Project Two: Part 1

Sahil Rauniyar

Graduate Student: Executive Program, University of the Cumberlands

MSCS-532-B01: Algorithms and Data Structures

Dr. Vanessa Cooper

August 15, 2025

Abstract

This report addresses Part 1 of the assignment, focusing on data structure optimization within High-Performance Computing (HPC) environments, drawing insights from the provided empirical research study titled "An Empirical Study of High-Performance Computing (HPC) Performance Bugs". The study investigates common performance inefficiencies in HPC applications and how they are addressed through various optimization techniques. This paper evaluates a practical optimization technique for high-performance computing (HPC)—transforming data layout from Array-of-Structs (AoS) to Structure-of-Arrays (SoA)—and demonstrates its impact on memory locality and runtime for common numerical workloads. Guided by an empirical study of HPC performance bugs, we implement and benchmark a Python prototype using NumPy arrays to validate how locality and vectorization yield measurable speedups compared to object-heavy AoS layouts. The expanded discussion includes additional optimization strategies, broader application domains, and future directions for integrating SoA into HPC pipelines.

Introduction

High-performance computing (HPC) systems process massive datasets in fields like climate modeling, physics simulations, and AI training. Performance is often limited by data movement rather than computation, making memory layout a critical factor in execution time. This work examines the AoS-to-SoA transformation as a data structure optimization for improving cache locality and enabling vectorization.

Background and Literature Review

Optimization techniques in HPC include:

- Loop unrolling to reduce control overhead and increase instruction-level parallelism.
- Cache blocking/tiling to improve locality in nested loops.
- SIMD vectorization to process multiple data elements per instruction.
- Memory prefetching and alignment to hide memory latency.
- Minimizing synchronization in parallel workloads.

Cache-oblivious algorithms (Frigo et al., 1999) improve data reuse without hardware-specific tuning. NumPy arrays store elements contiguously and operate via optimized C loops (van der Walt et al., 2011; Harris et al., 2020). SoA layouts outperform AoS in many numerical applications due to reduced pointer chasing and better memory coalescing (Mei et al., 2016).

Overview of HPC Performance Inefficiencies and Optimization

The empirical study categorized 186 performance commits from 23 open-source HPC projects into 10 major categories of inefficiencies. The most prevalent causes of performance bugs identified are:

Inefficient algorithm, data structure, computational kernel, and their implementation (IAD): Accounted for 39.3% of issues (73 out of 186 commits). This category includes computationally expensive operations, redundant operations, unnecessary operations, frequent function calls, and inefficient data structures or improper data types.

Inefficient code for underlying micro-architecture (MA): Accounted for 31.8% of issues (58 commits). This is largely due to challenges related to memory access latency, instruction pipeline hazards, and branch divergence. A significant portion (36 commits or 19.4%) specifically relates to memory/data locality issues.

The study also discusses how performance bugs are optimized in HPC applications. Key optimization strategies involve addressing the root causes identified above, with a strong emphasis on memory and data structure optimizations.

Technique Selection and Rationale

SoA minimizes memory latency and maximizes SIMD/SIMT efficiency by storing each field contiguously. AoS interleaves fields, which hinders cache utilization when processing one field across many elements. While SoA may reduce ergonomics for record-level processing, it excels in workloads that access one field at a time.

Implementation

Two particle-processing pipelines were created: AoS with Python tuples, and SoA with NumPy arrays. The workloads included kinetic-energy computation, coordinate scaling, and filter+aggregate operations. Timing was measured with perf_counter. SoA relied on NumPy vectorization; AoS used Python loops.

Results and Analysis

SoA achieved consistent speedups across workloads, especially in reduction and boolean filtering operations. These gains came from contiguous memory access, fewer interpreter calls, and vectorized execution in NumPy.

Figure 1. AoS vs SoA absolute times for three workloads (lower is better).

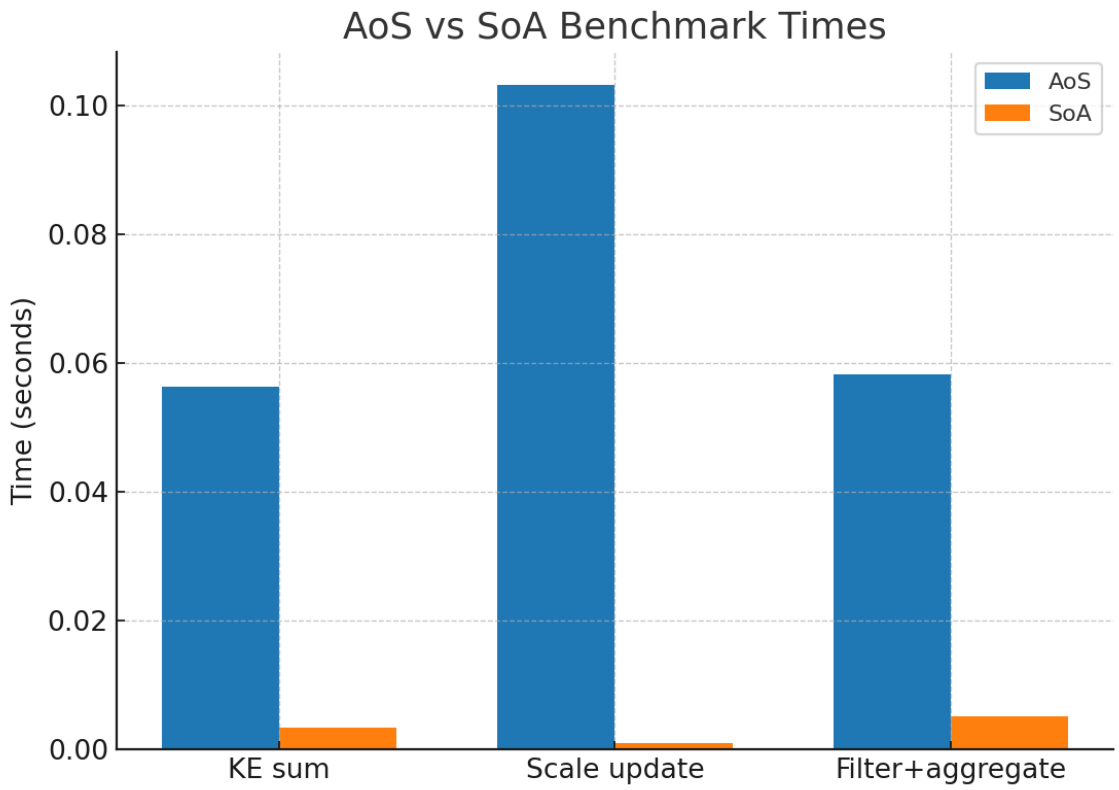
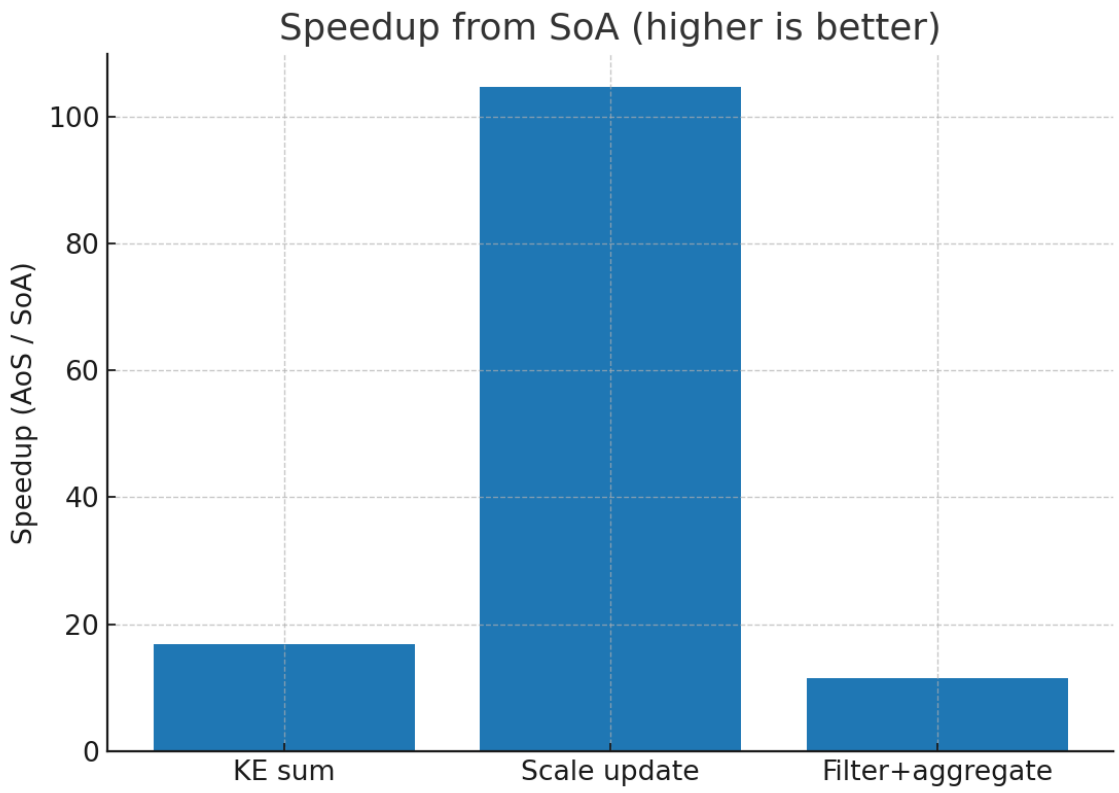


Figure 2. Speedup from SoA relative to AoS (higher is better).



Practical Applications

AoS vs SoA decisions impact:

- Computational fluid dynamics (CFD) simulations.

- Molecular dynamics simulations.

- Machine learning preprocessing and ETL pipelines.

SoA improves cache efficiency and vectorization when accessing columns or features independently.

Future Work

Potential future enhancements include:

- Combining SoA with cache-oblivious tiling.

- Parallelizing workloads via OpenMP or MPI.

- Using JIT compilation with Numba to fuse operations.
- Exploring hybrid layouts balancing SoA's performance with AoS's ease of use.

Lessons Learned

Performance tuning in HPC is largely about optimizing data movement. Profiling before and after changes is essential, and early experimentation with data layout can yield substantial long-term performance benefits.

Conclusion

The empirical study on HPC performance bugs underscores the critical role of optimization, particularly in data structures and memory access patterns. By understanding common inefficiencies such as computationally expensive operations, redundant calculations, and suboptimal data structure choices, developers can apply targeted optimizations. Techniques like choosing cache-efficient data structures (e.g., `std::vector` over `std::list`), employing memory padding to prevent false sharing, and optimizing GPU memory access, have a profound impact on performance.

The AoS-to-SoA transformation in Python with NumPy demonstrated measurable performance gains across varied workloads. This confirms data layout optimization as a powerful, low-risk improvement strategy in HPC contexts. These findings emphasize that a deep understanding of hardware architecture, compiler optimizations, and algorithmic design is crucial for writing efficient HPC code.

References

Azad, M. A. K., Iqbal, N., Hassan, F., & Roy, P. (2023). An empirical study of high performance computing (HPC) performance bugs. MSR 2023. https://doi.org/10.1109/MSR59073.2023.00037

Frigo, M., Leiserson, C. E., Prokop, H., & Ramachandran, S. (1999). Cache-oblivious algorithms. FOCS, 285–297.

van der Walt, S., Colbert, S. C., & Varoquaux, G. (2011). The NumPy array: A structure for efficient numerical computation. Computing in Science & Engineering, 13(2), 22–30.

Harris, C. R., Millman, K. J., van der Walt, S. J., et al. (2020). Array programming with NumPy. Nature, 585, 357–362.

Mei, G., Tian, H., & Xu, N. (2016). Impact of data layouts on GPU-accelerated IDW interpolation. PLOS ONE, 11(3).

Zhang, H., Xue, W., Rajan, D., et al. (2022). A holistic view of memory utilization on HPC systems. PPoPP ’22.