Bridging Performance Portability and Scalability for Plasma Simulations on Heterogeneous Systems

Nigel Tan
Los Alamos Nat. Lab
Los Alamos, NM, USA

Scott V. Luedtke
Los Alamos Nat. Lab
Los Alamos, NM, USA

Michela Taufer
U. of Tennessee, Knoxville
Knoxville, TN, USA

Brian Albright
Los Alamos Nat. Lab
Los Alamos, NM, USA

Abstract

Bridging portability and scalability is essential for HPC applications. The Vector Particle-In-Cell (VPIC) code, widely used in plasma physics simulations, historically required extensive platform-specific optimizations to achieve high performance. VPIC 2.0 addresses this challenge by adopting Kokkos for performance portability, enabling it to scale effectively across diverse architectures, including CPUs and GPUs. However, the abstractions introduced by Kokkos can obscure hardware-specific capabilities and introduce performance overhead. In this work, we mitigate these overheads by enhancing vectorization and optimizing memory access patterns through platform-targeted particle sorting in VPIC 2.0. These optimizations enable VPIC 2.0 to match the performance of the highly tuned, hardware-specific VPIC 1.2 on CPUs and to achieve superlinear scaling on GPUs.

CCS Concepts

• **Computing methodologies** → **Massively parallel and high-performance simulations**; *Massively parallel algorithms*; • **Applied computing** → **Physics**.

Keywords

performance-portability, sorting, gather-scatter, vectorization, scalability

ACM Reference Format:
Nigel Tan, Scott V. Luedtke, Michela Taufer, and Brian Albright. 2025. Bridging Performance Portability and Scalability for Plasma Simulations on Heterogeneous Systems . In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3731599.3767493

1 Introduction

Ad hoc reengineering of HPC codes for each new hardware platform can deliver high performance, but it is not sustainable due to the rapid evolution of heterogeneous systems. Frameworks like Kokkos [22] and RAJA [10] aim to address this challenge by enabling code portability. However, they can introduce performance overhead and make it harder to exploit hardware-specific characteristics. To overcome these challenges, we ported the Vector Particle-In-Cell (VPIC) code [3] to GPUs using Kokkos and applied portable optimizations to the sorting algorithm and vectorization routines. The original version of the code, VPIC 1.2, was extensively optimized for CPUs using architecture-specific techniques. In contrast, our new version, VPIC 2.0, introduces portable optimizations based on Kokkos to target both CPUs and GPUs without relying on hardware-specific rewrites. Our optimizations are applied once and preserved across platforms, enabling VPIC 2.0 to match the performance of the original CPU-optimized version while also delivering high performance on GPUs. Our evaluation shows that VPIC 2.0 scales efficiently across diverse CPU and GPU architectures. Our work demonstrates that a one-time investment in portable, performance-aware optimization—enabled by Kokkos—can eliminate the need for repeated ad hoc tuning while retaining scalability.

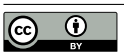
1.1 Limitations of State-of-the-Art

Several projects highlight solutions for performance portability overhead. Tools such as Cabana [17] and LLAMA [7] offer better memory layout control, but do not cover scenarios with changing dynamic memory access patterns. Portable vectorization libraries such as VecCore [2] and Vc [12] boost CPU performance, but add dependencies and require substantial code alterations. These efforts address specific aspects of portability or performance, but fall short in providing a unified solution that supports dynamic access patterns, minimizes code disruption, and performs well across both CPUs and GPUs. This gap motivates our work on VPIC 2.0.

1.2 Our Contributions

To address the limitations of existing solutions, we make the following contributions.

- We pinpoint inefficiencies and bottlenecks in computation and data movement introduced by performance portability frameworks' abstraction layers, which obscure hardware capabilities.
- We develop strategies to eliminate ad hoc vectorization and improve memory access patterns with hardware-targeted sorting while retaining portability.
- We show that our optimizations enable the new VPIC 2.0 to match the CPU performance of the custom-optimized VPIC 1.2 and excel on GPUs.



SC Workshops '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1871-7/2025/11
https://doi.org/10.1145/3731599.3767493

2 Portability Challenges in VPIC

To understand the performance portability challenges in VPIC, we examine limitations in its legacy implementation and review the capabilities and constraints of existing portability frameworks.

2.1 Challenges in the VPIC Codebase

We evaluate our performance portability strategy using the Vector Particle-In-Cell (VPIC) codebase, a production-grade plasma simulation widely used in scientific computing. VPIC is a high-performance particle-in-cell (PIC) simulation code designed to model complex plasma phenomena, including magnetic reconnection, fusion, solar weather, and laser-plasma interactions. VPIC 1.2 was optimized exclusively for CPU architectures using platform-specific vector intrinsics to minimize data movement and maximize performance. Most MPI communication in VPIC is non-blocking point-to-point with up to six neighbors, allowing it to scale efficiently as more nodes are added. However, achieving and maintaining this scalability required extensive ad hoc reengineering for each new hardware platform, an increasingly unsustainable approach given the rapid evolution of heterogeneous computing. For example, 57% of the VPIC 1.2 codebase is dedicated to a custom SIMD library, which must be reengineered whenever new instruction sets are introduced. Figure 1 shows the distribution of SIMD-related code lines by platform and vector width. Due to the fixed-length nature of most instruction sets, significant code duplication occurs across platforms. Only 11% of the SIMD code implements the actual physics kernels. This heavy reliance on hardware-specific capabilities presents a substantial engineering burden, particularly when adding support for new instruction sets such as AVX-512 (non-Xeon Phi), Scalable Vector Extensions (SVE), and SVE2.

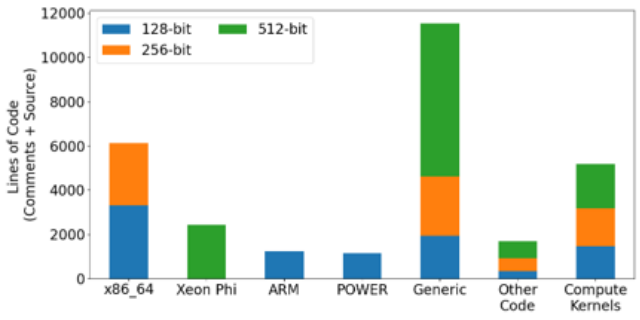


Figure 1: Breakdown of VPIC 1.2 code by SIMD vector length and platform. Over 57% of the total VPIC 1.2 code is dedicated to SIMD support, while only 11% is used to implement the computational kernels.

2.2 Overview of Portability Frameworks

Performance portability solutions are tools that allow a single source code to run on multiple platforms. These tools help developers avoid platform-specific code and ad hoc optimization for specific machines. Performance portability solutions can be generally classified as either low-level code generators or high-level libraries based on the level of abstraction and whether they use other tools as a backend.

Low-level tools (such as OpenMP [6], OpenACC [23], OpenCL [14], HIP [1], SYCL [11], or DPC++ [15]) are often closely integrated with compilers and generate target-specific code directly. OpenMP and OpenACC are compiler directives and libraries that enable the compiler to generate the necessary parallel code automatically. Developers annotate sections of the code for parallelization with compiler directives. Additional directives and clauses manage the data environment, memory management, and GPU offloading. The compiler then generates code to create and execute the parallel sections. These annotation-based tools enable parallel execution with minimal changes to the original serial code, but are heavily reliant on the compiler, can be challenging to maximize performance, and targeting accelerators can make the annotations very complex. OpenCL and HIP are runtime APIs and kernel languages used to create portable code. These frameworks are similar to CUDA in how they are programmed. These solutions can be complex for developers due to the different programming styles, additional memory management challenges, and hardware knowledge that is necessary for high performance. SYCL and DPC++ are programming models that expose an interface built on standard C++. The developer writes kernels in standard C++ using parallel execution policies and portable data structures for parallelizing code. These solutions are relatively new and are less mature compared with other solutions.

High-level tools (such as Kokkos [22], RAJA [10], OCCA [13], and Chapel [5]) generally attain portability by mapping code to different backends, such as CUDA, which then generate the necessary code. Kokkos and RAJA have programming models similar to those of SYCL, where developers use execution policies and data structures to parallelize code. Kokkos and RAJA differ in the degree to which the libraries are tightly coupled. Kokkos keeps the core programming model in a single library. RAJA, on the other hand, keeps execution abstractions, array abstractions, memory management, and meta-programming capabilities in separate libraries. OCCA is another framework for creating portable code using a directive-based kernel language. OCCA is more transparent than Kokkos and RAJA, as users can examine the translated source code. Chapel is a language built for parallel programming. Unlike other portability solutions, parallel programming languages often have more complex features, such as support for distributed or task-based parallelism. These high-level solutions can leverage vendor-optimized back-ends but introduce more levels of abstraction.

2.3 Extensions and Limitations of Existing Tools

Several works focus on extending portability frameworks to improve performance in key areas. Some works implement portable SIMD abstractions [4, 16] to improve compiler-based auto-vectorization. Abstractions for controlling memory layouts have demonstrated significant benefits for HPC applications [9, 18]. The Cabana project extends the Kokkos framework to apply these optimizations of memory layout while remaining portable [17]. The LLAMA project provides abstractions for applying custom memory layouts to more complex data [7]. However, these tools do not address cases where memory access patterns change as applications run.

VPIC 2.0 prioritizes portability across heterogeneous computing architectures by leveraging Kokkos [22]. However, as computational resources per node increase, performance becomes more sensitive to portability-related overhead. Previous work investigated using mixed precision to improve problem size scalability [19, 20]. To maintain the performance scalability of VPIC 1.2, VPIC 2.0 must efficiently scale with GPU capabilities while maximizing vectorization.

3 Portable Optimizations for VPIC 2.0

To address the performance portability challenges introduced above, we develop novel optimizations in VPIC 2.0 that target two key areas: compute and data movement. These optimizations are designed to eliminate the need for ad hoc tuning while ensuring scalable performance across heterogeneous platforms. Specifically, we improve vectorization strategies to support compute efficiency and introduce hardware-targeted sorting algorithms to optimize memory access.

3.1 Compute Optimizations: Portable Vectorization Strategies

Taking full advantage of the computing capabilities of any platform requires vectorization. The difficulty of vectorization changes based on the hardware platform and the programming model. GPUs naturally have large amounts of parallelization which can be seen in the programming model. CUDA and HIP both assume that kernels will run on many small cores in a SIMD like fashion, making it easy to vectorize code. CPUs and languages like C++ are designed primarily for serial execution. The HPC community has built multiple layers of abstraction (i.e., SIMD intrinsics and SIMD libraries) and tools (i.e., compiler auto vectorization) to vectorize code with varying degrees of effectiveness and productivity. Portability layers introduce another abstraction layer that can inhibit vectorization. Kokkos relies on compiler auto vectorization for portable SIMD. Compiler auto vectorization is easily broken by a number of factors such as branches, math functions, memory layouts, and kernel size. Additional vectorization strategies are necessary to fully leverage the hardware and close the gap between the ad hoc optimized VPIC 1.2 and performance portable VPIC 2.0.

We refine *vectorization strategies* to eliminate the need for ad hoc vectorization. We leverage compiler *auto-vectorization* as a baseline while implementing a *guided vectorization* strategy that enhances auto-vectorization using OpenMP SIMD and developer knowledge without requiring major code restructuring. Furthermore, we refactor the code using the Kokkos SIMD library for *manual vectorization* to maximize performance across common CPU architectures supported by Kokkos SIMD. Manual vectorization requires more effort than auto or guided but much less than ad hoc vectorization.

3.2 Data Movement Optimizations: Hardware-Aware Sorting

Efficient memory access patterns are vital for achieving performance and vary depending on the hardware platform. Portability layers hide the differences in memory characteristics from the user which results in portability overhead for optimizations such as sorting. VPIC sorts particles according to the cell index to improve the memory access pattern. Particle sorting must keep the target platform in mind to maximize memory efficiency. CPU main memory has low bandwidth and low latency. Ideally, each thread gets a different cell and processes all particles in the cell. This allows the threads to reuse cell-related data and prevents multiple threads from writing in the same location. The classification of particles according to the cell index, which we denote as a standard classification, ensures the optimal access pattern. GPU-based platforms require coalesced memory accesses to leverage the high memory bandwidth and hide the high memory latency. Coalesced memory accesses occur when successive threads access successive memory. Particle sorting produces the opposite pattern, with all threads accessing the same data. Threads accessing the same data prevent the GPU from hiding memory latency. To achieve portability and performance scalability, we need hardware-targeted sorting that produces the optimal order for the platform.

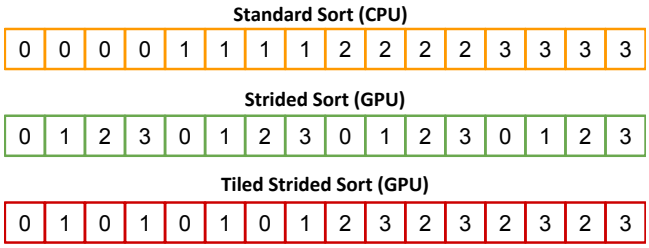


Figure 2: Example of different sorting algorithms.

3.2.1 Strided Sort. We create a different sorting algorithm that produces repeating and strictly monotonically increasing sequences. Our algorithm is listed in Algorithm 1 and an example is shown in Figure 2. First, we create a copy of the keys and identify the minimum and maximum keys. Second, we initialize a histogram to track the number of instances of each key. Third, we iterate through each key, add the current histogram count for the key multiplied by the maximum key, and update the histogram. Finally we sort the values using the new keys. This algorithm, which we denote as strided sort, produces coalesced memory access patterns.

3.2.2 Tiled Strided Sort. Our tiled strided sort algorithm builds on strided sort by reducing the amount of data read from main memory. Strided sort produces coalesced memory access patterns, but also requires reading the data multiple times. With tiled strided sort we produce repeating tiles where indices within each tile follow strided sort. Our algorithm is listed in Algorithm 2 with an example in Figure 2. This pattern allows the GPU to read the coalesced data and reuse them, increasing the arithmetic intensity and performance of the application.

4 Implementation of Portable Optimization

To realize the portable optimizations described in the previous section, we implement vectorization and sorting strategies in VPIC 2.0 using the Kokkos portability framework. Our implementation approach begins with the setup of the software environment and is followed by details of our compute and data movement optimizations.

4.1 Framework and Environment

We leverage the Kokkos performance portability framework for implementing our gather-scatter benchmarks and VPIC 2.0. Kokkos [21, 22] is a suite of libraries and abstractions to develop high performance C++ applications that are portable across many platforms, including all major CPUs and GPUs. Kokkos achieves high performance and portability by mapping its own programming model to various native backends such as OpenMP, CUDA, and HIP. This allows Kokkos to take advantage of the work done on optimizing the backend code generation and focus on mapping the model and library features for rapid development. Toolchain maturity and software environments still have limitations that can inhibit portability solutions. For our performance portability study, we use Kokkos 4.6.01 built with GCC and OpenMPI to provide a neutral environment across all platforms. On x86_64 platforms, we use GCC 12.2.0 and OpenMPI 4.1.5 while on ARM platforms, we use GCC 12.1.0 and OpenMPI 4.1.4. We use CUDA 12.3.1 and ROCm 6.4.1 for the Nvidia and AMD GPU platforms respectfully. For the Nvidia Grace CPU and Grace Hopper superchips we use NVHPC 25.1 because of the lack of support for the Grace architecture in GCC 12. The Grace CPU and A64FX ARM platforms have more toolchain limitations than the x86_64 CPUs due to having less mature compiler support and Kokkos 4.6.01 not supporting SVE or SVE2 vector instructions.

4.2 Implementation of Vectorization Strategies

We implement the four vectorization strategies using the Kokkos abstractions for vectorization loops and SIMD along with a custom SIMD library used by VPIC 1.2. The *auto vectorization* strategy uses the hierarchical parallelism mechanisms provided by Kokkos. Internally, the loops are marked with `#pragma ivdep` to inform the compiler that there are no loop-carried dependencies. The compiler uses heuristics and code analysis to vectorize the code. Our *guided vectorization* replaces `#pragma ivdep` with `#pragma omp simd` to force the compiler to vectorize the loop. The compiler may not vectorize the code if it cannot guarantee the correctness or is incapable of generating vectorized code. Additional changes, such as splitting kernels to separate difficult-to-vectorize mathematical functions, are also applied. We use the KokkosSIMD library for *manual vectorization*. The library is an implementation of the C++26 SIMD library. The library covers most common SIMD operations and includes SIMD masks for handling branches. The library supports AVX2, AVX512, and Neon instruction sets. We also implement functions for transposing data in registers. These functions help accelerate data loading and storing in VPIC and require much less instruction set specific code than the *ad hoc vectorization* strategy. The *ad hoc vectorization* strategy uses the custom SIMD library in VPIC 1.2 for vectorization. The library uses intrinsics and supports the AVX, AVX2, AVX512 (Xeon Phi only), Neon, and Altivec instruction sets.

4.3 Implementation of Hardware-Aware Sorting

We implement two sorting functions that change the order of the final sorted array. Both sorting functions create and modify a copy of the keys to generate the desired final order. The adjustment of the keys is $O(N)$. Once the new keys are generated, we use the parallel `sort_by_key` function provided by Kokkos for sorting the keys and values.

Algorithm 1 Strided Sort.

Input: $Keys, Values$
1: $new\_keys \leftarrow Keys$
2: $(max_k, min_k) \leftarrow MINMAX(new\_keys)$
3: $key\_counts \leftarrow Histogram(max_k - min_k + 1)$
4: **for all** $key \in new\_keys$ **do in parallel**
5: $\quad i = atomic\_fetch\_add(key\_counts(key - min_k), 1)$
6: $\quad key = (key - min_k) + i * (max_k + 1)$
7: **end for**
8: $SORT\_BY\_KEY(new\_keys, Keys)$
9: $SORT\_BY\_KEY(new\_keys, Values)$

Algorithm 2 Tiled Strided Sort.

Input: $Keys, Values, TileSz$
1: $new\_keys \leftarrow Keys$
2: $(max_k, min_k) \leftarrow MINMAX(new\_keys)$
3: $key\_counts \leftarrow Histogram(max_k - min_k + 1)$
4: **for all** $key \in new\_keys$ **do in parallel**
5: $\quad atomic\_inc(key\_counts(key - min_k))$
6: **end for**
7: $max_r \leftarrow MAX(key\_counts)$
8: $chunk\_sz \leftarrow TileSz * max_r$
9: $RESET(key\_counts)$
10: **for all** $i \in len(Keys)$ **do in parallel**
11: $\quad id = Keys(i) - min_k$
12: $\quad tile = atomic\_fetch\_add(key\_counts(id, 1)$
13: $\quad chunk = Keys(i)/TileSz$
14: $\quad new\_keys(i) = chunk * chunk\_sz + tile * TileSz + id$
15: **end for**
16: $SORT\_BY\_KEY(new\_keys, Keys)$
17: $SORT\_BY\_KEY(new\_keys, Values)$

4.3.1 Strided Sort. Our *strided sort* pseudocode is shown in Algorithm 1. A copy of the keys is made and the minimum and maximum keys are found. We initialize a View key_counts to track the number of duplicates for each unique key. Finally, we iterate over all keys in parallel and update each key by adding an offset based on the maximum key value and the number of times the key has been seen. This ensures that the final keys are sorted such that the key View is split into one or more strictly monotonically increasing subsets.

4.3.2 Tiled Strided Sort. Our algorithm *tiled strided sort* builds on *strided sort* by splitting the data into tiles and applying *strided sort* to each tile. Keys are split into chunks and each chunk contains one or more repeating tiles where all keys in each tile are in *strided sort* order. Similarly to strided sort, we start by finding the minimum and maximum keys and setting up a View for counting repeated keys. Next, the maximum number of repeated keys is found to determine how many tiles are in each chunk. The counting View is reset to zeros. For each key, we atomically fetch and update `key_counts` to get the tile ID, determine the chunk in which the key belongs, and update the new key with the chunk and tile offset.

5 Performance Portability Evaluation

We evaluate these optimizations on a diverse set of CPU and GPU architectures to assess both performance portability and scalability.

5.1 Heterogeneous System Configurations

We evaluate portability and scalability across four robust computing systems: Darwin at LANL, Selene at Nvidia, plus Sierra and Tuolumne at LLNL. Darwin at Los Alamos provides a wider mix of x86_64, ARM, and GPU architectures than typical clusters. Our testing utilizes Intel Xeon Max 9480, Xeon Platinum 8480, EPYC 7763, A64FX, and Grace CPUs, alongside Nvidia A100, H100, AMD MI100, and MI250 GPUs. Selene, Nvidia's DGX SuperPod, offers 79 petaflops with nodes comprising two 64-core AMD EPYC 7742 CPUs, 2 TB DRAM, and eight 80 GB NVIDIA A100 GPUs. Sierra, a 125-petaflop machine at Lawrence Livermore, is built on IBM Power Systems AC922, with 4,320 nodes featuring dual 22-core Power9 CPUs, 256 GB RAM, and four 16 GB NVIDIA V100 GPUs. Tuolumne, a 288-petaflop system at Lawrence Livermore, akin to El Capitan, has 1,152 nodes equipped with four AMD MI300A APUs (24 CPU cores plus a GPU, 128 GB of HBM3). Table 1 provides a summary of the hardware architectures, memory capacities, and capabilities of the systems. The values highlight the architectural diversity (i.e., x86_64, ARM, and GPU accelerators with DDR4, DDR5, HBM2, and HBM3 memory) used in our portability and scalability evaluation.

Table 1: CPU and GPU specifications for the four test systems, including core counts, memory type/capacity, last-level cache, and STREAM Triad bandwidth.

Platform	Core count	Main Memory	Last Level Cache	Main Memory Bandwidth
A64FX	48	32 GB HBM	4*8MB	424 GB/s
EPYC 7763 (Zen 3)	2*64	512 GB DDR4	256 MB	165 GB/s
Platinum 8480 (SPR DDR)	2*56	256 GB DDR5	105 MB	96.77 GB/s
Xeon Max 9480 (SPR HBM)	2*56	128 GB DDR5	105 MB	266.05 GB/s
Grace	2*72	480 GB	114 MB	390 GB/s
MI300A (CPU)	24	128 GB	256 MB	202.18 GB/s
V100S	5120	32 GB	6 MB	886.4 GB/s
A100	6912	80 GB	40 MB	1,682 GB/s
H100	16896	96 GB	50 MB	3,713 GB/s
MI100	7680	32 GB	8 MB	970.9 GB/s
MI250	13312	128 GB	16 MB	2,498 GB/s
MI300A (GPU)	14592	128 GB	256 MB	3,254 GB/s

5.2 Evaluation Methodology

We evaluate how each optimization impacts the performance and scalability of VPIC 2.0 by defining clear metrics (runtime, bandwidth, and resource utilization), varying key testing parameters (CPU types, key patterns, and GPU counts), and organizing experiments into three classes (vectorization, sorting, and scalability).

5.2.1 Performance Metrics.

- **Runtime** Execution time for the particle push kernel and the full simulation.
- **Bandwidth** Memory bandwidth in gather–scatter benchmarks to assess data movement efficiency.
- **Resource Utilization** Comparison of compute intensity achieved (FLOPs) with arithmetic intensity (FLOPs / byte).

5.2.2 Parameters Varied.

- **Chip Configurations** A range of x86_64 and ARM CPUs to evaluate vectorization portability.
- **Key Patterns** Contiguous and repeating gather–scatter patterns for sorting performance studies.
- **Number of GPUs** Strong scaling from 1 to 512 GPUs.

5.2.3 Experiment Classes.

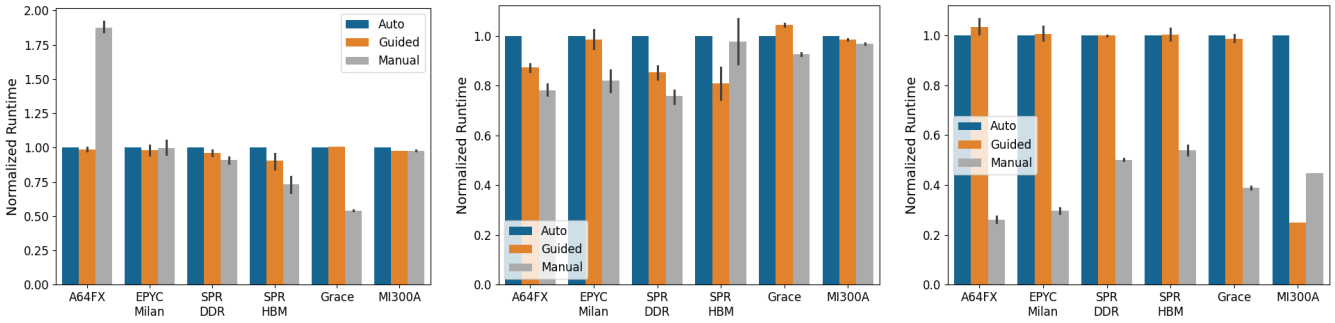
- **Vectorization** Impact of vectorization strategies on CPUs, using microbenchmarks and the particle push kernel.
- **Hardware-Targeted Sorting** Performance effects on gather–scatter patterns, stencils, and the particle push kernel.
- **Scalability** Exploration of new super-linear strong scaling opportunities in VPIC 2.0.

5.3 Vectorization Evaluation

To understand the impact of different vectorization strategies on performance portability, we first apply the auto, guided, and manual vectorization strategies to three kernels derived from the RAJAPerf microbenchmark suite. The AXPY kernel represents the simplest SIMD code without mathematical functions or branching. The PLANCKIAN kernel models calculations involving Planck's law and uses the exponential function, which may hinder compiler vectorization. The third kernel, PI_REDUCE, approximates π in parallel with a reduction across all samples.

Figure 3 shows runtime normalized to auto vectorization. For most platforms, AXPY performs similarly across all strategies, indicating that compilers handle very simple kernels well. Manual vectorization is nearly twice as slow on A64FX due to the lack of Kokkos SIMD support for 512-bit SVE instructions. Grace, which uses 4×128-bit SIMD units, benefits more from manual vectorization despite also using SVE/SVE2 due to the SIMD units better aligning with the 128-bit NEON instruction set. PLANCKIAN gains up to 20% from guided vectorization, while PI_REDUCE reveals how common operations can inhibit vectorization; manual vectorization is up to 80% faster than auto and guided on non-MI300A CPUs.

We next evaluate auto, guided, manual, and ad hoc vectorization strategies on the particle push kernel in VPIC (Figure 4). We run the tests on Darwin using the laser–plasma instability benchmark, with the A64FX using two separate nodes instead of dual-socket configurations. The particle push kernel is significantly more complex than the RAJAPerf kernels, involving complex memory access patterns, common math functions, and branch divergence.



(a) AXPY Kernel (b) Planckian Kernel (c) PI Reduce Kernel

Figure 3: Normalize runtime comparison of auto, guided, and manual vectorization strategies across CPU architectures for three RAJAPerf kernels (AXPY, PLANCKIAN, and PI_REDUCE). Runtimes are normalized to the auto vectorization baseline.

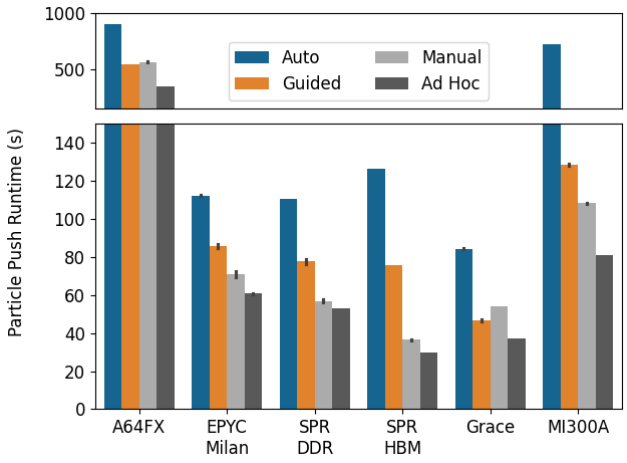


Figure 4: Runtime comparison of auto, guided, manual, and ad hoc vectorization for the VPIC particle push kernel across CPU architectures. Results are from the laser–plasma instability benchmark run on the Darwin testbed.

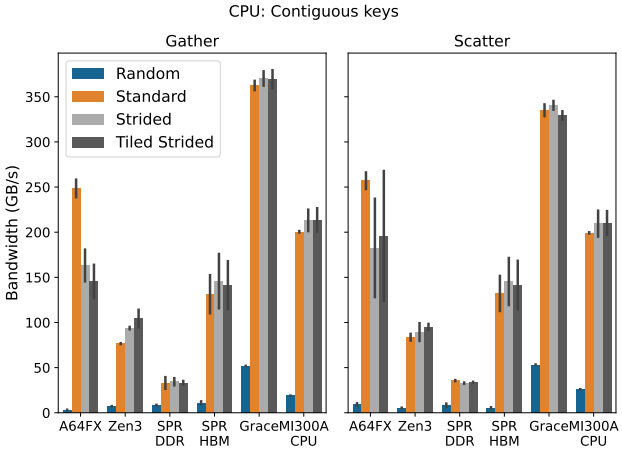
Guided and manual vectorization consistently outperform auto vectorization, with guided achieving competitive performance on all platforms and manual matching VPIC 1.2 on x86_64 CPUs. Guided vectorization is up to 83% faster on the MI300A and 25–55% faster on other platforms. Greater gains on A64FX and Grace are limited by the lack of SVE/SVE2 support in manual/ad hoc strategies. Moving from DDR to HBM (SPR DDR vs. SPR HBM) improves manual/ad hoc performance because compilers cannot easily generate the optimized load/store code. The complexity of the particle push kernel amplifies the vectorization benefits observed in the microbenchmarks.

5.4 Sorting Evaluation

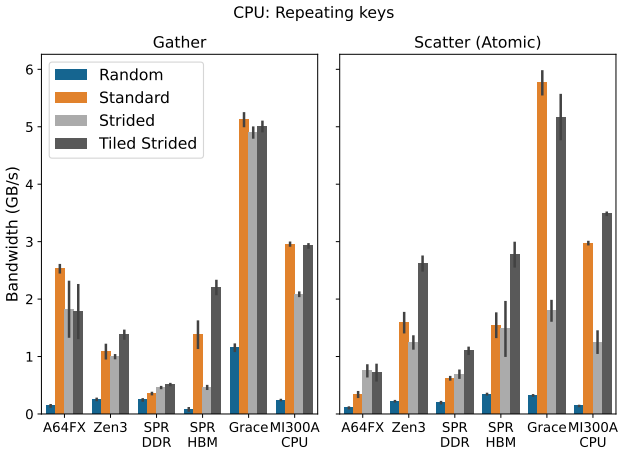
To evaluate the importance of hardware-targeted sorting for controlling memory access patterns, we develop a microbenchmark for gather–scatter patterns common in hash tables, stencils, and particle methods. The benchmark processes one billion double-precision numbers using three key patterns: contiguous keys with unique keys in sorted order representing the ideal case with fully coalesced accesses; repeated keys with 10 million unique keys, each repeated 100 times, introducing high atomic contention; and a 5-point stencil access pattern as used in the VPIC particle push kernel. Test sizes are set to guarantee test data is large enough that it cannot fit in cache. Memory bandwidth is calculated based on the total amount of data movement for each kernel divided by the measured runtime. We test three sorting algorithms (i.e., standard, strided, and tiled-stride) on multiple CPU and GPU platforms. Tile sizes match the number of CPU threads or three times the number of GPU cores. The same sorting variants are applied to the full VPIC particle push kernel, which includes a gather phase, atomic scatter phase, and additional physics computations.

On CPUs, the results in Figure 5 show that with contiguous keys (Figure 5a), sorting has minimal effect because accesses are already coalesced. GPUs and high-bandwidth CPUs such as A64FX and SPR HBM sustain near-peak STREAM bandwidth, but SPR DDR underperforms due to problem-size sensitivity. When keys are repeated (Figure 5b), bandwidth drops by nearly two orders of magnitude on all CPUs, with a more severe drop for HBM-based platforms. Strided sorting often matches or underperforms standard sort, whereas tiled-strided improves cache utilization and reduces atomic serialization. In the stencil case (Figure 5c), patterns resemble the repeated keys case but with more irregular accesses and lower bandwidth. Tiled-strided generally performs best, though performance gaps widen for architectures such as A64FX that are more sensitive to complex memory access patterns.

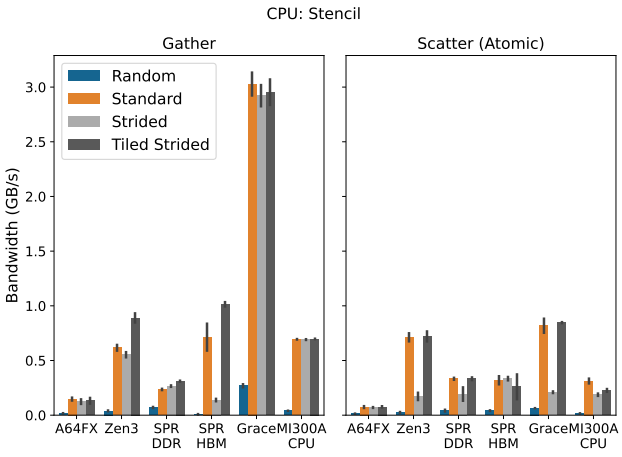
On GPUs, the results in Figure 6 indicate that for contiguous keys (Figure 6a), all sorting algorithms perform identically since accesses are already optimal. With repeated keys (Figure 6b), standard sort suffers from high memory latency and atomic conflicts, especially on V100, MI100, and MI250. Strided and tiled-strided restore coalescing, with tiled-strided nearly doubling bandwidth on the A100 and H100. Vendor differences emerge in that on AMD GPUs, strided sometimes outperforms tiled-strided. In the stencil case (Figure 6c), both strided and tiled-strided improve over standard sort, but the benefits are smaller because irregular accesses still require multiple data reads.



(a) Gather-Scatter performance on different CPUs with unique contiguous keys.

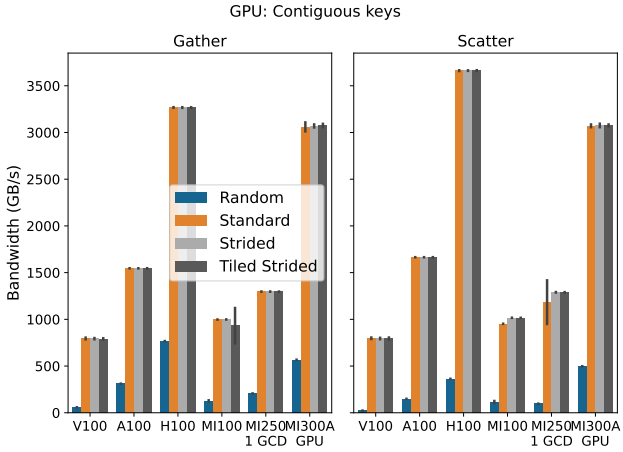


(b) Gather-Scatter performance on different CPUs where each key is repeated 100 times.

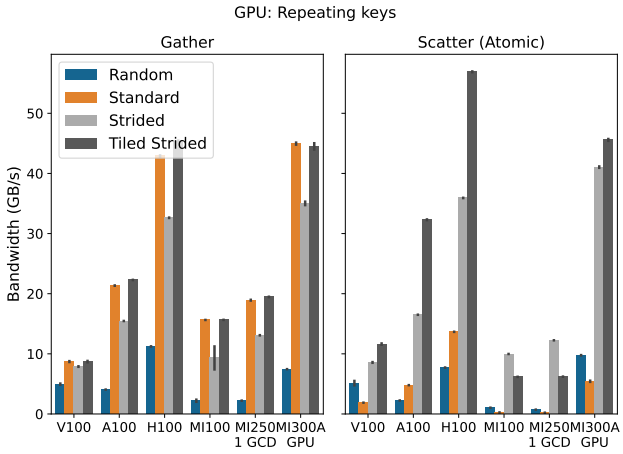


(c) Gather-Scatter performance for a 5-point stencil on different CPUs where each key is repeated 100 times.

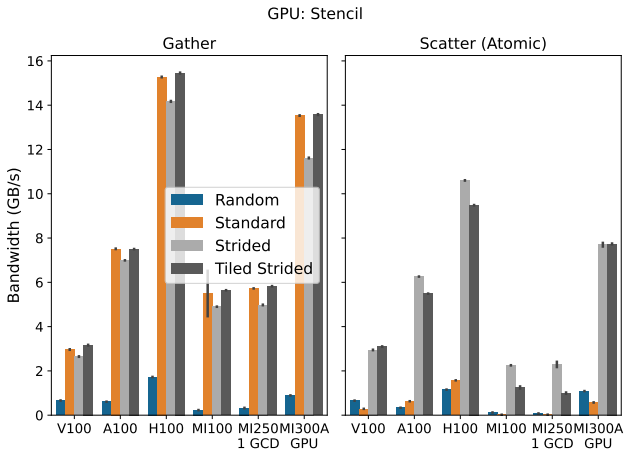
Figure 5: Performance of different gather-scatter patterns using contiguous and repeating keys on various CPU platforms.



(a) Gather-Scatter performance on different GPUs with unique contiguous keys.



(b) Gather-Scatter performance on different GPUs where each key is repeated 100 times.



(c) Gather-Scatter performance for a 5-point stencil on different GPUs where each key is repeated 100 times.

Figure 6: Performance of different gather-scatter patterns using contiguous and repeating keys on various GPU platforms.

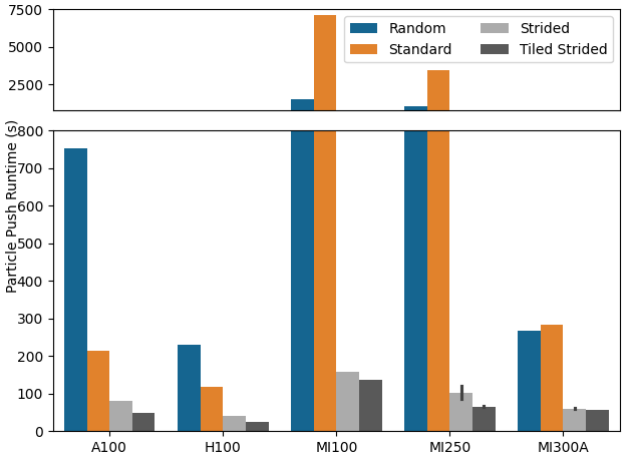


Figure 7: Impact of different sorting orders on the VPIC particle push kernel across four GPU architectures.

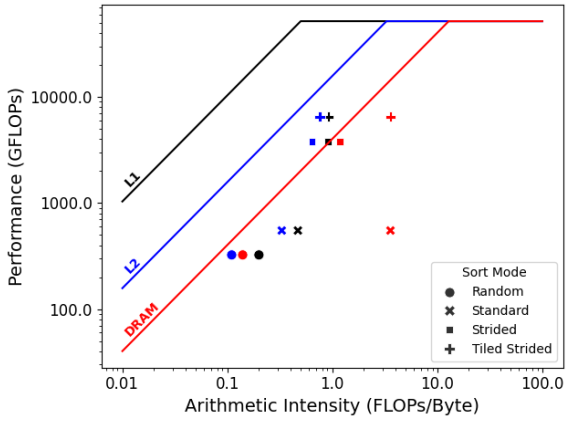
Applying these sorting algorithms to the full VPIC particle push kernel (Figure 7) shows that on NVIDIA GPUs, strided sort is more than two times faster than standard sort, and tiled-strided nearly doubles the performance of strided. On AMD GPUs, random and standard sorts are over an order of magnitude slower than strided or tiled-strided. Vendor-specific cache and memory differences play a key role, highlighting that hardware abstraction layers can hide important tuning opportunities.

Runtime metrics alone do not indicate whether the hardware is being fully utilized. To assess utilization, we perform roofline analyses using nsight-compute and rocprof-compute profilers (Figure 8). On the H100, standard sort shows high arithmetic intensity (3.58) but poor utilization (1% of peak FP32). Strided sort improves utilization but lowers intensity (1.18). The tiled strided sort matches the intensity of the standard sort (3.59) while improving utilization, increasing throughput from 550 GFlop/s to 6.51 TFlop/s, an 11.8× improvement. On the MI250 (single GCD), strided and tiled-strided improve utilization over standard sort, though less dramatically than on the H100. The tiled strided sort changes the arithmetic intensity from 16.9 to 4.47 FLOP/byte and boosts throughput from 38.8 GFlop/s to 800 GFlop/s, a 20.6× improvement. The MI300A behaves differently, with all ing methods having low arithmetic intensity (< 1.0) and being memory-bandwidth bound. Standard (0.51), strided (0.65), and tiled-strided (0.85) all show unexpectedly low compute utilization, indicating additional portability overheads and hardware-specific effects that limit performance.

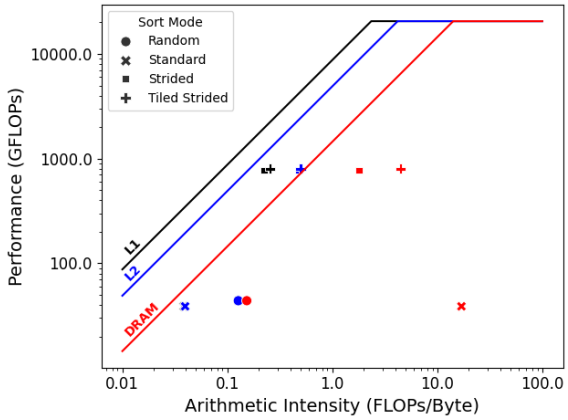
These results confirm that the choice of sorting strategy must be tuned to each architecture to maximize both bandwidth and computational throughput, especially when targeting performance portability across heterogeneous systems. Performance anomalies in Figure 8c show that there is more opportunities to improve memory access patterns in future work.

5.5 Scalability and Cache Effects

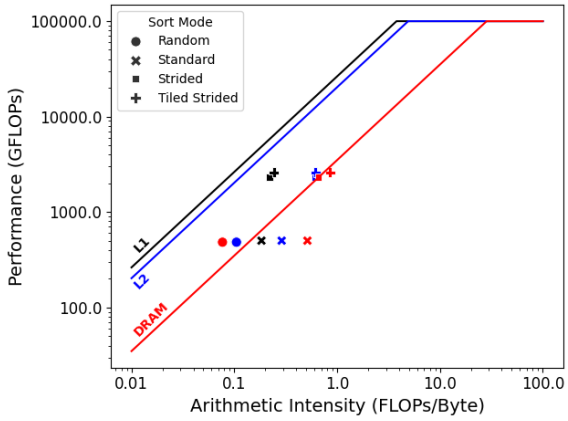
Modern GPUs with larger caches (e.g., A100, H100, MI300A) enable the possibility of storing the entire VPIC grid in cache, eliminating



(a) H100 roofline of the particle push kernel for different sorting algorithms.



(b) MI250 roofline of the particle push kernel for different sorting algorithms.



(c) MI300A roofline of the particle push kernel for different sorting algorithms.

Figure 8: VPIC 2.0 particle push GPU rooflines using different sorting algorithms to reorder particles.

the need for particle sorting and allowing super-linear speedup in strong scaling. In VPIC 2.0, particle sorting minimizes the number of grid points updated at once so that they remain in the cache. If the entire grid fits in cache, sorting overhead can be avoided altogether.

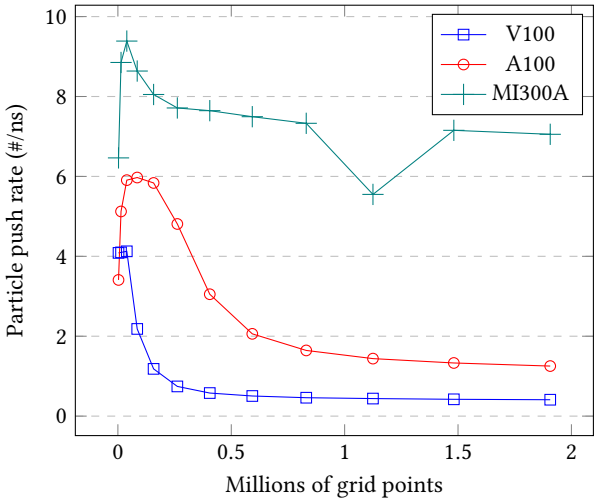
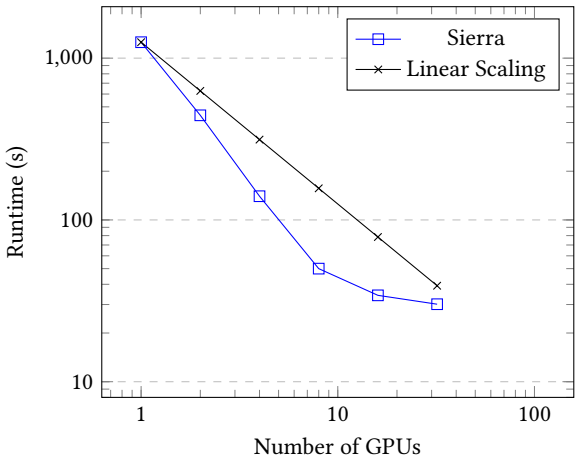


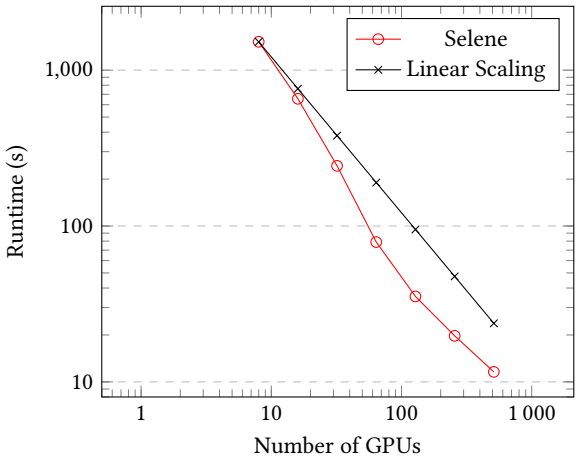
Figure 9: Number of particle pushes per nanosecond as a function of grid size on Sierra with V100 and Selene with A100. The total number of particles is held constant and only particle push time is included.

We repeat the laser–plasma interaction benchmark from the sorting tests, but with particle sorting disabled, using Sierra (V100), Selene (A100), and Tuolumne (MI300A). The particle count is fixed, while the grid size varies. Figure 9 shows the particle pushes per nanosecond versus the grid size. Each GPU exhibits a sharp performance peak: V100 reaches ~4 pushes/ns at 13,824 grid points, A100 reaches ~6 pushes/ns at 85,184, and MI300A reaches ~9 pushes/ns at 39,304. For the A100, the peak grid size is about 6× that of the V100, matching its cache increase. The performance boost for the A100, compared to the V100, aligns with an approximately sixfold increase in grid points, mirroring the cache growth. If this rise is due to cache effects, subsequent strong scaling tests should exhibit super-linear speedup. We hypothesize that the A100's performance decrease with very few grid points (resulting in very high particles per cell) is an effect of colliding writes during current deposition. The MI300A architecture is distinctly different from the V100 and A100 GPUs and features a significantly larger cache, complicating predictions about the influence of cache size on the data. Although there is a rare decrease in performance to below six particle pushes per ns, this is an anomaly.

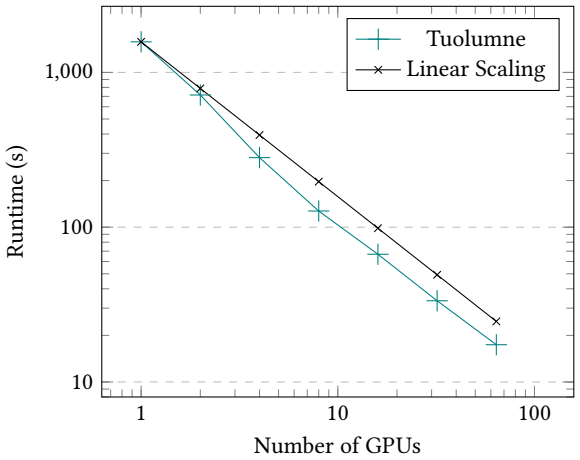
Carefully selecting the size of our grid to match the peak performance in Figure 9, we run strong scaling tests on Sierra with V100 GPUs (i.e., from 1 to 32 GPUs), as shown in Figure 10a, on Selene with A100 GPUs (i.e., from 8 to 512 GPUs), as shown in Figure 10b, and on Tuolumne with MI300A GPUs (i.e., from 1 to 64 GPUs). As suggested by Figure 9, we do indeed observe super-linear scaling on all three platforms.



(a) The V100 GPUs on Sierra achieves super-linear speedup but runs into communication limits at larger scales.



(b) The A100 GPUs on Selene achieves better super-linear speedup.



(c) The MI300A GPUs on Tuolumne achieves super-linear speedup.

Figure 10: Strong scaling on the Sierra, Selene, and Tuolumne systems.

Figure 10a presents the strong scaling performance of VPIC 2.0 on the Sierra supercomputer using NVIDIA V100 GPUs. As the number of GPUs increases, more of the computational grid is kept in cache, leading to superlinear speedup up to a point. The superlinear speedup observed in this scaling test highlights the effectiveness of caching strategies in maximizing performance. We observe a 25x speedup for an 8x increase of GPUs, from 1 to 8. Beyond 8, the GPUs are very empty, and communication overhead starts to cancel out the super-linear speedup, revealing a key scalability limitation.

Figure 10b presents the strong scaling results on the Selene supercomputer with A100 GPUs. This result shows that newer GPU architectures with larger caches can sustain superlinear scaling for a larger range of GPU counts before communication overhead dominates. On Selene, we see a 19x speedup for an 8x increase of GPUs, from 8 to 64. Unlike the V100, we see near-ideal scaling all the way to 512 GPUs, the largest allocation we were able to test. It validates the effectiveness of smart caching strategies in enabling scalable GPU performance. Recent advances in GPU-GPU communication and growing cache sizes continue to make the superlinear strong scaling case more practical.

Figure 10c shows that the superlinear strong scaling behavior applies to AMD as well as Nvidia GPUs. Superlinear scaling is maintained on up to 256 GPUs even as communication costs grow. We see a 90.5x speedup for a 64x increase in GPUs. Improvements in shared-memory and cache utilization can further improve the strong scalability on the MI300A. The 256 MB of last level cache in the MI300A can fit more than 3.5 million grid points, allowing more problems to take advantage of the superlinear strong scaling behavior. Additional features like GPU-aware MPI will reduce the communication overhead for exchanging particles and enable greater superlinear scaling in the future.

6 Scientific Impacts

VPIC transforms computational plasma physics by enhancing performance scalability and portability. It allows high-fidelity simulations of phenomena such as magnetic reconnection and fusion across various hardware setups. Furthermore, VPIC's vectorization and memory strategies are broadly applicable in fluid dynamics, astrophysics, and gahter scatter patterns in general. It exhibits scalability on large supercomputers, making VPIC a promising choice for exascale computing, where efficient resource use is paramount. VPIC 2.0 enables longer simulations for greater insights into plasma behavior, advanced diagnostics that can be run in the timestep for more accurate analysis and insights in plasma modeling [8], and multiple runs of previously expensive simulations for stochastic analysis and generation of AI/ML training datasets. Such capabilities can accelerate scientific discovery and open up avenues of study that were previously intractable. The superlinear strong scaling behavior is a promising optimization for running large batches of smaller simulations. Such simulations can be used as training datasets for upscaling the resolution of plasma simulations or creating fast models to guide other simulations.

7 Conclusions

Our key contribution is optimizing VPIC 2.0 to achieve performance portability and scalability across heterogeneous architectures. Our vectorization strategies ensure that VPIC 2.0 attains performance parity with the ad hoc optimized VPIC 1.2 on CPU platforms. We optimize particle sorting by adapting the sorting order to the target hardware, significantly enhancing particle push performance on GPUs with improvements of up to 37× faster than using the standard sorting order on GPUs. To explore the new scalability capabilities made available, we implement caching strategies that store grid data in the shared cache of the GPU, enabling strong superlinear scaling. These optimizations collectively allow VPIC 2.0 not only to match but also to exceed the scalability of VPIC 1.2, unlocking new opportunities for high-fidelity plasma simulations and broader scientific discovery.

Acknowledgments

This work was supported by the Advanced Simulation and Computing, Integrated Codes Program and the Laboratory Directed Research and Development (LDRD) Program at Los Alamos National Laboratory. Los Alamos National Laboratory is managed by Triad National Security, LLC, for the National Nuclear Security Administration of the U.S. Department of Energy under Contract No. 89233218CNA000001. LA-UR-25-28139

References

[1] [n. d.]. HIP: C++ Heterogeneous-Compute Interface for Portability. https://github.com/ROCm-Developer-Tools/HIP. Accessed: 2023-09-23.
[2] G Amadio, P Canal, D Piparo, and S Wenzel. 2018. Speeding up software with VecCore. In *Journal of Physics: Conference Series*, Vol. 1085. IOP Publishing, 032034.
[3] Kevin J Bowers, BJ Albright, L Yin, B Bergen, and TJT Kwan. 2008. Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation. *Physics of Plasmas* 15, 5 (2008), 055703.
[4] Berenger Bramas et al. 2017. Inastemp: A Novel Intrinsics-As-Template Library for Portable SIMD-Vectorization. *Scientific Programming* 2017 (2017).
[5] Bradford L Chamberlain, David Callahan, and Hans P Zima. 2007. Parallel Programmability and the Chapel Language. *The International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312.
[6] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An Industry Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering* 5, 1 (1998), 46–55.
[7] Bernhard Manfred Gruber, Guilherme Amadio, Jakob Blomer, Alexander Matthes, René Widera, and Michael Bussmann. 2023. LLAMA: The low-level abstraction for memory access. *Software: Practice and Experience* 53, 1 (2023), 115–141.
[8] Fan Guo, Xiaocan Li, William Daughton, Patrick Kilian, Hui Li, Yi-Hsin Liu, Wangcheng Yan, and Dylan Ma. 2019. Determining the Dominant Acceleration Mechanism During Relativistic Magnetic Reconnection in Large-scale Systems. *The Astrophysical Journal Letters* 879, 2 (2019), L23.
[9] Holger Homann and Francois Laenen. 2018. SoAx: A Generic C++ Structure of Arrays for Handling Particles in HPC Codes. *Computer Physics Communications* 224 (2018), 325–332.
[10] Richard D. Hornung and Jeffrey A. Keasler. 2014. *The RAJA portability layer: overview and status.* Technical Report. Lawrence Livermore National Lab.
[11] Ronan Keryell, Ruyman Reyes, and Lee Howes. 2015. Khronos SYCL for OpenCL: A Tutorial. In *Proceedings of the 3rd International Workshop on OpenCL*. 1–1.
[12] Matthias Kretz and Volker Lindenstruth. 2012. Vc: A C++ library for explicit vectorization. *Software: Practice and Experience* 42, 11 (2012), 1409–1430.
[13] David S Medina, Amik St-Cyr, and Tim Warburton. 2014. OCCA: A Unified Approach to Multi-Threading Languages. *arXiv preprint arXiv:1403.0968* (2014).
[14] Aaftab Munshi. 2009. The OpenCL Specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE, 1–314.
[15] James Reinders, Ben Ashbaugh, James Brodman, Michael Kinsner, John Pennycook, and Xinmin Tian. 2021. *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems Using C++ and SYCL.* Springer Nature.
[16] Damodar Sahasrabudhe, Eric T Phipps, Sivasankaran Rajamanickam, and Martin Berzins. 2020. A Portable SIMD Primitive Using Kokkos for Heterogeneous Architectures. In *Accelerator Programming Using Directives: 6th International Workshop, WACCPD 2019, Denver, CO, USA, November 18, 2019, Revised Selected Papers 6*. Springer, 140–163.
[17] Stuart Slattery, Samuel Temple Reeve, Christoph Junghans, Damien Lebrun-Grandié, Robert Bird, Guangye Chen, Shane Fogerty, Yuxing Qiu, Stephan Schulz,

Aaron Scheinberg, et al. 2022. Cabana: A performance portable library for particle-based simulations. *Journal of Open Source Software* 7, 72 (2022), 4115.

[18] Robert Strzodka. 2012. Abstraction for AoS and SoA Layout in C++. In *GPU Computing Gems Jade Edition*. Elsevier, 429–441.

[19] Nigel Tan, Robert F Bird, Guangye Chen, Scott V Luedtke, Brian J Albright, and Michela Taufer. 2022. Analysis of Vector Particle-In-Cell (VPIC) memory usage optimizations on cutting-edge computer architectures. *Journal of Computational Science* 60 (2022), 101566.

[20] Nigel Tan, Robert F. Bird, Guangye Chen, and Michela Taufer. 2021. Optimize memory usage in Vector Particle-In-Cell (VPIC) to break the 10 trillion particle barrier in plasma simulations. In *Proceedings of the 21st International Conference on Computational Science (ICCS)*. Springer *(Best Track Paper Award)*, Krakow, Poland, 452–465.

[21] Christian Trott, Luc Berger-Vergiat, David Poliakoff, Sivasankaran Rajamanickam, Damien Lebrun-Grandie, Jonathan Madsen, Nader Al Awar, Milos Gligoric, Galen Shipman, and Geoff Womeldorff. 2021. The Kokkos EcoSystem: Comprehensive Performance Portability for High Performance Computing. *Computing in Science Engineering* 23, 5 (2021), 10–18. doi:10.1109/MCSE.2021.3098509

[22] Christian R Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S Hollman, Dan Ibanez, et al. 2021. Kokkos 3: programming model extensions for the exascale era. *IEEE Trans. on Parallel and Distributed Systems* 33, 4 (2021), 805–817.

[23] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. 2012. OpenACC—First Experiences with Real-World Applications. In *Euro-Par 2012 Parallel Processing: 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012. Proceedings 18*. Springer, 859–870.