15-745 F25 Project Proposal

Sun A Cho and Yufei Shi

Revised October 30, 2025

1 Project Title

Optimizing Data Structure Layout for Improved Memory Performance

2 Group Info

Sun A Cho (sunac@andrew.cmu.edu) and Yufei Shi (yshi2@andrew.cmu.edu)

3 URL for Project Web Page

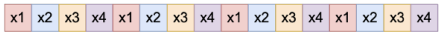
https://cmu-15745-f25.github.io/project-website/

4 Project Description

4.1 Background

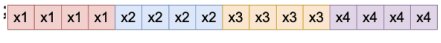
As processor speed continue to increase, their memory system performance has failed to keep pace, leading to the growing disparity between CPU performance and memory access latency. This phenomenon has made the memory system a primary bottleneck in modern computing performance. To mitigate this, computers nowadays employ a hierarchy of multi-level cache memory to improve memory access speed by exploiting spatial and temporal locality within the user programs. However, the effectiveness of such caching mechanisms depends critically on the memory access patterns exhibited by applications.

In imperative low-level programming languages like C and C++, programmers can directly influence the memory access patterns by organizing and accessing data in structured ways. Prior works have identified two memory access patterns commonly used for organizing collections of data: *arrays of structures* (AoS) and *structures of arrays* (SoA) [1, 3, 4, 5], as shown in Figure 1. The choice between these layouts has significant performance implications through their effects on cache utilization, auto-vectorization, and false sharing.

C++ Data Structure Definition	Memory Layout	Access Syntax
`struct S { double x1, x2, x3, x4; };` `using AoS = std::array<S, 4>;`		`auto first = aos[0].x1;`

(a)

C++ Data Structure Definition	Memory Layout	Access Syntax

`struct SoA {`
`  std::array<double, 4> x1,x2,x3,x4;`
`};`
		`auto first = soa.x1[0];`

(b)

Figure 1: Data structure definition, memory layout, and access syntax of *array of structures* (Figure 1a) and *structure of arrays* (Figure 1b) [1].

4.2 Motivation

Current compilers leave data layout decisions entirely to programmers, who often choose based on intuition or momentum rather than profiling. Even when profiling is used, it reflects the current code structure instead of what performance could be achieved with data layout optimizations. It is also not always clear which data layout will better serve an application without deliberating the structure of the program. Worse, manually converting existing code from using one data layout to another is labor-intensive and error-prone.

Imperative programmers naturally think in terms of objects and data structures, leading to a preference for AoS layouts. However, experiments show that using SoA is preferable to AoS unless data is being accessed in a striding pattern [1]. This motivates a compiler optimization that applies automated analysis to identify and replace suboptimal AoS layouts to SoA layouts to improve performance.

4.3 Goals

75% Goal: Implement a compiler analysis that identifies memory accesses to suboptimal AoS layouts (AoS with non-strided access patterns).

100% Goal: Implement a compiler pass that automatically converts suboptimal AoS layouts to SoA layouts and updates AoS accesses to SoA accesses.
Measure performance improvements through benchmarking.

125% Goal: Implement cache-line-aligned padding between SoA arrays.
Extend analysis with a cost model that uses memory access frequency and pattern analysis to determine whether SoA or AoS is preferred, and selectively apply the transformation only when profitability analysis indicates clear benefit.

5 Logistics

5.1 Plan of Attack and Schedule

Week 1 (Oct 20): Both: Literature review and brainstorm project ideas.

Week 2 (Oct 27): Both: Revise and finalize project proposal.
Both: Set up development environment and build system.
Both: Create an initial benchmark suite (3-5 small kernels with different data layouts).
Both: Study LLVM's `GEP` instruction and `StructType`.
Yufei: Skeleton analysis pass.

Week 3 (Nov 3): Sun A: Access pattern analysis.
Yufei: Loop traversal and struct array detection.
Both: Integrate and finish the analysis implementation, validate on benchmarks.

Week 4 (Nov 10): Sun A: Start SoA struct type generation.
Sun A (if time permits): Add cache-line-aligned padding between SoA arrays.
Yufei: Start memory access rewriting for SoA accesses.
Yufei (if time permits): Draft a basic profitability heuristic.

Week 5 (Nov 17): *Nov 20 milestone.*
Sun A: Complete AoS to SoA struct transformation.
Yufei: Complete memory access rewriting.
Both: Integrate, initial testing on the benchmarks, and debug if necessary.

Week 6 (Nov 24): *Week of Thanksgiving.*
Both: Evaluate the optimization using the benchmarks.
Both: Start working on the project report and poster.
Both (if time permits): Evaluate the optimization using applications from well-established benchmark suites.

Week 7 (Dec 1): Both: Work on the project report and poster.

5.2 Milestone

By November 20th, we hope to have completed the basic analysis (75% goal) and are close to completing the compiler pass (100% goal).

5.3 Literature Search

We reviewed several literature related to data layout optimization. The AoS to SoA transformation has been introduced as a high-level data structure optimization by Rompf et al [5]. The work by Chen et al. [1] demonstrates the performance implications of AoS versus SoA on cache utilization. Their prototype implementation inspired us, though we implement transformations at the compiler level rather than through a C++ library. Radtke and Weinzierl [4] uses C++ attributes to guide the compiler to achieve semi-manual conversions between data layouts, demonstrating the practicality of a compiler-based automatic layout transformation. Finally, the ispc compiler [3] introduces an interesting *hybrid SoA* layout and serves as our reference on how data layout choices affect vectorization.

5.4 Resources Needed

We anticipate only using the LLVM framework which is open-source and available at `https://releases.llvm.org` [2]. We will be primarily developing this project on macOS with arm CPUs, which is available to all members of the group. We will potentially need access to machines with AVX to evaluate how our optimization works with autovectorization; we plan to use the GHC clusters for this, but have yet to verify the AVX support.

5.5 Getting Started

We have studied relevant literature on data layout transformations and memory access pattern optimizations. We have written a simple LLVM pass to familiarize ourselves with LLVM's `GEP` instruction and `StructType`. There is nothing blocking us from getting started immediately.

References

[1] Jolly Chen, Ana Lucia Varbanescu, and Axel Naumann. Optimizing memory access patterns through automatic data layout transformation (work in progress paper). In *Companion of the 16th ACM/SPEC International Conference on Performance Engineering*, ICPE '25, page 47–53, New York, NY, USA, 2025. Association for Computing Machinery.

[2] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004.

[3] Matt Pharr and William R. Mark. ispc: A spmd compiler for high-performance cpu programming. In *2012 Innovative Parallel Computing (InPar)*, pages 1–13, 2012.

[4] Pawel K. Radtke and Tobias Weinzierl. Compiler support for semi-manual aos-to-soa conversions with data views. In Roman Wyrzykowski, Jack Dongarra, Ewa Deelman, and Konrad Karczewski, editors, *Parallel Processing and Applied Mathematics*, pages 301–314, Cham, 2025. Springer Nature Switzerland.

[5] Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing data structures in high-level programs: new directions for extensible compilers based on staging. *SIGPLAN Not.*, 48(1):497–510, January 2013.