# Title Page

**Title:**  
Optimization in High-Performance Computing: AoS vs SoA

**Student Name:**  
Murali Krishna Chintha

**Course:**  
Algorithms and Data Structures (MSCS-532-B01) – Second Bi-term

**Instructor:**  
Vanessa Cooper

**Date:**  
08/17/2025

# Optimization in High-Performance Computing: AoS vs SoA

## Abstract

High-Performance Computing (HPC) applications are highly sensitive to data structure design and memory access patterns. This study explores the optimization of particle simulation workloads using Array of Structures (AoS) and Structure of Arrays (SoA) layouts. A baseline AoS implementation in Python is compared with optimized SoA implementations using NumPy vectorization and Numba Just-In-Time (JIT) compilation. Benchmarks demonstrate speedups of up to 1800x over the baseline, validating empirical research findings that memory locality and vectorization are critical to HPC performance. The results highlight both strengths and limitations of layout optimization, with implications for designing efficient scientific codes.

## Introduction

High-Performance Computing (HPC) underpins modern scientific discovery, from weather forecasting to molecular dynamics. Central to HPC is not just raw compute power but how efficiently algorithms interact with memory hierarchies. Data structure design plays a vital role in reducing cache misses, improving vectorization, and eliminating performance bottlenecks.

HPC systems rely on complex architectures with multiple layers of cache, high-bandwidth memory, and multicore processors. These features demand that software developers carefully consider how data structures map to hardware capabilities. Poorly designed layouts can negate the benefits of powerful CPUs, leading to performance degradation that scales poorly with problem size. Conversely, even small optimizations in data representation can unlock dramatic improvements in throughput.

This project focuses on data layout optimization—a technique often highlighted in empirical HPC performance studies. Specifically, we compare Array of Structures (AoS) and Structure of Arrays (SoA) layouts for a particle update workload. The objective is to demonstrate how SoA improves cache locality and enables vectorization, resulting in significant performance gains.

## Background and Related Work

The empirical study *An Empirical Study of High Performance Computing (HPC) Performance Bugs* highlights how poor data locality, lack of vectorization, and inefficient data structures are recurrent sources of performance bugs in HPC applications. These bugs are not limited to incorrect results but manifest as hidden inefficiencies, where programs technically run correctly but waste significant computational resources.

AoS layouts, though intuitive, interleave unrelated fields, reducing spatial locality and hindering SIMD (Single Instruction Multiple Data) optimizations. For example, storing particle position and velocity in a single structure causes cache lines to fetch unused fields, leading to wasted bandwidth. By contrast, SoA separates fields into contiguous arrays, aligning well with cache line fills and vector units. This layout allows a CPU to fetch and operate on a vector of values (e.g., all x-coordinates) without skipping irrelevant data.

Prior literature corroborates these findings: - Williams et al. (2009) showed that SoA layouts yield better performance in stencil computations due to improved cache reuse. - Datta et al. (2008) emphasized cache-oblivious optimizations, where algorithms are designed to perform well across different cache sizes without explicit tuning. - Lam, Pitrou, and Seibert (2015) demonstrated how Numba provides high-level access to low-level optimization techniques, bridging productivity and performance. - NVIDIA developer documentation frequently highlights how memory coalescing in GPUs mirrors the same principle: contiguous memory access patterns are critical for performance.

These studies provide both theoretical and practical context for why AoS is often inefficient, and why SoA is a more sustainable choice in high-performance contexts.

## Methodology

### Baseline (AoS)

* Implemented as a NumPy structured array with fields: x, y, z, vx, vy, vz.
* Updated using pure Python loops, mimicking naïve implementations often found in scientific prototypes.
* Each iteration requires multiple pointer dereferences and incurs Python interpreter overhead.

### Optimized (SoA)

1. **NumPy SoA:** Separate arrays for positions and velocities. Updates and aggregations performed with vectorized operations. This approach leverages broadcasting and SIMD instructions through NumPy’s underlying C implementation.
2. **Numba SoA:** Same layout as above, but loops compiled with @njit, exploiting fast math and reducing interpreter overhead. Numba’s LLVM backend enables explicit parallelism (optional) and optimizes tight loops into machine code.

### Experiment Setup

* **Hardware:** MacBook Air M2 (student environment), 8-core CPU.
* **Problem Sizes:** 50,000; 100,000; 200,000 particles.
* **Steps:** 50 timesteps to simulate multiple updates.
* **Trials:** 5 per implementation to average variability.
* **Metrics:** Mean runtime per trial (ms), standard deviation, kinetic energy, mean distance from origin.
* **Outputs:** CSV logs, runtime plots, speedup plots.

This experimental design provides both quantitative performance data and qualitative insights into code maintainability and complexity.

## Results

The benchmark results reveal striking differences between AoS and SoA implementations.

### Table 1

**Runtime performance (mean of 5 trials in milliseconds)**

| N | Baseline AoS (ms) | NumPy SoA (ms) | Numba SoA (ms) |
| --- | --- | --- | --- |
| 50,000 | 11,746 | 14.47 | 6.32 |
| 100,000 | 24,416 | 29.59 | 13.35 |
| 200,000 | 47,148 | 83.88 | 32.02 |

*Note.* Values represent mean runtime over five trials. Lower values indicate better performance.

### Key Observations

* **Baseline AoS** grows linearly but remains prohibitively slow (seconds per run). The interpreter overhead compounds as N increases.
* **NumPy SoA** achieves ~400–800x speedup by leveraging vectorization. However, performance gains plateau slightly for larger N due to array operation overhead.
* **Numba SoA** achieves ~1800x speedup, bringing runtimes to single-digit milliseconds for smaller workloads. It scales better at large N by eliminating Python overhead entirely.
* All implementations preserve correctness: kinetic energy and mean distances match across approaches, verifying functional equivalence.

### Figures

A graph with a line and numbers

AI-generated content may be incorrect.

*Figure 1.* AoS vs SoA performance (absolute runtime)

A graph with blue and orange lines

AI-generated content may be incorrect.

*Figure 2.* Speedup of SoA over AoS baseline.

*Note.* Figures were generated from benchmark results. Figure 1 shows runtime in milliseconds; Figure 2 shows relative speedup against the baseline implementation.

Both figures confirm the orders-of-magnitude improvement due to layout optimization.

## Discussion

The results align closely with findings from the empirical study: performance bugs often arise from non-vectorized code and inefficient memory layouts. AoS, though intuitive, is cache-unfriendly and incurs massive Python interpreter overhead. By contrast, SoA enables vectorization and efficient cache utilization.

**Strengths of SoA:** - Contiguous memory layout improves cache locality. - Supports SIMD/vectorization and JIT compilation. - Well-suited for large-scale numerical simulations. - Portable: the same code can benefit from both CPUs and GPUs with minimal changes.

**Weaknesses:** - Code complexity increases when handling heterogeneous data. - Requires careful redesign of existing AoS-based applications. - Less intuitive for developers accustomed to object-oriented patterns. - Debugging vectorized code can be more challenging due to abstraction layers.

**Comparison to Theory:** - The empirical study predicted that vectorization and memory-aware optimizations produce significant speedups. Our results validate this prediction with 1000x+ improvements. - Minor divergence: NumPy’s vectorization shows diminishing returns at larger N, suggesting additional optimizations such as blocking or tiling might further improve scalability.

## Lessons Learned

* Python, though traditionally slower, can achieve HPC-like performance when paired with libraries like NumPy and Numba.
* Data layout is not a minor detail but a fundamental determinant of computational efficiency.
* Benchmarks reinforce that theoretical optimizations (cache locality, vectorization) translate directly into measurable gains.
* While Python allows for rapid prototyping, reaching peak performance requires awareness of underlying hardware constraints.

## Conclusion

This project demonstrated that SoA layout, combined with vectorization and JIT compilation, dramatically outperforms AoS in HPC workloads. The findings echo empirical research: memory locality and data structure choice are fundamental to achieving high performance. Future work may extend this prototype with: - Parallel kernels using Numba’s prange. - GPU acceleration with CuPy or Numba CUDA. - Testing on larger particle counts to observe cache saturation. - Applying similar optimization principles to graph algorithms or sparse matrix operations.

By integrating lessons from both theoretical research and practical experimentation, developers can write Python code that approaches the performance of low-level C or Fortran in HPC contexts.

## References

* Datta, K., et al. (2008). *Optimization and performance modeling of stencil computations on modern microprocessors.* SIAM Review.
* Williams, S., Waterman, A., & Patterson, D. (2009). *Roofline: An insightful visual performance model for multicore architectures.* Communications of the ACM.
* Lam, S. K., Pitrou, A., & Seibert, S. (2015). *Numba: A LLVM-based Python JIT compiler.* Proceedings of the Second Workshop on the LLVM Compiler Infrastructure.
* Harris, M. (2005). *Optimizing memory access in CUDA.* NVIDIA Developer Blog.
* Empirical Study Reference: *An Empirical Study of High Performance Computing (HPC) Performance Bugs.*
* van der Walt, S., Colbert, S. C., & Varoquaux, G. (2011). *The NumPy array: A structure for efficient numerical computation.* Computing in Science & Engineering.
* Dongarra, J., Beckman, P., Moore, T., et al. (2011). *The International Exascale Software Project roadmap.* International Journal of High Performance Computing Applications.
* Henretty, T., et al. (2011). *Data layout transformation for stencil computations on multicore CPUs.* Proceedings of the International Conference on Supercomputing.