Plasma Physics and
Controlled Fusion

PURPOSE-LED
PUBLISHING™

PAPER • **OPEN ACCESS**

A generalized parallelization algorithm for particle-in-cell simulations

To cite this article: Ricardo A Fonseca 2026 *Plasma Phys. Control. Fusion* **68** 085054

View the article online for updates and enhancements.

You may also like

- HiPACE: a quasi-static particle-in-cell code
T Mehrling, C Benedetti, C B Schroeder et al.
- Synergia CUDA: GPU-accelerated accelerator modeling package
Q Lu and J Amundson
- Toward the modeling of chains of plasma accelerator stages with WarpX
J.-L. Vay, A. Almgren, L. D. Amorim et al.

This content was downloaded from IP address 193.136.189.182 on 02/09/2026 at 11:12

Plasma Physics and Controlled Fusion

Check for updates

PAPER

OPEN ACCESS

RECEIVED
27 December 2025

REVISED
10 August 2026

ACCEPTED FOR PUBLICATION
20 August 2026

PUBLISHED
1 September 2026

Original content from this work may be used under the terms of the Creative Commons Attribution 4.0 licence.

Any further distribution of this work must maintain attribution to the author(s) and the title of the work, journal citation and DOI.

A generalized parallelization algorithm for particle-in-cell simulations

Ricardo A Fonseca*

DCTI—Dep. Ciências e Tecnologias da Informação, ISCTE-IUL, Lisboa, Portugal
* Author to whom any correspondence should be addressed.

E-mail: ricardo.fonseca@iscte-iul.pt

Keywords: plasma simulation, particle-in-cell codes, graphics processor units

Abstract

Particle-in-cell (PIC) codes are a cornerstone of plasma-based accelerator research. We present a generalized parallelization algorithm for PIC simulations that achieves high efficiency across modern high-performance computing (HPC) architectures, including CPUs (x86 and Arm) and graphics processing units (GPUs) (NVIDIA, AMD). The algorithm employs a micro-spatial domain decomposition combined with a high-performance particle manager for efficient particle migration between domains. Each domain is mapped to a CPU thread or GPU thread block, ensuring balanced parallel workloads even in realistic scenarios. Implementations use architecture-specific programming models—OpenMP for CPUs and CUDA/HIP for GPUs—while maintaining a common algorithmic structure, making porting straightforward. Benchmark results for a representative 2D problem demonstrate excellent performance on both single devices and large-scale HPC systems. The algorithm is also show to be extendable for other PIC variants and architectures.

1. Introduction

Building on the pioneering work of Buneman and Dawson in the early 1960 s [1, 2], particle-in-cell (PIC) codes [3–5] have been a fundamental tool in plasma-based accelerator development since its inception [6]. These codes (e.g. VLPL [7], OSIRIS [8], CALDER [9], and VORPAL [10]) operate at the most fundamental, microscopic level, making few physical approximations and are ideally suited to these problems. However, this also makes them among the most computationally expensive models in plasma physics, requiring efficient use of large-scale high-performance computing (HPC) systems [11, 12] and exploiting multiple levels of parallelism available on these systems [13, 14].

Since the early 2000 s, graphics processing units (GPUs) have emerged as an alternative to traditional CPU-based computing [15], offering increased throughput and energy efficiency [16, 17]. The current ecosystem of scientific computing systems [18] still relies on diverse hardware approaches and vendors, each with specific programming models, memory architectures, and processor types. Nevertheless, GPU/accelerator-based systems now dominate the Top500 list, with 255 systems in the November 2025 edition.

As GPU-based systems gradually became the prevailing HPC architecture, the plasma simulation community began rethinking PIC algorithm implementations for this platform. Early efforts demonstrated the feasibility of electromagnetic (EM) PIC on GPUs and explored the use of CUDA-based particle push, field interpolation, and current deposition kernels [19, 20]. Subsequent work addressed key algorithmic bottlenecks in GPU-based PIC simulations, including particle reordering, charge and current deposition, memory coalescing, and load balancing [21–23]. More complete GPU-oriented implementations and performance studies subsequently demonstrated the potential of accelerators for large-scale plasma simulations [24–26], and subsequent work extended these developments toward production-oriented and multi-architecture environments through optimized CUDA implementations and portability strategies for modern accelerator-based systems [27, 28]. However, the irregular memory-access patterns inherent to the PIC algorithm, together with the limited capabilities of early GPU devices, constrained

© 2026 The Author(s). Published by IOP Publishing Ltd

the performance and scope of these first implementations. Continued community efforts, combined with evolving GPU capabilities, have since led to the deployment of user-oriented codes targeting modern GPU hardware, including PIConGPU [29], Smilei [30], WarpX [31–33], HiPACE++ [34], VPIC [35], and OSIRIS [36, 37].

Despite the rise of GPUs, CPU architectures remain critically important, and the community continues to pursue platform-agnostic solutions that retain high performance across different architectures. This is not straightforward because implementations optimized for NVIDIA GPUs commonly rely on CUDA and on architecture-specific optimization techniques (e.g. [37]). Some codes pursue portability through programming models such as OpenMP, as used by Smilei [30], or SYCL [31]. Others employ libraries that provide hardware-portability layers, including AMReX in WarpX [31] and Alpaka in PIConGPU [29]. Although these approaches facilitate portability across CPU and GPU architectures, abstraction layers often fail to fully exploit hardware-specific features. An alternative is to use native programming models that directly expose the characteristics of each target architecture while retaining a common underlying algorithm. Such an approach avoids the need to develop two fundamentally different parallelization strategies while preserving as much portability as possible across architecture-specific implementations without sacrificing performance.

In this paper, we present a generalized parallelization strategy for the PIC algorithm that delivers high performance on both CPU and GPU architectures by exploiting their similarities. The implementation uses architecture-specific backends with native programming models, while keeping the core algorithm identical, making porting straightforward. Section 2 provides an overview of the two prevailing compute architectures. Section 3 discusses PIC algorithm bottlenecks and introduces the proposed parallelization strategy. Section 4 details implementation aspects for both architectures, and section 5 extends the algorithm to multi-device setups for large-scale HPC systems. Section 6 presents benchmarks across multiple architectures, hardware configurations, problem sizes, and scaling scenarios. Section 7 explores extensions to other PIC variants, such as quasi-3D geometry, and additional hardware platforms. Finally, section 8 summarizes the main results and outlines future work.

2. Modern HPC hardware

Current HPC systems [18] primarily rely on two architectures: multicore/manycore CPUs and general-purpose GPUs, as illustrated in figure 1. While these architectures historically evolved independently, recent iterations exhibit notable similarities in parallelism and memory hierarchy at the socket level.

Modern CPUs, such as Intel's Sapphire Rapids family, feature up to ~ 64 cores per socket, with two hardware threads per core. These cores provide strong scalar performance and include single-instruction multiple-data (SIMD) units up to 512 bits wide for vector processing. CPUs employ a multi-layer memory hierarchy, with large per-core caches and main memory banks using DDR5 RAM with capacities on the order of ~ 100 GB, complemented by smaller high-bandwidth memory (HBM) banks. This design offers good performance even for non-sequential memory access through the cache system.

Conversely, modern GPUs, such as NVIDIA's Ampere architecture and successors, include up to ~ 108 streaming multiprocessors (SMs), supporting up to 2048 threads per SM. Threads execute in groups of 32, called warps, in a SIMD fashion, managed by a highly efficient thread scheduler capable of rapid context switching. The GPU memory system also uses a hierarchical design, with explicit control over shared memory within each SM and large HBM pools (on the order of ~ 100 GB) delivering extremely high bandwidth for coalesced (sequential) memory access.

Both architectures achieve performance through massive parallelism and hierarchical memory systems: a large, slower memory pool (main memory on CPUs, device memory on GPUs) complemented by smaller, faster memory blocks (CPU caches, GPU caches/shared memory). These commonalities can be leveraged to design a parallelization strategy for PIC codes that runs efficiently on both platforms. For optimal performance, our implementation uses native programming models, namely OpenMP for CPUs and CUDA/HIP for GPUs, adding additional specialization when needed.

3. Generalized parallelization for the PIC algorithm

The PIC algorithm [3–5] implements a particle-mesh approach for modeling kinetic plasma scenarios by tracking charged particles that interact through EM fields discretized on a grid. The EM-PIC algorithm consists of four main steps:

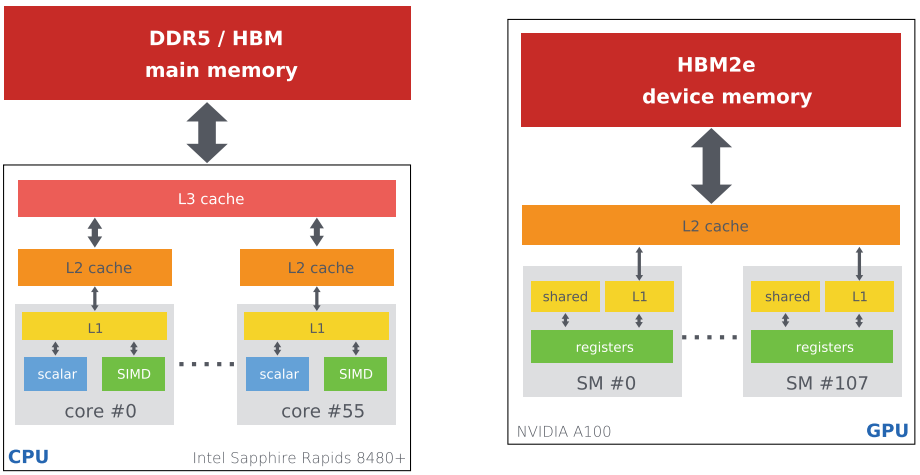


Figure 1. Comparison of modern CPU (Intel Sapphire Rapids 8480+, left) and GPU (NVIDIA A100, right) architectures showing the memory hierarchy and processing units. The solid line marks the device boundary.

1. Interpolating EM fields at particle positions;
2. Advancing particle velocities and positions using these fields;
3. Depositing self-consistent electric current on the simulation grid based on particle motion;
4. Advancing EM fields using the deposited current.

The most computationally demanding components are particle advance and current deposition (steps 1–3), which account for the bulk of the calculations and scale linearly with the number of particles. Fortunately, operations on individual particles are essentially independent, enabling parallel processing and efficient use of the floating-point capabilities of modern architectures. However, achieving the required level of performance on a single CPU or GPU requires keeping all processing units fully utilized while ensuring efficient access to particle and field data, making memory access the dominant challenge.

The primary limitation on effective memory bandwidth in PIC algorithms arises from irregular memory access patterns during field interpolation and current deposition, which depend on particle positions as the simulation evolves. Additionally, current deposition can lead to memory collisions when multiple particles processed in parallel attempt to update the same grid cell, reducing parallel efficiency.

To address these challenges, we propose a micro-spatial domain decomposition strategy, inspired by approaches originally developed for distributed memory systems [11]. Here, we describe a 2D implementation, which can be extended to higher dimensions. The simulation domain is divided into small spatial regions of approximately 20^2 cells, referred to as tiles. To simplify the algorithm and improve efficiency, all tiles in the simulation have the same size. Currently, the algorithm requires the global simulation domain to be rectangular; however, it can be straightforwardly extended to support tile distributions with irregular boundaries, for example, to approximate a circular domain.

The goal of this strategy is to minimize irregular memory access to main/device memory. Before interpolation, tile EM fields are copied from main memory to faster local memory using sequential (coalesced) access. Interpolation and current deposition are then performed in this fast memory, minimizing the overhead of irregular access. After computation, updated data is copied back to main memory using sequential access. Each tile operates independently, relying only on local data, allowing straightforward parallel processing across tiles. Additional parallelism can be exploited within each tile, as particles and cells can also be processed concurrently. Memory collisions during current deposition can be mitigated through serialization or, alternatively, atomic operations, which are typically efficient in local memory.

As with any spatial domain decomposition, after advancing EM fields and depositing current, tile edge values must be synchronized with neighboring tiles. More importantly, after particle advancement, particles may cross tile boundaries and must be reassigned to the appropriate tiles. This migration process, which we refer to as *particle tile sort*, represents the most significant challenge of the algorithm and must also be handled efficiently in parallel.

3.1. Particle tile sort

The particle tile sort reorganizes the global particle buffer so that particles belonging to the same tile occupy contiguous positions in memory, enabling sequential access to particle data for each tile. Many

relevant scenarios exhibit significant particle number imbalance across tiles, with some tiles containing orders of magnitude more particles than others, and this distribution evolves dynamically during the simulation. To handle this efficiently, we maintain a single contiguous particle buffer for all tiles rather than separate buffers per tile. This approach reduces overall memory requirements and avoids frequent checks or resizing of individual buffers. Two auxiliary arrays are used: one stores the number of particles per tile, and the other stores the offset (start position) of each tile within the global particle buffer.

On modern architectures, this sort must be both highly parallel and minimize data movement. Exploiting the fact that particles will at most cross into one of the 8 neighboring tiles in each iteration, the algorithm is divided into four parallelizable steps, as each tile operates independently:

1. **Count particles crossing tile boundaries.** Each tile iterates over its particles, counting those leaving in each direction and those remaining. It stores this information, along with the indices of outgoing particles, in temporary buffers. The new particle count per tile is computed by adding (via atomic operations) the number of particles staying and incoming from neighbors.
2. **Compute new tile offsets.** Using the updated particle counts, a parallel prefix scan determines the new offset for each tile in the global particle buffer, accounting for both local and incoming particles. These will be the final offset values after the sort.
3. **Stage outgoing particles.** Using a temporary particle buffer as a staging area, configured using the new offsets, each tile reserves space for its outgoing particles in the appropriate destination tile using atomic operations. Each tile then iterates over the list of outgoing particles and copies them into the appropriate neighbor's staging area. Since space is preallocated, this copy avoids memory collisions. Holes left in the main buffer are filled with particles from the end of the local list. If the tile offset changes, some particles staying in the tile may need to be shifted; these are copied as a block into the tile's staging area. After this step, all data in the main buffer already reflects the new offsets, leaving space for staged particles at the end of each tile region.
4. **Finalize copy from staging area.** Each tile copies its staged particles into the main buffer using the reserved space. This is a simple sequential memory operation, as the staging area is already organized according to the new tile distribution.

This algorithm minimizes unnecessary data movement by keeping most particles that remain in their tile in the same position within the global buffer. The only exception occurs when a tile's offset changes; in such cases, a small number of particles are shifted to maintain contiguity. For example, if a tile's offset increases by N positions (e.g. due to particle accumulation in lower-index tiles), N particles from the beginning of the tile's list are moved to the end. As shown later, this has negligible impact on performance.

3.2. Particle advance / current deposit

Once particles are organized by tiles, advancing particles and depositing current becomes relatively straightforward. We assign one computational kernel per tile, namely one CPU thread or one GPU thread block per tile. On GPU architectures, we use one thread per particle, while on CPUs, explicit vectorization of the particle advance should be employed for optimal performance, as discussed below. This process consists of two main steps: (i) interpolating EM fields and updating particle velocities, and (ii) moving particles and depositing current.

For the velocity update, the kernel begins by copying the corresponding tile EM fields into fast memory, namely local memory on CPUs and block shared memory on GPUs, effectively creating a software-defined cache for this data. Then, for each particle, we read its position and velocity from global memory, interpolate EM fields from local memory, advance the velocity using the appropriate algorithm, and store the updated velocity back in global memory.

For particle motion and current deposition, we follow a similar approach. The kernel allocates an electric current grid for the tile in fast memory and initializes it to zero. Then, for each particle, we read its position and velocity from global memory, update its position, and accumulate the current corresponding to this motion using a charge-conserving algorithm (e.g. [38]). Given the additional level of parallelism inside the kernel, special care must be taken to avoid memory collisions during current deposition. To address this, we use atomic add operations on GPU architectures (which are generally efficient in block shared memory) or serialize the current deposit for vectorized code on CPUs. Once all particles have been processed, the local tile current grid is added to the global current grid using sequential memory access.

3.3. Field advance

For a finite-difference-based field solver, such as the Yee algorithm [39], we adopt a similar strategy by assigning one computational kernel per tile for efficient field advance calculations. On GPU architectures, we use one thread per cell, while on CPUs, given the simplicity of the algorithm, we rely on compiler auto-vectorization as it typically provides good performance. The main challenge, though less severe than in particle operations, is again the nonlinear memory access pattern, this time introduced by finite-difference operators. To mitigate this, the kernel begins by copying the corresponding tile EM fields into fast memory, performs the field update locally, and then writes the updated fields back to main memory. In our implementation, the electric current is accessed directly from global memory since it follows a sequential pattern, allowing more fast memory to be allocated for the remaining grid data.

For spectral-based field solvers, such as the pseudo-spectral time-domain (PSTD) method [40], the primary computational overhead typically lies in the fast Fourier transform (FFT) routines. In these cases, it is generally preferable to use optimized, high-performance FFT libraries available for the target system. These libraries often require a specific memory layout for optimal performance, so we implement a lightweight computational kernel (again, one per tile) to convert between the tile-based grid and the FFT-compatible layout.

4. Implementation details

Ideally, we would like to maintain a single code base for all architectures to simplify development. However, while the proposed algorithm is similar across platforms, this approach typically incurs a performance penalty. 'One-size-fits-all' programming models enforce hardware abstraction layers that do not always align with our algorithm, and achieving peak performance with these models often requires specialized versions, limiting the benefits of portability. Alternatively, we can achieve optimal performance by using native programming models that explicitly manage hardware details. Specifically, this involves using OpenMP for CPU threading (along with SIMD vector intrinsics) and CUDA/HIP for NVIDIA and AMD GPUs.

Although this results in at least two separate code bases, the shared algorithm significantly simplifies development. Many code blocks remain identical or very similar, with the main differences occurring in how computational kernels described in the previous section are launched. Minor variations in kernel implementations are required, but these are specific to the computational backends. At a higher level, the code exposes the same classes and interfaces to the user, enabling the same problem to run seamlessly on different architectures.

4.1. CPU implementation

Parallelism on CPUs is primarily achieved using OpenMP [41], with kernel launches handled by an OpenMP loop that assigns one thread per tile. Although OpenMP does not provide explicit control over local (fast) thread memory, this behavior can be approximated by copying EM fields into a locally defined buffer. Due to the small tile size, these buffers typically reside in the L1 cache. For particle routines, dynamic or guided OpenMP scheduling is essential to ensure parallel load balance, as different tiles may contain varying numbers of particles and thus exhibit different workloads.

An additional level of parallelism for particle advance and current deposition is introduced through explicit vectorization, following the algorithm developed for the OSIRIS code [14]. This approach leverages the fact that particle computations are identical and essentially independent. The algorithm loads a batch of particles corresponding to the SIMD vector width and processes them in parallel using the SIMD unit. The main challenge here is memory collisions during current deposition, which are mitigated by serializing the final step of this operation. To simplify deployment of vector code across different CPU architectures, we implemented a hardware-agnostic abstraction layer. This layer provides generic vector operations (including gather instructions) for particle advance, which are translated at compile time into architecture-specific intrinsics. Currently, all major SIMD instruction sets are supported, including AVX2 and AVX-512 for x86, and NEON and SVE for Arm.

4.2. GPU implementation

Since GPU accelerators have a separate memory space, we execute the entire algorithm on the GPU to avoid the bottleneck of CPU–GPU communication during normal operation. The only data transfer between host and device occurs when saving simulation results to disk. Computational kernels are launched using the standard GPU thread engine, typically assigning one thread block per tile and using

block shared memory as fast local memory. For problems with a small number of tiles, multiple blocks per tile can be used to improve performance by increasing thread count.

The work presented here focuses mainly on NVIDIA GPUs using CUDA. We assign a large number of threads per block (approximately 1 k) and use one thread per particle or grid cell, depending on the kernel. Since, unlike the CPU code, multiple threads operate on the same tile, explicit synchronization within the kernel is required, which is achieved using block-level synchronization functions. Additionally, during current deposition, atomic operations are employed to prevent memory collisions in shared memory among threads in the same block.

5. Extension to multiple devices

Extending the algorithm to multiple devices introduces an additional parallelization challenge, as the problem must now be split across multiple devices and memory spaces, requiring communication of field and particle data between devices. The most general and portable solution for this purpose is MPI, which is widely available on all HPC platforms. While other models could be used for multiple devices within the same hardware node (as is already the case for the OpenMP CPU implementation), MPI provides a unified approach that works regardless of device location.

To parallelize the algorithm across multiple devices, we introduce an additional level of spatial domain decomposition, assigning one MPI process (or computational element) to each domain. The core algorithm remains unchanged for intra-domain parallelization; the only modifications involve updating information at tile edges, which may now reside on different MPI domains.

For grid data, the implementation is straightforward. We pack data from multiple tiles destined for the same MPI node into a single message buffer and initiate communication with neighboring nodes. We then perform tile edge updates within the domain using standard single-device routines. Once this step is complete, we wait for incoming data and unpack it into the corresponding local tiles.

For particle data, MPI communication is interleaved with the standard particle tile sort steps described in section 3.1. The algorithm proceeds as follows:

1. After counting particles crossing tile boundaries (Step 1), exchange the number of particles leaving MPI domains with neighboring nodes. Message data is organized by tile so that upon unpacking, each edge tile knows how many particles it will receive.
2. Compute new offsets in the particle buffer (Step 2), also accounting for particles incoming from different MPI nodes.
3. When copying particle data to the staging area (Step 3), also handle particles leaving the MPI domain. Once complete, pack outgoing particle data into message buffers (one per MPI neighbor) and initiate communication with all neighbors simultaneously.
4. While particle data messages are being exchanged, proceed with copying particles from staging memory into tiles (Step 4), processing particles that remain within the MPI domain.
5. After communication completes, unpack received particle data into the corresponding tiles.

The CPU and GPU implementations are nearly identical, using kernel-based parallel routines for message packing and unpacking. This is facilitated by CUDA-aware MPI, which allows GPU device memory to be used for message buffers. For both grid and particle data, MPI communication is overlapped with intra-node grid and particle tile updates, significantly reducing communication overhead.

6. Benchmarks

Benchmarks were performed using a simulation of the collision between electron and positron plasma clouds initially moving perpendicularly to the simulation plane, where the physics is dominated by the Weibel instability [42]. This well-studied scenario presents a challenging test case, as it produces cavitation, high-density filaments, and significant particle motion within the simulation plane, leading to uneven particle distribution across the grid, as shown in figure 2. This stresses both the particle tile sort routine and parallel load balancing. At the same time, the scenario scales easily to larger spatial domains, making it ideal for large-scale parallelization tests.

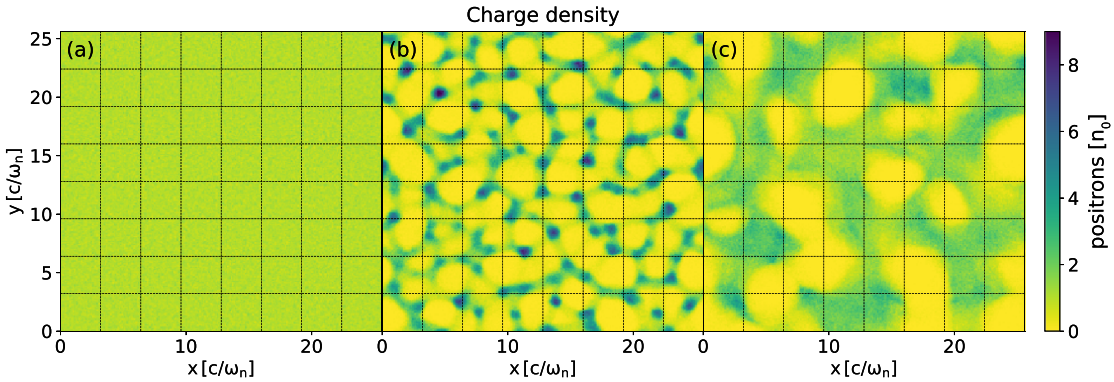


Figure 2. Benchmark simulation. The plot shows the structure of the in-plane charge density for the (a) initial ($t = 3.5/\omega_n$), (b) growth ($t = 18.2/\omega_n$) and (c) saturation ($t = 35/\omega_n$) stages of the Weibel instability. Dashed lines represent the tile boundaries. Only a 256^2 cell subset of the total simulation grid is presented.

Table 1. Single-socket performance.

Device	Programming model	Performance [GPart s^{-1}]
NVIDIA A100 80 GB SMX	CUDA	11.60
NVIDIA A100 80 GB PCIe	CUDA	10.97
NVIDIA A100 40 GB SMX	CUDA	10.78
Intel Xeon Gold 6336Y 24c @ 2.40 GHz	OpenMP / AVX2	0.77
Intel Xeon Gold 6336Y 24c @ 2.40 GHz	OpenMP	0.43
Fujitsu A64FX 48c @ 1.8 GHz	OpenMP / $SV10^{512}$	0.64
Fujitsu A64FX 48c @ 1.8 GHz	OpenMP	0.15
Apple M4 Pro 14c @ 2.59 GHz	OpenMP / NEON	0.88
Apple M4 Pro 14c @ 2.59 GHz	OpenMP	0.53

Regarding the tile size, the dimensions may be chosen to be anywhere from 3^2 to 40^2 cells, at which point the tile size may be too large to fit in the GPU shared memory / CPU L1 cache. Larger tile sizes will lead to fewer particles crossing tile boundaries per time step, which benefits the particle tile sort algorithm by reducing data movement. On the other hand, smaller tile sizes leave more fast memory available for the remaining computations, which may speed up other operations in the code. For this benchmark, the tile size was chosen to be 32^2 cells (see dashed lines in figure 2), which offered a good balance between these two effects. Smaller values, down to $\sim 20^2$ cells, were found to yield similar performance.

The main figure of merit is the total number of particle advances per second, computed by counting all particle updates over a fixed number of iterations and dividing by the total wall-clock time. This metric includes all algorithm steps: particle advance, field solve, and particle tile sort. Table 1 shows performance results for several GPU and CPU architectures using a 2048^2 grid, 8^2 particles per cell per species, and linear interpolation, run for 500 time steps (enough to reach instability saturation). For CPUs, we report both base OpenMP performance (compiler auto-vectorization) and explicit vectorization timings. The algorithm achieves excellent performance across all devices, reaching ~ 11 GPart s^{-1} on GPUs and ~ 0.8 GPart s^{-1} on CPUs with explicit vectorization. The large GPU vs CPU speedup of $\sim 14\times$ is a result of both the superior floating-point and memory bandwidth capabilities of GPUs.

The breakdown in table 2 shows that particle advance routines dominate runtime (87% on GPUs and 96% on CPUs), while the field solver is negligible ($\sim$1% on GPUs and $\sim 0.03\%$ on CPUs). The main difference between architectures lies in the particle tile sort, which accounts for $\sim 12\%$ of GPU time and $\sim 4\%$ of CPU time. While this is mainly due to particle advance being more efficient on GPUs, we can also see in table 2 that there are differences in the sort stages: CPUs handle the 'copy out' step (step 1 in section 3.1 above) better, likely due to cache effects. Overall, the particle tile sort introduces minimal overhead.

Performance depends strongly on problem size (figure 3). Both architectures show similar behavior, with reduced performance for small problems and peak performance for sizes above 1 GB/species, regardless of the number of particles per cell. This reflects the need for large thread counts to maximize floating-point throughput and hide memory latency, as well as the need to offset structural overhead. GPUs are particularly sensitive, requiring thousands of threads for efficiency. For small problems with

Table 2. Breakdown of time spent on each step for (a) total simulation and (b) particle tile sort.

(a)		
Step	GPU	CPU
Move / Deposit	53.38%	59.55%
Velocity Advance	33.81%	36.67%
Particle Tile Sort	11.54%	3.74%
Field Solver	1.09%	0.03%
Initialization	0.18%	0.00%
Other	0.01%	0.00%
(b)		
Step	GPU	CPU
Copy out	41.87%	29.02%
Bnd check	51.97%	70.05%
Copy in	5.90%	0.94%
Update offset	0.26%	0.00%

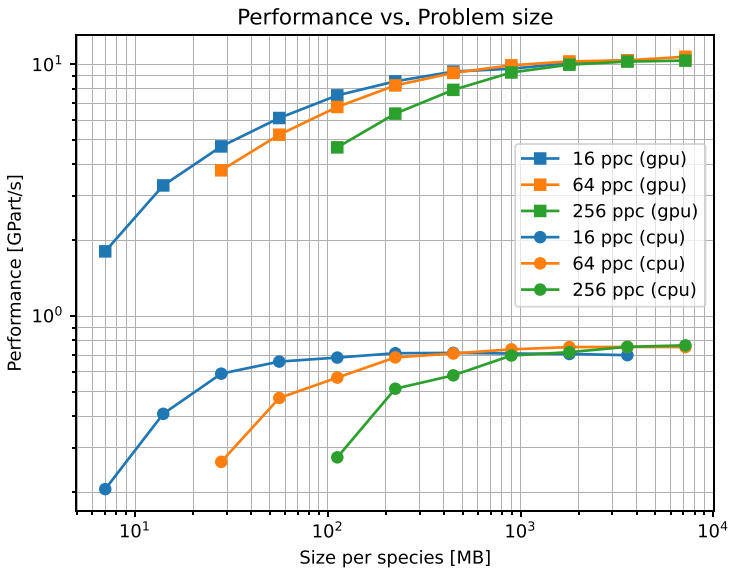


Figure 3. Algorithm performance vs problem size for GPU (NVIDIA A100 80 GB PCIe) and CPU (Intel Xeon Gold 6336Y 24c @2.40GHz). Square markers: GPU; circle markers: CPU. Problem size is measure in memory size per species. Performance measured for 4^2, 8^2, and 16^2 particles per cell.

few tiles, launching multiple thread blocks per tile is required to improve performance, with each block handling a subset of particles and depositing current into separate shared-memory buffers before accumulation in global memory via atomic operations. With this optimization, GPU performance exceeds 10 GPart s^{-1} for all problems above 1 GB/species and remains reasonable even for the smallest test (128^2 grid, 16 particles/cell, 256k particles total, 7 MB), achieving 1.85 GPart s^{-1}.

Explicit vectorization significantly improves CPU performance, particularly on architectures with wide SIMD units such as Fujitsu A64FX. This processor supports both 128-bit vectors via older NEON extensions and 128- to 512-bit vectors through scalable vector extensions (SVE), enabling us to evaluate the impact of different vector widths and instruction sets. Table 3 presents measured performance for various configurations, including compiler auto-vectorization. Using explicit SIMD intrinsics yields substantial gains, with speedup scaling with vector width and reaching approximately 4.3× over the baseline. However, overall efficiency on CPUs remains far below that of GPUs: although the peak floating-point performance of this CPU is roughly one-quarter that of the tested GPU, the observed performance is about 17× lower. This gap is primarily due to costly vector gather operations and serialized current deposition. Without efficient gather capabilities or fast atomic operations (features available on GPUs) increasing SIMD width further provides diminishing returns.

Table 3. Performance on Fujitsu A64FX vs SIMD vector width. 'Auto' refers to compiler auto-vectorization (up to 512 bits). Relative speedup is computed from 'Auto.'.

Vectorization	Performance [GPart s^{-1}]	Relative speedup
Auto	0.154	1.00×
NEON 128 bits	0.295	1.92×
SVE 128 bits	0.390	2.54×
SVE 256 bits	0.498	3.24×
SVE 512 bits	0.653	4.25×

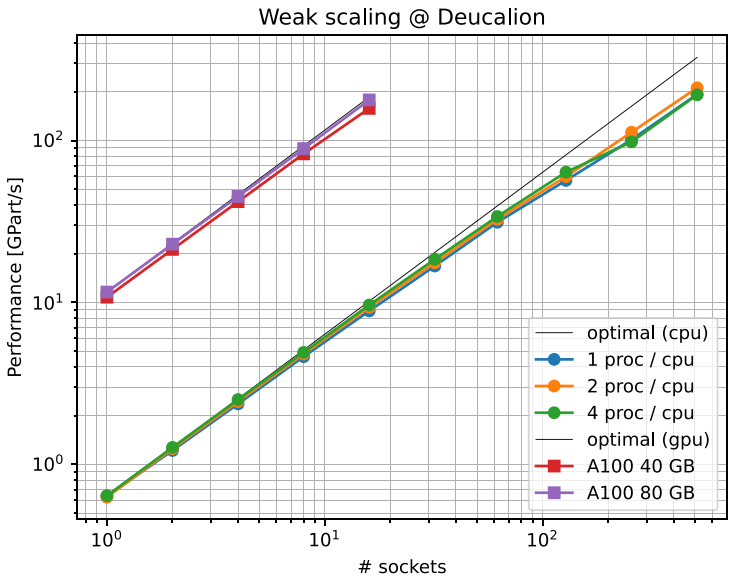


Figure 4. Parallel weak scaling on Deucalion for CPU and GPU partitions. GPU results shown for both models; CPU results include 1, 2, and 4 processes per CPU. Solid lines indicate ideal scaling.

For larger problems, multiple devices are required. As discussed earlier, this is achieved using MPI. Figure 4 shows weak scaling results on the Deucalion system, which includes CPU nodes (Fujitsu A64FX) and two types of GPU nodes (NVIDIA A100 SMX, 80 GB and 40 GB models). The CPU nodes feature a non-uniform memory access (NUMA) layout with four memory regions, each linked to 12 cores. Unlike the pure OpenMP version, MPI allows multiple processes per CPU bound to specific NUMA nodes. The CPU tests include configurations with 1, 2, and 4 MPI processes per CPU (48, 24, and 12 threads per process, respectively). The scaling tests start from the same problem used for the single-device benchmarks and scale up to the maximum number of devices available on the system, keeping the problem size per device constant.

CPU tests with different numbers of processes per node showed minimal differences. For single CPUs, using four processes per node provided negligible gains (<1%) due to intra-node communication overhead. For multi-CPU runs, increasing the number of processes per node raises network traffic, offsetting any benefits. Scaling efficiency from one CPU remains above 80% up to 64 CPUs (∼ 3k cores) and drops to 66% at 512 CPUs (∼24.5k cores), likely due to network topology requiring multiple switch traversals.

GPU tests showed slightly better performance (∼10%) for the 80 GB models compared to the 40 GB models, consistent with single-device benchmarks and attributable to higher memory bandwidth. Scaling efficiency reaches 96% up to 16 GPUs. Each node hosts four GPUs, minimizing network impact. CUDA-aware MPI ensures efficient intra-node communication at the level of NVIDIA GPUDirect.

7. Extension to other PIC algorithms and architectures

The generalized parallelization algorithm presented here can also be applied to other PIC variants, including those using spectral field solvers for improved dispersion properties, such as the PSTD solver. One variant of particular relevance to plasma-based accelerator modeling is the so-called 'quasi-3D'

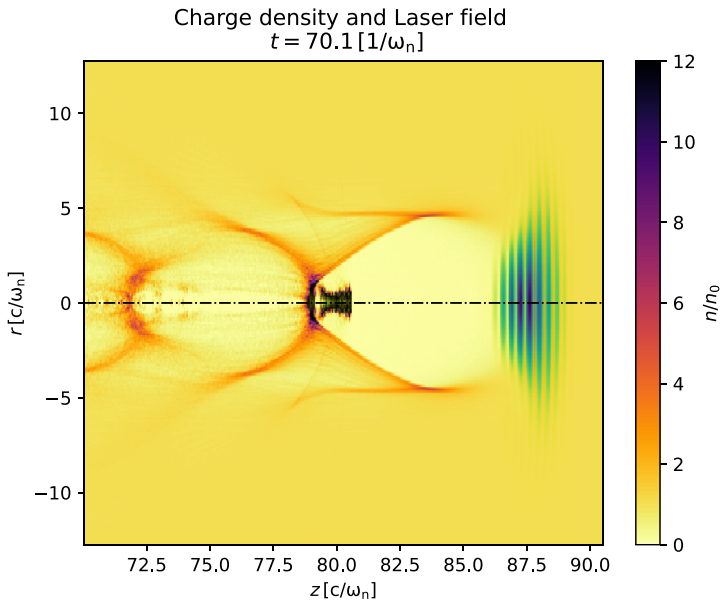


Figure 5. Simulation of a laser wakefield acceleration scenario using a cylindrical (z, r) geometry with azimuthal mode expansion. The plot overlays the radial laser field (mode $m = 1$, green) on top of the charge density (mode $m = 0$, yellow). The bottom half of the plot is a mirror image of the top half, separated by a dash-dot line.

geometry, which employs a cylindrical (z, r) grid combined with an azimuthal mode expansion [43, 44]. Figure 5 shows an example of a laser wakefield accelerator simulation modeled using this geometry with two azimuthal modes (the minimum required for laser driven scenarios) combined with the parallelization algorithm described in this work. The simulation was executed on a single NVIDIA A100 GPU, achieving a performance of approximately 3.7 GPart s^{-1}. Further details of this implementation will be discussed elsewhere.

The algorithm has also been tested on AMD GPUs using HIP. Given the similarities between AMD and NVIDIA architectures and programming models, we implemented a lightweight compatibility layer that maps general GPU-specific instructions to the appropriate backend, abstracting hardware differences and enabling a unified code base for both CUDA and HIP. Additionally, we developed a version based on SYCL for Intel GPU hardware. However, at the time of writing, we did not have access to Intel GPUs for testing and were only able to validate the SYCL implementation on NVIDIA boards, where performance was approximately 22% lower than the native CUDA version.

8. Overview and future work

Modern HPC systems are dominated by two main computing architectures—CPUs and GPUs—which, despite their distinct histories, now share key characteristics such as massive parallelism and hierarchical memory systems with multiple levels of speed and capacity. In this paper, we demonstrated that a single generalized parallelization algorithm can achieve high efficiency on both architectures, delivering top-level performance. Achieving this required implementing the algorithm using native programming models tailored to each architecture, but porting between platforms proved straightforward. Furthermore, the algorithm scales efficiently across multiple devices, whether within a single node or across distributed nodes, enabling effective use of large-scale HPC systems.

Ongoing work focuses on developing Python wrappers to enable interactive use and zero-copy integration into broader workflows, including AI-driven optimization studies. We are also extending this platform to support additional PIC variants, particularly reduced models such as QuickPIC [45] and the ponderomotive guiding center approximation [46]. Finally, we plan to explore alternative programming models, such as Kokkos, to further enhance portability and maintain high performance across diverse architectures.

Acknowledgment

The author would like to thank Dr Viktor Decyk at UCLA for many helpful discussions.

Data availability statement

The data that support the findings of this study are openly available at the following URL/DOI: https://github.com/ricardo-fonseca/zpic-parallel [47]. The source code is published under an AGPL 3.0 license.

Funding

This work was partially supported by the EuPRAXIA preparatory phase project, EU Horizon Europe research and innovation programme Grant Agreement No 101079773. Code development and testing was done primarily using the ISCTE Knowledge and Innovation computing center resources co-funded by LISBOA-01-0246-FEDER-000022. Parallel scaling and Arm performance tests were done using the Deucalion supercomputer through Grant No. 2024.00003.TEST.DEUCALION. AMD GPU development and testing was performed at the Lumi supercomputer through Grant No. EHPC-DEV-2023D07-006.

Author contribution

Ricardo A Fonseca 0000-0001-6342-6226
Conceptualization (equal), Funding acquisition (equal), Investigation (equal), Methodology (equal), Project administration (equal), Software (equal), Validation (equal), Visualization (equal), Writing – original draft (equal), Writing – review & editing (equal)

References

[1] Buneman O 1959 *Phys. Rev.* **115** 503–17
[2] Dawson J 1962 *Phys. Fluids* **5** 445–59
[3] Hockney R W and Eastwood J W 1981 *Computer Simulation Using Particles* (McGraw-Hill)
[4] Dawson J M 1983 *Rev. Mod. Phys.* **55** 403–47
[5] Birdsall C K and Langdon A B 1991 *Plasma Physics via Computer Simulation The Adam Hilger Series on Plasma Physics* (IOP Publishing (Adam Hilger))
[6] Tajima T and Dawson J 1979 *Phys. Rev. Lett.* **43** 267–70
[7] Pukhov A 1999 *J. Plasma Phys.* **61** 425–33
[8] Fonseca R *et al* 2002 *Lect. Notes Comput. Sci.* **2331 LNCS** 342–51
[9] Lefebvre E *et al* 2003 *Nucl. Fusion* **43** 629
[10] Nieter C and Cary J R 2004 *J. Comput. Phys.* **196** 448–73
[11] Liewer P C and Decyk V K 1989 *J. Comput. Phys.* **85** 302–22
[12] Fonseca R, Martins S, Silva L, Tonge J, Tsung F and Mori W 2008 *Plasma Phys. Control. Fusion* **50** 124034
[13] Bowers K J, Albright B J, Yin L, Bergen B and Kwan T J T 2008 *Phys. Plasmas* **15** 055703
[14] Fonseca R, Vieira J, Fiuza F, Davidson A, Tsung F, Mori W and Silva L 2013 *Plasma Phys. Control. Fusion* **55** 124011
[15] Owens J D, Houston M, Luebke D, Green S, Stone J E and Phillips J C 2008 *Proc. IEEE* **96** 879–99
[16] Huang S, Xiao S and Feng W 2009 On the energy efficiency of graphics processing units for scientific computing *IPDPS 2009 - Proc. 2009 IEEE Int. Parallel and Distributed Processing Symp.*
[17] Brodtkorb A R, Hagen T R and Saetra M L 2013 *J. Parallel Distrib. Comput.* **73** 4–13
[18] 2011 Top 500 supercomputer sites (available at: www.top500.org/lists/top500/2025/11/)
[19] Stantchev G, Dorland W and Gumerov N 2008 *J. Parallel Distrib. Comput.* **68** 1339–49
[20] Burau H, Widera R, Hönig W, Juckeland G, Debus A, Kluge T, Schramm U, Cowan T E, Sauerbrey R and Bussmann M 2010 *IEEE Trans. Plasma Sci.* **38** 2831–9
[21] Abreu P, Fonseca R A, Pereira J M and Silva L O 2011 *IEEE Trans. Plasma Sci.* **39** 675–85
[22] Decyk V K and Singh T V 2011 *Comput. Phys. Commun.* **182** 641–8
[23] Joseph R G, Ravunnikutty G, Ranka S, D'Azevedo E and Klasky S 2011 Efficient gpu implementation for particle in cell algorithm *Proc. - 25th IEEE Int. Parallel and Distributed Processing Symp., IPDPS 2011* pp 395–406
[24] Kong X, Huang M C, Ren C and Decyk V K 2011 *J. Comput. Phys.* **230** 1676–85
[25] Bastrakov S, Donchenko R, Gonoskov A, Efimenko E, Malyshev A, Meyerova I and Surmin I 2012 *J. Comput. Sci.* **3** 474–9
[26] Chen G, Chacón L and Barnes D 2012 *J. Comput. Phys.* **231** 5374–88
[27] Rossi F, Londrillo P, Sgattoni A, Sinigardi S and Turchetti G 2012 Towards robust algorithms for current deposition and dynamic load-balancing in a GPU particle in cell code *AIP Conf. Proc.* **1507** 184–92
[28] Decyk V K and Singh T V 2014 *Comput. Phys. Commun.* **185** 708–19
[29] Zenker E, Widera R, Huebl A, Juckeland G, Knüpfer A, Nagel W E and Bussmann M 2016 *Lect. Notes Comput. Sci.* **9945 LNCS** 293–301

[30] Derouillat J *et al* 2018 *Comput. Phys. Commun.* **222** 351–73
[31] Myers A *et al* 2021 *Parallel Comput.* **108** 102833
[32] Vay J L *et al* 2021 *Phys. Plasmas* **28** 023105
[33] Fedeli L *et al* 2022 Pushing the frontier in the design of laser-based electron accelerators with groundbreaking mesh-refined particle-in-cell simulations on exascale-class supercomputers *Int. Conf. for High Performance Computing, Networking, Storage and Analysis, SC*
[34] Diederichs S, Benedetti C, Huebl A, Lehe R, Myers A, Sinn A, Vay J L, Zhang W and Thévenet M 2022 *Comput. Phys. Commun.* **278** 108421
[35] Bird R, Tan N, Luedtke S V, Harrell S L, Taufer M and Albright B 2022 *IEEE Trans. Parallel Distrib. Syst.* **33** 952–63
[36] Tableman A R 2019 Kinetic plasma simulation: meeting the demands of increased complexity *Phd Thesis* University of California (available at: https://escholarship.org/uc/item/98q8g2m6)
[37] Lee R P, Pierce J R, Miller K G, Almanza M, tableman A, Decyk V K, Fonseca R A, Alves E P and Mori W B 2025 *J. Plasma Phys.* **91** E8
[38] Villasenor J and Buneman O 1992 *Comput. Phys. Commun.* **69** 306–16
[39] Yee K 1966 *IEEE Trans. Antennas Propag.* **14** 302–7
[40] Liu Q H 1997 *Microw. Opt. Technol. Lett.* **15** 158–65
[41] Chandra R, Dagum L, Kohr D, Menon R, Maydan D and McDonald J 2001 *Parallel Programming in Openmp* (Morgan kaufmann)
[42] Fonseca R A, Silva L O, Tonge J W, Mori W B and Dawson J M 2003 *Phys. Plasmas* **10** 1979–84
[43] Lifschitz A, Davoine X, Lefebvre E, Faure J, Rechatin C and Malka V 2009 *J. Comput. Phys.* **228** 1803–14
[44] Davidson A, tableman A, An W, Tsung F, Lu W, Vieira J, Fonseca R, Silva L and Mori W 2015 *J. Comput. Phys.* **281** 1063–77
[45] Huang C, Decyk V, Ren C, Zhou M, Lu W, Mori W, Cooley J, Antonsen T and Katsouleas T 2006 *J. Comput. Phys.* **217** 658–79
[46] Silva T, Helm A, Vieira J, Fonseca R and Silva L O 2019 *Plasma Phys. Control. Fusion* **62** 024001
[47] Fonseca R A 2026 ZPIC Particle-in-cell framework (available at: https://github.com/ricardo-fonseca/zpic-parallel)