To the Graduate Council:
I am submitting herewith a dissertation written by Nigel Phillip Tan entitled "Enhancing Code Portability, Problem Scale, and Storage Efficiency in Exascale Applications." I have examined the final paper copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Michela Taufer, Major Professor

We have read this dissertation
and recommend its acceptance:

Michael Berry

Jian Huang

Sanjukta Bhowmick

Scott V. Luedtke

Bogdan Nicolae

Accepted for the Council:

Dixie Thompson

Vice Provost and Dean of the Graduate School

To the Graduate Council:
I am submitting herewith a dissertation written by Nigel Phillip Tan entitled "Enhancing Code Portability, Problem Scale, and Storage Efficiency in Exascale Applications." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Michela Taufer, Major Professor

We have read this dissertation
and recommend its acceptance:

Michael Berry

Jian Huang

Sanjukta Bhowmick

Scott V. Luedtke

Bogdan Nicolae

Accepted for the Council:

Dixie Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Enhancing Code Portability, Problem Scale, and Storage Efficiency in Exascale Applications

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Nigel Phillip Tan

August 2024

I dedicate this work to my loving family. My parents, Sonny and Grace, who always support and look out for me. My brothers, William and David, who helped me relax and avoid burning myself out.

Acknowledgements

I would like to thank my advisor Michela Taufer for teaching me to be a better research and pushing me to excel. Michael Berry, Jian Huang, Sanjukta Bhowmick, Scott V. Luedtke, and Bogdan Nicolae for being on my committee. My friend Paula Olaya who went through the PhD process with me. The members of Global Computing Laboratory who were invaluable resources and friends. Robert Bird who was invaluable for setting the initial directions of my research. My collaborators, Kevin Assogba, Brian Albright, Franck Cappello, Guangye Chen, Stephen Harrell, Ali Khan, Nicolas Morales, Keita Teranishi, for all the help throughout the years.

Abstract

The growing diversity of hardware and software stacks adds additional development challenges to high-performance software as we move to exascale systems. Re-engineering software for each new platform is no longer practical due to increasing heterogeneity. Hardware designers are prioritizing AI/ML features like reduced precision that increase performance but sacrifice accuracy. The growing scale of simulations and the associated checkpointing frequency exacerbate the I/O overhead and storage cost challenges already present in petascale systems. Moving forward, the community must address performance portability, precision optimization, and data deduplication challenges to ensure that exascale applications efficiently deliver scientific discovery.

In this dissertation, we address each challenge posed to exascale applications by emerging heterogeneous hardware. We enhance the performance portability of the Vector Particle-In-Cell (VPIC) application using the Kokkos portability framework, achieving performance gain across different platforms. We develop techniques for leveraging lower precision formats such as 16-bit floating-point and 16-bit fixed-point for the VPIC application while preserving scientific accuracy, demonstrating similar scientific findings as for 32-bit floating-point. We design a Merkle tree-based data deduplication method that prunes spatiotemporal redundancy and creates a compact metadata representation for incremental checkpointing, mitigating I/O overhead and reducing storage costs.

Table of Contents.

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Problems, Motivations, and Solutions

Exascale supercomputers, capable of performing 10^{18} double-precision floating-point operations per second, are essential tools for scientific discovery because they can address impracticable problems for petascale supercomputers. However, exascale supercomputers are heterogeneous, integrating various accelerators, including different GPU-based platforms. Unfortunately, these heterogeneous supercomputers introduce challenges that need to be addressed to harness the full potential of exascale computing.

Challenge 1: Provide performance portability. The performance portability frameworks necessary to address hardware heterogeneity often introduce overhead costs and compromise performance. While portability solutions like Kokkos (1), Raja (2), and OpenMP (3) provide frameworks that ad hoc optimization of codes for diverse machines cannot accomplish, they do not necessarily fit onto every hardware platform. Programming a GPU like a CPU can lead to significant performance loss compared to directly targeting a GPU. Moreover, a portability layer can often obscure the realities of hardware characteristics, resulting in developers unknowingly creating inefficient software.

Challenge 2: Preserving accuracy. Performance portability alone is insufficient to maximize the use of heterogeneous hardware. Hardware manufacturers increasingly prioritize accelerating Artificial Intelligence (AI) and Machine Learning (ML) workloads with features such as half-precision and bfloat16 number formats that use less than 32 bits to represent real numbers. While these lower precision number formats are potentially useful for decreasing data movement and storage, they also reduce accuracy. This accuracy loss can be so significant in HPC simulations that it compromises scientific discovery. Methods for using alternative number formats while mitigating accuracy loss are necessary for HPC applications to take advantage of the new AI/ML-targeted features.

Challenge 3: Pruning redundancy. Storage costs and I/O overhead are fundamental problems that worsen as system scale increases. Reducing data movement with lower precision formats can help but is insufficient if using lower precision formats to increase problem size. I/O patterns such as checkpointing, where many processes distributed across multiple nodes simultaneously write vital data structures to disk, induce contention for the limited I/O resources. Checkpoint data may contain repeating patterns, leading to spatiotemporally redundant data. Existing solutions, such as asynchronous multi-level checkpointing and incremental checkpointing techniques, help mitigate the I/O overhead and storage problems but need to be revised for emerging use cases like adjoint computations, which perform more frequent checkpoints. Incremental storage techniques that eliminate temporal and spatial redundancy are necessary to reduce I/O overhead and storage costs.

1.2 Thesis Statement

Providing performance portability, preserving accuracy with lower precision number formats, and pruning redundancy form a chain of challenges we must overcome to develop HPC software for large heterogeneous supercomputers. Heterogeneous supercomputers necessitate performance portability solutions to support the many

combinations of hardware. However, portability solutions do not leverage more AI/ML-targeted hardware features, such as lower precision number formats. Reducing data movement and storage costs with lower precision number formats helps alleviate I/O bottlenecks and enables larger-scale simulations, ultimately changing the context rather than solving the I/O issue. Methods for pruning spatiotemporally redundant data are needed to mitigate I/O and storage overhead.

We claim that developing HPC software on exascale supercomputers requires a cohesive approach to provide performance portability, increase problem size while preserving accuracy, and enhance storage efficiency by pruning redundancy. To validate this thesis statement:

- We port the high-performance Vector Particle-In-Cell (VPIC) code using the Kokkos portability framework as a case study for porting an existing application to heterogeneous systems and mitigating portability costs (4);
- We evaluate half-precision and fixed-point representations for increasing simulation scale and demonstrate that it is possible to maintain scientific accuracy with lower precision (5; 6);
- We develop a massively parallel data deduplication library that takes advantage of spatiotemporal redundancy and extracts a compact metadata representation of repeating patterns within the data (7).

1.3 Contributions

This dissertation makes three contributions: providing performance portability for HPC software, preserving accuracy using lower precision formats, and pruning redundant data with deduplication.

1.3.1 Providing Performance Portability for HPC Software

To address the challenge of providing performance portability for HPC software, we port the Vector Particle-In-Cell (VPIC 1.2) plasma physics code, which was traditionally optimized ad hoc each time a new supercomputer was made available, to a single version for heterogeneous systems using the Kokkos performance portability framework (VPIC 2.0) (Section 3.1). We compare the runtime, scalability, and portability of VPIC 2.0 versus VPIC 1.2 across several GPU-based and CPU-based platforms (Section 3.2). Using insights from our performance evaluation, we identify key code transformations for mitigating portability layer overhead, such as adjusting sorting order and leveraging vectorization abstractions (Section 3.3). We identify the benefits and broader applicability of our performance portability study and our optimizations for mitigating portability overhead. We demonstrate how VPIC 2.0 achieves performance parity with VPIC 1.2 on CPUs and GPUs. (Section 3.4).

1.3.2 Preserving Accuracy with Lower Precision Formats

To fully leverage AI/ML-targeted lower precision number formats while maintaining scientific accuracy, we develop a set of optimizations to the particle format that enable large simulations in VPIC using reduced precision and control numerical accuracy by leveraging a local coordinate system (Section 4.1). We present a detailed memory usage and runtime performance analysis of our optimizations on CPU-based and GPU-based platforms (Section 4.2). We demonstrate that our optimizations can maintain scientific accuracy while enabling larger simulations (Section 4.3). We identify our techniques' impacts and broader applicability for improving simulation scale and performance using lower precision formats in grid-based simulations (Section 4.4).

1.3.3 Pruning Redundant Data with Deduplication

To mitigate storage costs and I/O overhead for checkpoints, we prune spatiotemporal redundancy in data during the writing process. We introduce a series of design principles at the core of our GPU-accelerated data deduplication method (Section 5.1). We propose a highly parallel algorithm based on these principles that coalesce contiguous chunks into a hierarchical set of repeating patterns using Merkle trees (Section 5.2). We illustrate the implementation of our algorithms by proposing a research prototype that leverages Kokkos for performance portability (Section 5.3). We demonstrate the benefits of our solution for a graph application through extensive experiments, comparing our method with state-of-the-art incremental checkpointing and compression methods for various graphs (Section 5.4). We identify the impacts of our deduplication work and its broader applicability for improving incremental checkpointing performance and reproducibility. Our solution eliminates spatiotemporal redundancy in checkpoints by identifying contiguous patterns in the data. (Section 5.5).

1.4 Organization

The remainder of this dissertation details how we overcome each challenge in the chain: providing performance portability, leveraging lower precision number formats to increase performance and scale while maintaining scientific accuracy, and mitigating I/O overhead and storage costs from large-scale simulations by pruning spatiotemporal redundancy. Chapter 2 presents each challenge's background and existing work. Chapter 3 describes how we port a particle-in-cell simulation and mitigate portability overhead to produce a portable version of the code that achieves performance parity with the existing code on CPUs and high performance on GPUs. Chapter 4 outlines how we apply lower precision numerical formats to grid-based simulations to enable larger-scale simulations while maintaining scientific accuracy.

Chapter 5 introduces our incremental data deduplication technique for removing spatiotemporal redundancy to mitigate I/O overhead and reduce storage costs. Chapter 6 summarizes our contributions, insights, and future research directions.

Chapter 2

Background and Existing Work

To fully understand the contributions of this dissertation we must review the current state-of-the-art for performance portability, mixed precision, and data storage. We discuss the existing performance portability solutions and the difficulties in evaluating portable applications. We review different low-precision number formats and existing work on using mixed precision in AI/ML and HPC applications without sacrificing accuracy. We examine existing incremental storage and compression techniques for reducing checkpointing overhead and storage costs.

2.1 Performance Portability Solutions

Growing hardware diversity poses challenges for HPC as the community moves towards exascale machines. The existing development strategy of refactoring and optimizing code for each target platform achieves excellent performance but becomes impractical as the diversity of CPUs and GPUs increases. Portability solutions are required to address these challenges in HPC software development.

2.1.1 Low-level and High-level Portability Solutions

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 it uses other tools as a backend.

Low-level tools (such as OpenMP (3), OpenACC (8), OpenCL (9), HIP (10), SYCL (11), or DPC++ (12)) 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. A runtime library allows for more dynamic control over parallelization and querying of device capabilities and settings. These annotation-based tools enable parallel execution with minimal changes to the original serial code, but are heavily reliant on the compiler, can be difficult 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 codes. These frameworks are similar to CUDA in how they are programmed. Developers define parallel kernels in special functions marked for execution. Each call of a computational kernel executes the kernel in parallel by a user-defined number of threads rather than by a single thread as in C/C++. The developer must explicitly allocate memory on the device and copy data from the host to the device. This programming style can be difficult for developers due to the different programming styles, additional memory management challenges, and knowledge of the hardware 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 to the other solutions.

High-level tools (such as Kokkos (1), RAJA (2), OCCA (13), and Chapel (14)) 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 how tightly coupled the libraries are. Kokkos keeps the core programming model in a single library. RAJA, on the other hand, keeps the execution abstractions, array abstractions, memory management, and metaprogramming capabilities as 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 backends but introduce more levels of abstraction.

2.1.2 Building on Portability Solutions

Several works focus on extending portability frameworks to improve performance in key areas. Some works implement portable SIMD abstractions (15; 16) to improve on compiler-based auto-vectorization. Abstractions for controlling memory layouts have demonstrated significant benefits for HPC application (17; 18). The Cabana project extends the Kokkos framework to apply these memory layout optimizations while remaining portable (19). The LLAMA project provides abstractions for applying custom memory layouts to more complex data (20). However, these tools do not address cases where memory access patterns change as the applications execute.

2.1.3 Evaluating Portability

Portability introduces another dimension to performance analysis. Performance results between different hardware platforms, in many cases, cannot be directly compared (e.g., CPUs vs GPUs). High performance on one platform does not guarantee similar results on another platform. Differences between hardware can have a large impact on performance. For example, chips with non-uniform memory access (NUMA) regions may perform poorly if the application does not take NUMA into account when handling data placement. Thus, cross-platform testing is becoming increasingly valuable. Comparing performance across different portability solutions has already been the subject of previous studies (21; 22; 23; 24; 25).

Comparing different performance portability solutions also complicates analysis. This is particularly important for high-level tools that have multiple possible backends. One approach by Pennycook, et al (23) outlines a clear definition for performance portability and proposes an empirical metric defined as the harmonic mean of the application or architecture efficiency. Productivity is another aspect frequently used when evaluating codes (The Three Ps: Performance - Portability - Productivity). Unlike performance and portability, productivity is subjective and difficult to quantify in a broadly useful way. One method for quantifying productivity is to look at maintenance effort. Work by Harrell, et al (24) presents an approach to quantify maintenance effort for multi-platform code bases, in which they defined a platform divergence metric that was proposed as an analog to the amount of ongoing maintenance work. Tools such as Intel Code Base Investigator (CBI) (26) can determine the divergence of portable codes based on CPU and GPU code paths.

2.2 Mixed Precision for HPC

Using a mixture of different floating-point formats for speeding up applications has a long history (27; 28; 29). HPC hardware can perform single-precision floating-point math twice as fast as double-precision math. Switching from double-precision to single-precision allows compute-bound applications access to more compute power. Memory-bound applications benefit from halving the amount of data movement while making more data available in caches. However, leveraging lower precision requires careful attention to algorithm design. Simply reducing the precision can lead to catastrophic numerical errors. Even lower precision formats have been introduced for the growing need for computation and bandwidth. AI/ML applications are a major driving force for developing mixed precision algorithms.

2.2.1 Using Low Precision Formats in AI/ML

AI and machine learning can tolerate even lower accuracy during training (30), leading to half-precision hardware support (such as Tensor cores) and software. Half-precision uses 5 bits for the exponent and 10 bits for the mantissa. Other formats such as Google's Bfloat16 (31), which sacrifices precision for range (8-bit exponent and 7-bit mantissa) and TensorFloat32 (32), which sacrifices storage space (8-bit exponent and 10-bit mantissa), have been implemented to control the range and precision further. Another option is fixed-point formats where the number of bits for the integer (m) and fractional (n) portions is defined by the format ($Qm.n$). Fixed-point allows for more control over range and precision (33) but can be restrictive compared to floating-point formats. Analysis of the precision for these formats is necessary due to the differences between the standards and the implementation (34).

2.2.2 Adoption of Low Precision Formats in HPC

Most half-precision usage outside of AI has been in numerical linear algebra, where techniques such as iterative refinement can compensate for lost accuracy (35; 36; 37). Grid-based techniques such as particle-in-cell codes can regain accuracy by switching to a local coordinate system. This technique has been used in PIC codes like VPIC to mitigate the accuracy loss from using single-precision floating-point values (38). Some works have successfully used fixed-point and mixed single/double precision for molecular dynamics simulations (39). Other works analyze codes to determine which sections can use reduced precision (40; 41). However, half-precision and other lower-precision formats remain difficult to use on many platforms due to the challenges of maintaining accuracy.

2.3 Data Movement and Storage Solutions

Data movement and storage challenges continue to grow as the scale of supercomputers increases. Checkpointing suffers from high I/O overhead and storage limitations which has led to extensive research into techniques for mitigating overhead and storage costs (42). Incremental storage techniques reduce storage costs by identifying and storing only the differences between checkpoints. Data compression techniques encode data with fewer bits than the original representation in either a lossy or lossless manner.

2.3.1 Incremental Storage

Incremental storage is a well-known technique to accelerate I/O and reduce storage space utilization. In addition to checkpointing, it is applied in many other scenarios: versioning and incremental snapshots of file systems, virtual machine images, and block devices. They use either dirty page (block) tracking or deduplication (43; 44; 45; 46) to identify incremental differences. Dirty page tracking for host memory can

be accelerated by the OS and hardware using techniques such as user-level faults that avoid expensive context switches triggered by more conventional methods that trap the SEGFAULT signal. However, dirty-page tracking requires specialized kernel support and is unavailable on all platforms. Furthermore, they are typically limited to memory page granularity (e.g., 4 KB), which limits their deduplication potential (e.g., single-byte changes or writing identical data to the same address can mark an entire page dirty). Such techniques are unavailable on GPUs because the GPU drivers handle the memory virtualization fully transparently (47; 48). Complementary to incremental storage is the problem of how to re-assemble checkpoints from differences, which involves metadata organization, indexing and search techniques (49; 50). Several works focus on enabling checkpointing for GPU applications (51; 52; 53; 54). However, each has drawbacks. Some works (51; 54) only perform essential temporal deduplication. Others (52; 53) only perform deduplication at the page or variable level. Methods such as libhashckpt (55) use a hybrid method with multiple change detection systems to reduce the checkpoint size.

2.3.2 Compression Techniques

Another alternative to incremental checkpointing is checkpoint compression, both lossless (56; 57) and lossy (58; 59). Lossless compression methods can reconstruct the data without loss of information. Lossy compression methods use approximations of the data and are of particular interest thanks to the greater compression ratios for floating-point data (60; 61; 62; 63; 64). Typical compression algorithms prioritize decompression performance, assuming that compression is a relatively infrequent operation, which does not hold in our high-frequency checkpointing scenario. Furthermore, many compression algorithms cannot leverage the temporal redundancy of data that evolves in time.

Chapter 3

Providing Performance Portability for HPC Software

The increasing heterogeneity in HPC presents a complex challenge to software development. As more vendors develop their exascale hardware, the potential CPU and accelerator combinations multiply, making the task of supporting software for heterogeneous hardware increasingly complex. Yet, this complexity must be overcome to achieve high performance on large-scale systems. While practical, the ad hoc optimization of HPC software for each target platform is not a sustainable solution. This has prompted the development of performance portability solutions, enabling a single source code to run on different hardware and simplifying the process. Portability solutions such as OpenMP, OpenACC, SYCL, and Kokkos enable code to run on both CPUs and GPUs from all major hardware vendors. These solutions greatly simplify porting and developing code for new hardware. However, porting code is not trivial and introduces additional performance analysis and optimization challenges. Comparing performance on multiple platforms requires more testing and hardware knowledge to understand application performance. Portability solutions also introduce overhead and conceal important hardware characteristics from developers, which leads to lower performance. For example, GPUs prioritize

memory bandwidth, while CPUs favor lower latency. Memory access patterns that perform well on CPUs may perform poorly on GPUs.

To maximize the efficient usage of heterogeneous hardware, we leverage portability layers, expand performance analysis to include portability, and take portability overhead into account. We port an existing HPC code to heterogeneous systems. We examine and address different challenges of the porting process and provide insights for software development on heterogeneous hardware. Our results demonstrate the performance parity of our portable code with its ad hoc optimized version. Our contributions towards performance portability are:

- We port the existing optimized VPIC 1.2 code to heterogeneous supercomputers using the Kokkos performance portability framework (VPIC 2.0). (Section 3.1)
- We evaluate and compare the runtime, scalability, and portability of VPIC 1.2 and our VPIC 2.0 port across several GPU and CPU-based platforms. (Section 3.2)
- We determine key code transformations for mitigating portability layer overhead, such as adjusting sorting order and leveraging vectorization abstractions. (Section 3.3)
- We identify the importance and broader applicability of our performance portability study and our optimizations for mitigating portability overhead. (Section 3.4)

3.1 Porting Applications to Accelerators

3.1.1 Case Study: Vector Particle-In-Cell

Vector Particle-In-Cell (VPIC) (65) is a high-performance particle-in-cell (PIC) code that models plasma phenomena such as magnetic reconnection (66), fusion (67), solar weather (68), and laser-plasma interactions (69). The original VPIC code (referred

to throughout as VPIC 1.2) was built to minimize data movement and maximize performance with platform-specific vector intrinsics; given a new HPC platform, the ad hoc re-engineering of the code guaranteed the continuation of performance. For example, VPIC was optimized for the Xeon Phi Knights Landing processor with AVX-512 intrinsics for maximum performance and use of the high bandwidth memory available on the chip (70). However, the increasing heterogeneity of emerging exascale platforms makes optimizing the VPIC code for every available computing platform increasingly impractical. Optimizing VPIC for the Xeon Phi Knights Landing chip added support for the AVX-512 SIMD extensions, but additional AVX-512 extensions and varying hardware support prevent the optimizations from working on non-Xeon Phi processors. Re-optimizing the code for new AVX-512 variants would introduce significant development effort and maintenance costs. A new multi-architectural VPIC code is needed to ensure portability and performance across exascale platforms. To develop VPIC 2.0, we use the Kokkos performance portability framework.

3.1.2 Kokkos Portability Elements

Kokkos (1; 71) is a suite of libraries and abstractions for developing high-performance C++ applications that are portable across many platforms, including all major CPUs and accelerators. 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.

The Kokkos ecosystem has three major portability elements: the Kokkos kernel libraries, the Kokkos tools, and the Kokkos core programming model. The Kokkos kernel libraries provide an interface for common linear algebra (e.g., BLAS, SparseBLAS) and graph algorithms with optimized backends, such as the MAGMA linear algebra library. The Kokkos tools comprise an interface and a set of utilities

that connect to the Kokkos runtime and enable lightweight debugging and profiling. Individual kernels or specified regions composed of many kernels may be profiled. Connectors for several common profilers and tools, such as Caliper and PAPI, are included for advanced debugging and analysis. The Kokkos programming model enables developers to control memory layout, data placement, parallel execution, and execution devices. Kokkos maps specified data characteristics and execution policies to the target hardware backends using a set of rules for automatically determining different hardware-specific parameters, removing the user from the mapping task. Kokkos provides both data structure and parallel execution abstractions. Data structures have three key components for defining the memory space, layout, and memory traits. The execution abstractions, including space, pattern, and policy, define the parallel executions. We choose Kokkos as our portability solution for its vendor agnostic approach and ecosystem of support libraries.

3.1.3 Porting VPIC with Kokkos

We define and follow a rigorous five-step methodology to port VPIC onto Kokkos. The methodology is easily modified for other portability solutions.

Step 1: We analyze application and identify key functionality and data structures.

Step 2: We convert key data structures to use Kokkos data structures (both host and device).

Step 3: We refactor code to synchronize data between the Kokkos and original data structures while also considering legacy support.

Step 4: We parallelize key functionality with Kokkos.

Step 5: We evaluate performance and portability

VPIC operates by defining a simulation space divided into a grid of cells and modeling particle movement across these cells in an iterative fashion. For Step 1,

we analyze VPIC's execution flow and data structures. In other words, particles are distributed across an n-dimensional (n-D) space that is decomposed into a n-D grid. The n-D grid is rectangular with fixed Cartesian geometry by default. Each simulated particle is a macroparticle with a defined weight. For each time step, particles move in accordance with the forces acting on them from the fields. The particle movement induces electrical current, which changes the fields. Four key data elements in VPIC—the macroparticles, electromagnetic (EM) fields, interpolators, and current accumulators—must be moved to Kokkos Views to ensure portability.

Porting VPIC to use Kokkos requires storing the data structures in Kokkos Views to enable GPU execution and leverage the Kokkos memory abstractions. In Step 2, we convert the macroparticles, fields, interpolators, and current accumulators to store data using Kokkos Views. We replace the array of structures with a two-dimensional View where one dimension corresponds to the array index and the other dimension maps to the different member variables. Care must be taken during the conversion to Kokkos Views because the conversion approach directly affects the memory layout. The memory layout is important for performance, with different hardware performing better with different layouts and parallelism techniques (i.e., thread parallelism and vector parallelism) (17). There are two major memory layouts that Kokkos Views support: array of structs (AoS) and struct of arrays (SoA). The AoS stores multiple instances of a `struct` in an array such that they are contiguous in memory. The SoA layout, on the other hand, stores multiple instances of a `struct` in a single `struct` with arrays for each structure member. Each member stores values in a contiguous array. The AoS and SoA layouts are effectively the same as row and column-major order for multidimensional data such as the grid. The interpolators and accumulators are data structures comprised of only floats (18 for interpolators and 16 for accumulators).

Data movement in Kokkos is done explicitly. In Step 3, we refactor the code to move data to the correct execution space. We restructure VPIC to avoid data transfers due to the high cost of data movement in heterogeneous systems. By default, data

and simulation settings are set on the host CPU during initialization. Simulation data is then copied to the execution device and kept there throughout the simulation. Data movement between device and host is only done for communication and user diagnostics (output). On CPU-only platforms, where the host and device are the same, data copies are eliminated automatically. This code structure ensures high performance on GPUs, while Kokkos prevents any unnecessary data movement for CPU-only systems.

The particle push is often the most costly part of the simulation. In Step 4, we port the code piece by piece. Starting with the particles, we parallelize each part of the code touching the particles, such as the particle push and particle communication. We then convert the code with special care to minimize data transfer between the host and the device.

In step 5, we evaluate the code's portability and compare performance with the original VPIC 1.2 code. Based on the evaluation, we further port or optimize parts of the code.

3.2 Performance Portability Evaluation

To evaluate VPIC 2.0, we first compare its performance against the existing VPIC 1.2 code in both weak and strong scaling scenarios across various platforms. Second, we assess the code's portability by running a benchmark problem across several CPU and GPU-based platforms. Third, we evaluate the effort required for porting VPIC to heterogeneous systems using Kokkos. Finally, we demonstrate the impact of mitigating portability overhead in sorting and vectorization.

3.2.1 Test Platforms

We present results from five GPU-based platforms and four CPU-based platforms. The platforms are sorted chronologically by release year.

GPU-based Platforms

NVIDIA Pascal P100 (2016) experiments are run on the Kodiak computer at Los Alamos National Laboratory. This computer consists of 133 nodes, each with an Intel Xeon E5-2695 v4 CPU with 256 or 512 GB of DRAM and four NVIDIA Tesla P100 GPUs on a Mellanox InfiniBand EDR Fat-Tree interconnect.

NVIDIA Volta V100 (2017) experiments are run on the Sierra supercomputer at Lawrence Livermore National Laboratory. Sierra is a 125 petaflop, IBM Power Systems AC922 hybrid architecture system comprised of 4320 nodes, each with two 22-core IBM Power9 processors running at 2.3–3.8 GHz with 256 GB of RAM and four NVIDIA V100 GPUs each with 16 GB of HBM2 DRAM.

NVIDIA Turing RTX 5000 (2018) experiments use the RTX partition of the Frontera cluster (72) at the Texas Advanced Computing Center. This partition consists of nodes containing two Intel Xeon CPU E5-2620 v4 running at 2.10 GHz with 128 GB of RAM. Nodes are connected by Mellanox Infiniband FDR at 56 Gb/s configured in a fat-tree topology with a 4:1 oversubscription. Each node also contains four NVIDIA Quadro RTX 5000 GPUs with 16 GB of GDDR6 DRAM.

AMD Vega Radeon VII (2019) experiments are executed on a single node consisting of two AMD EPYC 7302 16-core processors with 64 GB of DRAM and four AMD Radeon VII GPUs with 16 GB of HBM2 DRAM each.

NVIDIA Ampere A100 (2020) experiments are completed on the NVIDIA DGX SuperPod (73), Selene, at NVIDIA. Each node within this system contains dual AMD EPYC 7742 CPUs with 64 cores per socket running at 2.25 GHz and 2 TB of DRAM.

The nodes are connected using Melanox Infiniband HDR at 200 Gb/s within a fat-tree topology configured for a 5:4 oversubscription. Each node contains eight NVIDIA A100 GPUs with 80 GB of HBM2 DRAM each.

CPU-based Platforms

Intel Knights Landing (2016) experiments use the KNL partition of the Trinitite testbed at Los Alamos National Laboratory. This partition comprises 100 nodes of a Cray XC40 system configured with a 3D Aries Dragonfly interconnect. Each node contains one 68 core Knights Landing processor with 16 GB of high-bandwidth MCDRAM and 96 GB of DRAM.

AMD Zen 2 (2019) experiments are run on the Bell cluster at Purdue University (74). Each node contains two AMD EPYC 7662 64 core processors operating at 2.0 GHz and 256 GB of DRAM. Each node is connected with Mellanox Infiniband HDR at 100 Gb/s in a fat-tree configuration that is oversubscribed 3:1.

Intel Cascade Lake (2019) experiments are executed on the Frontera cluster at the Texas Advanced Computing Center. Each node in Frontera contains two Intel Xeon Platinum 8280 processors with 28 cores operating at 2.70 GHz and 192 GB of memory. The nodes are connected with Mellanox Infiniband HDR at 100 Gb/s. The interconnect is configured in a fat-tree topology with an oversubscription factor of 11:9.

Fujitsu A64FX (2020) experiments are run on the Ookami cluster at Stony Brook University. Ookami is an HPE Apollo 80 system with 174 nodes. Each node contains a Fujitsu A64FX Arm processor (75) with 32 GB of HBM2 and a 512 GB SSD. Nodes are connected in a 2-level tree using non-blocking HDR-200.

3.2.2 Weak Scaling

For a test problem to study weak scaling, we adapt a large 3D simulation of stimulated Brillouin scattering (SBS) in a single laser speckle (76) relevant for inertial confinement fusion experiments. Running the full simulation to completion would be prohibitive in our scaling tests, so we modify the simulation to be much shorter while preserving the numerical properties of the full simulation. Instead of a single very long laser pulse with a slow ramp in intensity, we use two counter-propagating laser pulses that ramp up linearly in field strength (quadratically in intensity) from zero at the start to twice the intensity of Ref. (76) at the end of the simulation. The simulation is stopped before the two pulses collide. This arrangement gets particles moving near the start of the simulation and thus is computationally similar to a full-length simulation, and less like the uniform cold plasma of early time steps.

We simulate a fully ionized nitrogen plasma of temperatures $T_e = 600$ eV and $T_i = 150$ eV, with density $n_e = 0.05 n_{cr}$, where $n_{cr} = m_e \varepsilon_0 \omega_L^2 / e^2$ is the critical density for a laser with angular frequency $\omega_0 = 2\pi c/\lambda_L$. The plasma fills a simulation volume of $80\,\mu\text{m} \times 20\,\mu\text{m} \times 20\,\mu\text{m}$. Two identical laser pulses are incident on the plasma from the positive and negative x direction. Each pulse is a focused Gaussian beam of wavelength $\lambda_L = 527$ nm and beam waist $w_0 = 4\,\mu\text{m}$, focused at the center of the simulation volume. The pulses increase linearly in field strength over time to a maximum intensity $I_0 = 5 \times 10^{15}$ W/cm^2 at the end of the simulation, $t_{\text{stop}} = (80\,\mu\text{m}/2)/c$. The grid has a resolution of 6000x1500x1500 for a cell size very near the Debye length and a domain decomposition of 60x15x15 across 13,500 GPUs. Tests smaller than this full scale simulate a proportional volume fraction and use only one laser pulse. Larger tests expand the simulation volume.

Tests using 32 or fewer GPUs have an adjusted number of particles per cell to account for boundary conditions, making the maximum number of particles a rank has at initialization equal for all tests. There is a "buffer" region at the simulation boundary where particles cannot be placed to separate particle and field boundary

conditions. We run the weak scaling on three cutting-edge, GPU-based platforms (i.e., Sierra with Power 9 CPUs and V100 GPUs, Selene with EPYC 7742 CPUs and A100 GPUs, and the Frontera with Xeon E5-2620 v4 CPUs and RTX 5000 GPUs). The reported runtime is measured on up to 17,100 GPUs on Sierra, 4,096 on Selene, and 256 on Frontera; the times are normalized to the 64 GPU runs because boundaries mean smaller runs have less communication per GPU. Figure 3.1 demonstrates the results from the three GPU-based platforms and their different GPUs, spanning both commercial and consumer-grade cards. VPIC runtime, in general, is very consistent. Sierra (V100) and Selene (A100) measurements are from a single run.

The weak scaling results show excellent performance across the GPU-based platforms. The Sierra supercomputer's NVIDIA Volta (V100) hardware shows the best scaling performance. The test on Sierra's 17,100 GPUs (shy of the full machine because of some offline nodes) shows only a 4.8% slowdown versus the 64 GPU run. This is a testament to the capability of VPIC 2.0 to preserve performance while gaining in portability. Other platforms (i.e., Selene and Frontera) scale about half as well as Sierra on the configurations that were made available to us for our testing. These platforms are newer and likely have less mature MPI implementations.

We demonstrate the ability of VPIC 2.0 to run on CPU-based platforms for two platforms: the Bell cluster at Purdue University using EPYC 7662 CPUs and the Frontera cluster with Xeon 8280 CPUs only (without the use of GPUs). Figure 3.2 shows the weak scaling on the two platforms. The runtime values are the average of five runs and are normalized to the 64 CPU runs, as in the GPU-based testing. The presented runtimes are the average of five runs. For our tests, we run multiple MPI ranks per socket. Thus, the per-socket communication is constant for 4 or more sockets. While these performance results are slower than the GPU runs, they still add valuable insights from the perspective of our portability study. Furthermore, as accelerators become the driving platforms for large-scale simulations, the community gathering around VPIC is expected to use GPU-based platforms rather than CPU-only ones.

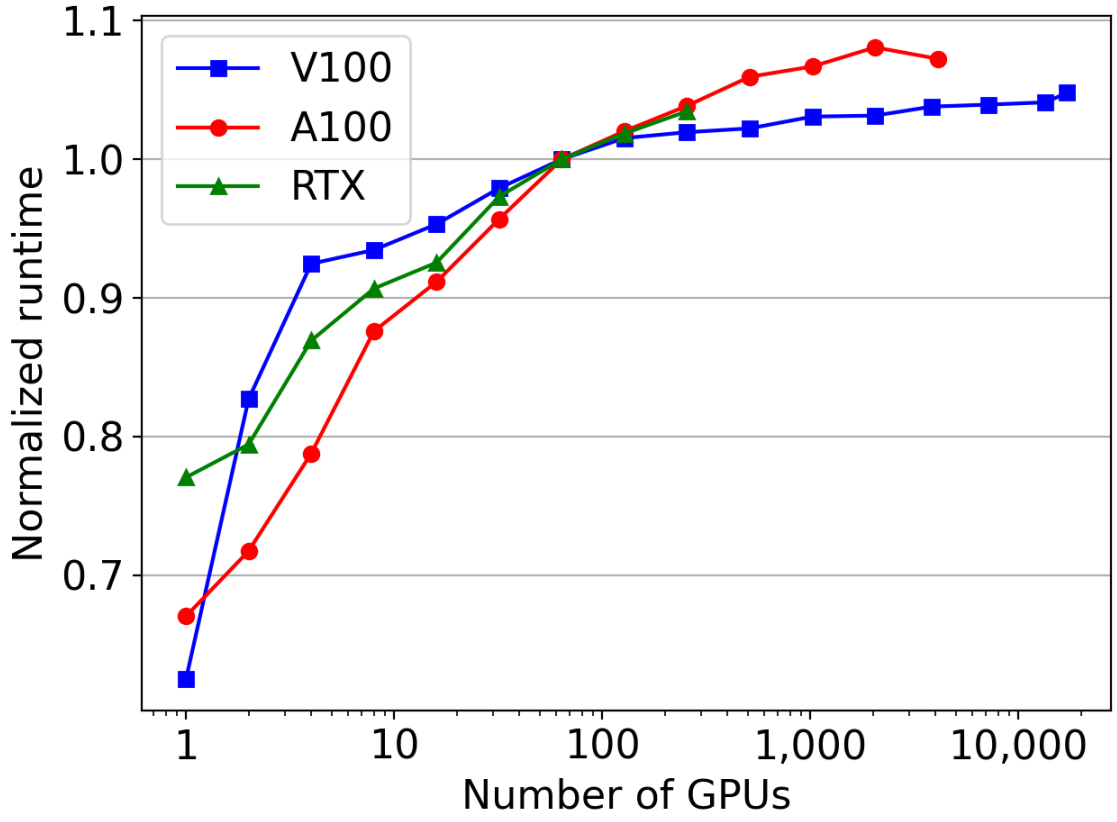


Figure 3.1: Weak Scaling on three GPU-based platforms (i.e., Sierra with Power 9 CPUs and V100 GPUs, Selene with EPYC 7742 CPUs and A100 GPUs, and the Frontera with Xeon E5-2620 v4 CPUs and RTX 5000 GPUs). Runtime is normalized to the 64 GPU runs, below which boundaries cause a reduction in per GPU communication.

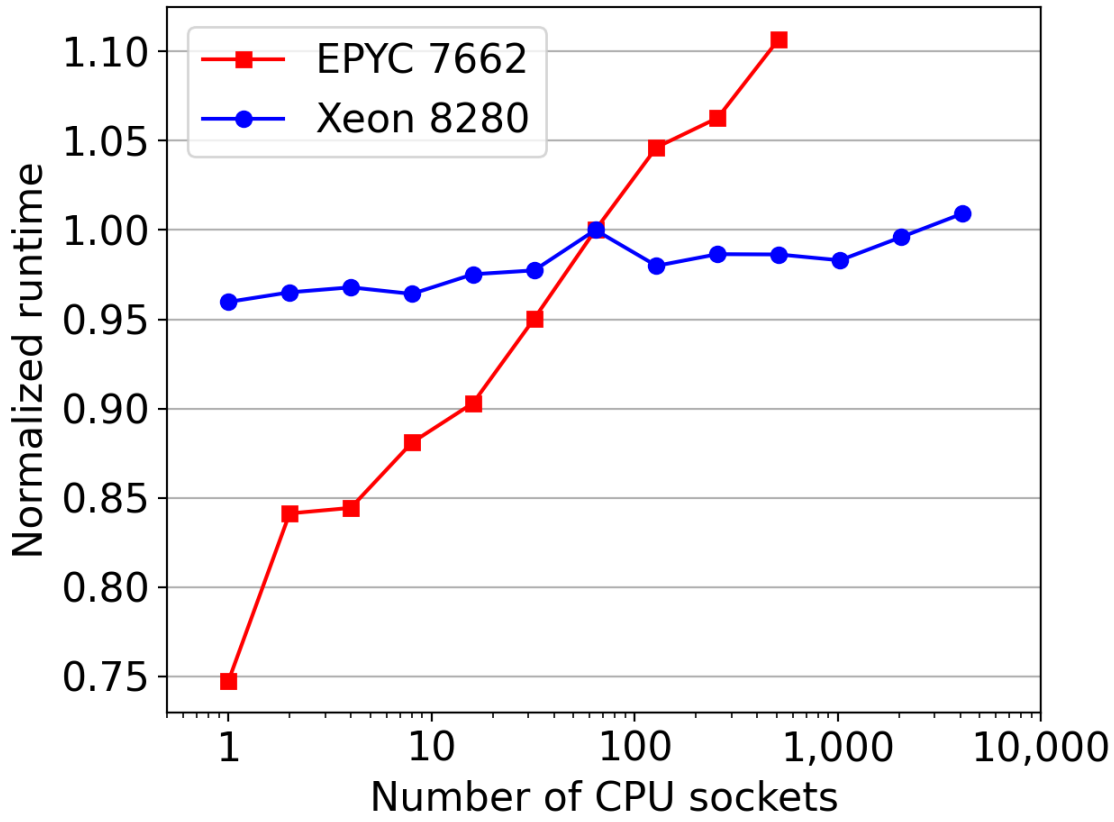


Figure 3.2: Weak scaling on two CPU-based platforms (i.e., Bell cluster at Purdue University using EPYC 7662 CPUs and the Frontera cluster with Xeon 8280 CPUs only). Runtimes are normalized to the 64 CPU runs to match the GPU tests.

3.2.3 Strong Scaling

The cache increase on the A100 to 40 MB from 6 MB on the V100 has made super-linear speedup possible which may change how these two cards are best utilized in VPIC 2.0. A significant part of the particle push deposits the current generated from particle movement to grid points. VPIC 2.0 periodically sorts the particles by voxel such that the number of grid points being written to is minimal at any given time. Minimizing the number of grid points to write allows the grid points to be stored in cache, resulting in faster writes. The A100 gives rise to the possibility of storing the entire grid in cache, obviating the particle sort, and possibly producing super-linear speedup behavior in a strong scaling test as the total available cache increases.

We investigate this feature with tests similar to the weak scaling tests above, but with particle sorting turned off and only using the two larger supercomputers (i.e., Sierra with the V100 and Selene with the A100). The number of particles is kept constant while the grid size varies. Figure 3.3 shows the number of particle pushes per nanosecond as a function of grid size. The figure confirms that there is indeed a grid size where particle push performance increases dramatically, both for V100 and A100. Performance peaks for the V100 with approximately 4 particle pushes per nanosecond at 13,824 grid points. On the other hand, the A100 reaches 6 particle pushes per nanosecond with a grid size of 85,184. Furthermore, the performance jump occurs at roughly 6 times more grid points for A100 vs. V100, similar to the increase in cache. If this increase is from cache effects, we should see a super-linear speedup in the following strong scaling tests. We suspect the A100 performance drop for very few grid points (and thus extremely high particles per cell) results from colliding writes in the current deposition.

Carefully selecting the size of our grid to match the peak performance in Figure 3.3, we run strong scaling tests on Sierra with V100 GPus (i.e., from 1 to 32 GPUs), as shown in Figure 3.4, and on Selene with A100 GPUs (i.e., from 8 to 512 GPUs), as shown in Figure 3.5. As suggested by Figure 3.3, we do indeed observe super-linear

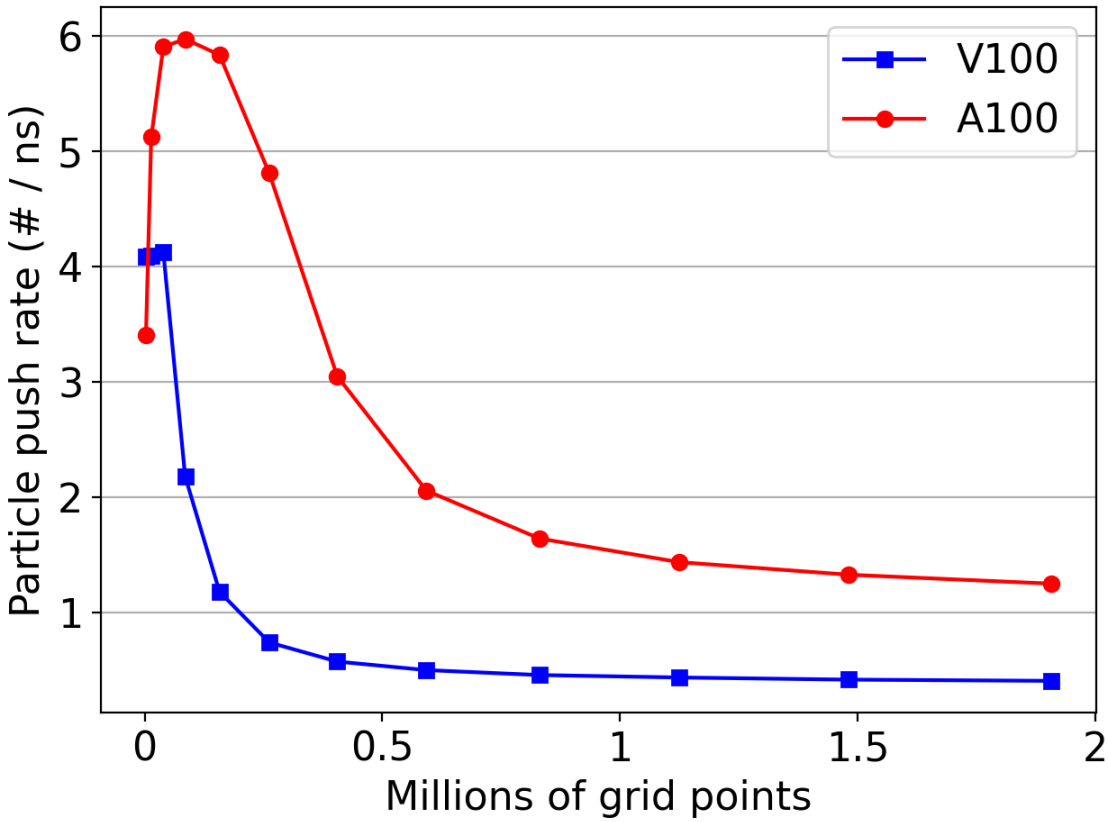


Figure 3.3: Number of particle pushes per nanosecond as a function of grid size on Sierra with V100 and Selene with A100. The total number of particles is held constant and all simulation time not in the particle push is excluded.

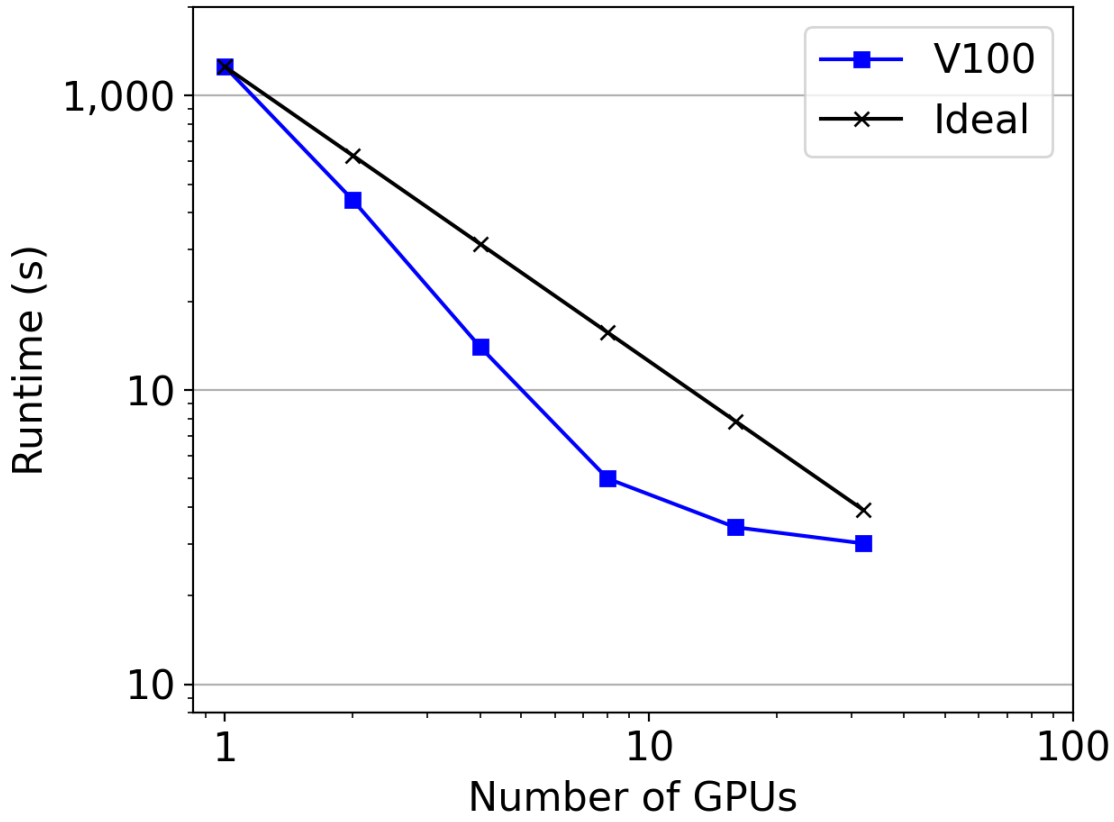


Figure 3.4: Strong scaling on the Sierra with V100 GPUs. As the number of GPUs increases, more of the grid is kept in the cache; as a result, we get a super-linear speedup. Communication costs eventually catch up with more than 16 GPUs.

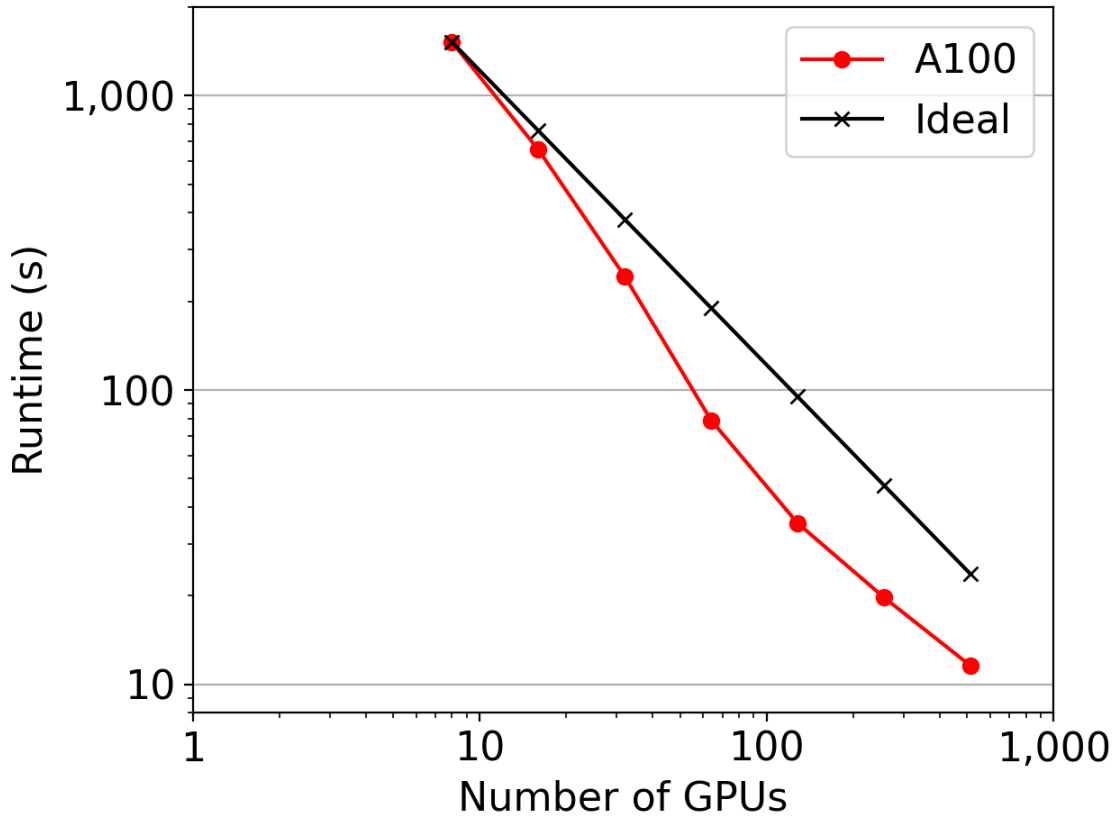


Figure 3.5: Strong scaling on the Selene with A100 GPUs. Thanks to the larger cache, the A100 GPUs maintain super-linear speedup with more GPUs than the V100 test.

scaling for both platforms. For V100, 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. On A100, 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.

The super-linear speedup from storing the grid in the cache has practical applications for certain class problems. For instance, simulations with fewer grid points or high particle per cell requirements (e.g., cross-beam energy transfer inertial confinement fusion simulations (77)) can cut their runtime down significantly. Smaller, non-full-scale simulations also benefit significantly from the super-linear speedup. The faster runtime enables domain experts and developers to quickly iterate through test parameters and tune their simulations before moving on to full-scale runs.

3.2.4 Portability Evaluation

Our portability study assesses how effectively VPIC 2.0, a single source code, can run across multiple platforms. To show portability, we present performance results on each platform listed in Sec. 3.2.1 for VPIC 2.0. VPIC 2.0 runs both AMD and NVIDIA GPU-based platforms as well as x86-64 and ARM CPU-based platforms. On CPU-based platforms, VPIC 2.0 is slower than VPIC 1.2. Figure 3.6 compares the different runtime measurements in seconds for the GPU-based platforms when running on 8 GPUs, with 1 MPI rank per GPU. The total simulation time is shown, predominately compute time, as communication time is minimal at 8 MPI ranks. There are no results for VPIC 1.2, as the original code was not developed for GPUs. For the NVIDIA-based platforms (i.e., P100, V100, RXT 5000, and A100), we see a large jump between P100 and V100, which can be explained by the increased atomic operation support in V100. We attribute the further jump from V100 to A100 to increased cache size and FP32 performance.

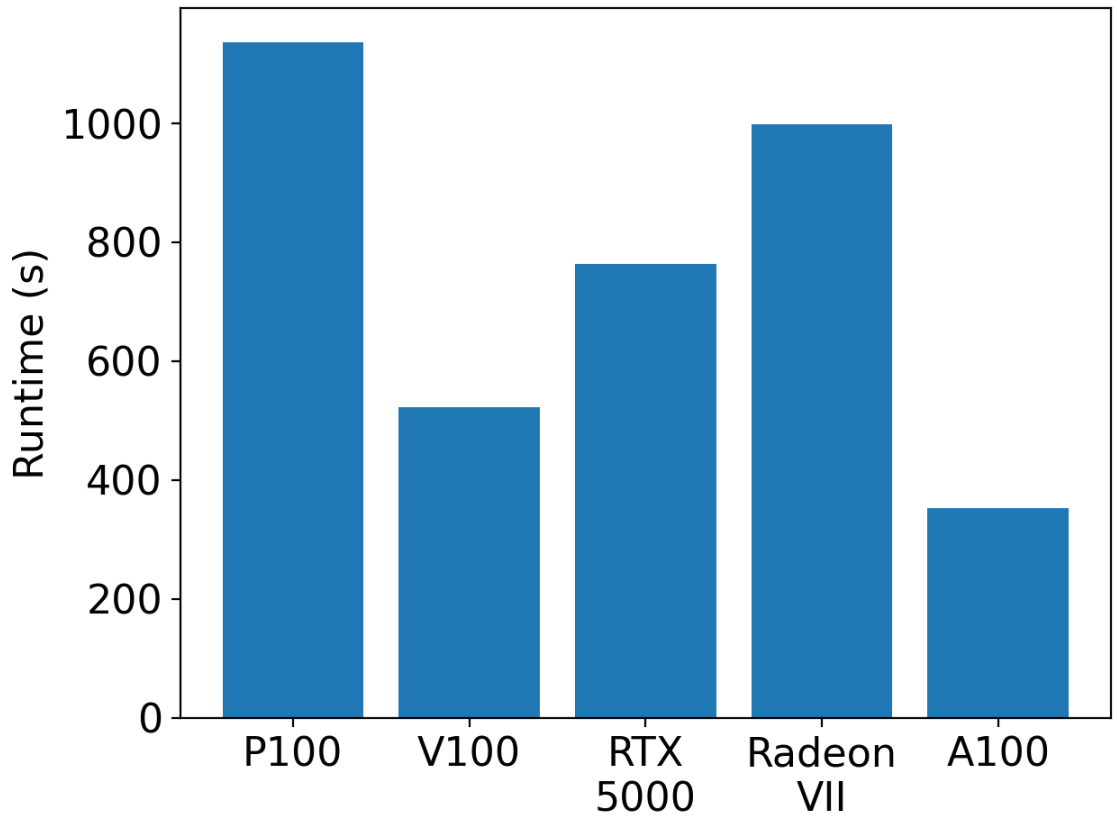


Figure 3.6: Total simulation time for small-scale GPU performance (8 cards). Results are ordered chronologically based on the GPU releases.

The RTX 5000, while slower than the V100, performs admirably for a workstation card and proves itself very capable of single-precision scientific simulations. The AMD Radeon VII underperforms when compared to the RTX 5000 despite having similar theoretical computing performance and superior memory bandwidth. The difference in performance can be attributed to two factors: (1) the maturity of the Kokkos AMD backend and (2) the fact that, to date, NVIDIA cards have been used as the development and testing platform for this work. The performance of VPIC 2.0 may further increase as Kokkos improves the AMD back-end support.

The runtime results comparing CPU-based platforms, including AMD, Intel, and ARM/Fujitsu chips, are shown in Figure 3.7. Results are shown for both VPIC 1.2 and VPIC 2.0. We compare 8 sockets of each platform for test normalization, representing a regime where MPI costs are non-negligible. While it is impressive that the same source code of VPIC 2.0 can run on such a diverse range of hardware, it is interesting to note that the AMD Zen 2 is the only platform that achieves performance comparable to that shown in the GPU study. We attribute the lower performance on CPU-based platforms to the limited CPU SIMD vectorization support in the version of Kokkos (Version 3). VPIC 1.2 is faster due to the hand-tuned SIMD code, which results in 8-wide vectors on the Xeon and EPYC platforms and 16-wide on KNL. Comparing just the time used in the particle push, VPIC 2.0 is about 8 times slower on Xeon, indicating no vectorization, and about 2 times slower on EPYC, indicating the use of 4-wide vectors. KNL is more complex, with VPIC 2.0 about 5 times slower, indicating some optimization. One could formulate a VPIC 2.0 code expression for further optimizations that can better vectorize. We do not address vectorization in this section as it introduces non-trivial code divergence, and this study focuses on achieving portability over platform-specific performance. We believe that some level of specialization will always be required for cutting-edge performance on a specific target platform. In many cases, the portability abstraction layer may be able to achieve this, but at other times, the burden will fall to the programmer.

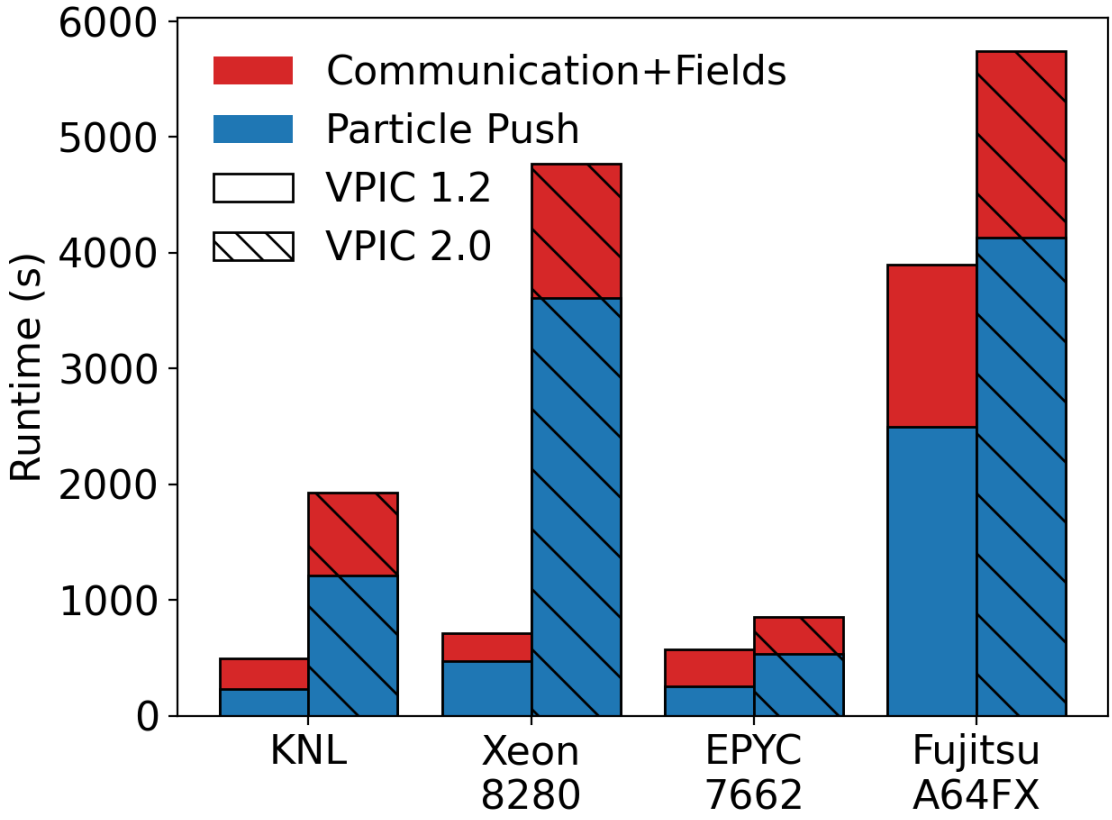


Figure 3.7: Compute and communication time for small-scale CPU performance (8 sockets). Results are ordered chronologically based on the CPU releases.

Our portability study relies on Kokkos to facilitate the code execution on various platforms. Therefore, the performance results inevitably depend on the priorities and maturity of the underlying framework and reflect the facets of the target hardware. Furthermore, it is clear from the presented results that the code maps best to GPU-based platforms as written in its present state. Proposals have been made to extend Kokkos SIMD support on CPUs, and we look forward to leveraging these improvements for VPIC 2.0.

3.3 Mitigating Portability Overhead

VPIC 2.0 is portable and performs excellently on GPUs but suffers from portability overhead and as a result is slower than VPIC 1.2 on CPUs.There are two major sources of portability overhead in frameworks such as Kokkos. First, there is abstraction overhead from mapping a single parallel programming model onto several different parallel programming APIs and hardware. Second, is inefficiencies from ignoring hardware properties. Identifying and mitigating portability overhead is a vital step for achieving high performance.

We identify two optimizations that are significantly impacted by portability overhead. Vectorization is a common parallelization technique that poses additional challenges for portability. Vectorization leverages single instruction multiple data (SIMD) instructions to exploit data parallelism for improved performance. However, vectorization is more difficult than thread or distributed parallelism. Any divergence of flow control can easily prevent vectorization. Kokkos, by default, relies on the compiler for auto-vectorization, which maximizes portability but is less effective than manually vectorizing the code with vector intrinsics or assembly. Manual vectorization is not portable and requires significant amounts of specialized code. This gap between auto-vectorization and manual vectorization leads to portability overhead.

Particle sorting is an important optimization in VPIC that controls the memory access pattern. VPIC's particle push kernel exhibits a gather-scatter pattern where

interpolator values are gathered to calculate the particle push and scatter the current generated by the particle movement to the cell where particle resides. Gather-scatter patterns can easily lead to poor memory access patterns that slow down execution. Gathering and scattering data from random locations will cause large amounts of cache misses and evictions. VPIC sorts the particles based on which cell they reside. This increases the likelihood of reusing cached data and minimizes the amount of data that needs to be loaded into the cache. However, the differences in memory architecture between CPUs and GPUs mean that normal sorting will not perform as well for GPUs. To mitigate portability overhead, we evaluate the spectrum of portable vectorization solutions and adjust the sorting order to maximize memory bandwidth for different hardware platforms.

3.3.1 Vectorization Strategy Evaluation

To mitigate the vectorization abstraction overhead, we evaluate the necessary code transformations and performance of three different vectorization strategies. The first strategy is to leverage the compiler's auto-vectorization capabilities to generate code. Auto vectorization requires no code changes, but the compiler may fail to vectorize code. Without any additional optimizations, VPIC 2.0 uses auto-vectorization. Manual vectorization with intrinsics or assembly is the opposite of auto-vectorization. The developer must refactor their code to operate with explicit vector types and intrinsics. This may involve changing data structures and using different algorithms. This strategy has the highest performance but is not portable. Different platforms support different SIMD instruction sets, which necessitates different code paths for different platforms. VPIC 1.2 uses manual vectorization using intrinsics.

The third strategy is for the developer to use auto-vectorization but provide more guidance for the compiler. Guiding the compiler enables portability while also avoiding common pitfalls that prevent vectorization. We guide vectorization of the particle push kernel by splitting the kernel into different sections and using OpenMP

SIMD pragmas to mark loops for vectorization. We split the kernel into different code sections so that parts of the code that are difficult to vectorize are isolated from the rest of the code. For example, the particle push kernel uses the square root operation to calculate the new position. Vectorized square root instructions are often less accurate than the scalar code, preventing the compiler from vectorizing unless fast math optimizations are enabled. Separating expensive math operations such as square root from the rest of the code ensures that the compiler can vectorize as much code as possible. Kokkos supports a vector execution policy for vector parallelism but the compiler still uses cost models to determine whether or not to vectorize the code. Relying on the compiler's cost models will lead to parts of the code failing to vectorize even though we know that the code can vectorize and will accelerate the application. Instead, we use OpenMP SIMD pragmas to force the compiler to vectorize code while remaining portable across different compilers.

The particle push runtime for different vectorization methods across a variety of x86-64 and ARM chips is shown in Figure 3.8. Auto vectorization is universally the slowest vectorization method, often more than twice as slow as guided vectorization. The guided method implemented in VPIC 2.0 significantly improves performance for the particle push kernel, reducing the runtime to only 34% more than the manual vectorization with vector intrinsics used in VPIC 1.2. For most of the x86-64 platforms, the guided method is much closer to the manual method in performance than the ARM chips. The A64FX differs from the other ARM and x86-64 chips. The A64FX supports the SVE vector extensions, whereas the ThunderX2 and Altra are limited to Neon vector extensions for SIMD vectorization. VPIC 1.2 supports only Neon, which means that the A64FX cannot use its full capabilities for the manually vectorized VPIC 1.2. Unlike with the other chips, the guided optimizations in VPIC 2.0 outperform the manual method used in VPIC 1.2. This highlights one of the benefits of using the guided vectorization strategy. Leveraging the compiler for vectorized code generation, while less effective than manual vector intrinsics, does not require additional developer effort as new vector extensions and hardware

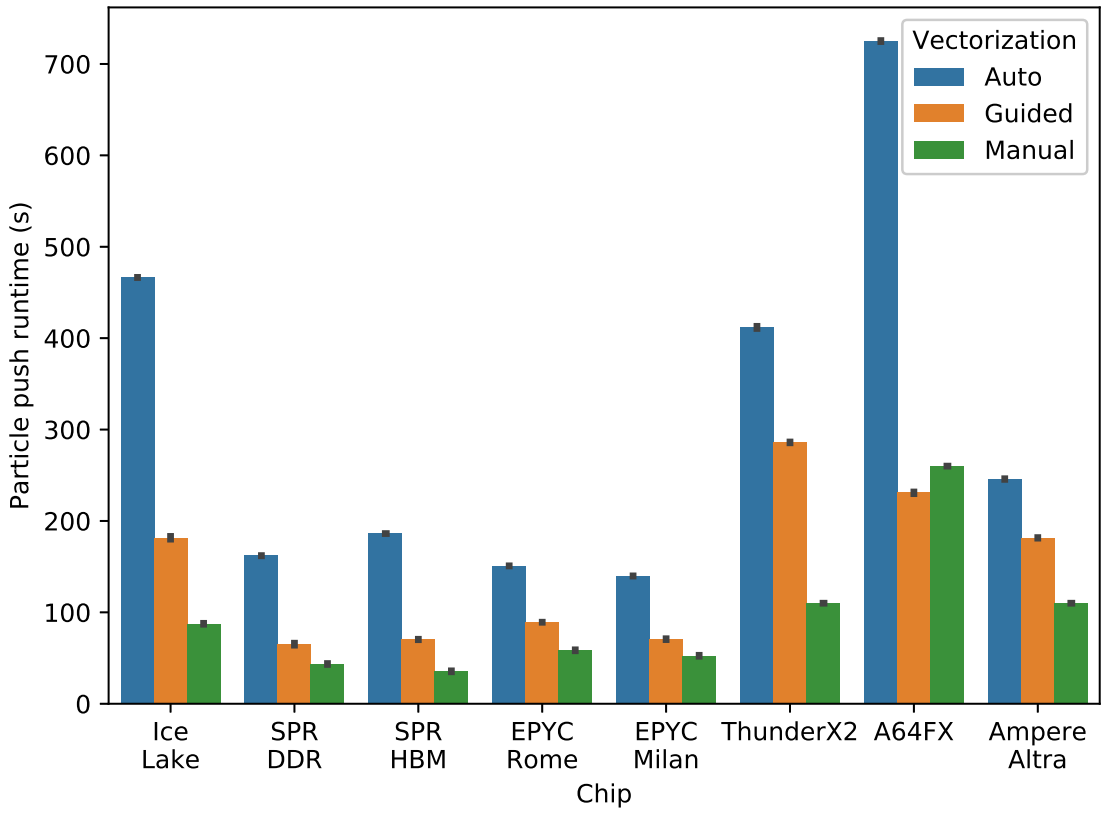


Figure 3.8: Runtime comparison of different vectorization strategies for various hardware platforms.

emerge. Developers can select their vectorization strategy based on their performance requirements and available developer resources.

3.3.2 Hardware Targeted Sorting Evaluation

Particle sorting must keep the target platform in mind to maximize memory efficiency. CPUs have low bandwidth and low latency caches that favor maximizing cache reuse. 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. Sorting particles based on the cell index, which we denote as standard sort, 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. Instead, we create a different sorting algorithm that produces repeating and monotonically increasing sequences. Our algorithm is shown in Algorithm 1. First we create a copy of the keys and identify the maximum key in the list. 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 list with the new keys. This algorithm, which we denote as strided sort, produces coalesced memory access patterns. We evaluate the performance of our different sorting algorithms on four GPU-based platforms: the NVIDIA A100, H100, AMD MI100, and MI250. We use a sample benchmark deck similar to the deck used in our portability test.

Algorithm 1 Strided Sort.

Input: $Keys, Values$

1: $new\_keys \leftarrow Keys$
2: $max\_key \leftarrow MAX(new\_keys)$
3: $key\_counts \leftarrow Histogram$
4: **for all** $key \in new\_keys$ **do in parallel**
5: $\quad offset = key\_counts(key) * (max\_key + 1)$
6: $\quad key = key + offset$
7: $\quad key\_counts(key) += 1$
8: **end for**
9: $SORT(new\_keys, Values)$

We evaluate the performance benefits of our sorting optimization for the particle push in Figure 3.9. Our strided sort cuts the particle push runtime to a third of the standard sort runtime on NVIDIA GPUs. On the MI100 and MI250 GPUs, the difference between the sorting order is even more extreme, with the strided sorting order being 37x faster than the standard order. Without the strided sorting order optimization, running on the AMD GPUs is impractical. Figure 3.10 shows the total runtime spent on sorting the particles with the different sorting orders. The strided sorting order is 34% to 66% more expensive than the standard sort, but the overall cost of sorting is smaller than the particle push. Unlike the particle push kernel, the sort kernel does not show the same drastic performance difference between NVIDIA and AMD GPUs. The large performance improvements and small additional sorting costs highlight the importance of hardware-targeted sorting for particle-in-cell applications.

3.4 Importance and Applicability of Performance Portability

The ability for any code to rapidly deploy itself on a new machine, with limited manual refactoring effort, is a powerful capability that actively drives down the time

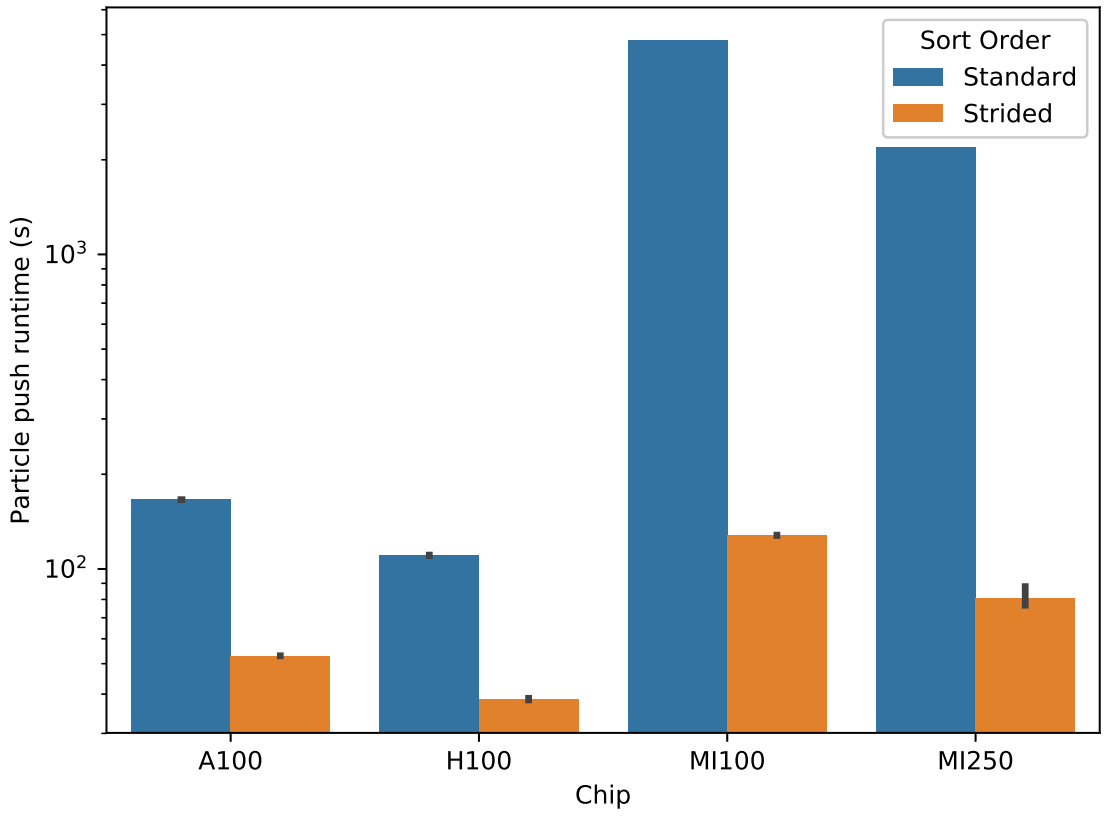


Figure 3.9: Runtime comparison of different sorting orders for various GPU-based platforms.

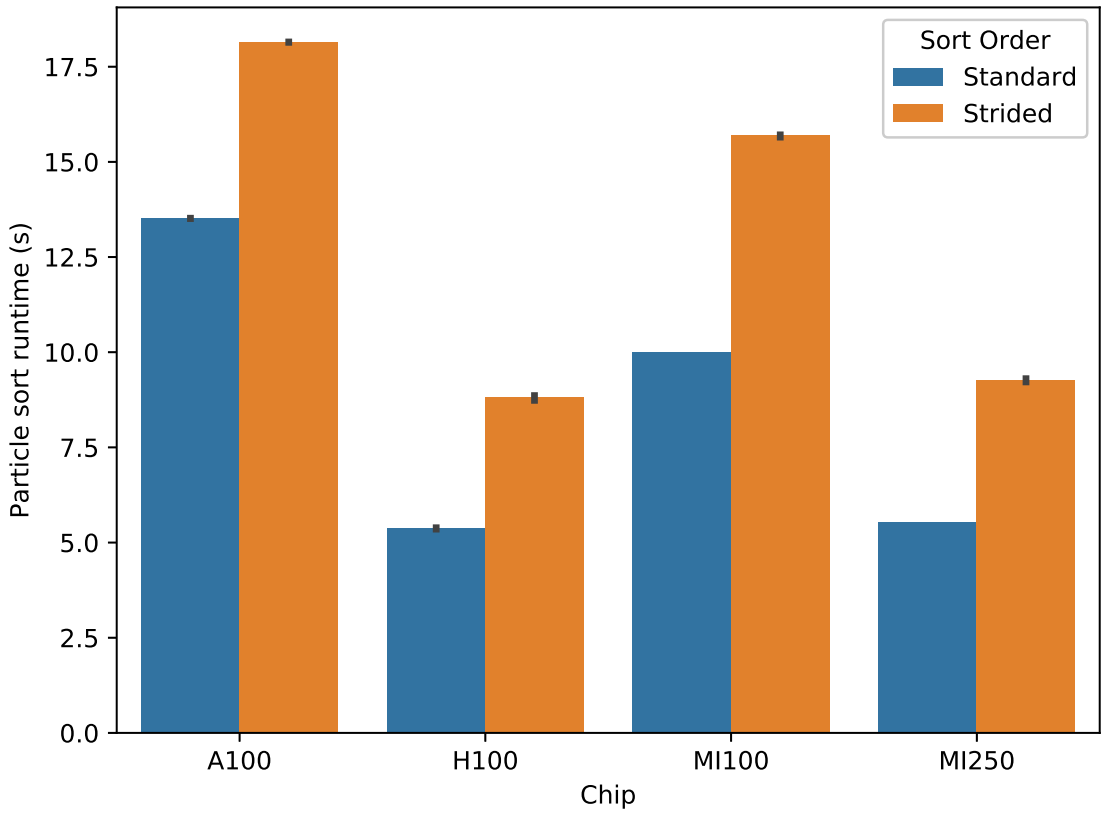


Figure 3.10: Sorting runtime comparison of different sorting orders for various GPU-based platforms.

to scientific discovery. As the heterogeneity of supercomputers continues to increase to meet the demands of exascale, effective use of these platforms translates directly to codes like VPIC being able to run simulations at unprecedented scales and opens the door to performing analysis and investigation which were previously intractable.

While enabling portability, VPIC 2.0 is also able to assure performance. The performance-portability trade-off allows simulations to be completed quickly and without multiple restarts by using accelerators and emerging processors; such platforms are not optimized or supported by the original VPIC code (VPIC 1.2). Portability frameworks, of course, come with drawbacks in the form of overhead from mapping a single code to different backends and concealing hardware properties. Techniques such as hardware-targeted sorting and guided vectorization mitigate portability overhead and further improve application performance. The reduction in execution costs when using accelerators, coupled with the portability presented in this chapter, facilitates the following changes in VPIC simulations:

- Simulating more timesteps in fewer core hours enables simulations that advance further in time, allowing us to set our sights towards performing longer fully resolved electromagnetic simulations than were previously possible.

- Advanced and expensive diagnostics can now be performed within the timestep, allowing for more accurate analysis. The extensive analysis enables new fundamental insights to particle acceleration modeling at unprecedented scales (78).

- Previous simulations, which took considerable resources, can now be run multiple times to analyze stochastic variation in, for example, short-pulse laser experiments, which in turn can generate sufficient data for machine learning analysis.

The new VPIC code release opens up new scientific avenues, extending the plasma physics research domain to exascale simulations. Our multi-layer software approach to performance portability is applicable to many different portability solutions and

applications. Our five-step methodology for porting existing applications is easily adapted for other portability solutions like SYCL and OpenMP with minor changes. The comparisons of vectorization strategies demonstrate that developers can greatly improve vectorization performance with a few simple code changes that guide the compiler in best vectorizing the code. The takeaways from our vectorization study are applicable to any application that can exploit data parallelism. The hardware-targeted sorting order optimizations apply to other particle-in-cell and grid-based applications with gather or scatter patterns. Gather scatter patterns are fairly common in HPC applications. For example, common mini-apps like LULESH (79), AMG (80), and Nekbone (81) that model hydrocodes, multigrid solvers, and computational fluid dynamics respectively. Molecular dynamics codes like LAMMPS (82) and adaptive mesh refinement codes also use sorting to improve performance for gather scatter patterns.

Chapter 4

Preserving Accuracy with Lower Precision Formats

Memory capacity and data movement costs limit large-scale simulation codes such as VPIC. As the VPIC code transitions from CPUs to accelerators (i.e., GPUs), the number of particles in its simulations becomes limited more by memory than compute capabilities. CPUs can access up to 4 TB of memory, while GPUs such as the NVIDIA A100 are limited to 80 GB, a factor of 50 difference. Additionally, data movement between CPUs and GPUs is costly, with PCIe 4.0 limited to 32 GB/s in one direction. Specialized protocols and hardware, such as NVLink 3.0, have been developed to address this issue, achieving up to 300 GB/s in one direction. Lower precision formats like half-precision can reduce data movement and memory usage. However, numerical accuracy loss limits the usability of reduced precision formats. Reduced precision formats, less accurate than single-precision, became prominent with AI/ML workloads that can tolerate some precision loss. Traditional HPC applications like VPIC require techniques for maintaining accuracy to benefit from the data movement and memory usage advantages of lower precision formats.

To minimize memory usage and maximize simulation size while maintaining preserving accuracy, we present a suite of optimizations for VPIC's particle storage

format, which enables larger-scale simulations using reduced precision number formats while maintaining accuracy. We leverage the local coordinate system to control numerical accuracy, a unique approach. We thoroughly examine the performance and accuracy of our optimizations on both CPU and GPU hardware, providing a comprehensive evaluation. The results of our study demonstrate that our optimizations not only reduce memory usage and improve the runtime performance of memory bandwidth-bound kernels but also produce accurate scientific results. Our contributions towards preserving accuracy with lower precision formats are:

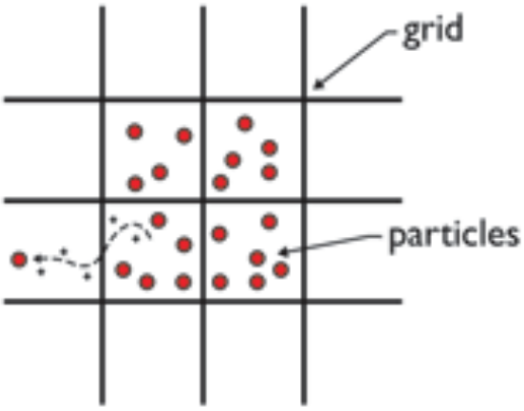
- We develop a set of optimizations to the particle format that enables large simulations in VPIC using reduced precision and control numerical accuracy by leveraging a local coordinate system. (Section 4.1)
- We present a detailed memory usage and runtime performance analysis of our optimizations on CPU and GPU-based platforms. (System 4.2)
- We demonstrate that our optimizations can maintain scientific accuracy while enabling larger simulations. (Section 4.3)
- We identify our techniques' importance and broader applicability for improving simulation scale and performance using lower precision formats in grid-based simulations. (Section 4.4)

4.1 Mixed Precision in the Particle-In-Cell Method

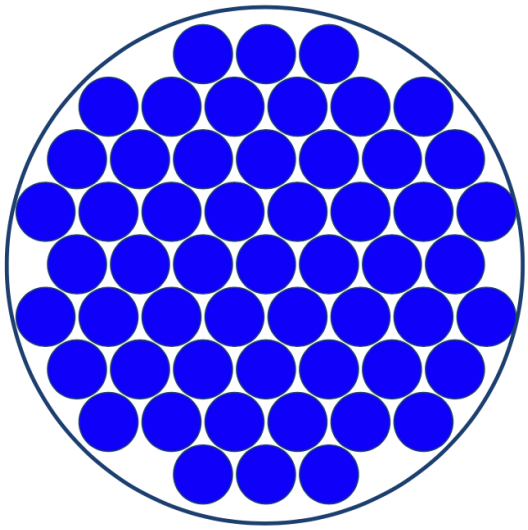
VPIC operates by defining a simulation space divided into a grid of cells and modeling particle movement across the cells. In other words, particles are distributed across an n-dimensional (n-D) space that is decomposed into an n-D grid. The resolution of the grid determines cell size and the maximum time step length. Figure 4.1a shows an example of a 2-D grid. Each simulated particle is a macroparticle (Figure 4.1b) with a defined weight (i.e., the number of real particles represented by each macroparticle).

The grid resolution and the number of particles affect simulation accuracy and computational costs. Fine-resolution grids better approximate a continuous n-D space. However, such fine-resolution grids increase computational costs due to field operations, and the time step size must shrink accordingly to ensure accurate physics. Thus, the number of time steps increases for the same period of simulated time, which further increases computational costs. Increasing the number of particles can also improve accuracy by more closely modeling real-world plasma phenomena. High particle count primarily affects the particle advance stage with computational costs scaling linearly with the number of particles. In typical PIC simulations, the cell size is close to the Debye length $\lambda_D = \sqrt{k_B T / 4\pi n_e e^2}$, where T is the electron temperature, k_B is Boltzmann's constant, n_e is the electron number density, and e is the electronic charge; the Debye length is the characteristic screening length in a plasma, generally the smallest physical length scale of interest. The time step is set as large as possible since the spatial constraint is often more strict, and for applications such as laser-plasma interaction, the dispersion properties of electromagnetic modes are more accurately captured with a larger time step.

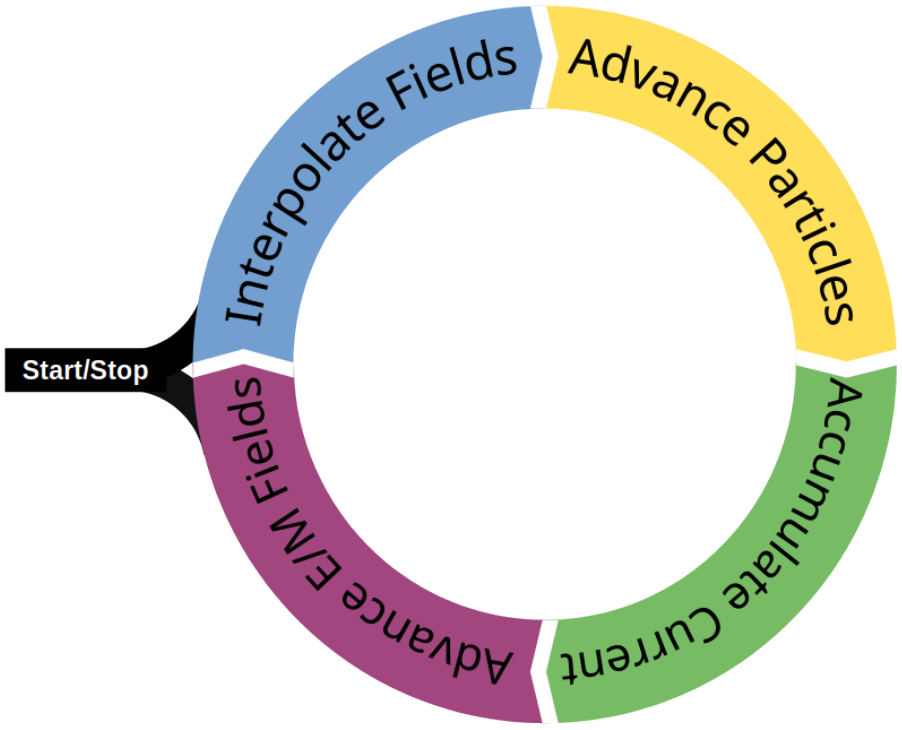
A VPIC simulation is an iterative process across a defined number of time steps: each iteration has four key stages (as shown in Figure 4.1c). First, the electric and magnetic fields are gathered from the grid points to each particle's location (interpolate fields). Second, particles move around based on the forces calculated from the electric and magnetic fields (advance particles). Third, the current generated by the particles' movements is calculated for each cell (accumulate currents). Last, electric and magnetic fields are advanced based on the accumulated current (advance EM fields), and the next iteration starts. The first three stages take most of the simulation time for large, particle-heavy simulations, as each stage must operate on all the particles. VPIC advances the electric and magnetic fields at each grid point in the fourth and last stage. The field advance stage is cheaper than the other stages since the number of particles tends to outnumber the number of grid points greatly. Furthermore, past I/O studies using VPIC noted that particles are responsible for



(a) Grid and particles



(b) Macroparticle



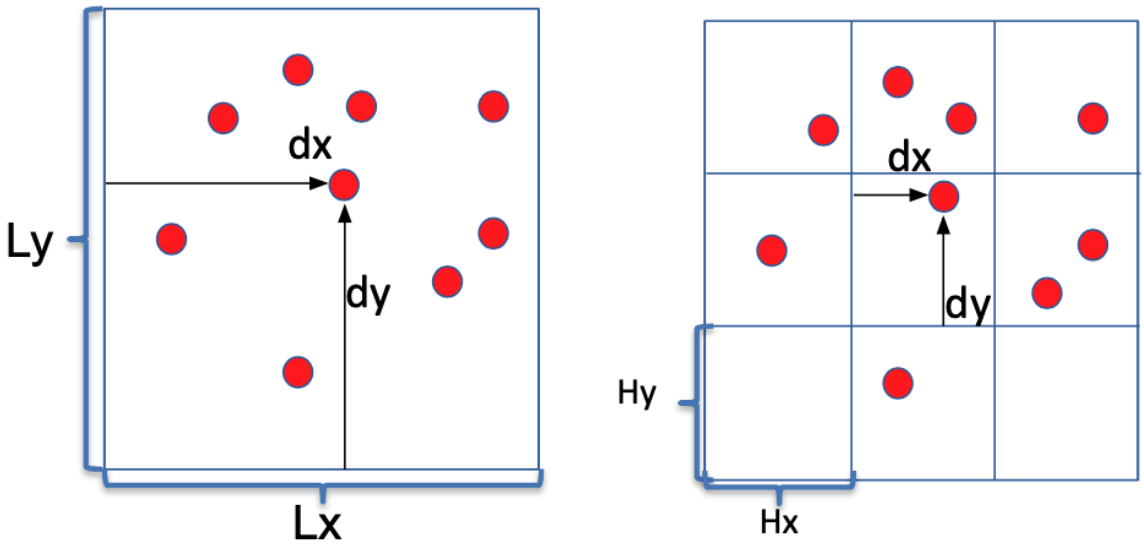
(c) VPIC iterative algorithm

Figure 4.1: VPIC features: grid and particles (a); concept of macroparticle (b); and VPIC stages (c).

most memory usage. The trillion particle run in (83) required only 80 GB of storage for the electric and magnetic fields compared with the approximately 30 TB necessary for the particles. Both fields and particles scale with the grid size, but particles tend to have a much higher constant factor. The 375-fold difference in memory requirement demonstrates that particle storage is a bottleneck for large-scale VPIC simulations. Using lower precision formats such as half-precision can improve performance by reducing storage costs and data movement. However, the numerical solvers in VPIC (i.e., Boris pusher (84), finite-difference time-domain solver (85) on a Yee staggered grid (86)) do not use matrix-based linear solvers and thus do not benefit from the existing work on mixed-precision solvers. Instead, we leverage the grid and coordinate system to use lower precision formats while maintaining accuracy.

4.1.1 Controlling Accuracy with Local Coordinates

The particle position can be stored in two different ways based on the coordinate system: globally and locally. Global coordinates of particles are calculated and stored based on their absolute position in the n-D space. In Figure 4.2a, the global positions of the particles are depicted in a 2-D space as (dx, dy). Alternatively, particle positions can be stored in local coordinates, with each particle storing its cell index and position within the cell. In Figure 4.2b, each particle position in the 2-D space is defined as $(i * H_x + dx, j * H_y + dy)$, where (i, j) is the cell index, (H_x, H_y) is the cell size, and (dx, dy) is the local position of the particle. This representation requires extra space for the cell index but allows VPIC to represent each particle position more accurately. This is because the distribution of floating-point values follows an almost logarithmic distribution (87). Values further away from zero are less precise, and approximately half of all floating-point values exist in $[-1, 1]$. Thus, local coordinates within the smaller cell have a more even and dense distribution of values to represent a particle position. VPIC uses local coordinates and single-precision for storing particle positions by default. In VPIC simulations, the larger the number of particles, the



(a) (dx, dy) **(b)** $(i * Hx + dx, j * Hy + dy)$

Figure 4.2: Global and local particle coordinates.

more physically accurate the simulations and the greater the memory usage. Each particle is a macroparticle of 32 bytes (as shown in Figure 4.3) that comprises 3 floats for position, 3 floats for momentum, 1 integer for the cell index, and 1 float for weight. There is a major disparity in memory usage between particle data and all the other data used in the code. Figure 4.4 shows an example of disparity for a VPIC simulation of 512 particles per cell; the particles are responsible for over 90% of the total memory usage. Larger VPIC simulations have thousands (or more) of particles per grid cell (88), making particles the primary focus when optimizing memory usage. For instance, the simulations described in (89), which study laser-plasma modeling, have up to $65,536$ particles per grid cell.

4.1.2 Particle Format Optimizations

Our suite of optimizations reduces memory usage associated with particles' weight and position through reduced precision (i.e., half-precision) and alternative number representation formats (i.e., fixed-point). Accuracy is maintained by leveraging simulation properties and characteristics of the particle-in-cell method.

The various types of precision formats for floating-point numbers supported by existing platforms are described in Table 4.1. VPIC 1.2 and VPIC 2.0 uses single-precision by default (i.e., float), as shown in Figure 4.3. The three candidates for reduced storage formats are FP16, TensorFloat (90), and Bfloat16 (91). Compared to FP16, TensorFloat requires more storage (19 bits), and Bfloat16 loses precision (decimal digits). Thus, we use FP16 for our optimizations.

Particle Weight

In VPIC, the macroparticle's weight represents the number of real particles modeled by the simulated particle. The particle weight changes within a limited range or does not change during a simulation. We use this property to optimize memory usage by adjusting how weight is stored.

```
struct particle {
  float w;           // Weight
  float dx, dy, dz; // Position
  int i;             // Cell index
  float ux, uy, uz; // Momentum
};
```

Figure 4.3: Original VPIC particle data structure.

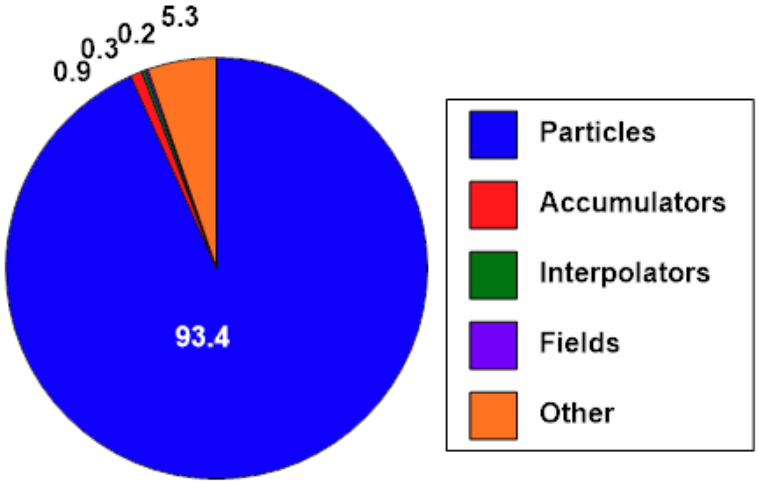


Figure 4.4: Breakdown of memory usage for a simulation using 512 particles per cell.

Table 4.1: Floating-point formats with their numerical representations.

Precision	Sign	Exponent	Fraction	Decimal Digits
Double (FP64)	1	11	52	≈ 15.9
Single (FP32)	1	8	23	≈ 7.2
Half (FP16)	1	5	10	≈ 3.3
TensorFloat	1	8	10	≈ 3.3
Bfloat16	1	8	7	≈ 2.4

When the particle weight varies but has a limited range of values, we can replace the single-precision weight with a 16-bit integer denoting a multiple of a known base weight. Alternatively, the 16-bit integer can be an index for a lookup table of particle weights. Both methods allow 65,536 different weights while reducing particle memory by 6.25%. We call this optimization short weight (SW).

When the particle weight remains constant for all particles of the same species throughout a simulation, particle weight can be removed entirely and replaced with a per-species constant weight. This solution reduces memory usage by 12.5% over the single-precision VPIC. We call this optimization constant weight (CW). Figure 4.5 shows the reduction in memory usage for both optimizations (i.e., SW and CW).

Particle Position

The position of a particle in VPIC 2.0 is represented by three float values in single-precision (32 bits). We optimize the code by switching the representation to half-precision (16 bits). Figure 4.6 shows that by deploying half-precision, we reduce memory usage by 18.75% compared to single-precision VPIC 2.0 and 31.25% when combined with the constant weight optimization (or CW). Half-precision can produce simulations of sufficient accuracy for a fine-resolution grid while enabling larger simulations, as we will show in Section 4.2.2.

From Half-Precision to Fixed-Point

Half-precision particle position may incur an accuracy penalty compared to the default single-precision. Particle positions in each cell in VPIC using local particle coordinates are normalized to [-1,1], and thus, we can use 16-bit fixed-point numbers instead of half-precision to minimize the loss in accuracy (92). Fixed-point numbers allow us to maximize the number of bits used for precision. The fixed-point $Qm.n$ format specifies m bits for the integer and n for the fractional portion. We use the $Q1.14$ fixed-point format for storing position. One bit is for the sign, one for the integer

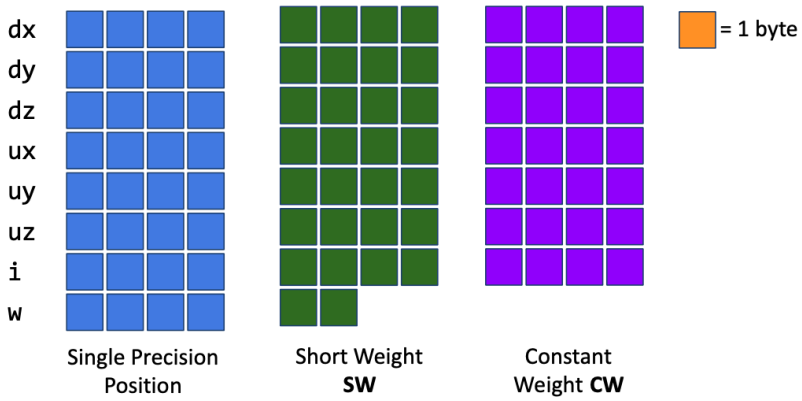


Figure 4.5: Particle memory usage for default single-precision VPIC and VPIC with our short weight (SW) and constant weight (CW) optimizations.

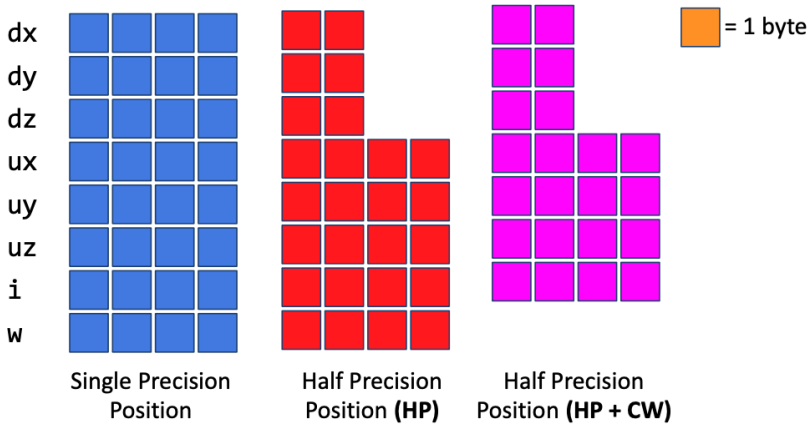


Figure 4.6: Particle memory usage comparison between single-precision VPIC 2.0, VPIC 2.0 with our half-precision (HP) optimization, and VPIC 2.0 with both half-precision and constant weight optimizations applied (HP+CW).

portion, and the remaining 14 bits for the fractional portion. The $Q1.14$ format uses the same amount of memory as half-precision but has an additional four bits (approximately one decimal digit) for improving accuracy.

Particle Momentum and Cell Index

Particle momentum and cell index are represented by three single-precision floats and a 32-bit integer, respectively. Momentum is left unchanged in this work. Momentum values are normalized to c (speed of light). Initial tests indicate that switching the momentum values from single-precision to half-precision would result in insufficient accuracy. The particle cell index determines which cell the particle resides in and is kept at the default 32-bit integer. A short 16-bit integer has insufficient range for large-scale simulations.

4.2 Performance Evaluation

We consider four test problems. The first test models laser-plasma interaction and is used to measure runtime performance and memory usage of VPIC 2.0 and the optimized version. The size of the simulation is controlled by adjusting the number of particles per cell. We evaluate the runtime and memory usage on a memory capacity-limited GPU, the effects of our optimizations on simulations using swap memory, and the impact of vectorization on the optimizations. The remaining three tests (i.e., 1-D periodic boundary problem, two-stream instability, and Weibel instability) constitute a set of simple benchmark problems for analyzing the impact of our optimizations on the VPIC's numerical accuracy. We show the impact of our optimization on the accuracy of particle movement, momentum conservation, and energy conservation.

4.2.1 Test Platforms

We run our tests on two clusters, targeting three platforms: the NVIDIA V100 GPU, IBM Power9 CPU, and Fujitsu A64FX CPU. We use a single NVIDIA V100 for GPU tests and a single node of each cluster for CPU tests. The use of a single node removes overheads associated with network resources in our measurements. Platforms are sorted chronologically based on the year of release.

The NVIDIA GPU V100 (2017) experiments are executed on a single GPU with 16 GB of HBM2 hosted on the IBM Power9 system described below. The experiments use CUDA 10.2, GCC 8.4.1, OpenMPI 4.0.4, and Kokkos 3.1.1.

The IBM Power9 (2017) experiments are executed on the Tellico high-end cluster at the University of Tennessee Knoxville. Tellico is an IBM Power Systems AC922 hybrid architecture system comprised of 4 nodes, each with two 16-core IBM Power9 processors running at 2.3–3.8 GHz with 155 GB of RAM. Two of the nodes host two V100 GPUs. Experiments use GCC 8.4.1, OpenMPI 4.0.4, and Kokkos 3.1.1.

The Fujitsu A64FX (2020) experiments are executed on the Ookami cluster at Stony Brook University. Ookami is an HPE Apollo 80 system with 174 nodes. Each node contains a 48-core Fujitsu A64FX Arm processor (75) with 32 GB of HBM2 and a 512 GB SSD. Nodes are connected in a 2-level tree using non-blocking HDR-200. Experiments use GCC 11.1.0, OpenMP 4.1.1, and Kokkos 3.4.1.

4.2.2 Test Settings

We analyze VPIC performance using our different sets of optimizations. The optimization sets used in this chapter are listed in Table 4.2. We run simulations modeling a laser-plasma interaction and measure memory usage and runtime performance of single-precision VPIC 2.0 versus our optimized versions for a diverse set of hardware platforms (i.e., V100, Power9, and A64FX). All the tests are conducted using a single, portable version of VPIC (4).

Table 4.2: Combinations of particle format optimizations used in this chapter.

Optimization set	Weight	Position
VPIC	single-precision	single-precision
VPIC+SW	short integer	single-precision
VPIC+CW	constant	single-precision
VPIC+CW+HP	constant	half-precision
VPIC+CW+FP	constant	fixed-point

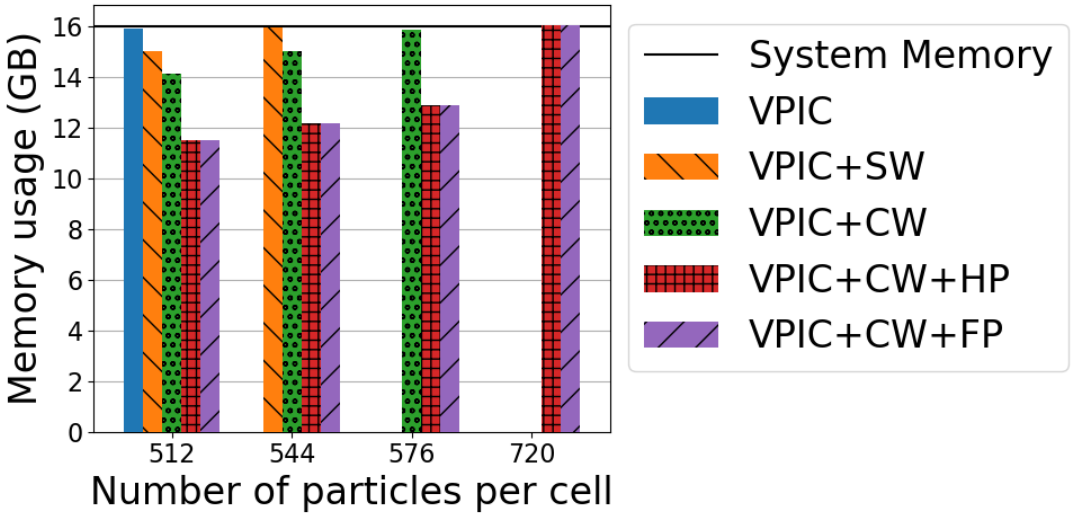
4.2.3 Runtime and Memory Usage on GPUs

For measuring peak memory usage and runtime performance on the GPU, we use four problem sizes designed to use the GPU's full 16 GB memory capacity. Problem size is determined by the total number of particles per cell. The base case comprises 512 particles per cell, and it fills the 16 GB DRAM when using single-precision VPIC. The remaining three cases (i.e., 544, 576, and 720 particles per cell) increase in size until only our optimized versions can successfully run.

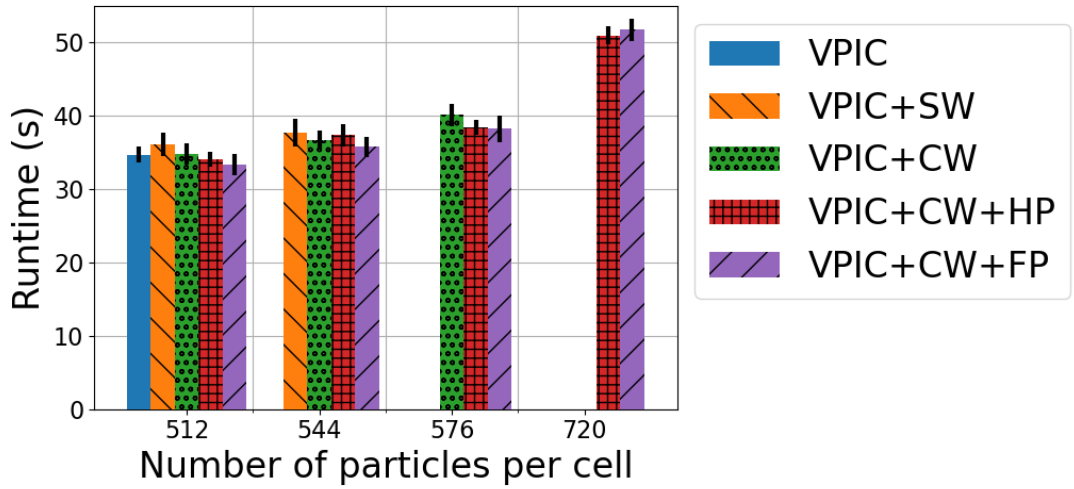
Peak memory usage is measured using the Space Time Stack tool provided by the Kokkos Tools profiling utilities (93). The tool tracks Kokkos's allocations and the maximum memory usage on the GPU. Runtime tests are repeated 10 times, and the average runtime is used to compare configurations; the standard deviation is also computed.

The scalability for the single-precision VPIC 2.0 code and when using our four optimizations (i.e., with short particle weight (SW), with constant particle weight (CW), using half-precision particle position (HP), and using fixed-point particle position (FP)) are shown in Figure 4.7a. Missing columns indicate failed simulations that crash due to insufficient memory. We control problem size by adjusting the number of particles per cell (Nppc). The grid resolution and number of time steps are kept constant. Our optimizations demonstrate a significant reduction in memory usage, as shown in Figure 4.7a. The optimizations to particle position (HP and FP) provide the greatest reduction in memory usage; when combined with constant weight (CW), they can increase the total number of particles by a factor of up to 40%.

We demonstrate that our optimizations have minimal effect on runtime performance in Figure 4.7b. Specifically, no negative effect is observed on runtime performance; in fact, runtime can improve thanks to the hardware's need to move less data through its memory system and between processors. In other words, the optimized VPIC minimizes the amount of data movement between CPU and GPU. This results in a relatively small performance improvement that largely cancels out



(a) V100's peak memory usage in GB of single-precision VPIC 2.0 and VPIC 2.0 with our optimizations.



(b) V100's runtime with our optimizations in seconds of single-precision VPIC 2.0 and VPIC 2.0 with our optimizations.

Figure 4.7: V100's peak memory usage in GB (a) and runtime in seconds (b) of single-precision VPIC 2.0 and VPIC 2.0 with our optimizations.

the additional overhead from converting data between different formats when entering and exiting the advance particle stage (i.e., from single-precision to half-precision and vice versa, from floating-point to fixed-point and vice versa), where the optimizations are deployed. Our results show that VPIC+CW+HP and VPIC+CW+FP perform similarly despite $Q1.14$ not being a natively supported format like half-precision. This indicates that the fixed-point optimization has very low overhead on the GPU.

4.2.4 Runtime and Memory Usage on CPUs

To measure peak memory usage and runtime performance on CPU-based platforms (i.e., Power9 and A64FX), we select the number of particles to stress the memory capabilities of the targeted platform. We increase the problem size until only our most heavily optimized versions of VPIC (i.e., VPIC+CW+HP and VPIC+CW+FP) can run the simulation. For each platform, we measure the peak memory usage with the Memory High Water Mark tool provided by the Kokkos Tools profiling utilities (93). In addition to the peak memory capacity stress test, we run two tests to investigate memory bandwidth-constrained scenarios. On a mature and broadly used HPC platform such as Power9, we push the memory bottleneck and simulation size further by using swapping, comparing runtime with swapping disabled and enabled. For both test configurations, we zoom into the VPIC kernels and identify those responsible for any performance losses or improvements with swapping disabled and enabled. On a cutting-edge platform such as A64FX, we study how our optimizations improve runtime performance in memory bandwidth-limited kernels by comparing a non-vectorized version of VPIC with a vectorized version.

Tests on Power9:

A method for running large simulations on memory-limited hardware is to leverage high-capacity storage devices to expand the available main memory with swapping. Swapping data from main to secondary memory allows for much larger available

memory at the cost of access latency and bandwidth. Our tests study the benefits of our particle optimizations for mitigating the drawbacks of interacting with slower memory hardware such as secondary memory. We run memory capacity stressing tests on the Power9 platform, without and with swapping, to quantify our optimizations' impact on performance.

CPUs have higher memory capacities than GPUs and can leverage swap memory to further increase capacity. Figure 4.8 shows the peak memory usage of VPIC with our optimizations on a single Power9 node. The system is set to allow swapping, and thus, if the memory usage exceeds the main memory limit (i.e., the back bar at 155 GB in the figure), the test uses swapping to complete the simulation. In the figure, we observe runs whose execution progressively exceeds the memory limit: single-precision VPIC 2.0 runs out of main memory already with 3040 particles per cell; VPIC+SW runs out with 3184 particles per cell; and VPIC+CW runs out with 3592 particles per cell. Optimizations such as VPIC+CW+HP and VPIC+CW+FP can keep in check swapping in and out of main memory for the largest number of particles, far beyond 3592 particles per cell, resulting in up to 22% increase in the number of particles per simulation compared to single-precision VPIC 2.0.

Enabling swap memory enables larger simulations but also slows down execution. Figure 4.9a shows the results of our memory stressing tests without swapping. As with the GPU memory stressing tests, we gradually increase the problem size until only our most optimized version of VPIC (i.e., VPIC+CW+HP and VPIC+CW+FP) can run. We further increase the problem size to 4,096 particles per cell, requires swap memory even with our best optimizations. A notable change from the GPU results is that VPIC+CW+FP consistently beats VPIC+CW+HP in runtime performance. Unlike the V100 GPU, Power9 only supports conversions between half-precision and single-precision using vector intrinsics. The additional overhead from using vector registers results in a small slowdown compared to using fixed points. Runtime performance shown in Figure 4.9b presents the results of the same tests but with swapping enabled. The runtime and standard deviation are much higher when

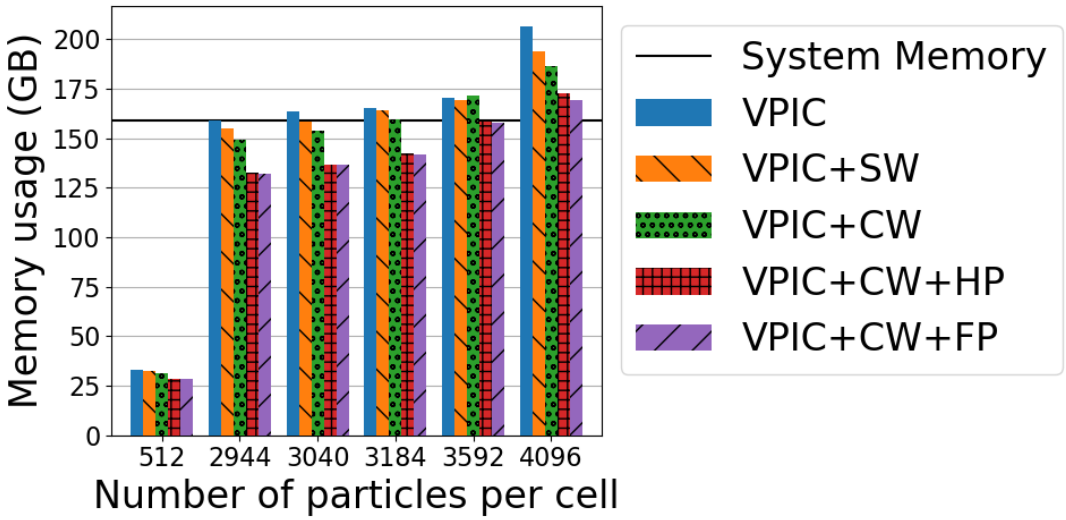
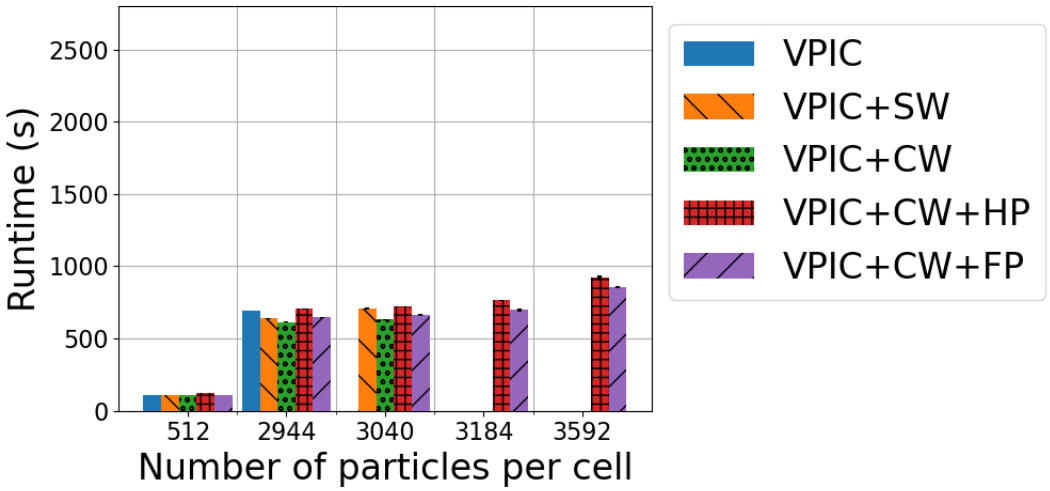
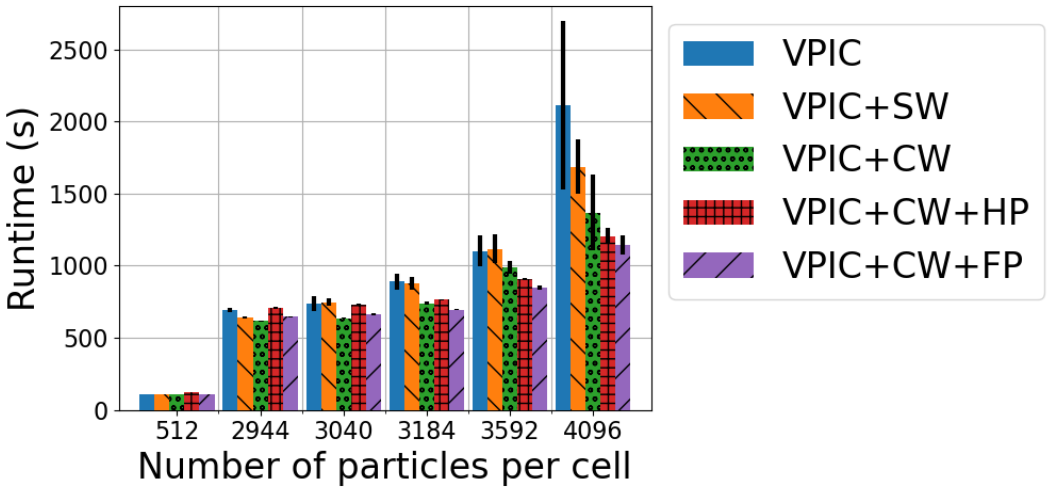


Figure 4.8: Power9's peak memory usage in GB of VPIC and VPIC with our optimizations. Simulations exceeding the system memory limit (black line) require swapping to complete successfully.



(a) Runtime with swap disabled.



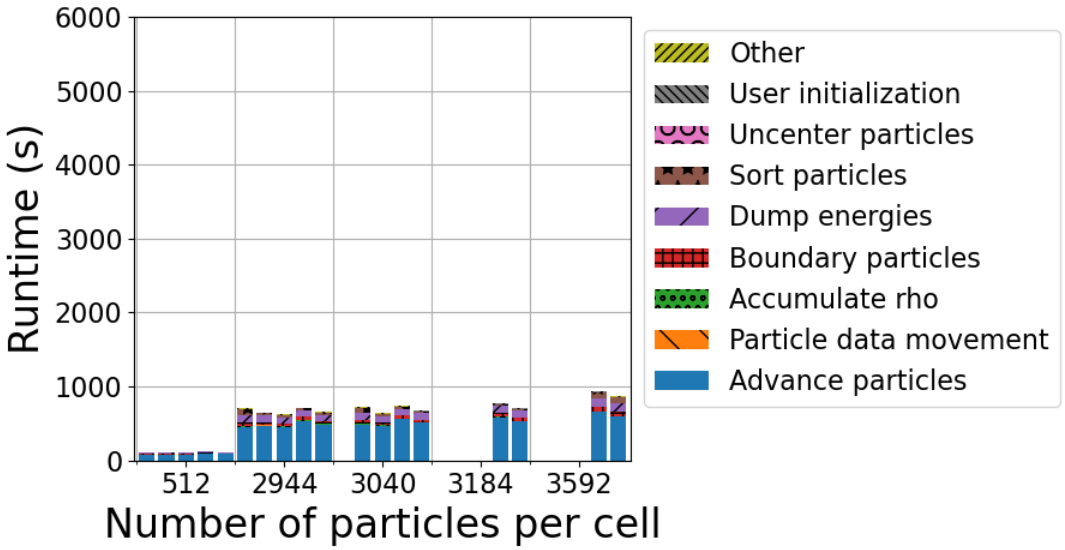
(b) Runtime with swap enabled.

Figure 4.9: Power9's runtime and standard deviation of VPIC using our optimizations and running on a single node with swap disabled (a) and enabled (b).

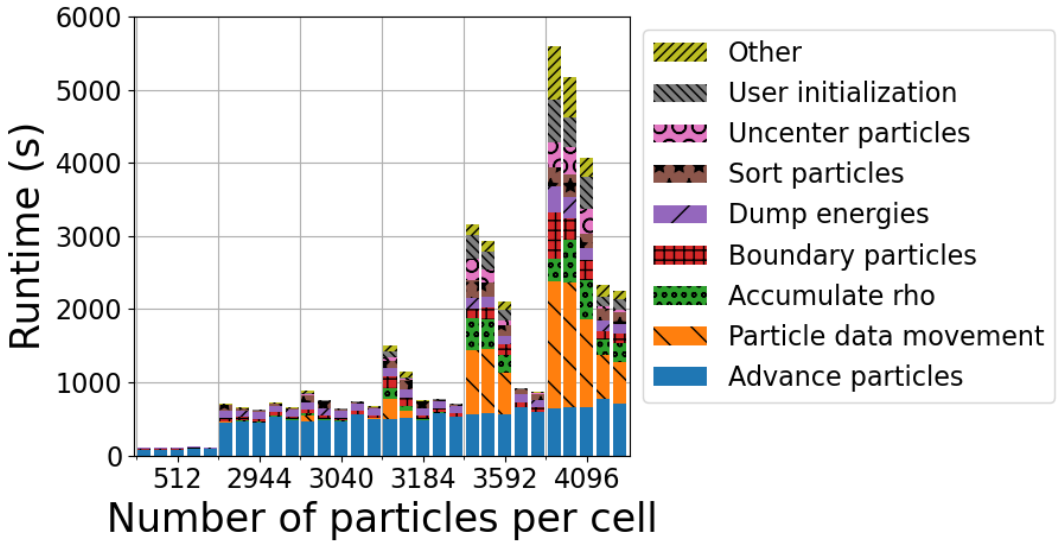
swapping from main memory to secondary memory is necessary. In other words, while swapping prevents the early termination of the simulations, it also results in longer and more variable runtime performance (the black bars in the figure). The runtime tests indicate that the VPIC+CW+HP and VPIC+CW+FP optimizations without swapping enable an approximately 15% reduction in runtime compared to no optimizations. With swapping, our optimizations further reduce runtime by up to 45% compared with single-precision VPIC 2.0 for the largest problem size. These improvements demonstrate the potential savings when interacting with slower storage hardware such as during checkpoints or when saving simulation data to disk for visualization and analysis.

To better understand the VPIC kernels that slow down due to swapping, we break down the runtimes for the two settings (without and with swapping) in Figure 4.10a and Figure 4.10b respectively. We consider eight key kernels and gather the remaining kernels under the same tag. The eight kernels are: user initialization for setting up the simulation data, uncenter particles for the sanity check of the particle advance; sort particles for improving the performance of particle-related kernels; dump energy for calculating and outputting the energy of the system at every time step; accumulate rho for adding the charge density of the particles to the field's charge density; particle data movement for moving particles among the Kokkos device/host and legacy data structures at the initialization and communication between parallel processes; and advance particles to update particle positions during the simulation. A production simulation's runtime would have a much smaller portion in the user initialization and dump energies kernels than these tests, which were artificially short and dumped the energies every time step.

The two figures reveal several interesting characteristics of the two sets of tests. When swapping is not enabled, simulations spend most of the time (more than 90%) advancing the particles (as also described in Section 4.1). When swapping is enabled and used by the simulation to complete successfully, several kernels experience a substantial degradation in performance. The detailed breakdown shows that the



(a) Without swap



(b) With swap

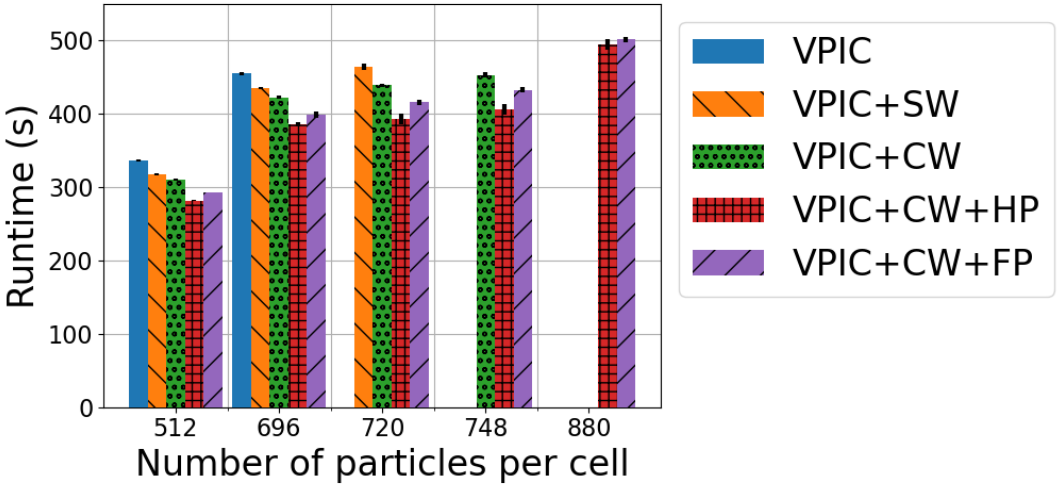
Figure 4.10: Power9's detailed runtime breakdown of key particle-focused kernels using the optimized version of VPIC without (a) and with (b) additional swapping functionalities. The remaining kernels are collected under the Other category.

kernels responsible for simulation initialization suffer the most significant performance degradation. Among those kernels, particle data movement, in particular, suffers with larger problems. Particle data movement is negligible in Figure 4.10a whereas it becomes the most expensive kernel in Figure 4.10b. Other memory bandwidth-bound kernels, such as particle sorting and dump energies, behave as expected, where memory swapping degrates performance. Those parts of the simulations in Figure 4.1c, such as interpolate fields, accumulate currents and advance EM fields that do not perform major computations but read and write particles, are suddenly heavily penalized by the lack of memory capacity, and the consequences of data movement from and to secondary memory. On the other hand, the particle advance kernel, which was the key kernel in tests that did not leverage swapping and is normally the most expensive kernel within the time-stepping loop, is practically unaffected by the use of swapping.

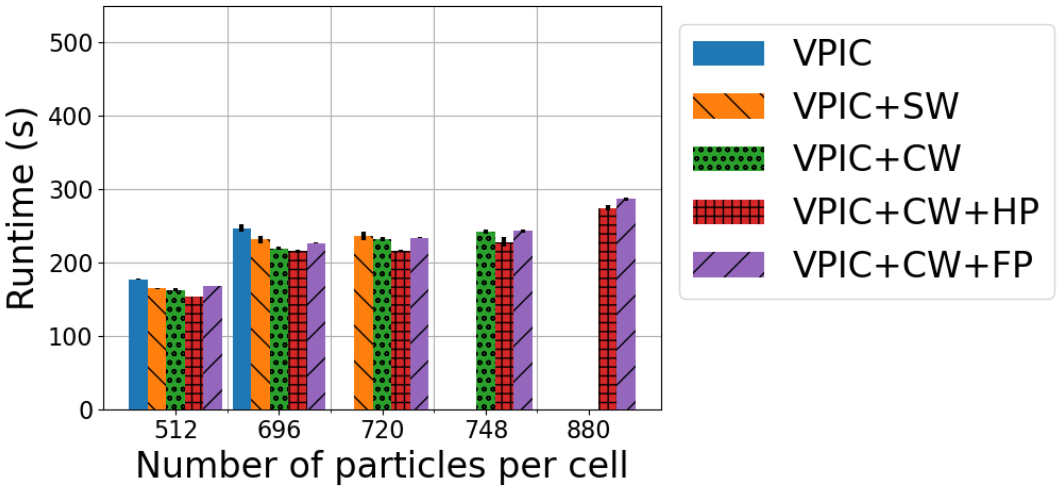
Tests on A64FX:

The Fujitsu A64FX was at the time of tests a cutting-edge platform that is still partially unveiled in its functionalities; it has 32 GB of HBM2 and 1 TB/s memory bandwidth. Its small but fast memory makes the A64FX an interesting platform to study the effects of our particle optimizations. Memory bandwidth is very important for VPIC. We use a similar methodology as Power9 to study the performance of VPIC on this cutting-edge platform, and we observe similar patterns in memory usage.

In this platform, we study the impact of vectorization on performance. We identified the advance particles kernel as the most expensive in VPIC and vectorized the kernel with OpenMP SIMD. In addition, optimization flags were adjusted to help vectorize the code. We verified that most of the kernel was vectorized by examining the produced assembly. Figures 4.11 show the runtime in seconds for our optimizations without and with vectorization. We progressively increase the number of particles for our tests to stress memory usage. Specifically, we consider 512, 696, 720, 748, and 880 particles per cell for our five optimizations.



(a) Non-vectorized



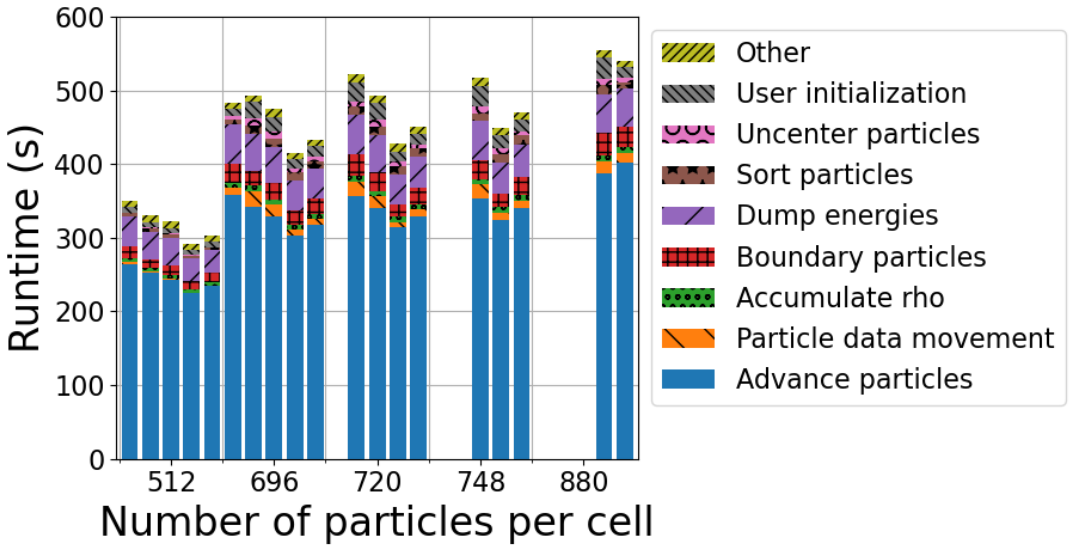
(b) Vectorized

Figure 4.11: A64FX's runtime and standard deviation of VPIC using our optimizations and running on a single node with vectorization disabled (a) and enabled (b).

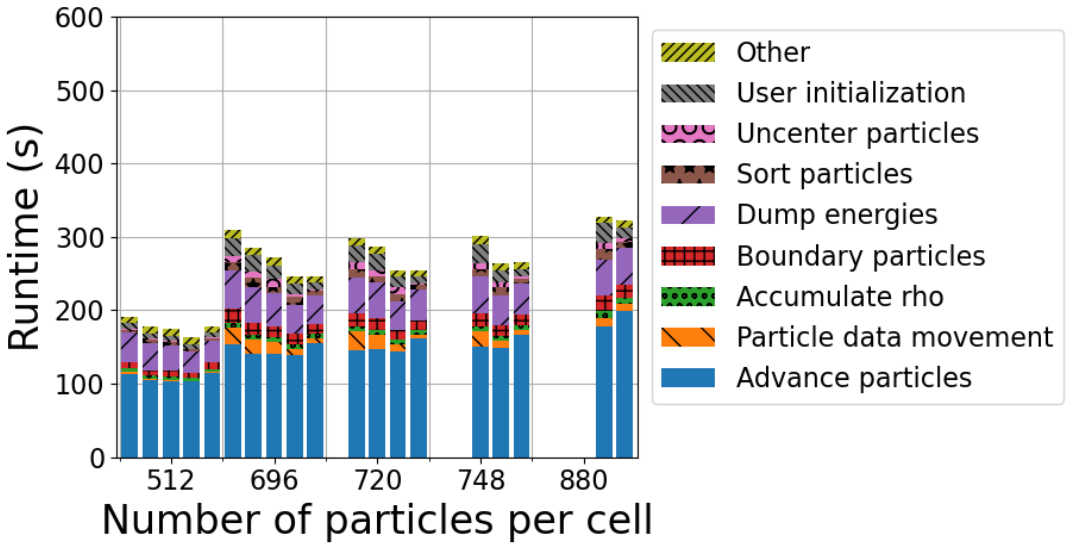
Reducing memory usage also reduces data movement costs which can mitigate the additional overhead from fixed-point formats. Figure 4.11a reveals several interesting characteristics of VPIC on the A64FX. Our optimizations with disabled vectorization improve performance across all tests with a consistent speedup pattern: given a particle size and as we move from single-precision VPIC 2.0 to the VPIC+CW+HP, the runtime decreases, and for VPIC+CW+FP the runtime slightly increases compared to VPIC+CW+HP but remains faster than the other configurations. These patterns suggest that our optimizations reduce data movement and, therefore, improve memory bandwidth utilization. As the problem size increases, the runtime increases roughly linearly with the number of particles. Compared with the V100 and Power9, A64FX exhibits high runtime and poor performance, indicating that we are not yet leveraging the full potential of this platform and its high memory bandwidth. As in the previous tests, missing runtime measurements are due to the main memory saturating and the consequent unsuccessful completion of the simulations.

Reducing data movement with our optimizations will still have an impact on performance even after accelerating the code with vectorization. Figure 4.11b shows the same series of tests as Figure 4.11a with vectorization enabled. The A64FX uses 512 bit wide SIMD vectorization for most of its computational power. To better utilize the A64FX we implement additional vectorization optimizations for the advance particles kernel. The vectorization improvements cut the runtime down by up to 47% compared to the measurements in Figure 4.11a. We also see that the overhead for the fixed-point optimization is greater than the half-precision overhead. The exact cause for the performance difference is unclear. Like the V100, the A64FX supports half-precision without vector intrinsics. The performance difference is likely due to architectural differences between the V100 and Power9.

To isolate how vectorization improves performance at such a high rate, we break down the execution times into the major kernels, similar to the tests with the Power9 platform. Figure 4.12a shows the kernel runtimes for the non-vectorized code, and Figure 4.12b for the vectorized code. The performance gain is due to the advance



(a) without vectorization



(b) with vectorization

Figure 4.12: A64FX's detailed runtime breakdown of key particle focused kernels using the optimized version of VPIC without (a) and with (b) vectorization. Remaining kernels are collected under the Other category.

particles kernel (blue bars in the figures) that we have further optimized for this test. Furthermore, the two figures confirm the speedup patterns observed for Power9 – once again, by reducing the data representation in VPIC, we reduce the data movement from and to the main memory, and ultimately we gain in performance for our fastest optimizations (i.e., VPIC+CW+HP). As the capabilities of the A64FX are revealed and refined, we expect our performance to improve further.

4.3 Accuracy Evaluation

Measuring VPIC accuracy is difficult due to the lack of rigorous error quantification for the PIC method. We measure numerical accuracy by testing three different benchmark problems. In each test, we compare the single-precision VPIC 1.2 against VPIC 2.0 with our half-precision and 16-bit fixed-point optimizations. The first benchmark problem is a simple 1-D problem with periodic boundary conditions and an arbitrary number of particles N_p. Each particle has a constant weight $\frac{1}{N_p}$ and an initial momentum of $V_0 = 0$. The initial potential and electric field are given by

$$\phi_i(x) = \begin{cases} x(1-x_i), & 0 < x < x_i \\ x_i(1-x), & x_i < x < 1 \end{cases},$$

$$\phi_b(x) = \frac{x^2 - x}{2}, \quad \phi(x) = \sum_{i=1}^{N_p} \phi_i(x) w_i + \phi_b(x), \quad E(x) = -\frac{\partial \phi}{\partial x}.$$

This test has an analytical solution for particle position described by

$$x_i = \frac{1}{N_p}\left[\left(\sum_k x_{k0}\right) - \frac{N_p+1}{2} + i\right] - \frac{1}{N_p}\left[\sum_k (x_{k0} - x_{i0}) - \frac{N_p+1}{2} + i\right]\cos(t).$$

We can track errors in an individual particle's position over time using the analytical solution. For simplicity, we set the number of particles to two. Table 4.3 describes the

Table 4.3: Relative memory usage, run-times, and accuracy for each potential particle position representation.

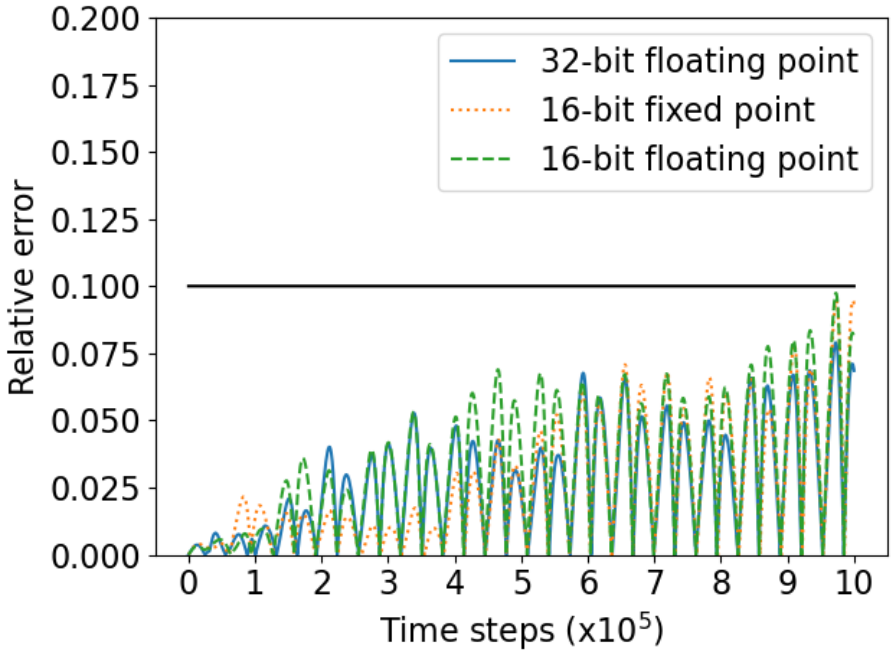
Format	Space reduction (# times)	Accuracy (Digits)
32-bit floating-point	1.00x	7.225
16-bit floating-point	0.81x	3.311
16-bit fixed-point	0.81x	4.515

relative memory usage and accuracy for each particle position representation. The 16-bit fixed-point setting guarantees an additional decimal digit of precision compared to the 16-bit floating point. This two-particle test problem can demonstrate the effects of our optimizations and grid resolution on particle position accuracy. Relative position error is tracked across 1,000,000 time steps with two different grid resolutions (i.e., $10,000$ and $6,000$ cells). The accuracy results for the two resolutions are shown in Figure 4.13a and Figure 4.13b, respectively.

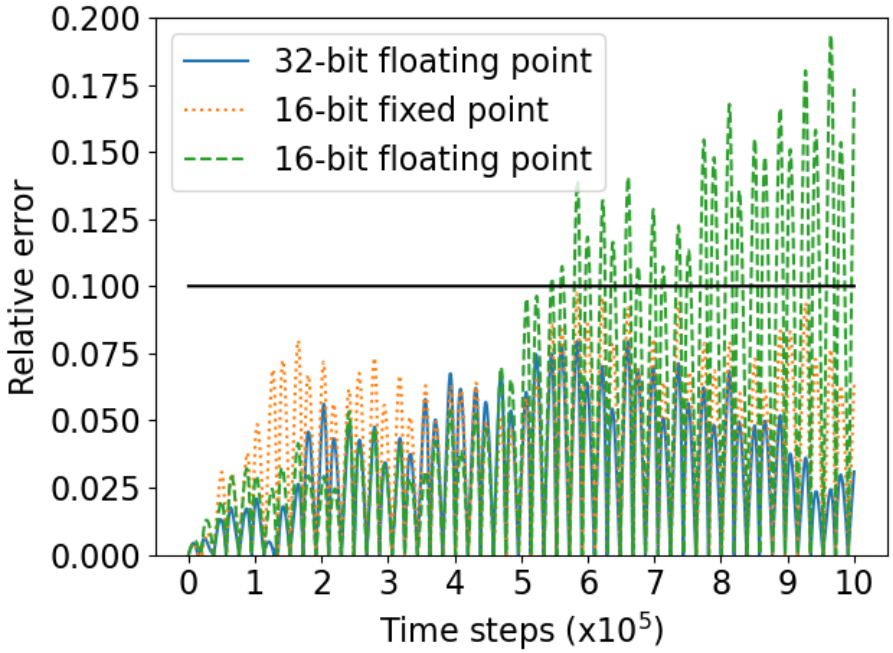
Both the 16-bit floating-point and the 16-bit fixed-point formats trade storage space for precision as shown theoretically in Table 4.3: 16-bit floating-point maintains 3.311 decimal digits of precision while 16-bit fixed-point maintains 4.515 digits, a significant drop from the 7.225 digits for 32-bit floating-point. Figure 4.13a demonstrates that a sufficiently high-resolution grid can make the difference in position precision immaterial since the bits available are used to describe a smaller physical offset from the closer together grid points.

It is important to note that grid resolution, time step length, and simulation length are directly related. High-resolution grids cause the time step length to shorten, which in turn causes the number of time steps and the overall runtime to increase for a fixed simulation length. As a result, most simulations keep grid resolution as coarse as possible to minimize the number of simulated time steps. Figure 4.13b shows the same test with a lower resolution grid (i.e., 6,000). The effect of grid resolution on accuracy is clearly shown by the 16-bit floating-point format failing to maintain sub 10% relative error in this test. The fixed-point format has a notable increase in accuracy over half-precision and is comparable to the 32-bit single-precision floating-point; both 16-bit fixed-point and 32-bit floating-point achieve sub 10% relative error throughout the test. Our results indicate that 16-bit fixed-point not only enables larger simulations than 32-bit single-precision but also improves accuracy over half-precision.

The second benchmark problem models the two-stream instability, which is an electrostatic instability commonly seen in plasma physics. Two counter-streaming



(a) 10,000 cells



(b) 6,000 cells

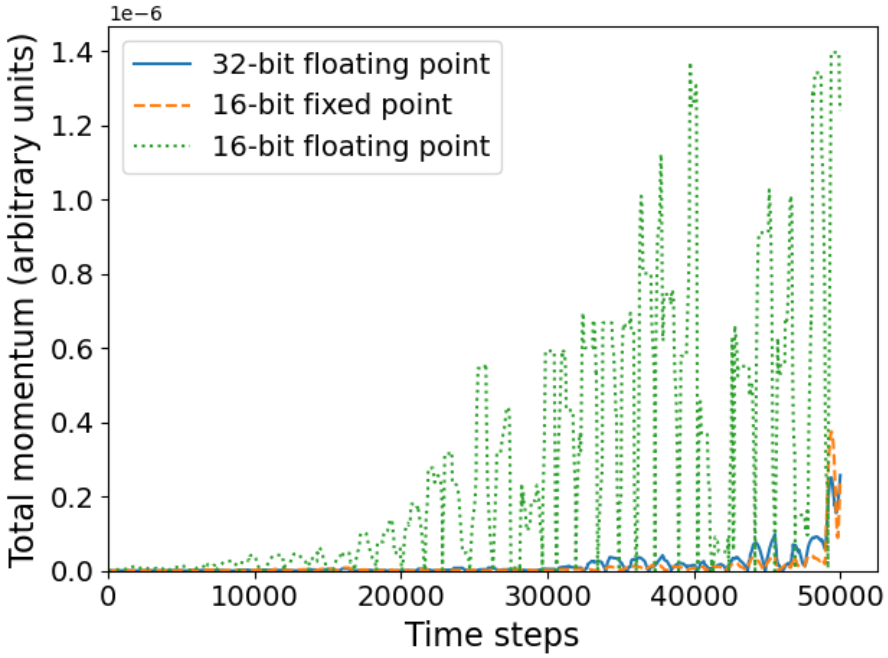
Figure 4.13: Accuracy comparison VPIC simulations with different grid resolution. Particle weight is kept constant and particle position is shown using 32-bit floating-point, 16-bit floating-point, and 16-bit fixed-point.

electron beams move in a stationary ion background (94). The streams are vulnerable to electrostatic perturbations that result in charge bunching behavior (95). The simulation is based on the two-stream instability deck described in (96). The two-stream test focuses on verifying the conservation of momentum. In Figure 4.14a, we observe how the different position representations affect the conservation of momentum. The two-stream instability is in 1D with 4,096 cells, and the system's total momentum is zero. Half-precision fails to conserve momentum adequately, while fixed-point performs similarly to single-precision. Increasing the number of cells would allow half-precision to conserve momentum similarly to single-precision at the cost of decreasing the time simulated per step.

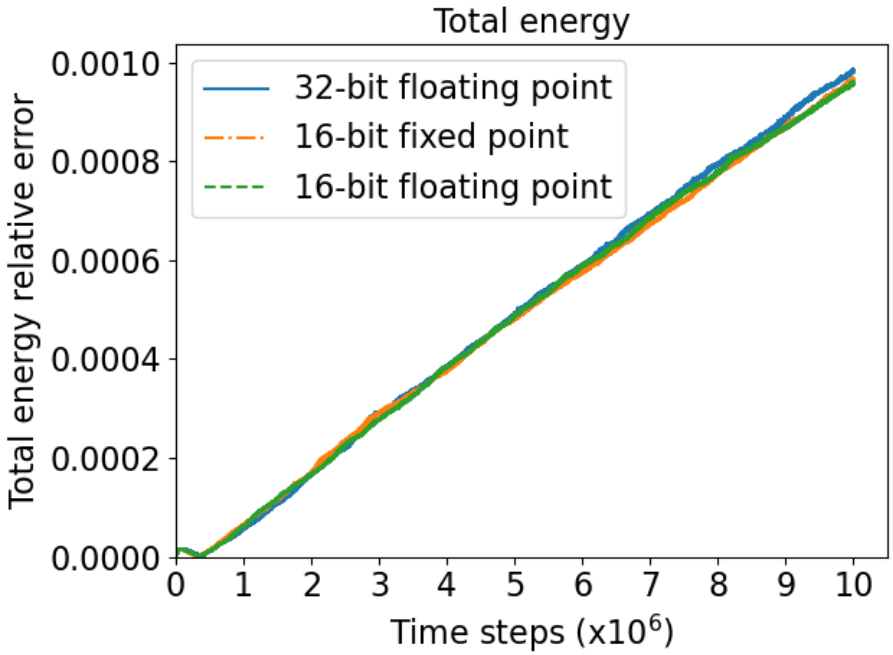
The third benchmark problem simulates the Weibel instability (97), of which the two-stream instability is a special case. Like the two-stream instability, the Weibel instability can arise from counter-streaming beams. Unlike the two-stream instability, the Weibel instability has electromagnetic perturbations resulting in elongated plasma structures (filamentation). The process converts the kinetic energy of the particle beams into magnetic field energy (98). This test aims to verify that energy is conserved with the particle format optimizations applied. Figure 4.14b clearly shows that total energy is conserved very well with less than 0.1% relative error across 10 million time steps for the Weibel instability. The curves for single-precision, half-precision, and fixed-point are tightly grouped. Thus, with our optimizations applied, energy conservation is upheld with minimal variation compared to single-precision VPIC 1.2.

4.4 Importance and Applicability of Lower Precision

Leveraging alternative numerical formats opens up many opportunities for improving simulation scale and execution speed, enabling advanced supporting features, and



(a) Two-stream instability.



(b) Weibel instability.

Figure 4.14: Conservation of momentum absolute error for the two-stream instability (a) and conservation of energy relative error for the Weibel instability.

accelerating scientific discovery. As more hardware manufacturers focus on AI/ML features, techniques for leveraging mixed precision in HPC software increase in value. Using alternate number formats increases the available compute and bandwidth by a factor of two or more on supported hardware. Controlling numerical precision by taking advantage of local coordinate systems in grid simulations enables lower precision formats, such as half-precision in a large class of simulations. Fixed point formats for normalized data can provide many of the half-precision benefits while incurring less precision loss. Alternate number formats combined with accuracy recovery techniques in HPC codes facilitate the following benefits:

- Leverage additional accelerator features normally left unused outside specialized AI/ML frameworks.
- Reduce storage costs and data movement and increase available computational capabilities while enabling larger, more realistic simulations on hardware such as the A64FX or commodity GPUs that have limited memory capacity and expensive data transfer times between host and device.
- Increase the available free memory for more complex diagnostics, in-situ analysis, resilience, advanced data management, and other additional features supporting the simulation.

Enhancing the software with mixed precision is important for maximizing hardware use. Lower precision formats require careful consideration and techniques for maintaining scientific accuracy. The benefits of lower precision formats enable more opportunities for science by accelerating simulations and allowing the use of older or consumer hardware for rapid testing without wasting allocation time on larger clusters. Our suite of particle format optimization can be applied to other grid based simulation codes such as WarpX (99) and GROMACS (100). One difficulty with our optimizations is that they are dependent on how position is represented in each applications data structures. The particle structure in VPIC differs from WarpX and

GROMACS, thus applying our optimizations requires application specific changes and knowledge. We can assist developers interested in our optimization by creating a portable library for mixed precision data types. An abstraction layer is necessary for number formats like half precision and bfloat16 which use hardware specific features. Fixed point formats do not need any specialized code and can easily be included in the library.

Chapter 5

Pruning Redundant Data with Deduplication

Storing the entire checkpoint history has high I/O overhead and requires too much storage space. Large-scale checkpointing involves many processes distributed on the compute nodes (typically one process per GPU) that compete for a limited amount of I/O bandwidth. This introduces large I/O overheads and, therefore, increases the end-to-end runtime. Emerging checkpointing use cases exacerbate the problem with higher checkpoint frequencies. Even in resilience, where the checkpointing frequency is low, I/O overheads are significant enough to warrant asynchronous multi-level checkpointing methods. In these methods, the processes write the checkpoints to the fastest local memory (e.g., GPU memory) and then let the application continue running. In the background, they flush the checkpoints asynchronously to slower memory tiers and, eventually, the external repository. Incremental checkpointing techniques help address the I/O challenges by only saving data that has changed since the previous checkpoint. However, at high checkpointing frequencies, such methods have two limitations. First, only a limited amount of spare space is available on the fastest memory tiers to cache checkpoints, so the HPC workflow may be delayed if it produces new checkpoints faster than they can be flushed to slower memory tiers.

Second, since the entire checkpoint record needs to be retained, its accumulated size may quickly explode to large sizes and produce unacceptable resource utilization, even if performance considerations were not a concern. Thus, there is a need to simultaneously reduce both the I/O overheads and space utilization of checkpoints.

To minimize I/O overhead and space utilization, we develop an incremental checkpointing method based on Merkle trees to deduplicate redundant data. Our method identifies and deduplicates repeating patterns across the checkpoints of the entire checkpoint record, extracts a compact metadata representation of these repeating patterns, serializes the differences and the compact metadata representation efficiently into host memory for asynchronous transfer to other storage tiers, and takes advantage of GPU accelerators to scale to tens of thousands of GPU cores. Our contributions towards pruning redundant data in checkpointing are:

- We introduce five design principles at the core of our GPU-accelerated data deduplication method. (Section 5.1)
- We propose a highly parallel algorithm based on our design principles that coalesce contiguous chunks into a hierarchic set of repeating patterns using Merkle trees. (Section 5.2)
- We illustrate how to implement our algorithms by proposing a research prototype that leverages Kokkos for performance portability (Section 5.3)
- We demonstrate the benefits of our solution for a graph application using extensive experiments comparing our method with state-of-the-art incremental checkpointing and compression methods for various graphs. (Section 5.4)
- We identify our deduplication work's importance and broader applicability for improving incremental checkpointing performance. (Section 5.5)

5.1 Design Principles

We define four fundamental design principles that guide our deduplication method. They are: (1) the deduplication of data chunks using spatiotemporal agnostic fine-grain hashing, (2) the compact hierarchic representation of contiguous repeating patterns with effective representation such as Merkle trees, (3) the combined serialization of metadata and unique chunks into a consolidate difference that is optimized for transfer to host memory; and (4) the fusion of GPU kernels for massive GPU parallelism. Figure 5.1 depicts a high-level overview of our deduplication method and how the principles guide its development.

5.1.1 Deduplication in Spatiotemporal Agnostic Format

We assume an HPC workflow in which processes are co-located on the same compute node, each assigned to a dedicated GPU. In this case, each process produces a high-frequency checkpoint record directly into the GPU memory. Since the spare GPU memory available for checkpointing is limited, we cannot afford to hold the entire checkpoint record in the GPU memory, even if we apply an incremental checkpointing technique that stores only the differences. Furthermore, even if we had enough free space on the GPU memory to hold the entire checkpoint record, it is not feasible to compare a new checkpoint with all previous checkpoints in the historical record to identify the differences. Thus, we propose a hash-based method that splits a new checkpoint into fine-grain chunks (in the order of tens or hundreds of bytes), then hashes each chunk to produce a set of unique chunk hashes, which can be compared with the accumulated set of unique chunk hashes of the checkpoint record to identify the data chunks that are unique to the new checkpoint. Using this method, a chunk must be stored only the first time it is encountered, regardless of how often it appears again in the same checkpoint (spatial duplication) or future checkpoints (temporal duplication). Although such methods have been proposed before in the context of incremental block storage (101), we operate directly in the memory of

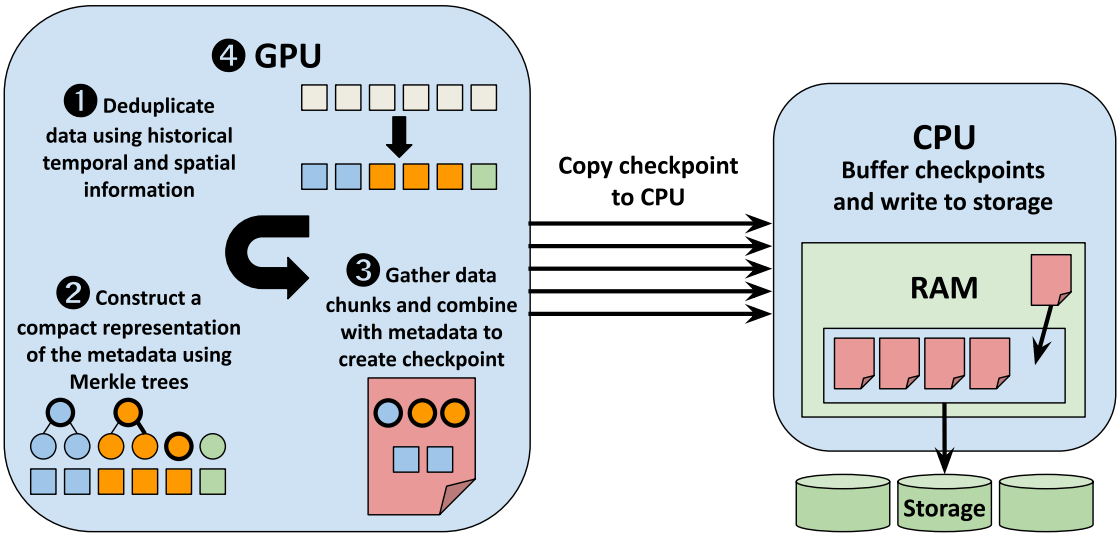


Figure 5.1: Summary of our incremental checkpointing method based on our design principles. Our method is designed for GPUs and can be combined with additional resilience features performed on the CPU. Each process performs deduplication on its own GPU.

GPU accelerators, which introduces additional challenges. First, unlike storage systems, we cannot afford to access a separate metadata repository with a historical record of unique hashes due to high I/O latency. Therefore, we propose to keep a distinct record for each process in the GPU memory. Second, as the historical record of unique hashes grows over time, there is a need for efficient indexing and lookup techniques that are GPU-optimized. To this end, we leverage specialized hash tables, as detailed in Section 5.3.

5.1.2 Hierarchic Representation with Merkle Trees

A GPU-optimized data structure that holds the entire checkpoint record of unique chunk hashes enables us to identify a minimal set of chunks representing the difference between new and older checkpoints. However, this set may be very large since we use fine-grain chunk sizes. Consequently, storing naive metadata about the chunks (e.g., an entry for each chunk that identifies the checkpoint ID and offset where it first occurred) leads to an explosion of the metadata size, which may dramatically reduce the space savings, even if the difference is negligible. On the other hand, it is essential to note that the chunks may form large contiguous regions that repeat both within the same checkpoint and across past checkpoints. Therefore, there is an opportunity to reduce the metadata sizes by identifying and leveraging such contiguous regions directly. To this end, we propose a hierarchical Merkle tree-based (102) method that stores hashes corresponding to different arrangements of non-overlapping adjacent regions in the historical record of unique hashes (bound by up to two times more hashes than the naive method) to identify a close to minimum number of contiguous regions that cover the difference. Using this method, the metadata size can be dramatically reduced under the right circumstances, as we need to store only the difference between the checkpoint ID and offset where an entire region appeared the first time. To this end, we introduce a specialized algorithm detailed in Section 5.2.

5.1.3 Combined Serialization of Metadata and Data Chunks

Once we have identified the smallest set of regions and unique chunks that make up the difference, we need to consolidate the data and metadata into the host memory to obtain a single checkpoint object that can be flushed asynchronously further down the storage hierarchy. However, these unique chunks may be scattered all over the GPU memory, especially if the deduplication is effective and the difference is negligible. Therefore, a naive strategy that initiates transfers of individual chunks from the GPU memory to the host memory suffers significant I/O bandwidth degradation since it involves non-trivial latencies to set up the transfer, not to mention cache misses. Therefore, we propose serializing the metadata and the unique chunks into a consolidated difference directly on the GPU memory. Then, for the consolidated difference, initiating a single device-to-host data transfer is enough, which can take advantage of the full PCIe bandwidth that links the GPU memory with the host memory. Even if the consolidation leverages high GPU-to-GPU memory bandwidth, it is non-trivial because coalesced memory accesses that occur when concurrently running threads access memory close to each other need to be considered. Doing so allows the hardware to predict and retrieve data ahead of time. Cache hierarchies are also better utilized. To this end, we design a specialized serialization method that pre-calculates offsets in the consolidated difference and assigns GPU threads to parallelize the data transfers by the observations above.

5.1.4 Massive GPU Parallelism

Without taking advantage of the massive parallelism of GPUs, even simple steps in our method, such as hashing the data chunks or performing GPU memory accesses and GPU-to-GPU data transfers, are time-consuming and lead to large overheads. Therefore, our method needs to scale to many GPU cores on the order of ten thousand on recent GPUs. However, this is non-trivial because hashing of data chunks, indexing and lookup in the hash historical record, generating compact metadata

representations, and consolidating checkpoint differences are complex operations with tight dependencies. Therefore, it is not enough to reason about these aspects as independent steps that can be parallelized using separate GPU kernels, as such a naive method would introduce unacceptable latencies associated with submitting and executing new kernels. To address this issue, we propose to use a single fused kernel that takes advantage of containers and abstractions offered by performance portability abstractions such as Kokkos to parallelize the execution into related "waves" that ensure the work is evenly distributed and maximize coalesced memory accesses without delays between waves, as detailed in Section 5.3.

5.2 Merkle Tree-based Compact Metadata Representation

We propose a heuristic algorithm to identify a close to minimal number of non-overlapping contiguous regions that fully describe the difference between a new checkpoint and all previous checkpoints in the checkpoint record. Using this method, in addition to the unique chunks encountered in the latest checkpoint, we only need to store a small amount of metadata to restore the checkpoint later fully.

Specifically, we assume the fine-grain chunks are the leaves of a (potentially incomplete) binary tree. Then, we compute the hashes for the leaves and, in a bottom-up fashion, for the intermediate tree nodes by hashing the left child's hash with the right child's hash, similar to Merkle trees. We only go one level up if the hash of the left and right child were found in the historical record of unique hashes and represent a contiguous region that can be consolidated. If this is the case, the new region is added to the historical record of unique hashes. Then, we collect the intermediate nodes that were touched by this process and are the closest to the tree's root. This yields a compact representation of the checkpoint difference, both with respect to the new chunks encountered the first time and the reused chunks from previous checkpoints.

To understand why the method works, consider an example depicted in Figure 5.2. The checkpoint historical record includes a single first checkpoint taken fully. The historical record of unique hashes consists of all possible non-overlapping regions corresponding to the intermediate Nodes 0-14. Then, a second checkpoint is taken, and the new Chunk 11 is identical to the previous Chunk 11 (which we refer to as *fixed* duplicate). In contrast, Chunk 12 is identical to another previous Chunk from the first checkpoint other than 12 (which we refer to as *shifted duplicate*). All other chunks between the first and second checkpoints are different.

The algorithm works as follows. First, we hash the chunks of the second checkpoint and insert any new hashes into the historical record of unique hashes, which results in four inserts I, J, K, L, referred to as *first-time occurrences*. Note that Chunks 13 and 14 are identical to Chunks 7 and 8. Even though they belong to the second checkpoint, they are still marked as shifted duplicates, just like Chunk 12. Chunks 11 and 12 are the only ones that form two non-contiguous regions that cannot be consolidated. Therefore, we stop. For all other leaves, we go one level up. We consolidate Chunks 7 and 8 into Region 3 and Chunks 9 and 10 into Region 4. Region 3 and Region 4 are added to the historical record of unique hashes. Then, we consolidate Chunks 13 and 14 into Region 6. Since the hash of Region 6 is identical to the hash of Region 3, which already is in the historical record of unique hashes, the entire Region 6 is marked as a shifted duplicate. The process continues only for Regions 3 and 4, which can now be consolidated into Region 1 and inserted in the historical record of unique hashes. We obtain the compact metadata representation of the difference as a set of three non-overlapping Regions: $1, 12, 6$. We can omit Chunk 11 from the difference since it remains unchanged from the first checkpoint. Finally, we obtain a mix of metadata describing the first-time occurrences and shifted duplicates, followed by the chunk content corresponding only to the first-time occurrences. Using this method, we save only three metadata entries compared with the naive method, which saves a metadata entry for each non-fixed duplicate chunk (referred to as the List method).

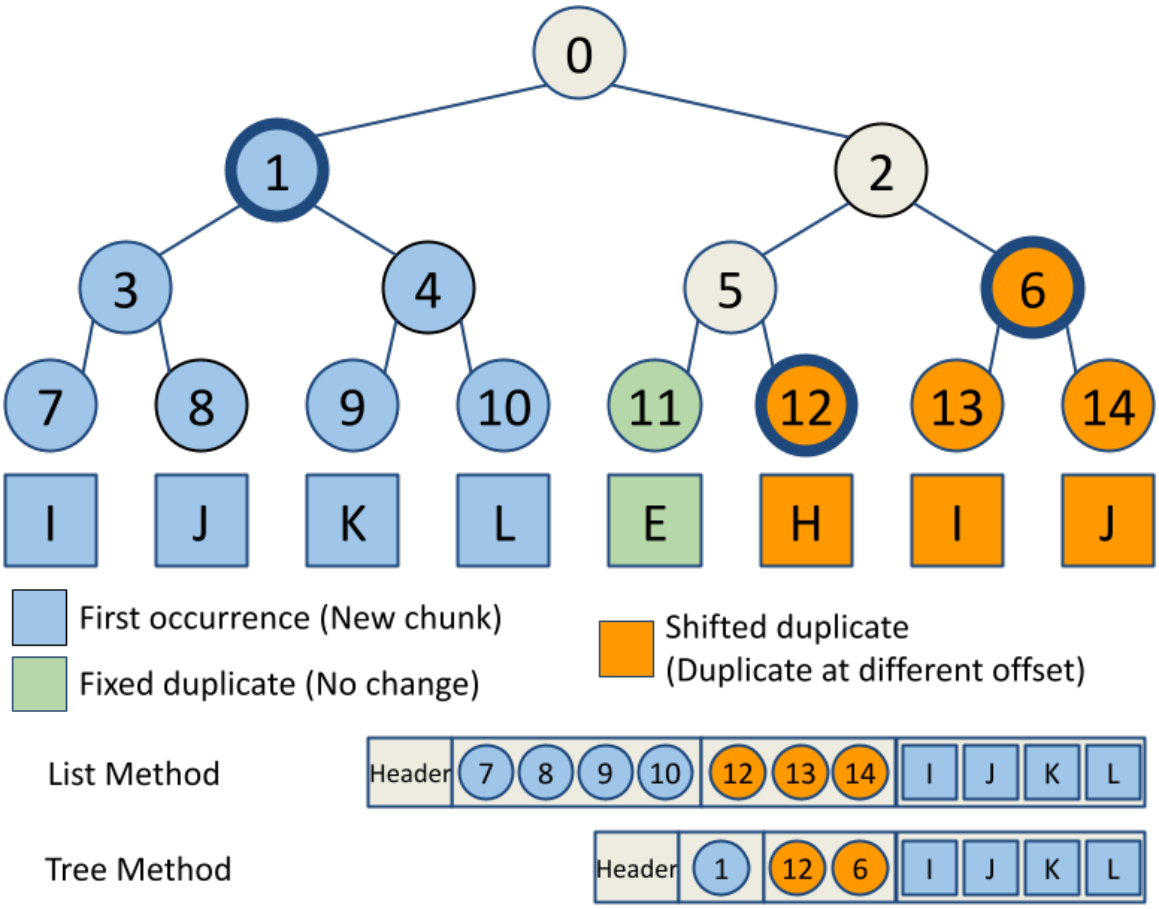


Figure 5.2: Example of compact metadata representation. Compared with a naive method, our method reduces the metadata amount from 7 entries to 3.

Now, the compact metadata and new unique chunks are ready to be serialized and transferred to the host memory.

We show our metadata compaction method in greater detail in Algorithm 2. The historical record of unique hashes is denoted $Map[hash]$, while $Tree[node]$ and $Label[node]$ are temporary data structures that hold the hashes of leaves (intermediate nodes) and, respectively, the type of the region covered by each leaf (intermediate node). Identical labels mean a region can be consolidated.

According to the design principle that aims to leverage the massive parallelism offered by GPUs, we parallelize this algorithm level-by-level. However, to avoid a situation where shifted duplicates are hashed faster than first-time occurrences (which leads to a missing entry in the historical record of unique hashes and, therefore, missed deduplication opportunities), we perform the parallelization in two stages: first, we process the sub-trees corresponding to the first-time occurrences, then we process the sub-trees corresponding to the shifted duplicates. To restore a checkpoint from the differences, it is enough to start from the first-time occurrences, fill the fixed duplicates, and assemble the shifted duplicates from the corresponding checkpoint ID (a previous checkpoint or the current checkpoint to be restored).

5.2.1 System Architecture

On a shared platform, a large number of compute nodes compete for the I/O bandwidth of an external repository, typically a parallel file system. Figure 5.3 depicts the general architecture of an asynchronous checkpointing that integrates with our GPU-accelerated method. Each compute node features multiple GPUs. Each application process uses a dedicated GPU and stops during checkpointing to perform the deduplication on the GPU according to the steps illustrated in Figure 5.1. Then, it transfers the consolidated difference to the host memory and resumes the computations. From this point forward, an asynchronous multi-level checkpointing runtime flushes the differences stored in the host memory down the storage hierarchy

Algorithm 2 Deduplication using compact metadata.

Input: $Chunks, Tree, Leaves, Labels, Map$

1: **for all** $leaf \in Leaves$ **do in parallel**
2: $digest \leftarrow$ Hash($Chunk(leaf)$)
3: **if** $digest == Tree(leaf)$ **then**
4: $Labels(leaf) \leftarrow$ FIXED_DUPL
5: **else**
6: $entry \leftarrow (leaf, chkptID)$
7: $success \leftarrow Map$.insert($digest$,$entry$)
8: **if** $success$ **then**
9: $Labels(leaf) \leftarrow$ FIRST_OCUR
10: **else if** not $success$ **then**
11: $(leaf_{old}, chkptID_{old}) \leftarrow Map[digest]$
12: $sameID \leftarrow chkptID_{old} == chkptID$
13: **if** $leaf < leaf_{old}$ && $sameID$ **then**
14: $Labels(leaf_{old}) \leftarrow$ SHIFT_DUPL
15: $Labels(leaf) \leftarrow$ FIRST_OCUR
16: $leaf_{old} \leftarrow leaf$
17: **else**
18: $Labels(leaf) \leftarrow$ SHIFT_DUPL
19: **end if**
20: **end if**
21: $Tree(leaf) \leftarrow digest$
22: **end if**
23: **end for**
24: **for** $level \in Tree$ **do**
25: **for all** $node \in level$ **do in parallel**
26: **if** $child_l$ and $child_r$ are FIRST_OCUR **then**
27: $Labels(node) \leftarrow$ FIRST_OCUR
28: $Tree(node) \leftarrow$ Hash($(Tree(child_l, child_r))$
29: $Map[Tree(node)] \leftarrow (node, chkptID)$
30: **end if**
31: **end for**
32: **end for**
33: **for** $level \in Tree$ **do**
34: **for all** $node \in level$ **do in parallel**
35: **if** $Labels(child_l) \neq Labels(child_r)$ **then**
36: save roots $child_l$ and $child_r$
37: **else if** $child_l$ and $child_r$ are SHIFT_DUPL **then**
38: $Tree(node) \leftarrow$ Hash($Tree(child_l), tree(child_r)$)
39: **if** $Tree(node) \in Map$ **then**
40: $Labels(node) \leftarrow$ SHIFT_DUPL
41: **else**
42: save roots $child_l$ and $child\_r$
43: **end if**
44: **end if**
45: **end for**
46: **end for**

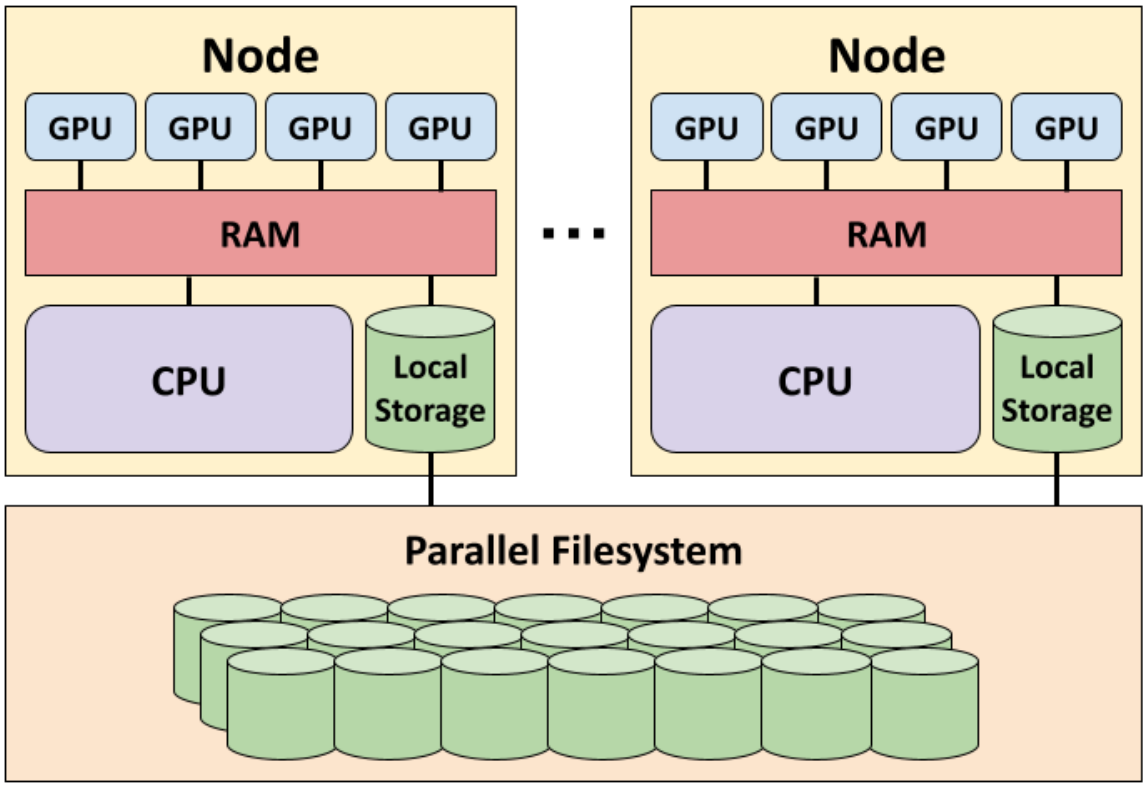


Figure 5.3: Architecture of multi-level asynchronous checkpointing that integrates our GPU-accelerated deduplication.

(local storage, parallel file system). Since each process maintains its record of unique hashes on its GPU, the only bottleneck is the competition for PCIe bandwidth between the GPUs when transferring the differences. This competition is nevertheless much lower than when the full checkpoints are transferred to the host memory, in which case much larger sizes are involved. Furthermore, intermediate storage tiers like host memory and local SSDs are filling up slower, which helps amortize the competition for the I/O bandwidth of the parallel file system. Ultimately, this leads to better utilization of the entire storage hierarchy, lower I/O overheads, and storage space savings.

5.3 Implementation Details

Our implementation builds on performance portable abstractions that enable efficient massive parallelization into fused kernels while offering optimized implementations of GPU-aware hash tables (used to maintain the historical record of unique hashes).

5.3.1 Parallelization with Kokkos

We implement our method using the Kokkos performance portability framework (1) for parallel execution on CPUs and GPUs. Kokkos includes several execution and data structure abstractions for developing scalable applications. We use the UnorderedMap provided by Kokkos. The UnorderedMap is designed to handle thousands of concurrent insertions. The map is lock-free and minimizes the use of atomic operations. This performance-focused design is essential for calculating and identifying differences between thousands of hashes. Kokkos provides flexible parallel execution constructs that allow direct control of work division. The ability to control which chunks are assigned to which threads and in what order significantly impacts performance, as outlined in the subsequent sections. Merkle trees are complete binary trees, so we store them in a flattened array and identify parent-child relationships

using simple formulas based on the offset in the array. This simplifies tree search and management on the GPU, as the array format does not waste space on unused pointers.

5.3.2 Efficient Hash Calculation Using GPUs

We use the 128-bit Murmur3 (103) hash function for comparing chunks. Murmur3 is a common non-cryptographic hash function used for hash tables. A fast hash function such as Murmur3 is necessary to maximize deduplication throughput. Slow cryptographic hash functions such as MD5 (104) would introduce a bottleneck. To efficiently leverage the GPU hardware, we structure the code such that successive threads compute hashes for consecutive chunks. By doing so, we ensure that the stride between memory accesses is reduced. Reducing the stride decreases the number of memory accesses to global memory, speeding up the hash calculation. Optimized memory access patterns are necessary to fully utilize the GPU's bandwidth and computational capabilities. This is particularly true for hashing data since the computational cost of Murmur3 is already low. Our method may compute and store up to twice the number of hashes in the historical record of unique hashes since the number of intermediate nodes equals the number of leaves minus one. However, this is a worst-case scenario encountered only when the checkpointed data fully changes during the checkpoint interval. This can be easily detected, and incremental checkpointing can be deactivated accordingly. Conversely, when the checkpointed data remains unchanged between checkpoints, our method may calculate intermediate nodes unnecessarily. This may be mitigated by adopting a top-down method. Another important aspect is the size of the chunks, which needs to be larger than double the size of the hash values. In our case, each Murmur3 hash digest is 16 bytes: so long as the chunk size exceeds 32 bytes, the cost of computing an inner node is lower than that of a leaf node. This represents a trade-off: hashing smaller chunks of data improves the memory access pattern by reducing the stride between memory accesses

but increases the number of inner nodes. Thanks to compact metadata representation, the latter aspect is of lesser concern. Finally, we did not consider hash collisions. If hash collisions are a concern, they can be mitigated by using a cache of chunks that can be directly compared in parallel with the metadata compaction.

5.3.3 High Throughput Serialization of Scattered Chunks

As with hash calculation, optimizing memory accesses for gathering scattered chunks is essential to obtain high GPU-to-GPU transfer throughput for serializing the consolidated checkpoint difference. Rather than having each thread copy a different chunk, we use a team of threads to copy each chunk into the contiguous buffer. This ensures that memory accesses coalesce and memory bandwidth utilization is maximized. Without such optimizations, even if the transfer from the GPU to the host memory is accelerated due to contiguity, this benefit would be negated by poor serialization throughput.

5.4 Performance Evaluation

5.4.1 Test Platforms

We perform our tests on two supercomputers at Argonne National Laboratory's Leadership Computing Facility (ALCF): ThetaGPU and Polaris. Theta GPU is a DGX A100-based system comprised of 24 NVIDIA DGX A100 nodes, each with eight NVIDIA A100 Tensor Core GPUs and two AMD EPYC 7742 64-core CPUs. Memory-wise, each node is equipped with 1 TB of DDR4 and 320 GB GPU memory for 24 TB DDR4 and 7.6 TB GPU memory. The nodes are interconnected using 20 Mellanox QM9700 HDR200 40-port switches wired in a fat-tree topology. External storage is provided by a Lustre parallel file system, which is mounted using POSIX and provides an aggregated I/O bandwidth of 250 GB/s. Polaris is a 560-node HPE Apollo 6500 Gen 10+ system. Each node consists of an AMD 32-core EPYC 7543P

CPU, 4 A100 GPUs, 512 GB of DDR4, two 1.6 TB SSDs, and 4 A100 GPUs. The nodes are connected using the Slingshot 10 network. The number of processes tested ranges from a single process to 64 processes. Each process has its own GPU and is isolated from the other processes.

5.4.2 Experiment Methodology

Use case: ORANGES

Our driver application for the experimental evaluation is ORbit ANd Graphlet Enumeration at Scale (ORANGES), a parallel graph application that takes a graph as an input and computes each vertex's graphlet degree vector (GDV) to enable graph matching. A graphlet is an induced subgraph of a small number of vertices. A graphlet degree vector can be seen as a generalization of the degree concept (105). The degree of a vertex represents the number of times a vertex is part of an edge. The graphlet degree vector represents the number of times a vertex is part of different graphlets, with one entry for each graphlet. GDVs are used for graph-matching applications, such as in comparing phylogenetic networks in bioinformatics and comparing event graphs in large-scale HPC applications. We create the GDV on all two to five vertices graphlets. Each vertex in the graph is associated with a vector of size 72, representing the different positions (or orbits) of the vertex in the 30 possible graphlets (more details on the graphlets and orbits are given in (106)). For each vertex in the graph, we identify the graphlets associated with it and its different orbits in the graphlets. Based on this calculation, we increase the count of each orbit in GDV associated with the vertex. If the graph is sparse, then the GDV is also sparse, as not all graphlets are formed. For example, triangles are rarely formed in event graphs representing communications in HPC simulations and almost planar road graphs. Due to this, only ten possible 30 graphlets are formed frequently, and the remaining 20 very rarely, if at all. Graphs will also have repeated substructures, which can result in some GDVs being similar to others. The updated pattern depends on the input graph, simplifying

the exploration of different patterns. These characteristics make ORANGES a good candidate to showcase the benefits of our work.

Alternative Deduplication and Compression Methods

We compare our method (henceforth denoted *Tree*) with a baseline checkpointing method that permanently stores a full checkpoint (denoted *Full*). We also implemented a *Basic* incremental checkpointing method that breaks the checkpoint into chunks, hashes the chunks, and then builds a bitmap to indicate what chunks are new and which chunks remain unchanged. It saves the bitmap and the new chunks. Furthermore, we implemented a *List* method that is identical to our method except for the metadata compaction, which is omitted. Instead, a full list of all first-time occurrences and shifted duplicates is stored along the new chunks. For fairness, both the Basic and List methods benefit from the same massive parallelization optimizations introduced by our method. Furthermore, we use several lossless compression algorithms included with NVIDIA's open-source *nvCOMP* (107) library. Since our application counts graphlets and needs an exact output for correctness, lossy compression is not applicable.

Performance Metrics

When evaluating our work, we focus on data deduplication ratio and deduplication throughput. The deduplication ratio is measured as the size of the full checkpoints divided by the size of the deduplicated checkpoints. Higher ratios indicate greater space savings. Throughput is calculated as the size of the original data divided by the time it takes to create and copy the incremental checkpoint from the GPU to the host memory. In the case of Full, this measures the GPU to host flush throughput. In the case of the other methods, the deduplication throughput includes both the overhead of compression/deduplication and the overhead of GPU-to-host transfers.

Test Scenarios

We present three scenarios that examine different factors affecting our method versus existing methods. The first scenario studies the impact of chunk size on deduplication performance. Chunk size determines the deduplication granularity, which directly affects checkpoint size reduction and the computational overhead of checkpointing. We vary the chunk size from 32 to 512 bytes and compare our method with the Full, Basic, and List method in terms of data deduplication ratio and deduplication throughput. This scenario leverages a single GPU. The second scenario investigates how deduplication benefits accumulate as the checkpointing frequency increases. Specifically, we capture a full initial checkpoint, then another $N-1$ incremental checkpoints evenly distributed during the runtime R (i.e., we use a fixed checkpoint interval R/N). We vary N from 5 to 20 and aggregate the metrics for all captured checkpoints (excluding the first checkpoint). Using this method, we can compare the results for input graphs that produce different runtimes (in the order of minutes). This scenario again leverages a single GPU. The last scenario performs strong scaling tests. We vary the number of GPUs from 1 to 64 and take checkpoints every 10 minutes. At scale, for larger dense graphs, the number of iterations rapidly increases, hence the longer checkpoint interval. We focus on comparing our Tree method with the Full method.

For the scenarios mentioned above, we use a set of graphs with different complexities in terms of vertices and edges (Table. 5.1). Message Race, Unstructured Mesh, Asia OSM, and Hugebubbles are used for the single process tests, and Delaunay is used for the scaling test. The Message Race and Unstructured Mesh scenarios are event graphs representing communication patterns in HPC benchmarks. Asia OSM, Hugebubbles, and Delaunay N24 are graphs from the SuiteSparse collection (108). The event graphs are more sparse than those from SuiteSparse, with fewer dense subgraphs. As noted later, this leads to improved results for the event graphs.

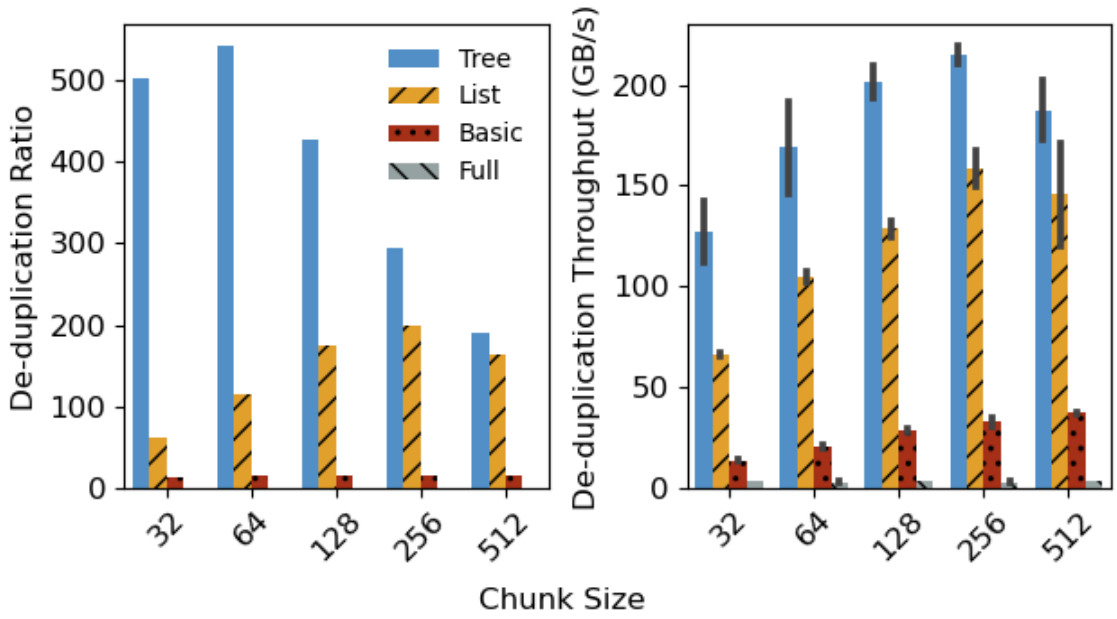
Table 5.1: Input graphs used for our tests. Delaunay N24 is used for the scaling test.

Graph	$\|V\|$	$\|E\|$	GDV size
Message Race	11,174,336	16,761,248	3.26 GB
Unstructured Mesh	14,418,368	21,627,296	4.21 GB
Asia OSM	11,950,757	25,423,206	3.49 GB
Hugebubbles	18,318,143	54,940,162	5.35 GB
Delaunay N24	16,777,216	100,663,202	4.9 GB

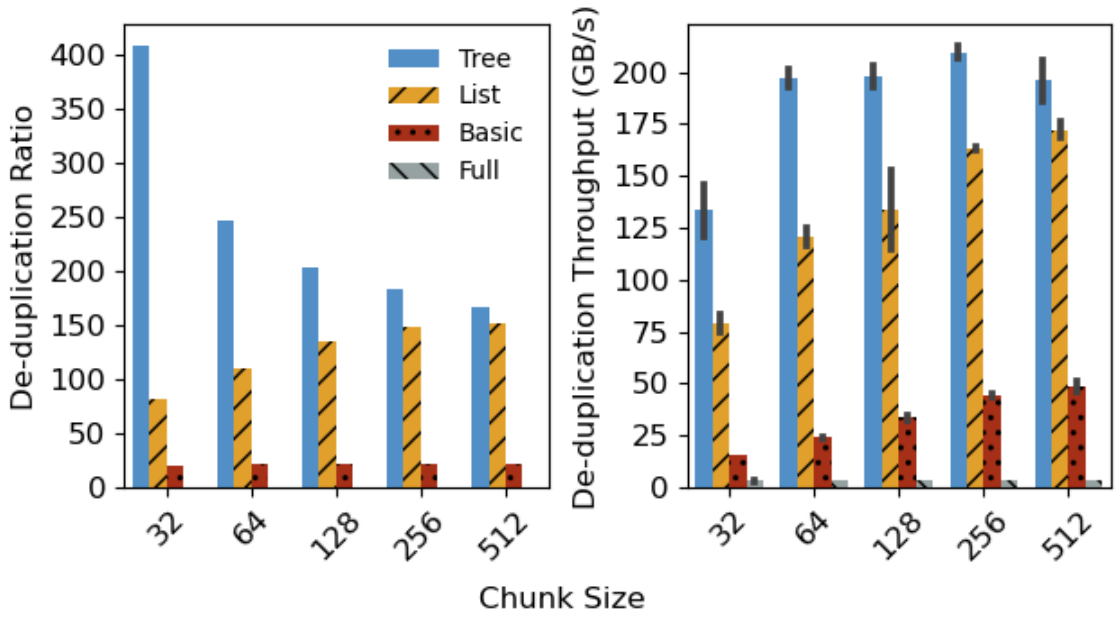
Before running ORANGES, we pre-process the graphs by reordering the vertices using Gorder (109). Gorder is a graph reordering application that relabels a graph's vertices to improve cache performance while maintaining the graph's topological structure. Gorder uses an approximate greedy algorithm with a priority queue to find a graph ordering where connected vertices are stored close together. Keeping the vertices close together improves cache reuse when operating on graphs and is a typical optimization applied by the graph community.

5.4.3 Deduplication Granularity

We evaluate how chunk size affects the deduplication ratio and throughput for our Tree method versus Full, Basic, and List methods on four different input graphs in Figure 5.4. Figure 5.4a highlights the trade-offs when deciding chunk size for the Message Race graph. With 64-byte chunks, our method achieves a five times better deduplication ratio than List (the best among the compared methods). The List method sees a decline in ratio with chunks smaller than 256 bytes—large amounts of metadata cause the decrease. As the chunk size shrinks, more of the checkpoint comprises metadata for tracking duplicate chunks. Our method compacts the metadata, which allows smaller chunk sizes without performance degradation. The throughput values suggest that significant reductions in checkpoint sizes can overcome the additional overheads of identifying compact metadata. Specifically, our method shows superior throughput, matching the improved deduplication ratio performance. Throughput performance starts to degrade with chunks smaller than 256 bytes, where the additional overhead exceeds the benefits of smaller checkpoints. This behavior is typical of all compared methods. Our Tree method benefits the most from small chunk sizes, allowing smaller chunks without decreasing throughput performance. Figure 5.4b shows similar performance characteristics for the Unstructured Mesh graph as the Message Race graph.

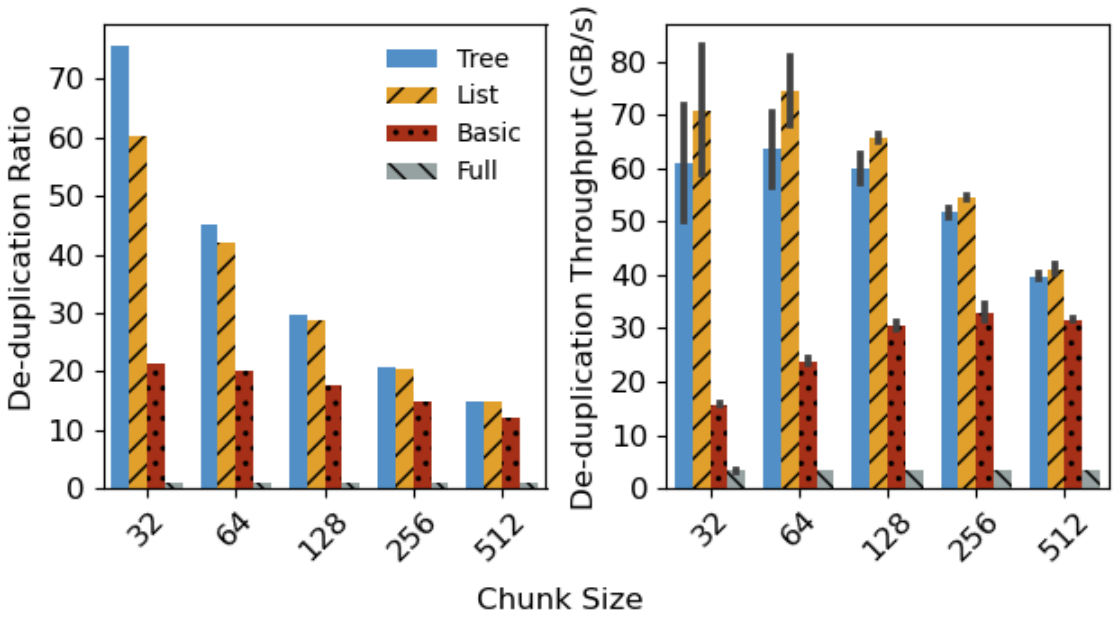


(a) Message Race

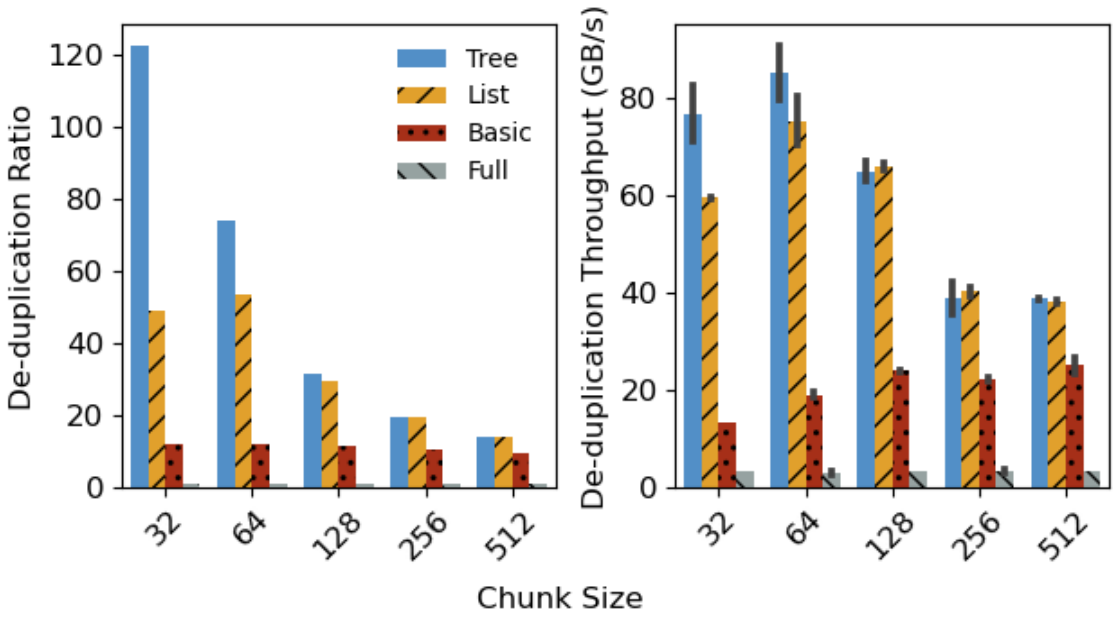


(b) Unstuctured Mesh

Figure 5.4: Impact of chunk size on deduplication ratio and throughput for our method (Tree) vs. other incremental methods (i.e., Full, Basic, and List) for the Message Race, Unstructured Mesh, Asia OSM, and Hugebubbles input graphs. Chunk size ranges from 32 to 512 bytes.



(c) Asia OSM



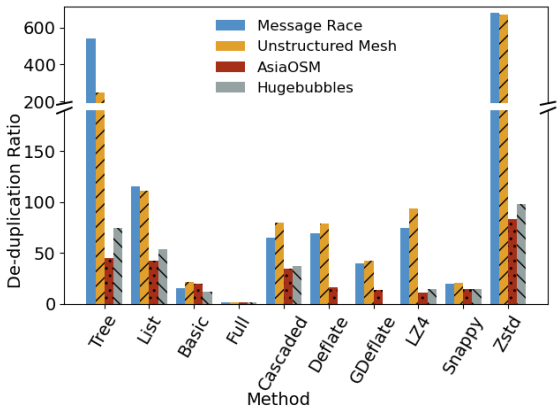
(d) Hugebubbles

Figure 5.4: Continued

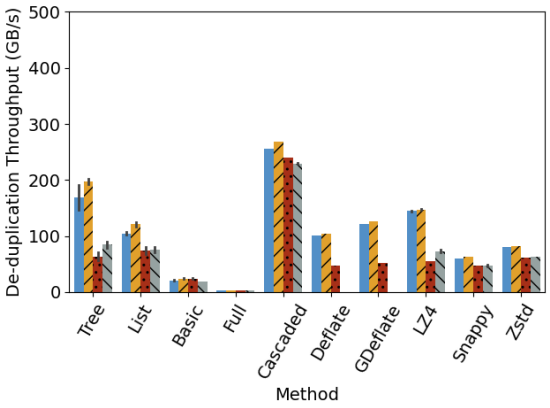
Asia OSM is more challenging to deduplicate than Message Race or Unstructured mesh. Figure 5.4c shows that the deduplication ratio is lower for all methods. Our Tree method only shows notable improvements with 32-byte chunks, outperforming the other methods. Figure 5.4d shows the same metrics for Hugebubbles. Similar to Asia OSM, the Hugebubbles graph is more challenging to deduplicate. Despite the difficulty, we see significant improvements with our Tree method with chunk sizes of less than 128 bytes. With 64-byte chunks and smaller ones, we see a 37% improvement in the deduplication ratio and a 13% increase in throughput.

5.4.4 Deduplication Frequency

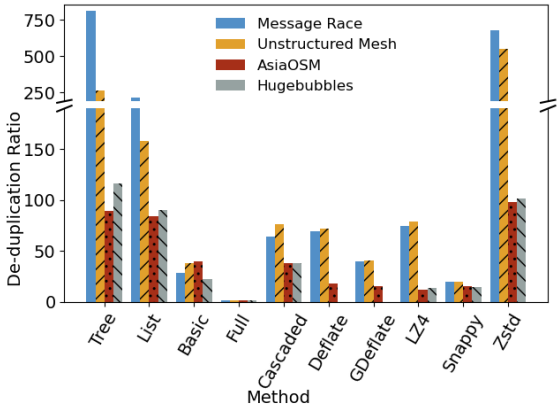
The frequency of checkpoints can affect our method's deduplication ratio and throughput. Figure 5.5 shows the impact of checkpoint frequency on our method compared with the state-of-the-art techniques and GPU compression methods. We capture 5, 10, or 20 checkpoints at moments evenly distributed during the runtime. The deduplication ratio results in Figures 5.5a- 5.5e demonstrate the benefits of using temporal information for deduplicating data. Increasing the checkpoint frequency reduces the number of updates at each checkpoint. Fewer updates and frequent checkpoints increase the temporal information our method can leverage. However, this does not always compare favorably with compression. For example, our method has worse deduplication ratios than Zstd for all four graphs. Increasing the number of checkpoints to 20 allows our Tree method to outperform Zstd for all input data, even with the more difficult Asia OSM and Hugebubbles graphs. This behavior is expected, as the compression techniques are limited to individual checkpoints. We are using our Tree method; taking 20 checkpoints results in a smaller total checkpoint size for all graphs except Asia OSM, which sees only a 2% increase in total size. This behavior makes our method excellent for applications seeking to conduct frequent checkpoints. Figures 5.5b- 5.5f show that throughput performance is less impacted by increasing checkpoint frequency. Throughput increases for our Tree method as well



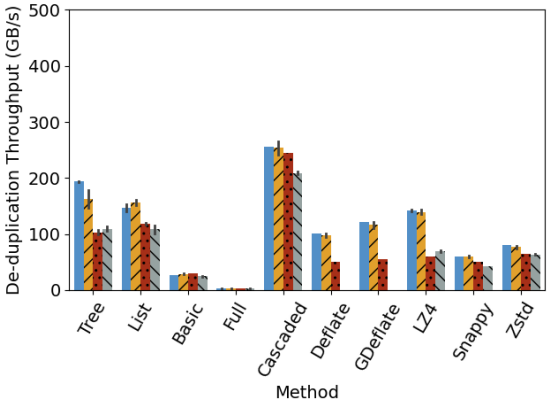
(a) Deduplication ratio for 5 checkpoints



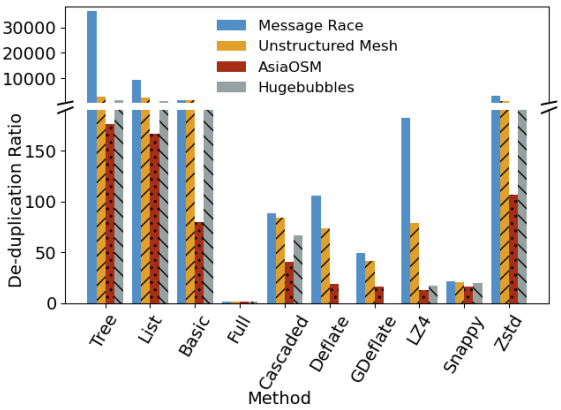
(b) Throughput for five checkpoints



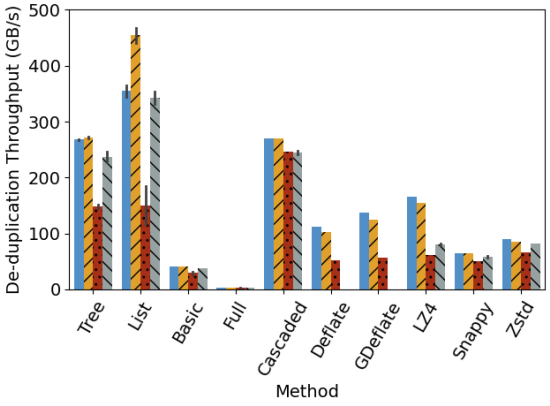
(c) Deduplication ratio for ten checkpoints



(d) Throughput for ten checkpoints



(e) Deduplication ratio for 20 checkpoints



(f) Throughput for 20 checkpoints

Figure 5.5: Impact of checkpoint frequency on deduplication ratio and throughput for our method (Tree) vs. state-of-art (i.e., Full, Basic, and List) and several *nvCOMP* compression algorithms. Results are shown for $N = 5, 10, 20$ checkpoints evenly distributed during the runtime (which varies for each input graph).

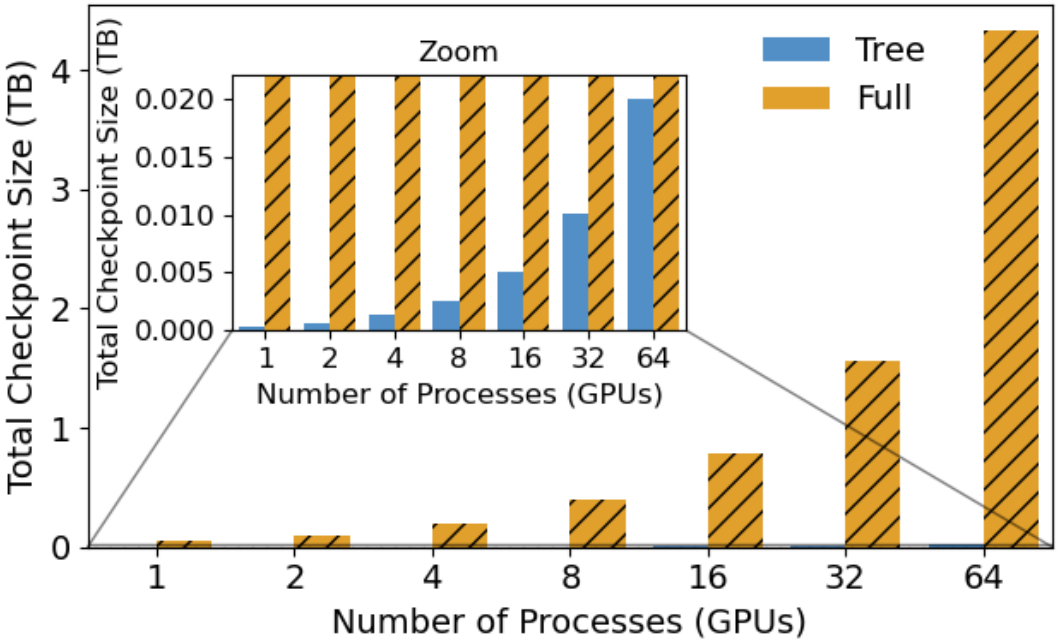
as for the List, and Basic methods, while the compression techniques are unaffected. The improvements to the Tree method range from 1.37× with the Unstructured Mesh to 2.77× with the Hugebubbles graph.

5.4.5 Strong Scaling

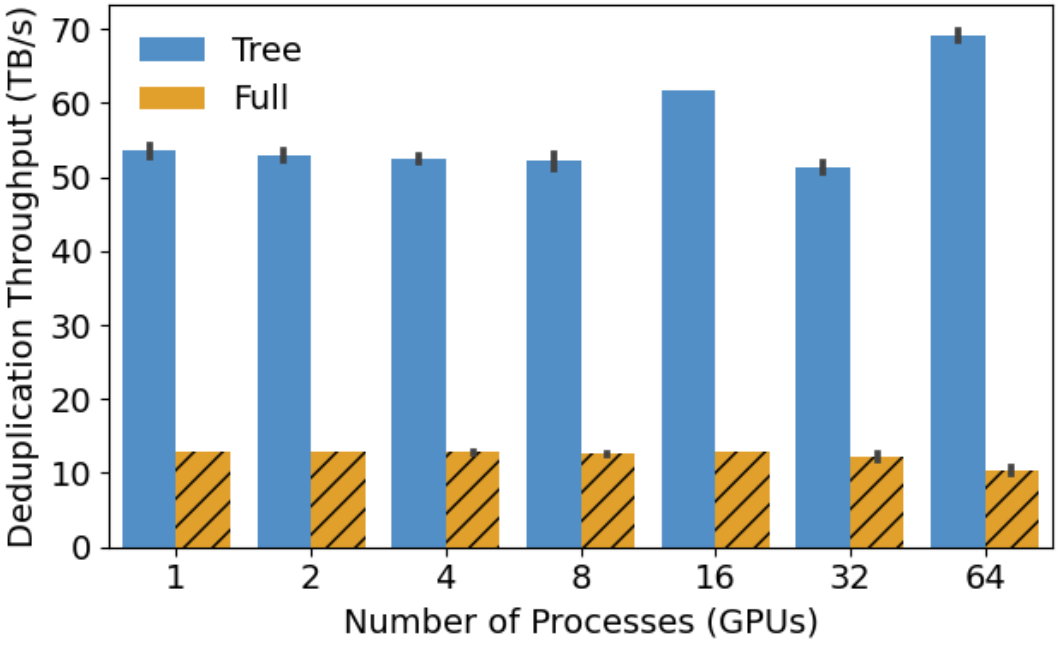
We show the strong scaling results comparing the Full method and our Merkle tree-based method in Figure 5.6. The Delaunay N24 graph is the input, and the number of GPUs varies from 1 to 64. Each process checkpoints independently, but multiple GPUs copying data to a shared CPU can impact performance. We measure the sum of the first ten checkpoints for all processes. Throughput is measured by taking the sum of 10 checkpoints and dividing it by the maximum runtime spent on deduplication across all processes. Figure 5.6a shows the sum of total checkpoint sizes for ORANGES running the Delaunay N24 graph. As the number of processes increases exponentially, so does the checkpointed data. At 64 processes, we see a 215× reduction in total checkpoint size compared with Full: 4.33 TB of checkpoints is reduced to 20 GB. The deduplication throughput is shown in Figure 5.6b. This indicates that our Tree method has greater throughput than Full, and the throughput maintains or improves as the number of processes increases. Since ORANGES is embarrassingly parallel (it finishes with a reduction of the independently obtained results on each GPU), we did not perform a full comparison at scale between all methods. We anticipate similar trends compared with the single GPU experiments.

5.5 Importance and Applicability of Pruning Redundancy

Pruning redundancy in checkpoint data using data deduplication techniques improves data storage efficiency and reduces I/O overhead. Incremental deduplication allows more frequent checkpoints and enables scientists to track and store more



(a) Total checkpoint size



(b) Deduplication throughput

Figure 5.6: Strong scaling results with up to 64 GPUs using the Delaunay N24 input graph. Our method (Tree) achieves over two orders of magnitude reduction in checkpoint size compared with Full and retains a high throughput at scale.

information about their simulations. Storing large quantities of simulation data opens up interesting avenues for training AI/ML models that can help scientists filter and identify quantities of interest. Pruning redundancy with incremental data deduplication contributes to scientific discovery in the following ways. First, we mitigate the ever-increasing storage challenges and I/O overhead as systems increase in scale and the gap between computational capabilities and data transfer speeds worsens. Second, we enable more frequent checkpoints for emerging checkpoint scenarios such as adjoint computation and reproducibility analysis that require frequent checkpoints. Lastly, we provide a compact metadata format for the search and retrieval of incremental checkpoint data, thus allowing the identification of repeating data patterns and the study of how simulation data evolves. Checkpointing is already an important aspect of HPC software development, but traditional methods are insufficient for emerging use cases and increasingly large-scale simulations. Our Merkle tree-based deduplication method is an effective tool for helping developers mitigate storage and I/O challenges. Our data deduplication method is well suited to applications with sparse updates like graph algorithms. Algorithms like PageRank and Single Source Shortest Paths (SSSP) have irregular communication and memory access patterns that result in sparsely updated data structures. Algorithms on dynamic graphs also exhibit significant amounts of redundant data which our method can eliminate and improve storage efficiency. The next step is to extend our work with lossy hash functions which enables developers to specify an error tolerance for identifying changes in data. This enables our method to deduplicate floating point data which may have small negligible changes in the data that normal hash functions would identify as changed. The metadata generated by our method can also accelerate data analysis. For example, an application may have non deterministic results where the output of different runs using the same settings differs from one another. Using our data deduplication method we can isolate and identify which parts of the intermediate checkpoints differ from prior runs. These insights can help verify whether non deterministic results is acceptable.

Chapter 6

Conclusions and Future Work

Heterogeneous hardware is essential for exascale systems, but this heterogeneity introduces challenges that must be addressed to harness the systems' potential fully. We highlight three key issues that arise from the transition to exascale machines with heterogeneous hardware. First, frameworks used for portability across platforms introduce their overheads and often conceal important hardware characteristics from the developer. Second, hardware designers focus on AI/ML features, such as lower precision number formats, which accelerate computation but introduce accuracy challenges when used for HPC software. Third, the growing scale of simulations and the emerging need for frequent checkpoints exacerbate the I/O overhead and storage cost challenges already present in petascale systems.

We examine each issue and develop solutions to mitigate these challenges and improve performance on heterogeneous systems. For the first challenge (Chapter 3), we port an existing HPC application (VPIC) using the Kokkos framework and study the particle sort to optimize code parts where portability introduces overhead. For the second challenge (Chapter 4), we demonstrate that lower precision number formats (i.e., half-precision and fixed-point), typically used for AI/ML training, can increase simulation scale while maintaining scientific accuracy for particle-in-cell codes. For the third challenge (Chapter 5), we develop a massively parallel data de-duplication

method using hash functions and Merkle trees to identify repeating patterns and eliminate spatial and temporal redundancy. Developing software on heterogeneous hardware requires tackling these challenges to maximize efficiency.

6.1 Providing Performance Portability for HPC Software

In Chapter 3, we address the challenge of mitigating portability overhead from performance portability frameworks. We develop a performance-portable version of VPIC, a kinetic Particle-in-Cell code used to do some of the world's largest plasma physics simulations. This version of VPIC (VPIC 2.0) relies on the Kokkos framework for portability. We identify memory layout and vectorization as key areas where VPIC 2.0 has performance issues due to Kokkos inducing overhead and concealing hardware characteristics. These portability overheads are mitigated by adjusting the sorting order for coalesced memory accesses and using additional SIMD abstractions.

In our contributions, we capture the VPIC 2.0 performance on nine different platforms (including three of the top ten systems on the Top500 list) and scale our tests to 17,100 GPUs. We demonstrate super-linear strong scaling on the NVIDIA A100 GPU and provide insights on the impacts of VPIC 2.0 for scientific discovery for the next generation of exascale supercomputers. VPIC 2.0 is portable across various NVIDIA, AMD, Intel, and ARM hardware. VPIC 2.0 achieves performance parity with the optimized VPIC 1.2 code on CPUs and attains a high percentage of peak compute utilization on GPUs for a real scientific application. The CPU and accelerator execution paths share over 99% of the same code. We demonstrate that the sort order greatly impacts memory access patterns and can result in over an order of magnitude improvement in runtime on GPUs. Our sorting order optimizations apply to any code using the gather scatter memory access pattern. We show that guided vectorization optimizations are competitive with explicit manual

SIMD vectorization. Combining performance portability frameworks with general optimizations for mitigating portability overhead is a practical solution for developing HPC applications on heterogeneous hardware.

6.2 Preserving Accuracy with Lower Precision Formats

In Chapter 4, we address the difficulties in leveraging lower precision formats for traditional HPC simulations such as particle-in-cell codes. Alternate number formats such as half-precision and Bfloat16 are frequently used in AI/ML applications to increase computing capabilities and reduce data movement, but they come at the cost of reduced accuracy.

We demonstrate that traditional HPC applications, such as the VPIC particle-in-cell code, can leverage lower precision number formats primarily used for AI/ML while maintaining scientific accuracy. We show the combination of constant weight and lower-precision position in the VPIC code reduces memory usage by up to 31.25% and enables up to 40% increase in the number of particles using the same amount of memory for particle-heavy simulations. The optimizations improve runtime performance by reducing data movement, compensating for any additional operations introduced. Unlike other work in mixed-precision algorithms, our optimizations use reduced precision for storage rather than accelerating computation. We present memory and runtime performance analysis across platforms such as the V100 GPU, Power9 CPU, and the A64FX CPU. The optimizations drastically improve performance when interacting with slower memory or storage devices such as swap memory. We also show that our optimizations not only greatly increase simulation scale on memory-constrained hardware but can also achieve similar accuracy as the original single-precision VPIC 1.2. The fixed-point optimizations, in particular, show great promise thanks to the higher precision than half-precision. The higher precision

can allow for lower grid resolutions, ultimately decreasing the number of time steps in the simulation. These optimizations have potential applications in similar grid-based simulations, such as molecular dynamics and fluid dynamics.

6.3 Pruning Redundant Data with Deduplication

In Chapter 5, we address large-scale simulations' growing storage costs and I/O overhead. The growing gap between memory bandwidth and computation capabilities gap, combined with the high I/O storage costs and limited I/O resources, creates significant data management challenges as we move to exascale machines.

We present a scalable GPU-aware Merkle tree-based incremental checkpointing method that leverages de-duplication to reduce the cost of storing the checkpoint history and increase the checkpointing throughput simultaneously. To this end, we identify contiguous repeating patterns across the entire checkpoint record and eliminate redundancy at the data and metadata levels. We use these fundamental design principles to create a method that removes spatiotemporal redundancy by leveraging repeating patterns. Unlike high-throughput compression techniques, our method improves the de-duplication ratio and throughput as the frequency of checkpoints increases, reducing the size of the checkpoint record by up to 67×. Furthermore, our method shows excellent scalability for large graphs, reducing the checkpoint sizes by two orders of magnitude and increasing the checkpointing throughput by almost an order of magnitude compared with full checkpoints. Such benefits are also useful for non-resilience scenarios where incremental checkpoints are used to analyze intermediate results (e.g., reproducibility efforts) or make progress (e.g., adjoint computations). Redundancy pruning techniques are useful tools that developers should leverage for data management. This is particularly true for accelerator-based systems where memory capacity and data movement costs impact application performance more than compute capabilities.

6.4 Future Work

For future work on mitigating portability overhead, we will analyze the performance of our optimizations to understand the trade-offs between vectorization strategies and explore the impacts of different memory access patterns on various platforms. The performance and ease of use for the guided vectorization strategy depend on several factors such as the platform, compiler, and code structure. We will analyze different configurations to get a better insight into the best practices for maximizing performance portability. We will also experiment with different sorting orders and memory layouts to optimize the memory access pattern on various platforms.

In the future, we plan to study the algorithmic changes necessary to enable lower precision storage for particle momentum and the impact of these changes on both scalability and accuracy. We also plan to develop heuristics and methodologies to help physicists use our suite of optimizations. Adjustments to simulation parameters (e.g., grid resolution, step size and number of steps) are necessary to maintain accuracy. Different classes of plasma simulations need to be studied to better understand which classes benefit from our optimizations the most and what changes to simulation settings are required to use our optimizations.

Encouraged by our data deduplication results, in future work, we will apply our incremental checkpointing method to enhance reproducibility. Reproducibility studies and analysis benefit from comparing intermediate checkpoints across different runs. The intermediate states provide more detailed information on how the application data evolves, allowing identification of both when and where simulations diverge between runs. Our Merkle tree methods can help accelerate checkpoint comparisons, improve reproducibility, and ensure trustworthy and reliable science.

Bibliography

Bibliography

[1] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, *et al.*, "Kokkos 3: Programming Model Extensions for the Exascale Era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2021. 1, 9, 16, 92

[2] R. D. Hornung and J. A. Keasler, "The RAJA Portability Layer: Overview and Status," tech. rep., Lawrence Livermore National Lab.(LLNL), 2014. 1, 9

[3] L. Dagum and R. Menon, "OpenMP: An Industry Standard API for Shared-Memory Programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998. 1, 8

[4] R. Bird, N. Tan, S. V. Luedtke, S. Harrell, M. Taufer, and B. Albright, "VPIC 2.0: Next Generation Particle-in-Cell Simulations," *IEEE Transactions on Parallel & Distributed Systems*, vol. 33, pp. 952–963, apr 2022. 3, 58

[5] N. Tan, R. F. Bird, G. Chen, and M. Taufer, "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)*, (Krakow, Poland), pp. 452–465, Springer *(Best Track Paper Award)*, June 16-18 2021. 3

[6] N. Tan, R. F. Bird, G. Chen, S. V. Luedtke, B. J. Albright, and M. Taufer, "Analysis of Vector Particle-In-Cell (VPIC) Memory Usage Optimizations on

Cutting-Edge Computer Architectures," *Journal of Computational Science*, vol. 60, p. 101566, 2022. 3

[7] N. Tan, J. Luettgau, J. Marquez, K. Teranishi, N. Morales, S. Bhowmick, F. Cappello, M. Taufer, and B. Nicolae, "Scalable Incremental Checkpointing using GPU-Accelerated De-Duplication," in *Proceedings of the 52nd International Conference on Parallel Processing*, pp. 665–674, 2023. 3

[8] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "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*, pp. 859–870, Springer, 2012. 8

[9] A. Munshi, "The OpenCL Specification," in *2009 IEEE Hot Chips 21 Symposium (HCS)*, pp. 1–314, IEEE, 2009. 8

[10] "HIP: C++ Heterogeneous-Compute Interface for Portability." https://github.com/ROCm-Developer-Tools/HIP. Accessed: 2023-09-23. 8

[11] R. Keryell, R. Reyes, and L. Howes, "Khronos SYCL for OpenCL: A Tutorial," in *Proceedings of the 3rd International Workshop on OpenCL*, pp. 1–1, 2015. 8

[12] J. Reinders, B. Ashbaugh, J. Brodman, M. Kinsner, J. Pennycook, and X. Tian, *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems Using C++ and SYCL*. Springer Nature, 2021. 8

[13] D. S. Medina, A. St-Cyr, and T. Warburton, "OCCA: A Unified Approach to Multi-Threading Languages," *arXiv preprint arXiv:1403.0968*, 2014. 9

[14] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel Programmability and the Chapel Language," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007. 9

[15] D. Sahasrabudhe, E. T. Phipps, S. Rajamanickam, and M. Berzins, "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*, pp. 140–163, Springer, 2020. 9

[16] B. Bramas *et al.*, "Inastemp: A Novel Intrinsics-As-Template Library for Portable SIMD-Vectorization," *Scientific Programming*, vol. 2017, 2017. 9

[17] R. Strzodka, "Abstraction for AoS and SoA Layout in C++," in *GPU Computing Gems Jade Edition*, pp. 429–441, Elsevier, 2012. 9, 18

[18] H. Homann and F. Laenen, "SoAx: A Generic C++ Structure of Arrays for Handling Particles in HPC Codes," *Computer Physics Communications*, vol. 224, pp. 325–332, 2018. 9

[19] S. Slattery, S. T. Reeve, C. Junghans, D. Lebrun-Grandié, R. Bird, G. Chen, S. Fogerty, Y. Qiu, S. Schulz, A. Scheinberg, *et al.*, "Cabana: A Performance Portable Library for Particle-Based Simulations," *Journal of Open Source Software*, vol. 7, no. 72, p. 4115, 2022. 9

[20] B. M. Gruber, G. Amadio, J. Blomer, A. Matthes, R. Widera, and M. Bussmann, "LLAMA: The Low-Level Abstraction for Memory Access," *Software: Practice and Experience*, vol. 53, no. 1, pp. 115–141, 2023. 9

[21] M. Martineau, S. McIntosh-Smith, M. Boulton, W. Gaudin, and D. Beckingsale, "A Performance Evaluation of Kokkos & RAJA Using the TeaLeaf Mini-app," in *The International Conference for High Performance Computing, Networking, Storage and Analysis, SC15*, 2015. 10

[22] R. O. Kirk, G. R. Mudalige, I. Z. Reguly, S. A. Wright, M. J. Martineau, and S. A. Jarvis, "Achieving Performance Portability for a Heat Conduction Solver

Mini-application on Modern Multi-core Systems," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 834–841, IEEE, 2017. 10

[23] S. J. Pennycook, J. D. Sewall, and J. R. Hammond, "Evaluating the Impact of Proposed OpenMP 5.0 Features on Performance, Portability and Productivity," in *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pp. 37–46, IEEE, 2018. 10

[24] S. L. Harrell, J. Kitson, R. Bird, S. J. Pennycook, J. Sewall, D. Jacobsen, D. N. Asanza, A. Hsu, H. C. Carrillo, H. Kim, *et al.*, "Effective Performance Portability," in *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pp. 24–36, IEEE, 2018. 10

[25] T. Deakin, S. McIntosh-Smith, J. Price, A. Poenaru, P. Atkinson, C. Popa, and J. Salmon, "Performance Portability Across Diverse Computer Architectures," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pp. 1–13, IEEE, 2019. 10

[26] J. Sewall, "Intel Code Base Investigator," 2019. 10

[27] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov, "Accelerating Scientific Computations with Mixed Precision Algorithms," *Computer Physics Communications*, vol. 180, no. 12, pp. 2526–2533, 2009. 11

[28] I. Kampolis, X. Trompoukis, V. Asouti, and K. Giannakoglou, "CFD-Based Analysis and Two-Level Aerodynamic Optimization on Graphics Processing Units," *Computer Methods in Applied Mechanics and Engineering*, vol. 199, no. 9-12, pp. 712–722, 2010. 11

[29] R. Olivares-Amaya, M. A. Watson, R. G. Edgar, L. Vogt, Y. Shao, and A. Aspuru-Guzik, "Accelerating Correlated Quantum Chemistry Calculations using Graphical Processing Units and a Mixed Precision Matrix Multiplication

Library," *Journal of Chemical Theory and Computation*, vol. 6, no. 1, pp. 135–144, 2010. 11

[30] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, *et al.*, "Mixed Precision Training," *arXiv preprint arXiv:1710.03740*, 2017. 11

[31] N. Burgess, J. Milanovic, N. Stephens, K. Monachopoulos, and D. Mansell, "Bfloat16 Processing for Neural Networks," in *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pp. 88–91, IEEE, 2019. 11

[32] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky, "NVIDIA A100 Tensor Core GPU: Performance and Innovation," *IEEE Micro*, vol. 41, no. 2, pp. 29–35, 2021. 11

[33] R. Yates, "Fixed-Point Arithmetic: An Introduction," *Digital Signal Labs*, vol. 81, no. 83, p. 198, 2009. 11

[34] M. Fasi, N. J. Higham, M. Mikaitis, and S. Pranesh, "Numerical Behavior of NVIDIA Tensor Cores," *PeerJ Computer Science*, vol. 7, p. e330, 2021. 11

[35] A. Haidar, P. Wu, S. Tomov, and J. Dongarra, "Investigating Half Precision Arithmetic to Accelerate Dense Linear System Solvers," in *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pp. 1–8, 2017. 12

[36] A. Haidar, H. Bayraktar, S. Tomov, J. Dongarra, and N. J. Higham, "Mixed-Precision Iterative Refinement using Tensor Cores on GPUs to Accelerate Solution of Linear Systems," *Proceedings of the Royal Society A*, vol. 476, no. 2243, p. 20200110, 2020. 12

[37] A. Sorna, X. Cheng, E. D'azevedo, K. Won, and S. Tomov, "Optimizing the Fast Fourier Transform using Mixed Precision on Tensor Core Hardware," in

2018 IEEE 25th International Conference on High Performance Computing Workshops (HiPCW), pp. 3–7, IEEE, 2018. 12

[38] K. J. Bowers, B. Albright, L. Yin, B. Bergen, and T. Kwan, "Ultrahigh Performance Three-dimensional Electromagnetic Relativistic Kinetic Plasma Simulation," *Physics of Plasmas*, vol. 15, no. 5, p. 055703, 2008. 12

[39] S. Le Grand, A. W. Götz, and R. C. Walker, "SPFP: Speed Without Compromise—A Mixed Precision Model for GPU Accelerated Molecular Dynamics Simulations," *Computer Physics Communications*, vol. 184, no. 2, pp. 374–380, 2013. 12

[40] O. Tintó Prims, M. C. Acosta, A. M. Moore, M. Castrillo, K. Serradell, A. Cortés, and F. J. Doblas-Reyes, "How to Use Mixed Precision in Ocean Models: Exploring a Potential Reduction of Numerical Precision in NEMO 4.0 and ROMS 3.6," *Geoscientific Model Development*, vol. 12, no. 7, pp. 3135–3148, 2019. 12

[41] S. C. G. A. I. Lasrib, E. Rohoub, and O. Sentieysb, "Implications of Reduced-Precision Computations in HPC: Performance, Energy and Error," *Parallel Computing is Everywhere*, vol. 32, no. 2018, p. 297, 2018. 12

[42] Hota, Alok and Raji, Mohammad and Hobson, Tanner and Huang, Jian, "A space-efficient method for navigable ensemble analysis and visualization," 10 2017. 12

[43] M. Vasavada, F. Mueller, P. H. Hargrove, and E. Roman, "Comparing Different Approaches for Incremental Checkpointing: The Showdown," in *Linux Symposium*, vol. 69, (Ottowa, Canada), pp. 69–80, 2011. 12

[44] R. Gioiosa, J. C. Sancho, S. Jiang, and F. Petrini, "Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel

Computers," in *SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, (Seattle, WA, USA), pp. 9–9, IEEE, 2005. 12

[45] D. Vogt, A. Miraglia, G. Portokalidis, H. Bos, A. Tanenbaum, and C. Giuffrida, "Speculative Memory Checkpointing," in *Proceedings of the 16th Annual Middleware Conference*, (New York, NY, USA), pp. 197–209, Association for Computing Machinery, 2015. 12

[46] B. Nicolae, "Towards Scalable Checkpoint Restart: A Collective Inline Memory Contents Deduplication Proposal," in *IPDPS '13: The 27th IEEE International Parallel and Distributed Processing Symposium*, (Boston, USA), pp. 19–28, Association for Computing Machinery, 2013. 12

[47] J. Ma, X. Zheng, Y. Dong, W. Li, Z. Qi, B. He, and H. Guan, "gMig: Efficient GPU Live Migration Optimized by Software Dirty Page for Full Virtualization," in *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, (New York, NY, USA), pp. 31–44, Association for Computing Machinery, 2018. 13

[48] K. Lee, M. B. Sullivan, S. K. S. Hari, T. Tsai, S. W. Keckler, and M. Erez, "GPU Snapshot: Checkpoint Offloading for GPU-Dense Systems," in *Proceedings of the ACM International Conference on Supercomputing*, (Phoenix, AZ, USA), pp. 171–183, Association for Computing Machinery, 2019. 13

[49] B. Nicolae, G. Antoniu, L. Bouge, D. Moise, and A. Carpen-Amarie, "BlobSeer: Next-generation Data Management for Large Scale Infrastructures," *J. Parallel Distrib. Comput.*, vol. 71, no. 2, pp. 169–184, 2011. 13

[50] B. Nicolae, "Scalable Multi-Versioning Ordered Key-Value Stores with Persistent Memory Support," in *IPDPS 2022: The 36th IEEE International Parallel and Distributed Processing Symposium*, (Lyon, France), pp. 93–103, IEEE, 2022. 13

[51] S. Kannan, N. Farooqui, A. Gavrilovska, and K. Schwan, "Heterocheckpoint: Efficient Checkpointing for Accelerator-based Systems," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, (Atlanta, USA), pp. 738–743, IEEE, 2014. 13

[52] R. Garg, A. Mohan, M. Sullivan, and G. Cooperman, "CRUM: Checkpoint-Restart Support for CUDA's Unified Memory," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, (Belfast, United Kingdom), pp. 302–313, IEEE, 2018. 13

[53] B. Pourghassemi and A. Chandramowlishwaran, "cudaCR: An In-kernel Application-level Checkpoint/Restart Scheme for CUDA-enabled GPUs," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, (Honolulu, USA), pp. 725–732, IEEE, 2017. 13

[54] K. Parasyris, K. Keller, L. Bautista-Gomez, and O. Unsal, "Checkpoint Restart Support for Heterogeneous HPC Applications," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, (Melbourne, Australia), pp. 242–251, IEEE, 2020. 13

[55] K. B. Ferreira, R. Riesen, R. Brighwell, P. Bridges, and D. Arnold, "libhashckpt: Hash-based Incremental Checkpointing Using GPU's," in *European MPI Users' Group Meeting*, (Santorini, Greece), pp. 272–281, Springer, 2011. 13

[56] F. Knorr, P. Thoman, and T. Fahringer, "ndzip: A High-Throughput Parallel Lossless Compressor for Scientific Data," in *2021 Data Compression Conference (DCC)*, (Snowbird, USA), pp. 103–112, IEEE, 2021. 13

[57] D. Ibtesham, K. B. Ferreira, and D. Arnold, "A Checkpoint Compression Study for High-performance Computing Systems," *The International Journal of High Performance Computing Applications*, vol. 29, no. 4, pp. 387–402, 2015. 13

[58] N. Sasaki, K. Sato, T. Endo, and S. Matsuoka, "Exploration of Lossy Compression for Application-level Checkpoint/Restart," in *2015 IEEE International Parallel and Distributed Processing Symposium*, (Hyderabad, India), pp. 914–922, IEEE, 2015. 13

[59] J. Zhang, X. Zhuo, A. Moon, H. Liu, and S. W. Son, "Efficient Encoding and Reconstruction of HPC Datasets for Checkpoint/Restart," in *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 79–91, IEEE, 2019. 13

[60] P. Lindstrom, "Fixed-rate Compressed Floating-point Arrays," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2674–2683, 2014. 13

[61] S. Di and F. Cappello, "Fast Error-bounded Lossy HPC Data Compression with SZ," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 730–739, IEEE, 2016. 13

[62] T. Lu, Q. Liu, X. He, H. Luo, E. Suchyta, J. Choi, N. Podhorszki, S. Klasky, M. Wolf, T. Liu, *et al.*, "Understanding and Modeling Lossy Compression Schemes on HPC Scientific Data," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 348–357, IEEE, 2018. 13

[63] J. Calhoun, F. Cappello, L. N. Olson, M. Snir, and W. D. Gropp, "Exploring the Feasibility of Lossy Compression for PDE Simulations," *The International Journal of High Performance Computing Applications*, vol. 33, no. 2, pp. 397–410, 2019. 13

[64] F. Cappello, S. Di, S. Li, X. Liang, A. M. Gok, D. Tao, C. H. Yoon, X.-C. Wu, Y. Alexeev, and F. T. Chong, "Use Cases of Lossy Compression for Floating-point Data in Scientific Data Sets," *The International Journal of High Performance Computing Applications*, vol. 33, no. 6, pp. 1201–1220, 2019. 13

[65] K. J. Bowers, B. J. Albright, L. Yin, W. Daughton, V. Roytershteyn, B. Bergen, and T. Kwan, "Advances in Petascale Kinetic Plasma Simulation with VPIC and Roadrunner," in *Journal of Physics: Conference Series*, vol. 180, p. 012055, IOP Publishing, 2009. 15

[66] M. Yamada, L.-J. Chen, J. Yoo, S. Wang, W. Fox, J. Jara-Almonte, H. Ji, W. Daughton, A. Le, J. Burch, *et al.*, "The Two-fluid Dynamics and Energetics of the Asymmetric Magnetic Reconnection in Laboratory and Space Plasmas," *Nature communications*, vol. 9, no. 1, p. 5223, 2018. 15

[67] L. Yin, W. Daughton, B. Albright, K. Bowers, D. Montgomery, J. Kline, J. Fernández, and Q. Roper, "Nonlinear Backward Stimulated Raman Scattering from Electron Beam Acoustic Modes in the Kinetic Regime," *Physics of Plasmas*, vol. 13, no. 7, 2006. 15

[68] E. Lichko, J. Egedal, W. Daughton, and J. Kasper, "Magnetic Pumping as a Source of Particle Heating and Power-law Distributions in the Solar Wind," *The Astrophysical Journal Letters*, vol. 850, no. 2, p. L28, 2017. 15

[69] J. C. Fernández, D. Cort Gautier, C. Huang, S. Palaniyappan, B. J. Albright, W. Bang, G. Dyer, A. Favalli, J. F. Hunter, J. Mendez, *et al.*, "Laser-plasmas in the Relativistic-transparency Regime: Science and Applications," *Physics of Plasmas*, vol. 24, no. 5, 2017. 15

[70] W. Nystrom, B. Bergen, R. Bird, K. Bowers, W. Daughton, F. Guo, A. Le, H. Li, H. Nam, X. Pang, *et al.*, "Recent Performance Results of VPIC on Trinity," in *APS Division of Plasma Physics Meeting Abstracts*, vol. 2017, pp. GP11–152, 2017. 16

[71] C. Trott, L. Berger-Vergiat, D. Poliakoff, S. Rajamanickam, D. Lebrun-Grandie, J. Madsen, N. Al Awar, M. Gligoric, G. Shipman, and G. Womeldorff, "The Kokkos EcoSystem: Comprehensive Performance Portability for High

Performance Computing," *Computing in Science Engineering*, vol. 23, no. 5, pp. 10–18, 2021. 16

[72] D. Stanzione, J. West, R. T. Evans, T. Minyard, O. Ghattas, and D. K. Panda, "Frontera: The Evolution of Leadership Computing at the National Science Foundation," in *Practice and Experience in Advanced Research Computing*, PEARC '20, (New York, NY, USA), p. 106–111, Association for Computing Machinery, 2020. 20

[73] C. Tierney, P. Savla, S. Ellis, , and R. Sohigian, "NVIDIA DGX SuperPOD: Scalable Infrastructure for AI Leadership," Tech. Rep. RA-09950-001, NVIDIA, 11 2020. 20

[74] G. McCartney, T. Hacker, and B. Yang, "Empowering Faculty: A Campus Cyberinfrastructure Strategy for Research Communities," *Educause Review*, 2014. 21

[75] M. Sato, Y. Ishikawa, H. Tomita, Y. Kodama, T. Odajima, M. Tsuji, H. Yashiro, M. Aoki, N. Shida, I. Miyoshi, K. Hirai, A. Furuya, A. Asato, K. Morita, and T. Shimizu, "Co-Design for A64FX Manycore Processor and "Fugaku"," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20, IEEE Press, 2020. 21, 58

[76] B. Albright, L. Yin, K. Bowers, and B. Bergen, "Multi-dimensional Dynamics of Stimulated Brillouin Scattering in a Laser Speckle: Ion Acoustic Wave Bowing, Breakup, and Laser-seeded Two-ion-wave Decay," *Physics of Plasmas*, vol. 23, no. 3, p. 032703, 2016. 22

[77] L. Yin, B. J. Albright, D. J. Stark, W. D. Nystrom, R. F. Bird, and K. J. Bowers, "Saturation of Cross-beam Energy Transfer for Multispeckled Laser Beams Involving both Ion and Electron Dynamics," *Physics of Plasmas*, vol. 26, no. 8, p. 082708, 2019. 30

[78] F. Guo, X. Li, W. Daughton, P. Kilian, H. Li, Y.-H. Liu, W. Yan, and D. Ma, "Determining the Dominant Acceleration Mechanism During Relativistic Magnetic Reconnection in Large-scale Systems," *The Astrophysical Journal Letters*, vol. 879, no. 2, p. L23, 2019. 42

[79] I. Karlin, "LULESH Programming Model and Performance Ports Overview," tech. rep., Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2012. 43

[80] Lawrence Livermore National Laboratory (LLNL), "AMG Benchmark," 2017. 43

[81] P. Fischer, J. Lottes, and H. Tufo, "Nek5000," tech. rep., Argonne National Laboratory (ANL), Argonne, IL (United States), 2007. 43

[82] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. In't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, *et al.*, "LAMMPS-a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales," *Computer Physics Communications*, vol. 271, p. 108171, 2022. 43

[83] S. Byna, J. Chou, O. Rubel, H. Karimabadi, W. S. Daughter, V. Roytershteyn, E. W. Bethel, M. Howison, K.-J. Hsu, K.-W. Lin, *et al.*, "Parallel I/O, Analysis, and Visualization of a Trillion Particle Simulation," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, IEEE, 2012. 48

[84] C. K. Birdsall and A. B. Langdon, *Plasma Physics via Computer Simulation*. CRC Press, 2004. 48

[85] A. Taflove and S. C. Hagness, *Computational Electrodynamics*. Artech house publishers, 2000. 48

[86] K. S. Yee *et al.*, "Numerical Solution of Initial Boundary Value Problems Involving Maxwell's Equations in Isotropic Media," *IEEE Trans. Antennas Propag*, vol. 14, no. 3, pp. 302–307, 1966. 48

[87] D. Goldberg, "What Every Computer Scientist Should Know about Floating-Point Arithmetic," *ACM Computing Surveys*, vol. 23, p. 5–48, mar 1991. 48

[88] K. J. Bowers, B. J. Albright, B. Bergen, L. Yin, K. J. Barker, and D. J. Kerbyson, "0.374 Pflop/s Trillion-particle Kinetic Modeling of Laser Plasma Interaction on Roadrunner," in *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pp. 1–11, IEEE, 2008. 50

[89] T. Arber, K. Bennett, C. Brady, A. Lawrence-Douglas, M. Ramsay, N. Sircombe, P. Gillies, R. Evans, H. Schmitz, A. Bell, *et al.*, "Contemporary Particle-in-cell Approach to Laser-plasma Modelling," *Plasma Physics and Controlled Fusion*, vol. 57, no. 11, p. 113001, 2015. 50

[90] NVIDIA Corporation, "NVIDIA A100 Tensor Core GPU Architecture ," tech. rep., NVIDIA Corporation, 2020. 50

[91] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen, *et al.*, "A Study of BFLOAT16 for Deep Learning Training," *arXiv preprint arXiv:1905.12322*, 2019. 50

[92] O. Catrina and A. Saxena, "Secure Computation with Fixed-Point Numbers," in *International Conference on Financial Cryptography and Data Security*, pp. 35–50, Springer, 2010. 54

[93] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling Manycore Performance Portability through Polymorphic Memory Access Patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 –

3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing. 60, 62

[94] L. E. Thode and R. N. Sudan, "Two-Stream Instability Heating of Plasmas by Relativistic Electron Beams," *Phys. Rev. Lett.*, vol. 30, pp. 732–735, Apr 1973. 76

[95] T. H. Stix, *Waves in Plasmas*. Springer Science & Business Media, 1992. 76

[96] G. Chen, L. Chacón, L. Yin, B. J. Albright, D. J. Stark, and R. F. Bird, "A Semi-implicit, Energy-and Charge-conserving Particle-in-cell Algorithm for the Relativistic Vlasov-Maxwell Equations," *Journal of Computational Physics*, vol. 407, p. 109228, 2020. 76

[97] B. D. Fried, "Mechanism for Instability of Transverse Plasma Waves," *Physics of Fluids*, vol. 2, no. 3, pp. 337–337, 1959. 76

[98] R. Morse and C. Nielson, "Numerical Simulation of the Weibel Instability in One and Two Dimensions," *The Physics of Fluids*, vol. 14, no. 4, pp. 830–840, 1971. 76

[99] J.-L. Vay, A. Almgren, J. Bell, L. Ge, D. Grote, M. Hogan, O. Kononenko, R. Lehe, A. Myers, C. Ng, *et al.*, "Warp-X: A new exascale computing platform for beam–plasma simulations," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 909, pp. 476–479, 2018. 78

[100] M. J. Abraham, T. Murtola, R. Schulz, S. Páll, J. C. Smith, B. Hess, and E. Lindahl, "GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers," *SoftwareX*, vol. 1, pp. 19–25, 2015. 78

[101] D. Meister, J. Kaiser, A. Brinkmann, T. Cortes, M. Kuhn, and J. Kunkel, "A Study on Data Deduplication in HPC Storage Systems," in *SC'12: Proceedings*

of the International Conference on High Performance Computing, Networking, Storage and Analysis, (Salt Lake City, UT, USA), pp. 1–11, IEEE, 2012. 82

[102] R. C. Merkle, "A Digital Signature Based on a Conventional Encryption Function," in *Advances in Cryptology—CRYPTO'87: Proceedings 7*, (Santa Barbara, CA, USA), pp. 369–378, Springer, 1988. 84

[103] A. Appleby, "SMHasher & MurmurHash." Retrieved from https://code.google.com/p/smhasher, 2012. 93

[104] R. L. Rivest, "The MD5 Message-Digest Algorithm." RFC 1321, Apr. 1992. 93

[105] N. Shervashidze, S. Vishwanathan, T. Petri, K. Mehlhorn, and K. Borgwardt, "Efficient Graphlet Kernels for Large Graph Comparison," in *12th International Conference on Artificial Intelligence and Statistics (AISTATS), Society for Artificial Intelligence and Statistics, 488-495 (2009)*, (Clearwater beach, USA), pp. 488–495, Proceedings of Machine Learning Research, 2009. 95

[106] O. Kuchaiev, T. Milenković, V. Memišević, W. Hayes, and N. Pržulj, "Topological Network Alignment Uncovers Biological Function and Phylogeny," *Journal of the Royal Society Interface*, vol. 7, no. 50, pp. 1341–1354, 2010. 95

[107] NVIDIA, "nvcomp: A library for fast lossless compression/decompression on the gpu," 2023. 96

[108] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011. 97

[109] H. Wei, J. X. Yu, C. Lu, and X. Lin, "Speedup Graph Processing by Graph Ordering," in *Proceedings of the 2016 International Conference on Management of Data*, (New York, NY, USA), pp. 1813–1828, Association for Computing Machinery, 2016. 99

Appendix

Appendix A

Research Artifacts

A.1 Conference and Journal Publications

1. **N. Tan**, K. Assogba, J. Ashworth, B. Bogale, M. M. Rafique, F. Cappello, B. Nicolae, and M. Taufer. "Towards Affordable Reproducibility Using Scalable Capture and Comparison of Intermediate Multi-Run Results". In Review

2. **N. Tan**, B. Nicolae, J. Luettgau, J. Marquez, K. Teranishi, N. Morales, S. Bhowmick, M. Taufer, and F. Cappello. "Scalable Checkpointing of Applications with Sparsely Updated Data". In *Proceedings of the 52nd International Conference on Parallel Processing (ICPP)*, 2023.

3. **N. Tan**, R. Bird, G. Chen, S. V. Luedtke, B. Albright, and M. Taufer. "Analysis of Vector Particle-In-Cell (VPIC) Memory Usage Optimizations on Cutting-Edge Computer Architectures". In *Journal of Computational Science Volume 60, April 2022, 101566*, 2022.

4. R. Patel, B. Roachell, S. Caino-Lores, R. Ketron, J. Leonard, **N. Tan**, K. Vahi, D. Brown, E. Deelman, and M. Taufer. "Reproducibility of the First Image of a Black Hole in the Galaxy M87 from the Event Horizon Telescope (EHT) Collaboration". *Journal of IEEE Computing in Science and Engineering*, 2022.

5. P. Bell, K. Suarez, D. Chapp, **N. Tan**, S. Bhowmick, and M. Taufer. "ANACIN-X: A software Framework for Studying Non-determinism in MPI applications". Software Impacts, 10:100151, 2021.

6. **N. Tan**, R. Bird, G. Chen, and M. Taufer. "Optimize Memory Usage in Vector Particle-In-Cell (VPIC) to Break the 10 Trillion Particle Barrier in Plasma Simulations". In *Proceedings of the International Conference on Computational Science (ICCS)*. 2021. Best Paper Award.

7. R. Bird, **N. Tan**, S. Luedtke, S. Harrell, M. Taufer, and B. Albright. "VPIC 2.0: Next Generation Particle-in-Cell Simulations". *Journal of IEEE Transactions on Parallel and Distributed Systems (IEEE TPDS)*, 2734-2748, 2021.

8. D. Chapp, **N. Tan**, S. Bhowmick, and M. Taufer. "Identifying Degrees and Sources of Non-Determinism in MPI Applications via Graph Kernels". *Journal of IEEE Transactions on Parallel and Distributed Systems (IEEE TPDS)*, 2021.

A.2 Software Artifacts

Table A.1: Software Artifacts

Software	
Artifact	**Description**
Vector Particle-In-Cell (VPIC) 2.0	New version of the Vector Particle-In-Cell code using Kokkos for portability. https://www.github.com/lanl/vpic-kokkos
GPU-Based De-duplication Library	Scalable GPU accelerated de-duplication code. Removes spatial and temporal redundancy in data while forming a compact representation of repeating regions.
ICPP23 Reproducibility Chameleon Appliance	A virtual image for the Chamelon cloud testbed that includes all the de-duplication software for the ICPP publication. https://www.chameleoncloud.org/appliances/113/
ICPP23 Reproducibility Artifacts	1. Input graphs: https://zenodo.org/record/8102190 2. De-duplication code: https://zenodo.org/record/8102017 3. ORANGES: https://zenodo.org/record/8102034 4. Results: https://zenodo.org/record/8102024

A.3 Conference Posters

1. **N. Tan**, M. Taufer, "Modernizing Simulation Software for the Exascale Era". Poster at the *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC) Doctoral Showcase*, Denver, CO, 2023.

2. **N. Tan**, S. V. Luedtke, M. Taufer, B. Albright, "Efficient Particle Tracing for Scalable Kinetic Plasma Simulation Analysis", Poster at the *IEEE International Conference on Cluster Computing (CLUSTER)*, Santa Fe, NM, 2023.

3. **N. Tan**, J. Luettgau, J. Marquez, K. Teranishi, N. Morales, S. Bhowmick, F. Cappello, M. Taufer, and B. Nicolae, "Scalable Checkpointing of Applications with Sparsely Updated Data". Proceedings of the *52nd International Conference on Parallel Processing (ICPP)*, 2023.

4. **N. Tan**, B. Nicolae, N. Morales, K. Teranishi, S. Bhowmick, and M. Taufer. "Towards Access Pattern Aware Checkpointing". Poster at the *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC), ACM/IEEE SC21 Graduate Research Competition*, St. Louis, MO, 2021.

5. **N. Tan**, R. Bird, and M. Taufer. "Optimizing Vector Particle-In-Cell (VPIC) for Memory Constrained Systems". Poster at the *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC), ACM/IEEE SC20 Graduate Research Competition*, Virtual, 2020. Won 3rd place in the graduate division.

Vita

Nigel Tan received his BS degree in both computer science and applied math from the University of California Merced, CA, USA, and his MS degree in computational and applied math from Rice University. He is currently working toward the PhD degree in computer science under Dr. Michela Taufer at the University of Tennessee, Knoxville. His research interests include in high-performance computing with an emphasis on performance portability and optimization across multiple architectures.