An efficient mixed-precision, hybrid CPU-GPU implementation of a fully implicit particle-in-cell algorithm

G. Chen, L. Chacón,

Oak Ridge National Laboratory, Oak Ridge,TN 37831, USA

D. C. Barnes

Coronado Consulting, Lamy, NM 87540, USA

Abstract

Recently, a fully implicit, energy- and charge-conserving particle-in-cell method has been proposed for multi-scale, full-f kinetic simulations [G. Chen, *et al.*, *J. Comput. Phys.* **230**, 18 (2011)]. The method employs a Jacobian-free Newton-Krylov (JFNK) solver, capable of using very large timesteps without loss of numerical stability or accuracy. A fundamental feature of the method is the segregation of particle-orbit computations from the field solver, while remaining fully self-consistent. This paper describes a very efficient, mixed-precision hybrid CPU-GPU implementation of the implicit PIC algorithm exploiting this feature. The JFNK solver is kept on the CPU in double precision (DP), while the implicit, charge-conserving, and adaptive particle mover is implemented on a GPU (graphics processing unit) using CUDA in single-precision (SP). Performance-oriented optimizations are introduced with the aid of the roofline model. The implicit particle mover algorithm is shown to achieve up to 400 GOp/s on a Nvidia GeForce GTX580. This corresponds to 25% absolute GPU efficiency against the peak theoretical performance, and is about 300 times faster than an equivalent serial CPU (Intel Xeon X5460) execution. For the test case chosen, the mixed-precision hybrid CPU-GPU solver is shown to over-perform the DP CPU-only serial version by a factor of ~ 100, without apparent loss of robustness or accuracy in a challenging long-timescale ion acoustic wave simulation.

1. Introduction

Particle-in-cell (PIC) methods were developed in the 1960's for the simulation of plasma systems with many particles interacting via electromagnetic fields. The technique employs the method of characteristics to follow discrete volumes (finite-size particles) in phase space, with fields defined on a discrete mesh in physical space. In a typical PIC timestep, particle orbits are integrated to find new positions and velocities for given fields. New fields are found by solving Maxwell's equations (or a subset thereof) using new moments (charge density and/or current) computed from particles. Interpolation operations are defined to exchange information between particles and mesh quantities. The method has been very successful in its application to many areas in plasma physics [1] and beyond.

Due to the intrinsic data-parallel nature of the orbit integrals, PIC methods have been particularly successful in exploiting current (petascale) supercomputers [2, 3]. However,

with the current trend toward million-way parallelism, achieving high performance and high efficiency for PIC simulations on future supercomputers will be non-trivial. Future parallel computing systems will place strong constraints on algorithms both in the amount of memory available and in the cost of accessing it (memory operations are much slower than processor computations in modern computers, with the gap most likely enlarging in the near future [4]). As a result, memory-bounded algorithms will be more challenged to utilize the hardware efficiently. Most current PIC time-stepping algorithms are explicit, with numerical stability constraints limiting work per particle with a single, small timestep. As a result, explicit PIC algorithms are typically memory-bounded, and thus are critically affected by the memory bottleneck.

Nevertheless, there have been fairly successful efforts in porting explicit PIC algorithms to new computing architectures such as graphics processing units (GPUs) [5–9]. In [5], a 2D electrostatic PIC algorithm is implemented on a GPU. A particle data structure is proposed to match the GPU architecture so that the algorithm can be adapted to different problem configurations. Reference [6] describes a GPU implementation of a 2D fully relativistic, electromagnetic PIC code, and introduces a strategy to improve performance of a charge-conserving current deposition scheme (which would otherwise require many conditional branches). Reference [7] further describes an implementation of a 2D fully relativistic, electromagnetic PIC code on a GPU-cluster via domain decomposition, featuring MPI for inter-node communication. References [8–10] report efforts to improve the memory efficiency of particle-grid interpolations and to reduce memory usage in gyrokinetic codes. In Ref. [11], a speedup of about 2 is reported with a hybrid CPU-GPU simulation approach vs. a CPU-only one on tens of thousands of nodes.

In this study, we focus on implementing a novel fully implicit PIC algorithm for electrostatic simulation[12] on a heterogeneous CPU-GPU architecture. Unlike explicit PIC, the fully implicit PIC algorithm does not feature a numerical stability timestep constraint. The method can solve either the coupled Vlasov-Ampère (VA) or Vlasov-Poisson (VP) system of equations, discretized to feature exact local charge and global energy conservation theorems. A Jacobian-free Newton-Krylov (JFNK) nonlinear solver is used to converge the nonlinear residual to a tight nonlinear tolerance. Crucial to the implicit PIC algorithm is the concept of particle enslavement, whereby the integration of particle orbits is an auxiliary computation, segregated in the evaluation of the nonlinear residual. Particle enslavement has several advantages. Firstly, it results in much smaller residual vectors (and therefore in a much reduced memory footprint) because only fields are dependent variables in the solver. Secondly, it affords one significant freedom to perform the particle orbit integration. In Ref. [12], this freedom was exploited to implement a self-adaptive, sub-stepping, charge-conserving particle mover.

Despite the advantages of particle enslavement, the particle integration step remains the most expensive element in the fully implicit PIC algorithm. This is so because particle orbits need to be computed for every nonlinear residual evaluation, and many such residuals are computed as JFNK converges to a solution. This issue is exacerbated further by the ability of implicit PIC to use large timesteps, which requires many orbit integration sub-steps per timestep in the particle mover.

Being the most time-consuming operation in the fully implicit PIC algorithm, the particle orbit integration is the obvious target for hardware acceleration. Our implementation

exploits the flexibility afforded by particle enslavement, and targets the particle orbit integration step for the GPU, while the field solver remains on the CPU. Contrary to most PIC algorithms, we show that the implicit particle mover is compute-bounded, and therefore has the potential of efficiently utilizing both the extreme multi-threading capability and the large operational throughput of the GPU architecture. Communication between CPU and GPU routines involves particle moments only (which are grid quantities), and not the particles themselves, thus minimizing the impact of memory bandwidth bottlenecks.

We use the roofline analytical performance model[13] to help understand the performance limitations and bottlenecks of the algorithm and achieve high performance on the GPU. There are several design constraints for optimizing code on a GPU [14–18]:

- Global memory operations are very expensive, and can severely limit the throughput of the simulation.
- Not all arithmetic operations are equally fast. The slow operations, such as square root and division, can hinder the computational throughput.
- CUDA employs a lockstep execution paradigm within a warp, which comprises 32 threads. Divergent control flow is allowed, but results in performance degradation.
- Memory collisions occur when many threads in parallel try to access the same memory location at the same time. Resolving them serializes the code and can become the bottleneck in parallel computations.

To mitigate the impact of these constraints on GPU performance, we have implemented a series of thread-level and warp-level optimizations in the particle orbit computation without sacrificing accuracy. As a result of these optimizations, our implicit particle mover achieves up to 300-400 GOp/s for VA and VP, respectively, on an Nvidia GeForce GTX580, with the VP approach performing better on account of the memory-collision issue. The corresponding GPU efficiency is $20-25\%$ of peak performance. The accuracy and performance of the overall hybrid CPU-GPU implicit PIC algorithm is demonstrated using a challenging, long-timescale ion acoustic wave (IAW) simulation. It is shown that a mixed-precision implementation, in which the CPU JFNK code uses double-precision and the GPU particle mover code uses single-precision, can be sufficient for accuracy. For the test case chosen, this setup results in speedups of the hybrid CPU-GPU algorithm vs. the CPU-only one up to 100. A defect-correction approach has also been implemented that enables the hybrid algorithm to deliver double-precision results. In this case, about a third of the GPU calls per time step are made in double precision, and the speedup drops to ~ 40. This should be compared to a factor of 25 speedup obtained when all GPU calls are made in double precision. These speedups are consistent with Amdahl's law, as the particle computation takes $\gtrsim 98\%$ of the overall computation time for the test case chosen.

The rest of the paper is organized as follows. Section 2 introduces the Nvidia GPU Fermi architecture and the roofline model. Section 3 describes the specific GPU optimizations introduced in the adaptive, charge-conserving particle mover. Section 4 shows the performance and efficiency results of numerical experiments, including the complete IAW test case integrated with the JFNK solver. Finally, we provide some discussion and conclusions in Sec. 5.

2. An analytical performance model for GPU computing

As modern computer architectures shift from single- to multi-core or many-core processors, parallel programs must be able to exploit increasingly large concurrency efficiently. This, in turn, places strong emphasis on identifying performance bottlenecks and sources of latencies. This task is facilitated when programmers have some basic understanding of the underlying hardware, such as the memory hierarchy and the processor computing capabilities, and target the optimizations for that hardware. In this study, we focus on the GPU architecture, which we introduce next.

2.1. Nvidia GPU architecture

Contemporary Nvidia GPUs[19] are capable of performing scientific computations programmed in high-level languages such as CUDA C/C++ and CUDA Fortran, with high accuracy (supporting IEEE standards) and high performance (theoretical throughput over trillion floating-point operations per second or TFLOPS). GPUs consist of many processing units. For instance, the newest Nvidia GPU to date, named Fermi, has up to 16 streaming multi-processors (SM). Each SM contains 32 processors, or CUDA cores, which perform floating-point, integer, and logic operations. SMs also contain 4 special function units (SFU), which calculate fast floating-point approximations to certain complex operations such as reciprocal, reciprocal square root, etc.

GPUs also contain its own memory system. From slow to fast, one finds off-chip global memory, on-chip shared memory, and on-chip register files. In addition, Fermi GPUs are equipped with a two-level (L1 and L2) read/write cache hierarchy. Specifically, Fermi GPUs contains 2 to 6 GB global memory and 768 KB L2 cache; each SM contains 64 KB configurable shared memory/L1 cache, and 128 KB registers. A unique feature of the GPU memory system is that all levels (except for the L2 cache) can be explicitly managed by the programmer.

In order to gain insight into the maximum performance and efficiency of a given algorithm running on a GPU, we adopt an analytical performance model, the so called roofline model[13]. We proceed to introduce the roofline model and its application for the Nvidia GPU architecture. In the sequel, we assume that all computational operations are on 32-bit words (e.g., single-precision) unless otherwise specified.

2.2. Roofline model

The roofline model is motivated by the fact that the bandwidth of current computer off-chip memory is often much slower than the throughput of the processing unit[20]. For instance, the Nvidia GeForce GTX580 GPU has a DRAM bandwidth of 192 GB/s, whereas its peak floating-point operational throughput is 1581 GFLOPS. This large discrepancy makes identifying whether the program is memory-bounded or compute-bounded critical to target optimizations. If memory-bounded, the program should maximize the use of fast memory; if compute-bounded, it should minimize the number of operations.

The roofline model provides a simple method to determine whether an algorithm is either memory-bounded or compute-bounded. The key figure of merit is the operational intensity (OI), defined as the ratio of compute operations to memory operations. An algorithm is compute-bounded for OI higher than the balanced OI, and is memory-bounded otherwise.

The balanced OI of the target device is defined as the ratio of peak-operational throughput to memory bandwidth (which is about 8 FLOP/B for the Nvidia GeForce GTX580 GPU).

An algorithm's compute efficiency is commonly defined as the ratio of its FLOPS vs. the peak theoretical compute performance. On a Nvidia GPU, the latter is calculated as

$$(\text{number of cores}) \times 2\,\text{Flop/CC} \times (\text{clock rate}), \tag{1}$$

where the factor of 2 comes from the fused multiply-add (FMA) operation, which computes one AXPY operation (i.e., $a \times x + y$) per clock cycle (CC). On the GeForce GTX580 GPU, number of cores = 512, and clock rate = 1.544 GHz, resulting in 1.58 TFLOPS. Note that, by definition, the peak theoretical performance may be reached by algorithms based on FMA operations only. However, most algorithms in scientific computing mix floating-point, integer, and logic operations. Such algorithms *cannot* reach the peak theoretical performance, no matter how well optimized. Therefore, for a given algorithm, it is useful to define an intrinsic efficiency based on its specific operations.

We define the *theoretical operational throughput* as the maximum theoretical performance of a compute-bounded algorithm. It can be calculated as

$$(\text{number of SM}) \times (\text{average operational throughput}) \times (\text{clock rate}), \tag{2}$$

where

$$(\text{average operational throughput}) \equiv \sum(\text{operations}) / \sum(\text{clock cycles}) \tag{3}$$

is the average number of operations per clock cycle per stream multi-processor. The theoretical operational throughput assumes that all memory and instruction latencies are completely hidden or negligible, without performance overhead of any kind. The *intrinsic efficiency is* defined as the ratio of the actual operational throughput vs. the theoretical one (Eq. 2). It indicates an algorithm's real effectiveness in using a given target hardware.

In what follows, we compare the operational throughput of several basic operations, which are the building blocks of our particle mover algorithm, on the Nvidia Geforce GTX580 GPU in the context of the roofline model.

2.3. Operational throughput

Each CUDA SM of the Nvidia GPU Fermi GF100 architecture[21] has 32 floating-point units (FPUs), 32 integer arithmetic logic units (ALUs), and 4 SFUs, capable of processing different types of computations simultaneously. Depending on the hardware implementations, the throughput of different computational operations varies[17]. To illustrate this, we have created a simple CUDA code to micro-benchmark (similar to those used in Ref.[22, 23]) the throughput of some basic floating-point and integer operations. In these tests, many identical operations (using unrolled loops) are performed by each thread, and many concurrent threads (with 100% occupancy) are employed. The test code is compiled with nvcc v4.0 compiler. The assembly code generated by the PTX (or cuobjdump) tool is examined to ensure that instructions are executed as intended. Results are shown in Fig.1, from which we can make the following observations:

- As expected, FMA reaches the peak theoretical performance for large OIs.

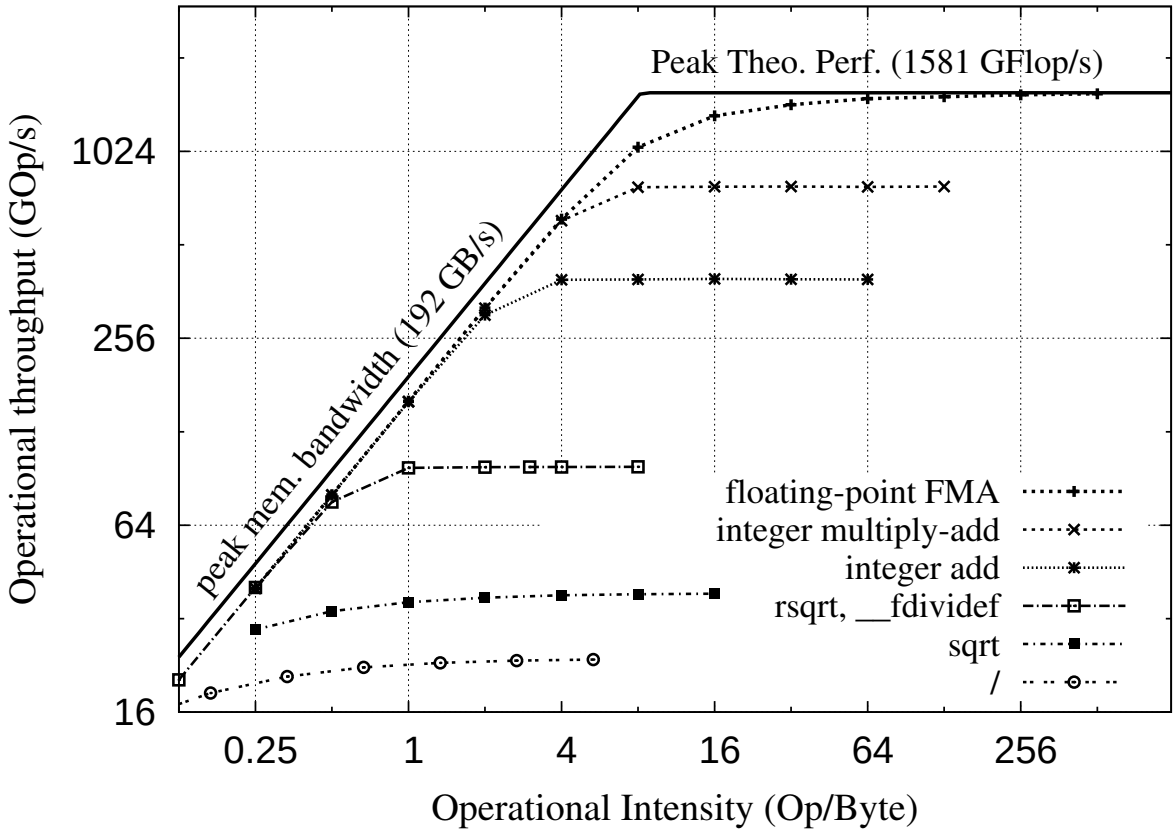

Figure 1: Roofline model for Nvidia GeForce GTX580 on a $\log_2$-$\log_2$ scale. The performance lines are obtained by artificial algorithms which repeat a single operation multiple times. Saturation of the curves shows the maximum operational throughput in the compute-bounded regime for a given operation. In the plot, sqrt, /, rsqrt, and __fdividef stand for IEEE compliant square root, IEEE compliant division, intrinsic reciprocal square root, and intrinsic fast division, respectively.

Table 1: Throughput of floating point (FP), arithmetic and logic (AL), and special function (SF) operations on GeForce GTX 580. Throughput is measured as operations per clock cycle per multiprocessor. The compiler nvcc v4.0 is used to produce the results.

FP add,mul, (fma)	AL add, mul, logic, cvt (mul-add)	SF rsqrt, __fdividef	/
32(64)	16(32)	4	$\lesssim$1

- The peak performance of integer operations (multiply-add) is at best half of the peak theoretical performance. In other words, integer operations are slower than floating-point operations.

- Standard (IEEE compliant) square root (sqrt) and division(/) operations are one to two orders of magnitude slower than simpler floating-point or integer operations.

- The throughput of intrinsic reciprocal square root (rsqrt), or intrinsic fast division in single-precision (__fdividef), hardware-implemented in SFUs, is exactly 16 times lower than the peak theoretical performance. This is expected, as it equals the ratio of the number of FPU (32) and SFU(4) multiplied by 2 (operations per FMA). Intrinsic operations are much faster than IEEE versions at the cost of being less accurate. We will show in later sections that, under some situations, fast intrinsic functions can be accompanied by a few FMA operations to improve accuracy. The hybrid use of SFUs and FPUs has the advantage of exploiting both units concurrently, and therefore offers the potential to achieve sufficient accuracy with enhanced performance.

Table 1 lists the maximum throughput of the operations on the GPU found by the micro-benchmarks. With the throughput of each basic operation known, we can now calculate the average operational throughput for a particular algorithm [Eq.(3)], with the total number of clock cycles required to execute those operations calculated as $\sum_i N_i \times 32/OT_i$. Here, N_i is the number of operations of type i, and OT_i is the operational throughput from Table 1.

We proceed to analyze in detail the GPU implementation of the implicit particle mover of Ref.[12].

3. GPU implementation and performance analysis of the adaptive, charge-conserving particle mover

We pursue the development of a hybrid CPU-GPU algorithm in which the particle push is farmed off to the GPU, while the nonlinear solver remains on the CPU. In our implementation, particles reside in the GPU global memory, and no particle transfer is needed between CPU and GPU for the field solver. Only grid quantities (i.e., the electric field and moments collected from the particles) are transferred between the two architectures. Nonlinear iterations are performed on the CPU until nonlinear convergence is achieved (convergence is enforced on field quantities, with particle quantities obtained self-consistently). The process is repeated every timestep.

This section focuses on the implicit particle mover [12], which features three main properties: self-adaptivity, automatic local charge conservation, and absolute numerical stability.

Self-adaptivity is achieved by particle sub-stepping, with the sub-timestep controlling the discretization errors of the orbit integral. In each sub-step, particles are pushed along a straight line. Local charge conservation is automatically enforced by ensuring particles land at cell boundaries during the orbit integration process. A Crank-Nicolson (CN) integration scheme, which is time-centered and implicit, is employed to guarantee numerical stability for arbitrary timesteps.

It is well known that contemporary GPUs are capable of launching a large number of threads (tens of thousands on current-generation devices) in a SPMD (single program, multiple data) style. They are very attractive for particle simulations, as particle orbits are independent of each other. However, the highly dynamic nature of the implicit mover may prevent the algorithm from achieving high performance and efficiency on the GPU. To begin with, the simulated systems are typically highly nonlinear and inhomogeneous. As a result, threads pushing particles with different positions in phase space will experience very different workloads. When large timesteps are employed, each particle follows an orbit according to local conditions, undergoing an indefinite number of sub-timesteps and cell-crossings. Similarly, an iterative solution of the coupled Crank-Nicolson equations introduces unpredictability in the algorithm, for the number of iterations for convergence is unknown and particle-dependent. The resulting unpredictable logical paths (per thread) create divergent branches, serializing the executions in a given warp (32 threads). Non-divergent branches may also be created, with some idle threads waiting for others to finish. It follows that it may be very difficult to keep all threads busy, resulting in parallel performance degradation.

Additional performance degradation can result from inefficient memory operations. One prime example is parallel scatter-gather operations, such as field interpolations to particles and moment integration from particles, which can significantly slow down the simulation. Random access to the field may have a large performance penalty, for instance, if the L1 cache miss-rate is high. Moment accumulation may hinder the computation due to the cost of resolving memory collisions. This occurs when two or more threads try to write the same memory location simultaneously. Ensuring correctness requires using atomic operations, which serialize the accumulations.

Despite these challenges, we demonstrate in the following sections that the implicit particle mover algorithm can in fact be relatively efficient on the GPU. With the aid of the roofline model, we identify that the implicit particle mover is compute-bounded, and therefore significant performance improvement can be achieved through targeted optimizations. We motivate specific optimizations with a baseline GPU implementation of the mover (described below), aiming at minimizing clock cycles (per thread) without accuracy degradation. We also describe warp-level optimizations, including our particle-sorting strategy and the use of a warp vote function, to minimize load imbalance and control-flow latencies.

Before getting into the detailed optimizations of the mover, we introduce its basic memory management. There are mainly two groups of memory operations. The first group involves reading particle quantities $\{v_p,x_p,i_p\}$ (denoting velocity, position, and cell index, respectively) at the timestep n from global GPU memory, and writing the updated quantities at the timestep $(n+1)$ to global GPU memory. Since the particle-orbit calculations are independent of each other, the load and store of the particle quantities, which we group in a structure-of-array fashion, are trivially coalesced (making stride-one access of the global memory for optimal performance[18]). The second group involves reading/writing grid quantities, such

as the electric field E, current density j, or charge density ρ. The E field is read-only; either L1 cache or texture memory can be exploited to accelerate repeated readings. Our approach to the accumulation of moments (such as j or ρ), which is often found to be the bottleneck of many PIC implementations [7, 8, 10], is discussed in detail below (Sec. 3.4).

To frame the subsequent discussion, we divide each particle sub-step into four algorithmic elements:

1. Estimate the sub-timestep $\Delta\tau$.
2. Integrate the orbit over $\Delta\tau$ using a Crank-Nicolson scheme.
3. If a particle crosses a cell boundary, make it land at the boundary.
4. Accumulate the current density on the grid points (for VA, but not for VP).

This process is repeated over many sub-steps until the end of the timestep, at which point we either take the orbit average of the current density (for VA), or accumulate the charge density (for VP). In the following, we describe the baseline algorithm and targeted optimizations for each element.

3.1. Algorithmic element #1: estimate of the sub-timestep

The first algorithmic element employs a standard local-error-control method[24] to estimate the sub-timestep, by taking the difference between the truncation errors of the Euler's scheme (a first-order method) and Heun's scheme (a second-order method) to be smaller than a specified tolerance. The resulting formula reads[12]

$$\|le(\Delta\tau)\|_2 < \varepsilon_a + \varepsilon_r \left\|r^0(\Delta\tau)\right\|_2, \tag{4}$$

where $\|\cdot\|_2$ denotes the L_2-norm of enclosed vector, $le(\Delta\tau) = \frac{(\Delta\tau)^2}{2}\{a_p^\nu, \left(\frac{\partial a_p}{\partial x} v_p\right)^\nu\}$ is the local truncation error of the sub-timestep ν, ε_a and ε_r are absolute and relative tolerances, respectively, and $r^0(\Delta\tau) \equiv \{v_p^\nu, a_p^\nu\}\Delta\tau$ is the initial residual. For $\partial a_p/\partial x$, we take

$$\partial a_p/\partial x \cong \frac{q}{m}(E_i - E_{i-1})/\Delta x, \tag{5}$$

when the particle is in cell i (or between grid points $i-1$ and i) at time level ν. This is exact for linear interpolations. It follows that the estimate can be found by solving a quadratic equation for $\Delta\tau$.

A direct implementation of the above method would require about 33 floating-point operations, 6 special function operations, and 1 division operation. To optimize the method, we replace the L_2-norm with the L_1-norm, e.g., $\|\{x_1, x_2\}\|_1 = |x_1| + |x_2|$, which suffices for the error estimate without requiring a square root. The equation for $\Delta\tau$ [from Eq.(4)] can be written as

$$\alpha\Delta\tau^2 - \beta\Delta\tau - \gamma^2 = 0, \tag{6}$$

where $\alpha = \left(\left|a_p^\nu\right| + \left|\left(\frac{\partial a_p}{\partial x} v_p\right)^\nu\right|\right)/2$, $\beta = \varepsilon_r\left(\left|a_p^\nu\right| + \left|v_p^\nu\right|\right)$, $\gamma^2 = \varepsilon_a$. To optimize the computation further, we avoid solving the quadratic equation by noting that it is sufficient to estimate $\Delta\tau$ from the absolute and relative tolerances separately:

$$\alpha\Delta\tau^2 = \gamma^2, \tag{7}$$

$$\alpha\Delta\tau = \beta, \tag{8}$$

and then take $\Delta\tau = \max(\gamma d, \beta d^2)$, with $d = \alpha^{-1/2}$ computed via the intrinsic rsqrt function only once.

As a result of these optimizations, the estimate of sub-timestep requires 30 floating-point operations (including 6 FMAs) and 1 special-function operation. The optimized algorithm has reduced the per-thread clock cycles from 106 to 28, largely achieved by replacing three square roots and one division with just one reciprocal square-root operation.

3.2. Algorithmic element #2: Crank-Nicolson particle move

The second element of the mover is a CN step using the estimated sub-timestep $\Delta\tau$:

$$\frac{x_p^{\nu+1} - x_p^{\nu}}{\Delta\tau^{\nu}} = v_p^{\nu+1/2}, \tag{9}$$

$$\frac{v_p^{\nu+1} - v_p^{\nu}}{\Delta\tau^{\nu}} = a_p^{\nu+1/2}, \tag{10}$$

where $1/2$ denotes a mid-point average, i.e., $v_p^{\nu+1/2} \equiv \left(v_p^{\nu} + v_p^{\nu}\right)/2$. These equations are implicit and coupled (but not stiff), and are generally solved by a fixed-point iterative method, e.g., Picard's method.

It turns out that, when employing first-order B-spline interpolations, Eq.(9) and (10) can be solved directly as (omitting the subscript p)

$$v^{\nu+1} = \frac{a^{\nu}\Delta\tau^{\nu} + \left[1 + \left(\frac{\partial a}{\partial x}\frac{(\Delta\tau)^2}{4}\right)^{\nu}\right] v^{\nu}}{1 - \left(\frac{\partial a}{\partial x}\frac{(\Delta\tau)^2}{4}\right)^{\nu}}, \tag{11}$$

where $\partial a/\partial x$ is given by Eq.(5). The division in Eq.(11) can be replaced by the intrinsic reciprocal function without loss of accuracy, as follows. We write $v^{\nu+1} = NR = N \times \mathrm{RCP}(D)$ where N, D stand for the numerator and denominator in Eq.(11), respectively, and $R \equiv \mathrm{RCP}(D)$ stands for the fast intrinsic reciprocal function of CUDA. Note that the maximum ULP (unit in the last place) error of the fast reciprocal is about 1[25] (only slightly lower than IEEE required 0.5 ULP precision). This is precise to the 8th significant digit, as the last bit of the single precision number corresponds to $5.96 \cdot 10^{-8}$. To reduce the error further, we take one more iteration of Newton-Raphson's method such that $v^{\nu+1} = NR(2 - DR)$[26].

The optimization introduced here involves two steps: it first eliminates the control-flow condition needed in the convergence test of the nonlinear iteration, and then replaces the division by the intrinsic fast reciprocal function. In the latter step, one extra step of Newton-Raphson is taken to ensure sufficient precision. The optimization slightly increases the number of operations from 11 to 14, but the number of clock cycles of this step is significantly reduced from 43 to 22.

3.3. Algorithmic element #3: Particle cell-crossing

The third algorithmic element checks whether the particle has moved into another cell after the CN move. If a crossing occurs, the particle is forced to stop at the cell boundary. The corresponding sub-timestep is found by solving the quadratic equation:

$$F(\Delta\tau) = \frac{a^{\nu+1/2}}{2}\Delta\tau^2 + v^{\nu}\Delta\tau - \Delta x^{\nu} = 0,$$

which is obtained by fixing the final particle position at the boundary in question, and combining Eq. (9) and (10). The corresponding particle velocity can be found according to the energy principle to be:

$$v^{\nu+1} = \mathrm{sng}(\Delta x^{\nu})\sqrt{(v^{\nu})^2 + 2a^{\nu+1/2}\Delta x^{\nu}}, \tag{12}$$

where $\mathrm{sng}(\Delta x^{\nu})$ returns the sign of $\Delta x^{\nu}(= x^{\nu+1} - x^{\nu})$, which signals the direction of particle motion.

The above treatment requires a square root and a division, which can be optimized by the following (inexact) Newton's method. The solution is first approximated by

$$\Delta\tau_0 = \frac{-v^{\nu} + \mathrm{sng}(\Delta x^{\nu})\sqrt{\left(v_p^{\nu}\right)^2 + 2a_p^{\nu+1/2}\Delta x_p^{\nu}}}{a^{\nu+1/2}} \tag{13}$$

using fast intrinsic functions rsqrt and __fdividef. Subsequent iterations are performed as:

$$\Delta\tau_k = \Delta\tau_{k-1} - F(\Delta\tau_{k-1})/F'(\Delta\tau_{k-1}), \tag{14}$$

where $F'(\Delta\tau_{k-1}) = a^{\nu+1/2}\Delta\tau_{k-1} + v^{\nu}$ is the Jacobian. A fast division can be applied to update Eq. (14). This is a safe approximation because the convergence of Newton's method is robust against small errors in the Jacobian[27]. Note that each of the applied intrinsic functions has a maximum error of 2 ULP[17], which provides excellent initial value for Newton iterations. Because of the quadratic convergence rate of Newton's method (i.e., the correct digits double for every iteration), this last step ensures that the solution is accurate to the last digit of a single-precision number.

Overall, we have replaced a square root and a division by a fast reciprocal square root and two fast divisions. Consequently, the number of clock cycles for particles moving inside the cell is reduced from 74 to 47.

3.4. Algorithmic element #4: particle current/charge accumulation

In the VA approach, the fourth element employs a standard interpolation procedure to accumulate the current density on the grid points from particles:

$$j = \frac{1}{\Delta x}\sum_p q_p v_p^{\nu+1/2} S(x - x_p^{\nu+1/2}),$$

where Δx is the cell size, and S is the shape function. This is done for every sub-timestep. In the VP approach, the charge density is collected from each particle as:

$$\rho = \frac{1}{\Delta x}\sum_p q_p S(x - x_p^{n+1}).$$

This is done only at the end of the orbit computation. In the baseline implementation, all the accumulations are performed on shared memory, using the floating-point atomicAdd function, and the final results are written back to global memory at the end of the timestep.

It is desirable to avoid memory collisions as much as possible in the accumulation process. On one hand, memory collisions are completely avoided when each thread accesses its own

copy of the physical domain in memory, but this is very demanding memory-size-wise. On the other hand, memory-size requirements are minimized when all threads (of one thread block) access the same memory domain, but this results in frequent memory collisions. To balance best efficiency of the accumulation and limited resources of shared memory, we provide each warp with its own memory domain[28]. This avoids memory collisions between threads of different warps. Collisions within a warp are resolved by the shared-memory, floating-point atomicAdd function.

A second optimization (for VA only) replaces the shared-memory accumulations of local current density in a given cell by register accumulations. This optimization exploits the fact that register access is faster than shared memory[18], and collisions with register accumulations are absent. Specifically, each thread employs two local register variables (one per cell face) to accumulate current density as the particle sub-steps within a cell. Register values are atomically added to shared memory and reset to zero only when the particle crosses a cell boundary.

Acceleration has been achieved in the moment accumulation by reducing memory collisions (for both VA and VP), and by reducing the usage of atomic operations (for VA). Quantitative results are presented in the following sections.

3.5. Roofline analysis

We proceed to show that the implicit particle mover algorithm is compute-bounded. The analysis is carried out for VA for brevity. A similar analysis, with similar conclusions, applies to VP.

The particle mover algorithm requires two (read/write) single-precision memory transfer operations from global memory of three particle quantities $\{x, v, i\}$ per particle orbit integration (recall that the electric field is read-only, and is cached for fast access). The corresponding (global) memory throughput is $2 \times 3 \times 4 = 24$ B per particle. For the sake of argument, we assume 20 sub-steps and 2 cell-crossings in a typical particle orbit per timestep, which results in 1762 (1410) computational operations per particle in the baseline (optimized) algorithm. It follows that the corresponding operational intensity is:

$$\mathrm{OI}_{\mathrm{im}} \simeq \begin{cases} 73 & [\text{baseline}] \\ 59 & [\text{optimized}] \end{cases} (\mathrm{Op/B}).$$

This is 7 to 9 times larger than the balanced OI (see Sec. 2.2),which confirms that the implicit particle mover algorithm is compute-bounded.

3.6. Particle sorting strategy

It is often beneficial to sort particles, as the memory operations are more efficient with improved data reuse. Previous studies[5, 6, 10, 29] have focused on explicit PIC algorithms. Explicit schemes employ small timesteps, so that only a small fraction of the particles that cross cells (or sub-domains) need sorting. The goal is to keep the memory space consistent with the physical space such that particles that are physically close are also co-located in memory. Those particles moving across cells (or sub-domains) need to be rearranged in the particle array to preserve memory collocation. The complexity of this process is $O(\eta N)$[6],

where N is the total number of particles and η is fraction of the particles crossing cell (or sub-domain) boundaries. This approach is appropriate for explicit schemes, where the timestep is very small.

Explicit PIC sorting strategies, however, are not suitable for implicit PIC when a large timestep is used. In this case, many particles may cross cells in an implicit timestep, thus increasing the overhead of shuffling particles in memory and resulting in frequent memory collisions. In addition, particles with different velocities require different amount of work per orbit integration, which causes load imbalances and divergent branches. Hence, in an implicit PIC context, particle sorting should focus on improving thread load-balancing and on minimizing memory collisions. We address the former by sorting particles such that each warp deals with particles with similar velocities (in sign and magnitude). The latter is addressed by having each thread within a warp integrate a particle located in a separate cell in physical space.

Our particle sorting approach features a two-pass implementation. In a first pass, we divide the 2D phase space $(x - v)$ into rectangular cells using a Cartesian grid. We label each cell with an integer number (starting from zero) via lexicographic ordering along the physical coordinate (rows). Assuming that the physical domain is discretized with N_g cells, the first row in phase space (corresponding to the same velocity interval) will be labeled with numbers 0,1,...,$N_g - 1$. The second row (corresponding to the next velocity interval) will be labeled with numbers N_g,...,$2N_g - 1$, and so on. Next, we label particles in each cell with the corresponding cell number. We then collect the total number of particles within each cell, and use a prefix sum [30] to aggregate particles in previous cells lexicographically. This gives:

$$N_{i0} = \sum_{j=0}^{i-1} N_j, \tag{15}$$

where N_j is the number of particles in cell j, and N_{i0} is the total number of particles up to (but excluding) cell i. Here, i, j are lexicographic indexes, and $N_{00} = 0$. In a second pass, we sort particles according to their velocities, and we place the particles into a 1D array in which each continuous and aligned N_g particles belong to consecutive N_g cells in physical space (x), and also have similar velocities. The key in this second pass is to reserve the particle locations in the 1D array from information collected during the first pass. In particular, N_{i0} provides the memory address (starting from index zero) of the 1D particle array for the first particle in cell i. Additional particles in the same cell with particle index $p \in [1, N_j - 1]$ are placed in the 1D array with memory address $N_{i0} + N_g \times p$.

When particles are sorted in this way, memory collisions within a warp are avoided as long as particles do not cross cell boundaries. Furthermore, the load balance improves because particles with similar velocities will have similar orbit lengths, thus improving the efficiency of the algorithm at the warp level. This approach needs two copies of the particle array (corresponding to timesteps n and $n + 1$), and the computational complexity of the sorting scheme scales with the number of particles. However, only a single sorting step is required per implicit timestep, making the overall overhead manageable.

3.7. Control-flow optimization

The implicit particle mover uses control flows extensively to ensure an accurate orbit integration with large timesteps. In this case, performance may degrade on the GPU because warp execution results in branches, which are executed sequentially.

Branches are created, for instance, when a particle crosses a cell boundary, or a particle orbit terminates, or memory accumulations collide, etc. Some optimizations introduced in the preceding sections already address branching and load balancing, e.g., by solving the Crank-Nicholson equations directly (see Sec. 3.2), and by reducing memory collisions (see Sec. 3.4). Particle cell-crossings introduce branches because they happen randomly in a given thread, thus forcing other threads in the same warp to wait. The quantitative performance impact of particle cell-crossing is shown in the next section.

Branches also appear when particle orbits within a warp do not terminate simultaneously. Their impact is ameliorated in our implementation by using the vote function with `all` reduction mode [17] as the exit condition for a given warp. The `vote.all` function returns `true` when the exit condition is satisfied for all the threads in a warp, and it can be used to break an infinite while-loop (which does not create conditional branches). We have found that the warp vote function is more effective for VP than for VA. This is most likely due to the fact that increased atomic operations of the VA approach hide load-imbalance latencies.

4. Numerical experiments

In this section, we first conduct several numerical experiments to characterize the baseline and optimized particle orbit integration algorithms described above. We measure their operational throughput on the GPU, and demonstrate the significant performance and efficiency gains of the proposed optimizations, resulting in a 50-70% intrinsic efficiency (vs. the application maximum operational throughput) for VA and VP, respectively, with a corresponding 20-25% overall efficiency (vs. peak throughput, 1581 GOp/s). We also study their performance sensitivity to various external parameters such as the number of threads and the timestep size, and conclude that performance is generally robust except for the timestep size (which directly affects the OI of the algorithm). We compare the performance of the particle pusher algorithm running on both a CPU (in single precision) and a GPU (in single precision), and demonstrate significant speedups $(200-300)$. Finally, we integrate the GPU particle mover with the full nonlinear solver [12] on a CPU, and test accuracy and wall-clock performance for a challenging ion acoustic wave simulation. We demonstrate that a mixed-precision hybrid CPU-GPU implementation (with the GPU running in single precision and the CPU in double precision) is sufficient from an accuracy standpoint, and that it is able to deliver very large wall clock speedups vs. the double-precision CPU-only implementation. In what follows, code running on the GPU is in single-precision unless otherwise specified.

4.1. GPU performance of the implicit particle mover

The GPU performance of the algorithm is best understood by comparing the theoretical operational throughput to that of a real execution. To compute the theoretical operational throughput, we adopt operations per second (Op/s), instead of FLOPS, as the figure of

Table 2: Breakdown of operations and clock cycles (per thread per sub-step) of the implicit particle mover algorithm. In the table, the left number counts operations, and the right one counts clock cycles.

OP/CC	FP add,mul, fma	AL add, mul, logic, cvt	SF rsqrt, __fdividef	division	total
baseline	64.2 / 48.0	31.8 / 63.6	6.2 / 49.6	2.1 / 67.2	104 / 228.4
optimized	60.1 / 43.5	26.6 / 53.2	2.2 / 17.6	0 / 0	88.9 / 114.3

merit for performance evaluations. This is because floating-point, integer, logic, and special-function operations play important roles in our algorithm. When computing the operational throughput, we need count all operations, especially the slower ones.

In the simulations, we set $\varepsilon_a = 10^{-8}$ and $\varepsilon_r = 0.02$ to be the absolute and relative tolerances of the orbit sub-time-step estimation, respectively (see Sec. 3.1). The corresponding average number of cell-crossings is about 10% of the number of sub-steps. Table 2 lists the theoretical number of arithmetic operations and their respective clock cycles for the baseline and optimized algorithms. We see that, even though the total number of operations does not change much from the baseline to the optimized version, there is a dramatic change in the number of clock cycles. By examining the operational throughput of each category, we see that the baseline algorithm spends most of the time computing only a small number of special functions and divisions, whereas the optimized algorithm significantly reduces the use of those operations. As a result, the theoretical performance nearly doubles after the optimization.

Figure 2 shows the operational throughput of the algorithms running on the GPU as a function of the number of threads. We see that perfect linear scaling continues beyond the physical number of CUDA cores, and performance only saturates when the number of threads exceeds the number of cores by a factor of about 20. This is consistent with Little's law[31], which predicts that the number of threads needed for maximum performance is equal to the number of cores multiplied by the instruction latency ($\sim$ 18 CC on Fermi GPUs[18]). Achieving the maximum performance of Little's law is possible when 1) latencies, including memory-level and instruction-level ones, are hidden by exploiting extreme concurrency with many threads, and 2) the architecture allows very fast switching between threads (GPUs can switch in one CC).

As implemented, the measured performance of the optimized algorithm (for VA) is 320 GOp/s, which corresponds to a 50% intrinsic efficiency and a 20% peak efficiency. This is to be compared with 130 GOp/s of the baseline algorithm (8% of peak). For the VP approach, we get 380 GOp/s (70% intrinsic efficiency, 24% peak efficiency). The performance increase of VP vs. VA can be traced to the much larger number of accumulations required by VA.

Figure 3 shows the run-time breakdown of the algorithm before and after the optimizations. The cost of global memory operations is negligible compared to other operations, confirming that the algorithm is compute-bounded. The most time-consuming parts are the time estimator and the Crank-Nicolson mover. They achieve a significant speedup, a result of both thread-level (through modifications of the algorithm) and warp-level (through particle sorting and vote function) optimizations. The optimizations in the cell-crossing algorithmic element are also effective.

Figure 4 shows the sensitivity of the performance and efficiency of the optimized particle

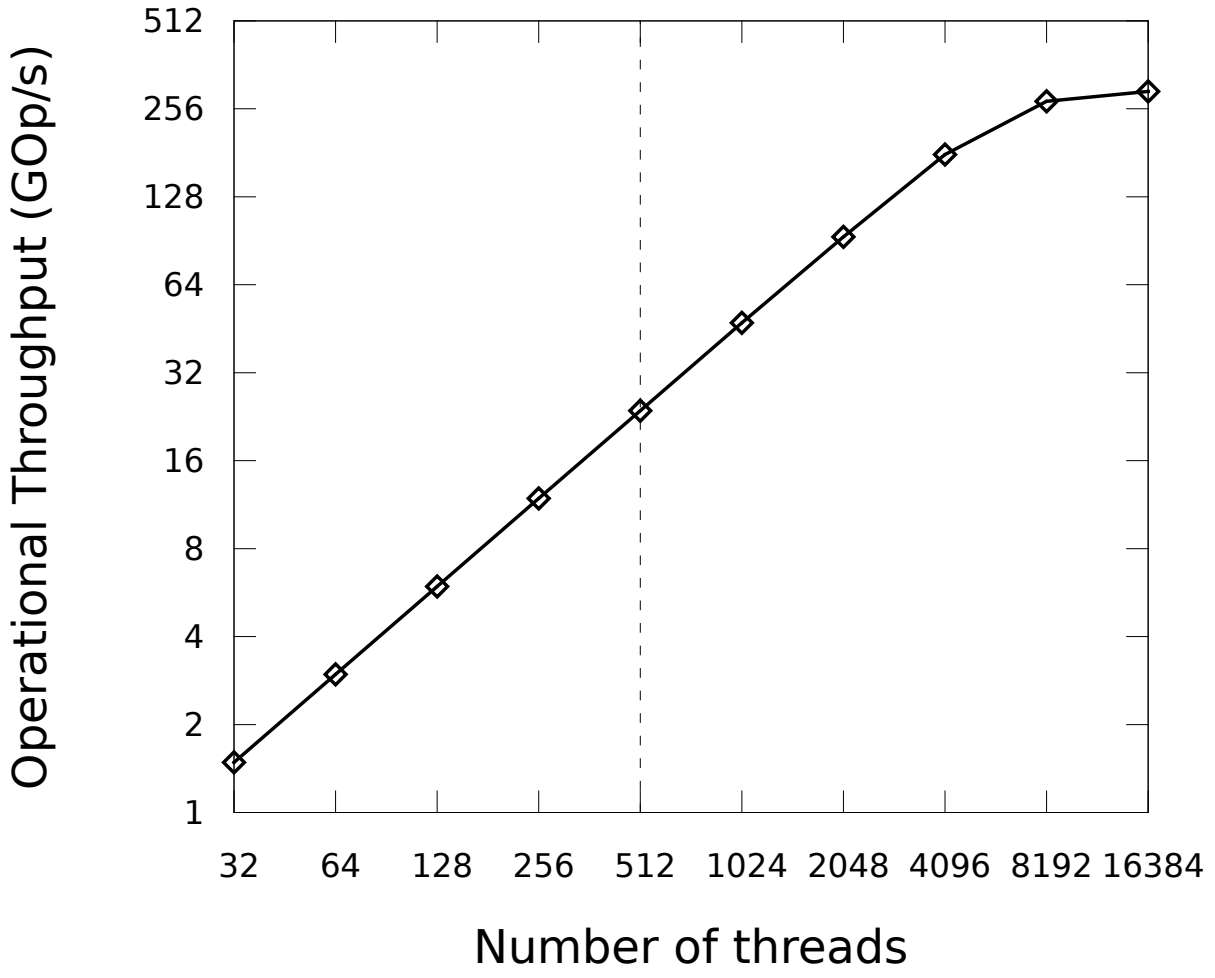

Figure 2: Scaling of implicit particle mover (optimized for VA). The dotted line indicates the number of parallel CUDA cores available on the GPU. The peak performance reaches 320 GOp/s.

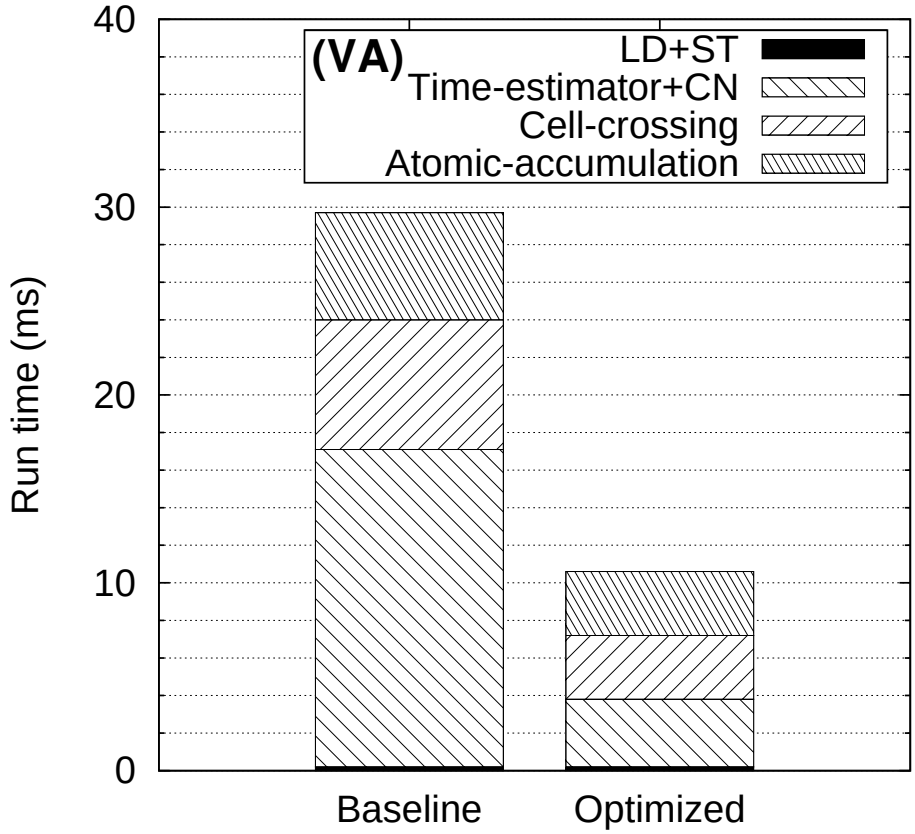

Figure 3: Run-time breakdown of the mover algorithm for the VA approach. The load (LD) and store (ST) of 1,048,576 particle quantities on the global memory take 0.2 ms. The VP algorithm features similar timings, except for the atomic-accumulation step (which becomes negligible).

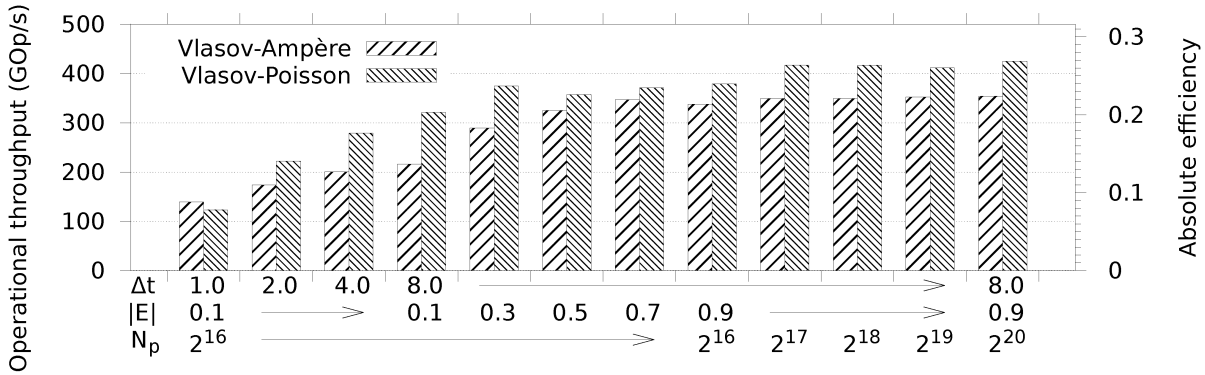

Figure 4: Performance and efficiency of the mover algorithm in single precision on the GPU under conditions varying the timestep (Δt), the field amplitude ($|E|$), and the number of particles (N_p). For large timesteps, the performance is insensitive to the field strength or the particle number.

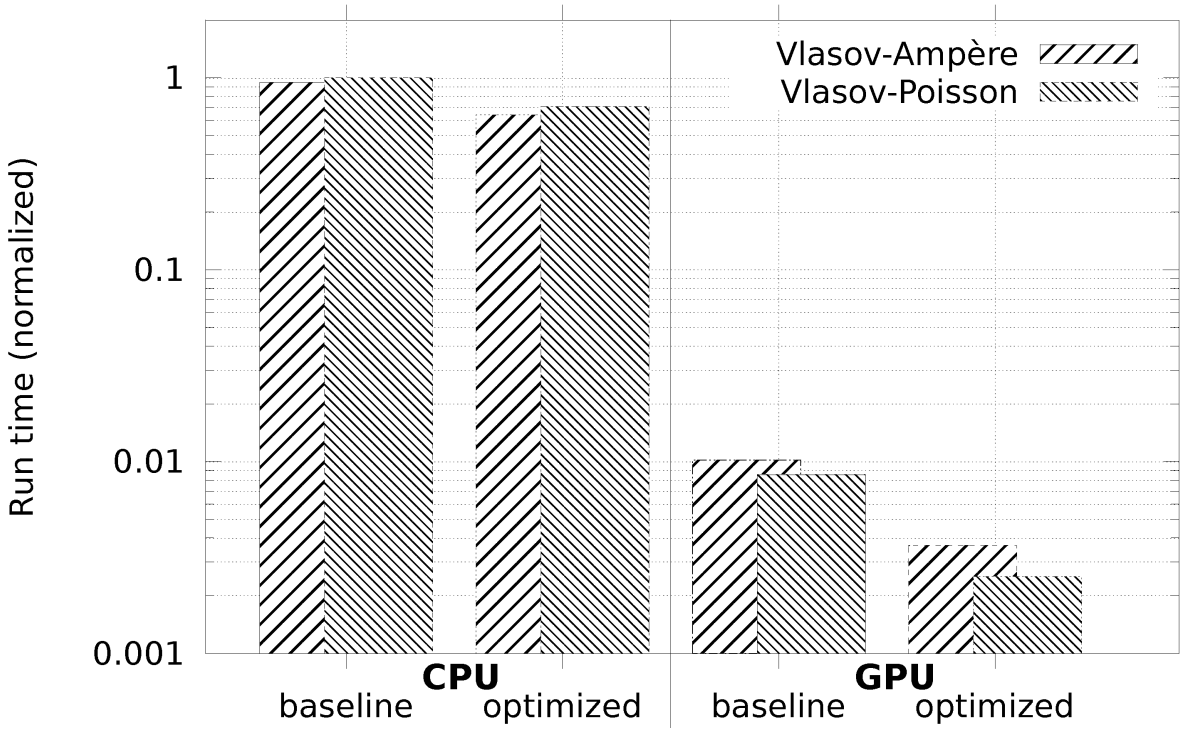

Figure 5: CPU and GPU run time comparison for the baseline and optimized mover algorithm of VA and VP approaches (log scale). The single-precision GPU versions are two orders of magnitude faster than the respective single-precision CPU versions.

mover algorithm for both the VA and VP approaches with respect to timestep, field strength, and number of particles. Clearly, while the performance is insensitive to changes in the field strength and the particles number, it is quite sensitive to changes in the timestep, improving with increasing Δt. For a timestep of 0.1 (typical in explicit PIC simulations), the algorithm is close to a memory-bounded regime, and the performance is low. As the timestep increases from 1 to 10 or larger (typical in implicit PIC simulations [12]), the algorithm becomes compute-bounded, and both the VA and the VP recover good performance as well as efficiency.

4.2. Performance comparison between the CPU and GPU implementations of the particle mover algorithm

For compute-bounded algorithms, a CPU-GPU performance comparison is informative when the speedup is measured against the theoretical peak performance speedup. For a single Intel Xeon CPU X5460 core at 3.16 GHz (used in this work), the peak theoretical performance for executing 2 operations on 4 single-precision variables in the SIMD (single instruction, multiple data) style per clock cycle is:

$$2\,\text{SIMD Op/CC} \times 4\,\text{value/Op} \times \text{clock rate} = 25.2\,\text{GOp/s}.$$

Therefore, assuming the same efficiency on both architectures, the nominal GPU-to-CPU speedup is around 60 ($\simeq 1581/25.2$).

Actual CPU vs. GPU timing comparisons are depicted in Fig. 5. We have programmed the CPU code with standard C/C++, without implementing any explicit Intel SSE SIMD instructions, or multi-threading using multi-cores. We have relied on the compiler (Intel C/C++ compiler version 12.0.4) to optimize automatically. We first compare the performance of the baseline algorithm between the CPU and GPU (this case is identical to the one shown in Fig. 3). The codes are very similar on both processors, except for fast memory management, which is explicit on the GPU (Sec.(3)). We find that the absolute GPU efficiency ($\sim 8\%$) is larger than that of the CPU code ($\sim 5\%$). The speedup scores 100. The increased speedup (100 vs 60) is consistent with the increase in efficiency (8% vs 5%).

After the optimizations, the speedup reaches about 200 and 300 for the VA and VP approaches, respectively, corresponding to a GPU efficiency of 20-25%. We note that some of the modifications introduced in Sec.3 (such as the optimized sub-timestep estimator) have also been used to improve the performance on the CPU.

4.3. Performance of the hybrid, mixed-precision CPU-GPU fully implicit PIC solver

We proceed to compare the performance of a hybrid, mixed-precision CPU-GPU implementation of the fully implicit PIC algorithm vs. the CPU-only serial implementation in Ref. [12] (but incorporating applicable algorithmic optimizations developed in this study). We follow this reference and use the ion acoustic wave (IAW) case for our numerical tests. As set up, the IAW features large-amplitude waves that can propagate in an unmagnetized, collisionless plasma without significant damping.

Figure 6 shows VA simulation results from both the hybrid implementation and the CPU-only one. We depict the ion and electron kinetic energy, as well as conserved quantities (local charge, total momentum, and total energy). The relative nonlinear tolerance of the JFNK solver is 2×10^{-4} for the hybrid CPU-GPU version, and 10^{-10} for the CPU-only version. The nonlinear tolerance is larger in the GPU version to prevent stalling of the nonlinear iteration due to lack of numerical precision in the calculation of the current density. As a result, the average number of Newton iterations per timestep is about 5 for the single precision implementation, vs. 8 for the double-precision one.

Figure 6 demonstrates very good agreement between the two implementations over a very long simulation span (100 IAW periods or 8000 plasma periods). Conservation of energy is enforced to the nonlinear tolerance level, i.e., 10^{-12} and 10^{-6} for double- and single-precision simulations, respectively. The total momentum is not conserved exactly in either implementation, but it remains bounded and fluctuates at similar levels in both simulations. The wall clock comparison of the two simulations shows a speedup of over 130. The speedup reduces to 70 when the same nonlinear tolerance (2×10^{-4}) is employed in both CPU-GPU and CPU-only simulations. These speedup factors are consistent with Amdahl's law, as the particle mover represents $\sim$98% of the overall cost of the algorithm in the CPU-only implementation. Larger speedups are possible for problem setups where the particle cost represents a larger fraction.

We have retrofitted the mixed-precision, hybrid CPU-GPU implementation with a defect-correction algorithm [32] that delivers a true double precision solution. In this version of the mixed-precision hybrid algorithm, the nonlinear residual in Newton's method is evaluated in double precision, while the Jacobian-vector product are evaluated in single precision. For the IAW problem, this hybrid implementation requires only about a third of the GPU particle

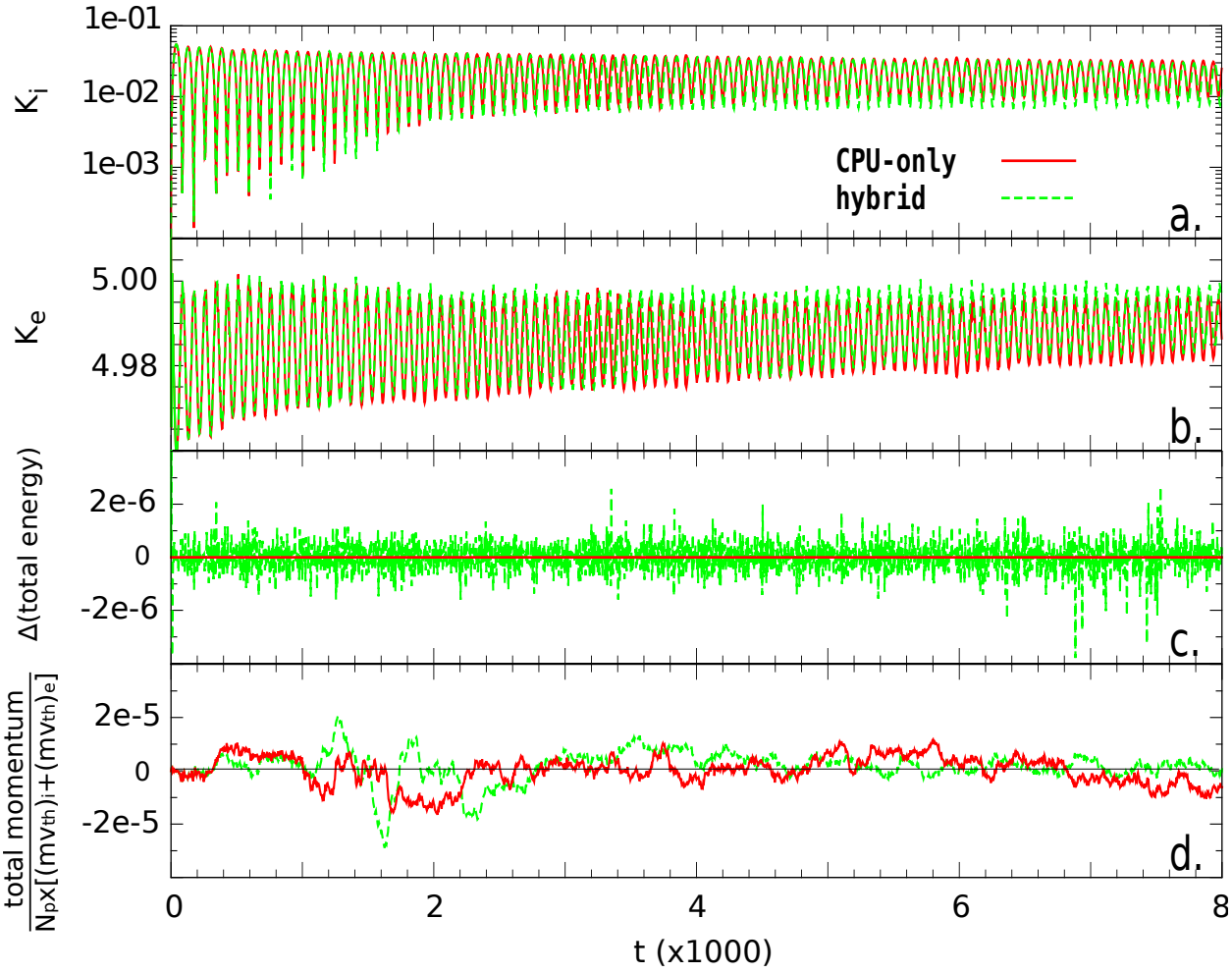

Figure 6: Long-timescale simulation of the IAW problem, comparing simulations with the particle mover on the CPU using double precision and on the GPU using single precision. Panel a,b,c,d are the time history of ion kinetic energy, electron kinetic energy, total energy variation every timestep, and normalized total momentum, respectively.

mover calls to be evaluated in double precision. It is important to note that double-precision computations in our GPU target architecture are expensive, not only due to a factor of 8 (or 2 in some of the newest GPUs) cost increase of double-precision operations, but also because current GPU architectures do not yet feature double-precision versions of the fast intrinsic functions or atomic operations. Nevertheless, our defect-correction implementation of the mixed-precision hybrid algorithm is still able to achieve a speedup of 40, which is 1.6 times better than a full double-precision hybrid CPU-GPU computation.

5. Discussion and Conclusions

This study has explored the hybrid, mixed-precision implementation of a recently proposed fully implicit PIC algorithm [12] on a heterogeneous (CPU-GPU) architecture. The implicit PIC algorithm is ideally suited for a hybrid implementation owing to the concept of particle enslavement, which segregates the particle orbit integration in the nonlinear residual evaluation. Accordingly, the particle mover can be farmed off to a GPU, while the rest of the nonlinear solver machinery remains on the CPU.

With the aid of the roofline model, we have demonstrated that the implicit adaptive, charge-conserving particle mover algorithm of Ref. [12] is compute-bounded, unlike memory-bounded explicit particle movers [5–7, 9]. The optimized parallelization of the particle mover on the GPU significantly boosts the performance compared to both the algorithm's original CPU implementation and a straightforward GPU implementation. The optimized particle mover exploits the powerful floating-point units and fast special-function units on the GPU without loss of accuracy. Significant acceleration has been achieved by eliminating the very-expensive IEEE divisions and square-roots, and by minimizing the creation of divergent branches. We have adopted a novel particle sorting strategy that sorts particles according to both their positions and their velocities to improve memory efficiency and load balancing. The operational throughput of the optimized particle mover algorithm reaches 300-400 GOp/s (for VP and VA, respectively), on the Nvidia GTX 580 GPU in single precision and with typical (large) implicit timesteps, corresponding to 50-70% intrinsic efficiencies (vs. the maximum algorithmic throughput) and 20-25% overall efficiencies (vs. peak throughput).

Moment accumulations from particles are often quoted as a main bottleneck in many previous explicit particle mover algorithms [7, 8, 10]. In contrast, the implicit mover algorithm spends most of the time pushing particles. This is true even for the VA approach, which requires memory accumulations at every sub-timestep. We find that the atomic accumulations in the VA approach take about 30% run time of the mover algorithm. If the VP approach is adopted instead, the accumulation overhead becomes almost negligible. For the other parts of the mover algorithm, we find that the time spent in particle cell-boundary-stopping (needed for exact local charge conservation) is comparable to that in the sub-timestep estimate and the Crank-Nicolson mover.

Significant speedup of the whole implicit PIC algorithm is achieved by the hybrid, mixed-precision CPU-GPU implementation (using single precision in the GPU and double precision in the CPU) vs. a CPU-only one (using double-precision) on a challenging multiscale test problem, the ion acoustic wave. Speedups about 100 are found, which are consistent with Amdahl's law. Careful comparison of relevant quantities (local charge, energy, momentum, and the electron/ion kinetic energy) shows very good quantitative agreement. Particularly

encouraging is the fact that errors in momentum conservation seem unaffected by the mixed-precision character of the hybrid implementation. Similarly, JFNK performance seems unaffected by the use of single precision particle computations on the GPU, as long as the nonlinear tolerance is adjusted accordingly. Overall, the mixed-precision hybrid implementation is found to provide a robust enough algorithm for this simulation.

A mixed-precision CPU-GPU implementation that delivers a true double-precision simulation capability has also been implemented using a defect-correction approach. The speedup (vs. the CPU-only double precision serial simulation) is about 40. This outperforms a full double precision CPU-GPU implementation, which results in a speedup of 25.

This study demonstrates at a proof-of-principle level that hybrid CPU-GPU implementations hold much promise for future investigation in the context of fully implicit PIC algorithms. Future work should focus on extending the approach to multiple dimensions and larger domains. With current limitations in GPU shared memory (a Fermi GPU currently has at most 48KB of shared memory per SM), larger problem sizes will require the use of domain decomposition among GPU nodes. Given that the particle orbit integrator in our implicit PIC algorithm may sample a relatively large fraction of the domain, special attention will need to be paid to devise strategies to minimize communication overhead. Solutions to these issues will be explored in future work.

Acknowledgments

The authors would like to thank Dr. David L. Green for his help with the Fortran-C/C++ mixed language implementation. This work has been funded by the Oak Ridge National Laboratory (ORNL) Directed Research and Development program (LDRD). ORNL is operated by UT-Battelle for the US Department of Energy under contract DE-AC05-00OR22725.

References

[1] C. Birdsall, A. Langdon, Plasma Physics via Computer Simulation, McGraw-Hill, New York, 2005. 1

[2] P. Liewer, V. Decyk, A general concurrent algorithm for plasma particle-in-cell simulation codes, Journal of Computational Physics 85 (2) (1989) 302–322. 1

[3] K. Bowers, B. Albright, L. Yin, W. Daughton, V. Roytershteyn, B. Bergen, T. Kwan, Advances in petascale kinetic plasma simulation with VPIC and roadrunner, in: Journal of Physics: Conference Series, Vol. 180, IOP Publishing, 2009, p. 012055. 1

[4] S. Fuller, L. Millett, The Future of Computing Performance: Game Over Or Next Level?, National Academies Press, 2011. 1

[5] V. Decyk, T. Singh, Adaptable particle-in-cell algorithms for graphical processing units, Computer Physics Communications. 1, 3.6, 5

[6] X. Kong, M. Huang, C. Ren, V. Decyk, Particle-in-cell simulations with charge-conserving current deposition on graphic processing units, Journal of Computational Physics. 1, 3.6

[7] H. Burau, R. Widera, W. Hönig, G. Juckeland, A. Debus, T. Kluge, U. Schramm, T. Cowan, R. Sauerbrey, M. Bussmann, PIConGPU: A fully relativistic particle-in-cell code for a GPU cluster, Plasma Science, IEEE Transactions on 38 (10) (2010) 2831–2839. 1, 3, 5

[8] K. Madduri, S. Williams, S. Ethier, L. Oliker, J. Shalf, E. Strohmaier, K. Yelicky, Memory-efficient optimization of gyrokinetic particle-to-grid interpolation for multicore processors, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, ACM, 2009, p. 48. 1, 3, 5

[9] K. Madduri, E. Im, K. Ibrahim, S. Williams, S. Ethier, L. Oliker, Gyrokinetic particle-in-cell optimization on emerging multi-and manycore platforms, Parallel Computing. 1, 5

[10] G. Stantchev, W. Dorland, N. Gumerov, Fast parallel particle-to-grid interpolation for plasma PIC simulations on the GPU, Journal of Parallel and Distributed Computing 68 (10) (2008) 1339–1349. 1, 3, 3.6, 5

[11] K. Madduri, K. Ibrahim, S. Williams, E. Im, S. Ethier, J. Shalf, L. Oliker, Gyrokinetic toroidal simulations on leading multi-and manycore hpc systems, SC11. 1

[12] G. Chen, L. Chacón, D. Barnes, An energy- and charge-conserving, implicit, electrostatic particle-in-cell algorithm, J. Comput. Phys. 230 (18) (2011) 7018 – 7036. 1, 2.3, 3, 3.1, 4, 4.1, 4.3, 5

[13] S. Williams, A. Waterman, D. Patterson, Roofline: an insightful visual performance model for multicore architectures, Commun. ACM 52 (4) (2009) 65–76. 1, 2.1

[14] S. Ryoo, C. Rodrigues, S. Baghsorkhi, S. Stone, D. Kirk, W. Hwu, Optimization principles and application performance evaluation of a multithreaded gpu using cuda, in: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, ACM, 2008, pp. 73–82. 1

[15] W. Hwu, C. Rodrigues, S. Ryoo, J. Stratton, Compute unified device architecture application suitability, Computing in Science & Engineering 11 (3) (2009) 16–26.

[16] D. Kirk, W. Hwu, Programming massively parallel processors: A Hands-on approach, Morgan Kaufmann, 2010.

[17] Nvidia, CUDA C programming guide version 4.0, (2011). 2.3, 3.3, 3.7

[18] Nvidia, CUDA C best practices guide version 4.0, (2011). 1, 3, 3.4, 4.1

[19] J. Nickolls, W. Dally, The GPU computing era, Micro, IEEE 30 (2) (2010) 56–69. 2.1

[20] W. Wulf, S. McKee, Hitting the memory wall: Implications of the obvious, Comput. Arch. News 23 (1995) 20–20. 2.2

[21] C. Wittenbrink, E. Kilgariff, A. Prabhu, Fermi GF100 GPU architecture, IEEE Micro 31 (2) (2011) 50–59. 2.3

[22] H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, A. Moshovos, Demystifying gpu microarchitecture through microbenchmarking, in: Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on, IEEE, 2010, pp. 235–246. 2.3

[23] V. Volkov, J. Demmel, Benchmarking gpus to tune dense linear algebra, in: High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for, IEEE, 2008, pp. 1–11. 2.3

[24] L. Shampine, Error estimation and control for ODEs, J. Sci. Comput. 25 (1) (2005) 3–16. 3.1

[25] J. Hennessy, D. Patterson, Computer organization and design, 4th Edition, Morgan Kaufmann, 2008. 3.2

[26] P. Markstein, IA-64 and elementary functions: speed and precision, Prentice Hall, 2000. 3.2

[27] A. Buttari, J. Dongarra, J. LANGOU, J. LUSZCZEK, S. TOMOV, Exploiting mixed precision floating point hardware in scientific computations. 3.3

[28] R. Shams, R. Kennedy, Efficient histogram algorithms for NVIDIA CUDA compatible devices, in: International Conference on Signal Processing and Communication Systems, 2007. 3.4

[29] K. Bowers, Accelerating a particle-in-cell simulation using a hybrid counting sort, J. Comput. Phys. 173 (2) (2001) 393–411. 3.6

[30] M. Harris, S. Sengupta, J. Owens, Parallel prefix sum (scan) with CUDA, GPU Gems 3 (39) (2007) 851–876. 3.6

[31] J. Little, A proof of the queuing formula L= λW, Operations Research 9 (3) (1961) 383–387. 4.1

[32] K. Böhmer, H. J. Stetter (Eds.), Defect Correction Methods – Theory and Applications, Springer-Verlag, Wien, New York, 1984. 4.3