CUDA C Programming Guide, Version 5.0, 2013.

[NVI13b] NVIDIA Corp., CUDA Occupancy Calculator, 2013.

[NVI13c] NVIDIA Corp., CUDA Toolkit Documentation: Thrust, 2013.

[Rei08] M. Reiser, *Theory and design of charged particle beams*, Wiley-VCH, Weinheim, Germany, 2. edition, 2008.

[SK10] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Addison Wesley, Upper Saddle River, NJ, USA, 2010.

[Wil05] K. Wille, *The Physics of Particle Accelerators*, Oxford University Press, New York, NY, USA, 2005, Reprint of the 2000 edition.