Optimization Technique and Implementation Project Report

Bereket Gebremariam

[School of Computer Information Sciences](https://www.ucumberlands.edu/school-computer-information-sciences),  
University of the Cumberlands

MSCE-532-M80

**Dr. Satish Penmatsa**

October 25, 2025

**Part 1: Optimization Technique Selection and Prototype**

**1. Selected Optimization Technique**

Based on the categories of performance fixes and root causes identified in the empirical study (e.g., "inefficient code for target micro-architecture," "locality optimization for cache and memory"), a powerful and demonstrable technique relevant to data structures in HPC is:

**Cache and Memory Locality Optimization (Data Layout)**

This technique addresses one of the most fundamental causes of performance bottlenecks in modern hardware: **poor memory access patterns**. In HPC, data is often accessed in large, repetitive patterns (like loops over large arrays), making cache efficiency critical.

| **Aspect** | **Discussion for the Report** |
| --- | --- |
| **Justification & Relevance** | HPC relies on fast, contiguous memory access to utilize CPU caches and vector units (SIMD). Data structures that are not **cache-aware** (e.g., a Python list of objects) force the CPU to constantly fetch data from the slow main memory (RAM). Optimizing data layout ensures data needed sequentially is stored sequentially, maximizing **data locality** and reducing memory latency. |
| **Strengths** | **Massive speedups** for data-intensive algorithms; directly leverages modern hardware architecture; a foundational optimization for *any* high-performance code. |
| **Weaknesses** | Can make the code less readable or less object-oriented (by separating related fields); requires understanding of the underlying memory model; changes to data structures can be complex. |
| **Use in Python (Prototype)** | The standard Python list is inherently inefficient for this, but libraries like **NumPy** are built specifically to handle data layout efficiently. NumPy stores data in **contiguous C-style arrays**. |

**2. Prototype Implementation Suggestion (in Python)**

Your prototype should compare two versions of a data structure operation: **Inefficient** vs. **Optimized**.

| **Prototype Detail** | **Description** |
| --- | --- |
| **Concept** | **Array of Structures (AOS) vs. Structure of Arrays (SOA)** |
| **Inefficient Data Structure (AOS)** | A standard Python list of objects, where each object contains a set of related data fields (e.g., a list of Particle objects, each with x, y, z coordinates). The fields for *all* particles are scattered in memory, leading to poor cache performance when accessing just one field across all particles. |
| **Optimized Data Structure (SOA)** | A **NumPy array** (or multiple arrays) where each array stores a single field (e.g., one array for all x coordinates, one for all y, one for all z). All x values are stored contiguously. When you iterate over all x values, you minimize cache misses. |
| **The Test** | Write a simple function (e.g., calculate the total distance traveled, or sum of all x coordinates) and run it on a **large dataset** (e.g., 1 million particles) for both the AOS (pure Python) and SOA (NumPy) versions. Use the time or timeit module to measure and compare the execution time. |
| **Observed Improvement** | The NumPy/SOA version should be **orders of magnitude faster**, directly demonstrating the impact of memory locality on performance. |