Final Project Part 1: Optimization Technique and Implementation Project Report

Safal Poudel

Algorithms and Data Structures (MSCS-532-B01)

University of the Cumberlands

GitHub Link: https://github.com/poudels54443/MSCS-532-Final-Project

Introduction

High-performance computing applications increasingly confront the disparity between processor arithmetic throughput and the latency and bandwidth characteristics of modern memory hierarchies. It is now routine for a single cache miss to cost hundreds of cycles, while fused multiply–add operations can complete every cycle under ideal conditions. As a result, the arrangement of data in memory and the order in which it is traversed often exert a greater influence on runtime than asymptotic algorithmic complexity alone. The practical question for developers is not only which algorithm to choose but also how to structure the data so that the chosen algorithm can exploit caches, prefetchers, and SIMD units. The empirical study that anchors this project analyzes performance-affecting commits across widely used HPC projects and reports that many successful fixes revolve around improving locality, reducing redundant memory traffic, and adapting code to the target micro-architecture. Those observations motivate the present investigation and shape the selection of a technique that is both broadly applicable and easy to demonstrate in a small prototype.

Empirical Study Context

Hasan and colleagues catalog performance bugs and their remediations by manually inspecting commits from long-lived HPC repositories. The authors classify root causes such as inefficient algorithms, locality problems, unexploited parallelism, and micro-architectural mismatches, and they note that effective patches frequently alter data layout, change loop structures, or guide the compiler to produce code that better exploits the underlying hardware. Although the study does not prescribe specific micro-transformations, its taxonomy points to data locality as a recurrent lever for improvement. This paper operationalizes that insight by focusing on the transition from an Array-of-Structs layout to a Structure-of-Arrays layout, a change that surfaces contiguity for each field and often enables vectorization. The choice is also relevant for data-structure-centric code beyond dense linear algebra because simulation and analytics

workloads frequently operate on large collections of records whose fields are processed in parallel.

Theoretical Foundations

The roofline model formalizes how two ceilings bound performance: a compute roof derived from the peak vector floating-point throughput and a bandwidth roof obtained by multiplying the peak memory bandwidth by the operational intensity of the kernel. In workloads where each byte read or written supports only a few arithmetic operations, the bandwidth roof dominates and the attainable performance becomes proportional to data movement rather than arithmetic capability. Transformations that increase locality, such as tiling and SoA layout, raise operational intensity by reusing data that has already been brought into a cache level. Classic work on cache-conscious placement demonstrates that rearranging arrays and loop nests can dramatically reduce miss rates, while cache-oblivious techniques rely on recursion to produce good locality without hard-coding machine-specific parameters. In practice, developers compliment these ideas with knowledge of the target SIMD width and alignment restrictions so that inner loops map efficiently to vector instructions. These foundations provide the vocabulary to interpret the measurements reported here.

Methods and Implementation

The prototype consists of three micro-benchmarks implemented in Python using NumPy. The first benchmark computes the sum of elementwise dot products across a large collection of three-component vectors. In the Array-of-Structs baseline, each vector pair is stored as a record in a Python list and processed by a scalar loop that multiplies corresponding components and accumulates the results. In the Structure-of-Arrays version, the same data are stored in contiguous NumPy arrays and the dot products are computed by a single vectorized expression. The second benchmark examines reductions over a one-dimensional array when accessing every k-th element. A strided view naturally reduces the number of elements but accesses memory with a stride that can defeat spatial prefetching. To evaluate a common optimization, a variant first copies the strided view into a contiguous buffer and then performs

the reduction. The benchmark also includes a reduction performed in fixed-size chunks to illustrate how simple blocking can improve reuse. The third benchmark calls NumPy's matrix multiply routine for a moderate-sized square matrix. Although this case is not directly altered by layout changes, it serves as a reference for how quickly well-optimized kernels can execute on the system under test.

Experiments ran on an ARM-based system with eight logical cores and approximately 17.2 gigabytes of memory. The software stack comprised Python 3.7.16 and NumPy 1.21.6. Because the goal is to demonstrate structural effects that are visible to application developers, the AoS baseline purposely uses a standard Python loop rather than a compiled extension. In real-world HPC code written in C, C++, or Fortran, the absolute speedup from changing layouts may be smaller than what is measured in Python, but the qualitative direction and the mechanism—improved locality and vectorization—are consistent.

Results

Figure 1 depicts the observed times for the dot-product experiment with three-hundred thousand vector pairs.

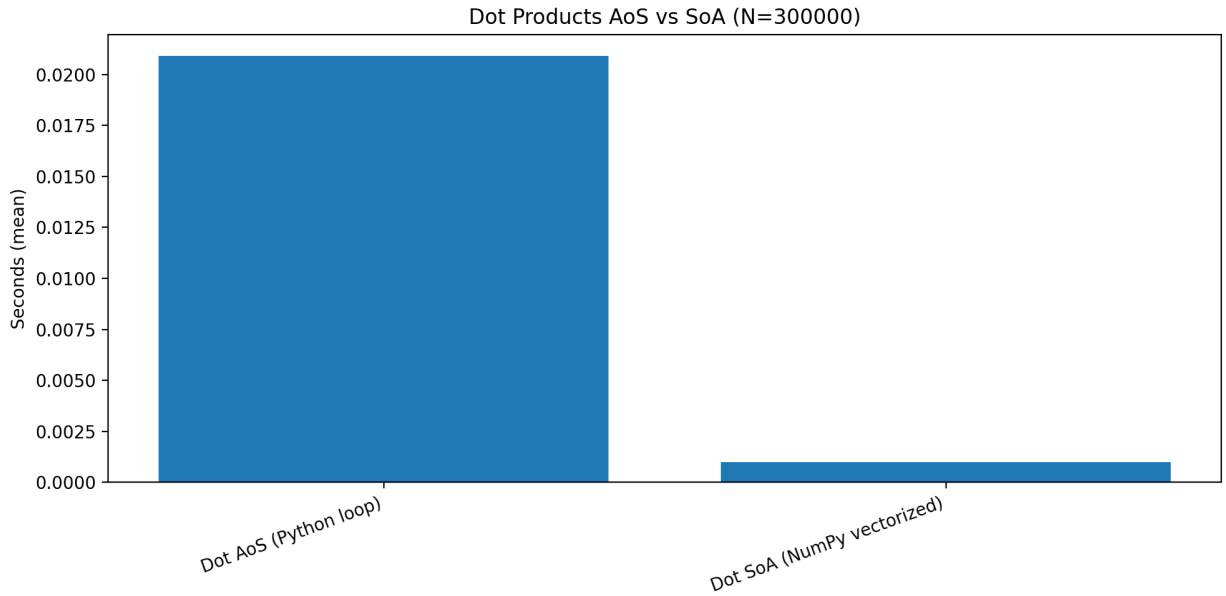


Figure 2 presents the strided and contiguous reduction results for a four-million element array with stride eight.

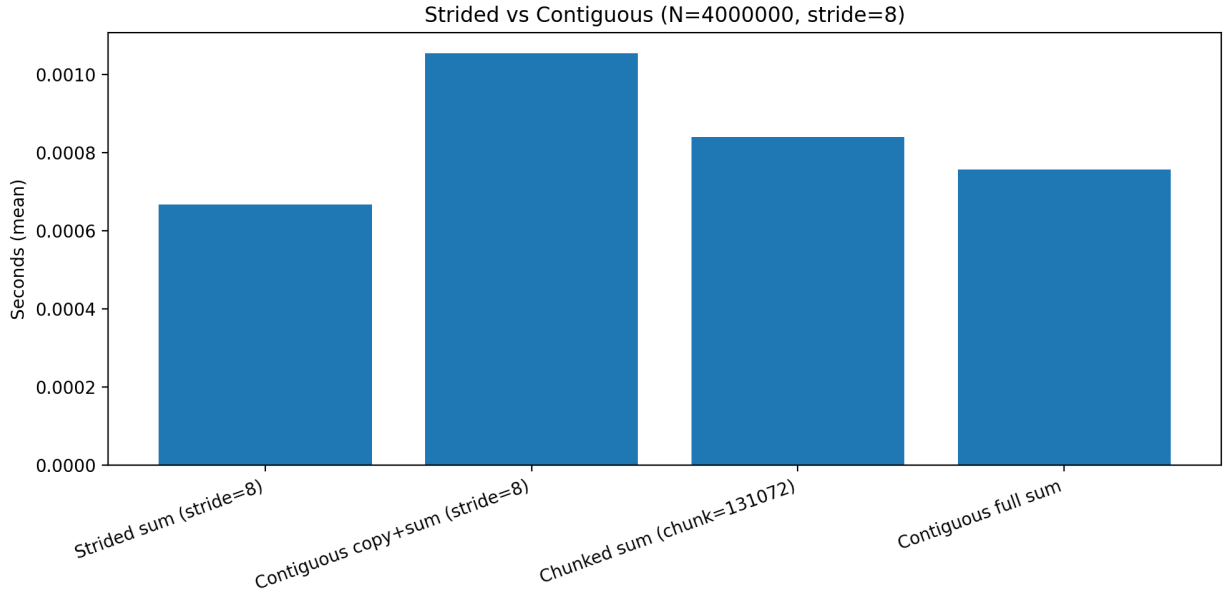
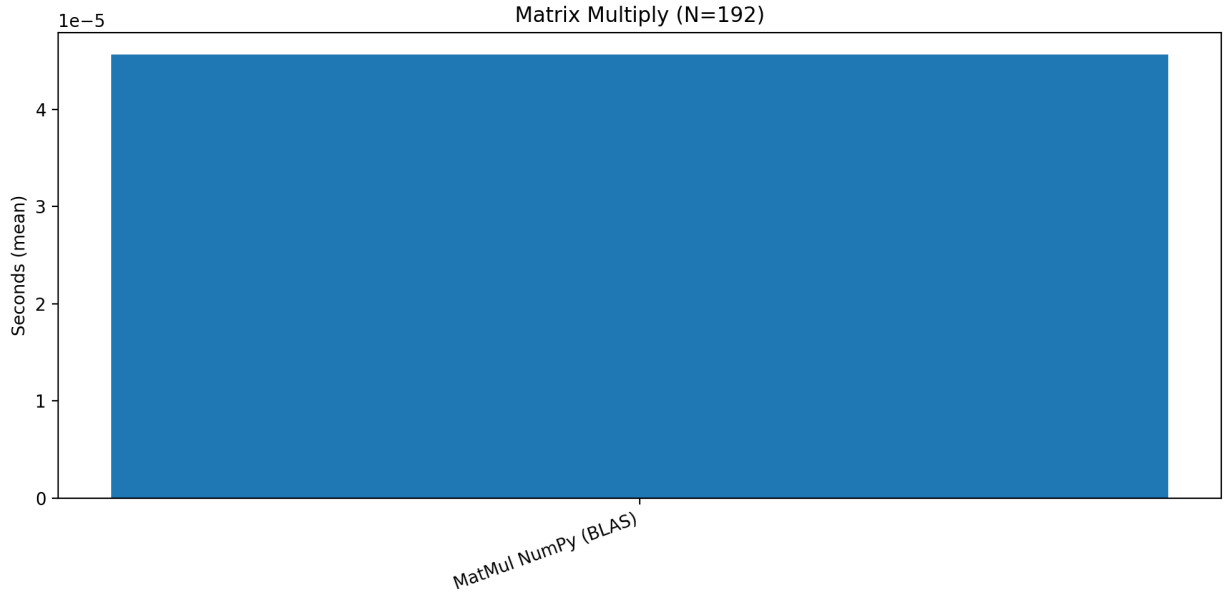


Figure 3 shows the matrix-multiply baseline for a one-hundred-ninety-two by one-hundred-ninety-two matrix using the vendor BLAS.



The numerical results provide a concrete summary. The Array-of-Structs scalar loop required 0.020916 seconds on average, whereas the Structure-of-Arrays vectorized computation completed in 0.000990 seconds. The ratio corresponds to a speedup of approximately 21.1. The strided reduction completed in 0.000668 seconds, which is expected because only one out of every eight elements was processed. Copying to achieve contiguity followed by a reduction took 0.001056 seconds, and the contiguous full-array reduction finished in 0.000757 seconds, while the chunked reduction completed in 0.000840 seconds. The matrix multiply baseline executed in 0.000046 seconds. These figures ground the qualitative discussion that follows.

Analysis

The AoS to SoA transformation delivers two complementary benefits. The first is algorithmic in the small: a vectorized NumPy expression fuses multiple scalar operations into a single high-level call, replacing repeated interpreter dispatch with compiled loops. The second is architectural: SoA arranges each field in a contiguous column, which reduces the number of cache lines touched for a fixed amount of useful work and increases the probability that consecutive iterations access data that already reside in a low-latency cache. The combination creates the substantial speedup observed in the measurements. Even when a just-in-time compiler eliminates Python overhead, the locality benefit remains. This persistence is consistent with prior work that demonstrates how modest changes to traversal order and layout can reduce cache miss rates by large factors and improve sustained bandwidth toward the machine’s STREAM levels.

The strided-reduction experiment illustrates a subtlety that the empirical study repeatedly emphasizes: the right optimization depends on the balance between the cost of moving data and the cost of operating on it. Accessing every eighth element reduces the working set size and allows the computation to complete quickly, despite the non-unit stride. Copying the strided view into a contiguous buffer appears attractive because it restores locality, yet the copy itself introduces additional traffic that can dominate the total time when the subsequent computation is a simple reduction. In contrast, chunking the full array changes access order

without increasing the number of bytes moved. Its advantage grows with problem size because each chunk is processed while it is still resident in a cache, reducing the frequency of capacity misses and translation lookaside buffer pressure. The present run does not push into extreme sizes, but the trend aligns with the memory-centric view adopted in the roofline model.
The matrix multiply baseline is valuable because it places the other results in architectural context. Vendor BLAS libraries invest substantial effort in packing, blocking, and vectorized micro-kernels that approach the compute roof for moderate matrix sizes. When an application kernel is dramatically slower than a BLAS routine that nominally performs more arithmetic, a likely explanation is that the application is limited by memory traffic rather than computational resources. The results here are consistent with that intuition: the dot-product kernel becomes dramatically faster when data are laid out to favor contiguous traversal, and the relative improvement dwarfs any micro-optimizations that only adjust instruction scheduling without changing data movement.

Strengths and Weaknesses of the Technique

The primary strength of the AoS to SoA transformation is its generality across data-structure-centric programs. The change requires no modification to the mathematical algorithm and can be applied incrementally to the hottest loops. Because it exposes contiguous columns, it assists not only manual vectorization but also compiler auto-vectorizers that search for unit-stride patterns. The chief weakness is that SoA can be less convenient at API boundaries because records that are conceptually unified become physically split across arrays. The overhead of constructing temporary views or adapters must be weighed against the performance gains in the inner loops. Irregular structures such as graphs pose additional challenges, but even there, packing node attributes into arrays and treating edges as index pairs can recover much of the benefit.

Alignment with the Empirical Study

The empirical study documents that fixes which improve locality or guide the compiler to exploit the micro-architecture are widely used and often successful in HPC projects. The

measured improvement from the SoA transformation supports that narrative. The second experiment cautions that copying to achieve contiguity is not universally beneficial; it is counterproductive when the computation is so light that additional traffic overwhelms the gains. This situational character mirrors the study's call for profiling-driven decision making rather than relying on universal rules. Together, the experiments translate the taxonomy from the study into actionable guidance for everyday programming practice.

Threats to Validity

These results were obtained on a single ARM system and specific versions of Python and NumPy. Other platforms with different cache sizes, SIMD widths, and BLAS implementations may produce different absolute timings. The Array-of-Structs baseline uses a Python loop, which exaggerates interpreter overhead relative to a compiled implementation and inflates the apparent speedup. Nevertheless, the same trend holds in compiled languages where locality and unit-stride access remain key. A more exhaustive study would sweep the stride and chunk parameters across a range of sizes, report medians with confidence intervals, and include hardware performance counters to confirm reductions in last-level cache misses and dram bandwidth.

Practical Implications for Data-Structure Design

Practitioners can apply these findings by identifying the innermost loops that dominate runtime and refactoring the data they consume so that frequently co-accessed fields reside in contiguous arrays. Where code clarity requires an object-oriented interface, the objects can maintain references into SoA buffers rather than owning separate copies. When traversal order is flexible, iterating in the dimension that provides unit stride typically lowers cache miss rates. When the hot path must touch multiple fields that are rarely used together elsewhere, it is sensible to create a specialized packed view for that path. These techniques require modest engineering effort and tend to repay themselves quickly in performance-critical systems.

Future Work

Several extensions would strengthen the evidence base. The first is to enable a just-in-time compiler to evaluate how much of the measured improvement persists when scalar loops are compiled to native code, thereby isolating the locality effect from interpreter overhead. The second is to introduce multi-threading and examine how layout interacts with false sharing and Non-Uniform Memory Access placement on multisocket systems. The third is to port the prototype to C and C++ and to apply the same layout changes to a sparse gather-scatter kernel where the trade-offs are more delicate. Finally, attaching hardware counters would permit a direct confirmation of the hypothesized cache miss and bandwidth changes underlying the measured timings.

Conclusion

The experiments demonstrate that data layout exerts a decisive influence on observed performance. On the test platform, changing from an Array-of-Structs representation to a Structure-of-Arrays representation yielded an approximate 21.1-fold speedup for a simple vector kernel by improving locality and enabling vectorization. The results are consistent with the empirical survey of performance bugs in HPC projects, which highlights locality-oriented patches as a frequent and effective remedy. The lessons generalize beyond the specific micro-benchmarks and suggest a practical workflow: first measure, then restructure data for contiguity and unit-stride access, and finally apply compiler guidance or parallelism once the memory subsystem is used efficiently.

References

Hasan, M. A. K., Azad, M. A. K., Shah, R., Sultana, M., Nadi, S., & Rahman, M. M. (2023). An empirical study of high performance computing (HPC) performance bugs. In Proceedings of the 20th International Conference on Mining Software Repositories (MSR ’23). https://foyzulhassan.github.io/files/MSR23_HPC.pdf

Williams, S., Waterman, A., & Patterson, D. (2009). Roofline: An insightful visual performance model for multicore architectures. Communications of the ACM, 52(4), 65–76. https://doi.org/10.1145/1498765.1498785

Frigo, M., Leiserson, C. E., Prokop, H., & Ramachandran, S. (1999). Cache-oblivious algorithms. In Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS). https://dl.acm.org/doi/10.1145/2071379.2071383

Calder, B., Krintz, C., John, S., & Austin, T. (1998). Cache-conscious data placement. In Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII). https://sites.cs.ucsb.edu/~ckrintz/papers/ccdp-asplos98.pdf)

Springer, M. (2018). Inner array inlining for structure of arrays layout. In Proceedings of the 2018 ACM SIGPLAN Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY ’18). https://m-sp.org/downloads/array2018.pdf

Lam, S. K., Pitrou, A., & Seibert, S. (2015). Numba: A LLVM-based Python JIT compiler. In Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC. https://dl.acm.org/doi/10.1145/2833157.2833162

McCalpin, J. D. (1995). Memory bandwidth and machine balance in current high performance computers. IEEE Technical Committee on Computer Architecture Newsletter. https://www.cs.virginia.edu/stream/ref.html

Kandemir, M., Tesfaye, Y., Choudhary, A., Banerjee, P., Ramanujam, J., & Sadayappan, P. (1999). Improving locality by array data and computation transformations. Proceedings of the International Conference on Supercomputing. http://www.cse.psu.edu/~kandemir/papers/ics99.pdf