# Optimization in High-Performance Computing: AoS vs SoA – Code and Execution Evidence

**Student Name:**  
Murali Krishna Chintha

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

**Instructor:**  
Vanessa Cooper

**Date:**  
08/17/2025

## Introduction

This document contains the source code, screenshots of execution, and repository link for the project *Optimization in High-Performance Computing: AoS vs SoA*. It complements the main APA report by providing direct implementation evidence and supporting artifacts.

## Python Source Code

Below is the implementation of optimization\_demo.py, which demonstrates the performance difference between Array of Structures (AoS) and Structure of Arrays (SoA) layouts using NumPy vectorization and Numba JIT compilation.

"""

Optimization Demo: AoS vs SoA with NumPy and Numba

Single-file benchmark to demonstrate how data layout and vectorization affect

performance in an HPC-flavored particle update workload.

What this file includes

1) Baseline AoS (Array of Structures) using a NumPy structured array and a pure

Python loop. This is intentionally slow and cache unfriendly.

2) Optimized SoA (Structure of Arrays) with two approaches:

a) Vectorized NumPy operations

b) Numba-compiled kernels (@njit) for fast loops that exploit contiguous data

3) A benchmark harness with warmup, multiple trials, scaling over N, and charts.

4) Deterministic random generation via a fixed seed unless overridden.

5) Outputs: a CSV of results and one or more PNG charts.

How to run

$ python optimization\_demo.py --sizes 50000 100000 200000 --steps 50 --trials 5 \

--outdir results

Dependencies

- Python 3.9+

- numpy

- numba

- matplotlib

- pandas (for CSV output)

Notes

- If you see very slow Baseline timing at large N, that is expected.

- The Numba path will trigger a JIT compile on first call; warmup is done to avoid

polluting the measured runs.

- Keep an eye on memory when pushing N to very large values.

"""

from \_\_future\_\_ import annotations

import argparse

import math

import time

from dataclasses import dataclass

from pathlib import Path

from typing import Dict, List, Tuple

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from numba import njit, prange

# -----------------------------

# Data generation and utilities

# -----------------------------

@dataclass

class Config:

sizes: List[int]

steps: int

trials: int

dt: float

seed: int

outdir: Path

no\_plots: bool

use\_parallel: bool

def set\_seed(seed: int) -> None:

np.random.seed(seed)

# --------------

# AoS Baseline

# --------------

def make\_aos(N: int) -> np.ndarray:

"""Create an Array-of-Structures layout using a NumPy structured dtype.

Each particle has fields: x, y, z, vx, vy, vz. Positions and velocities.

"""

dtype = np.dtype([

("x", np.float64), ("y", np.float64), ("z", np.float64),

("vx", np.float64), ("vy", np.float64), ("vz", np.float64),

])

arr = np.zeros(N, dtype=dtype)

arr["x"] = np.random.randn(N)

arr["y"] = np.random.randn(N)

arr["z"] = np.random.randn(N)

arr["vx"] = np.random.randn(N) \* 0.01

arr["vy"] = np.random.randn(N) \* 0.01

arr["vz"] = np.random.randn(N) \* 0.01

return arr

def baseline\_aos\_step(arr: np.ndarray, dt: float) -> Tuple[float, float]:

"""Pure Python loop update for AoS.

Updates positions by velocities, then computes kinetic energy sum and

average distance from origin. Returns (kinetic\_energy, mean\_distance).

"""

N = arr.shape[0]

ke\_sum = 0.0

dist\_sum = 0.0

for i in range(N):

arr[i]["x"] += arr[i]["vx"] \* dt

arr[i]["y"] += arr[i]["vy"] \* dt

arr[i]["z"] += arr[i]["vz"] \* dt

vx = arr[i]["vx"]; vy = arr[i]["vy"]; vz = arr[i]["vz"]

ke\_sum += 0.5 \* (vx\*vx + vy\*vy + vz\*vz)

x = arr[i]["x"]; y = arr[i]["y"]; z = arr[i]["z"]

dist\_sum += math.sqrt(x\*x + y\*y + z\*z)

return ke\_sum, dist\_sum / N

def baseline\_aos\_run(N: int, steps: int, dt: float) -> Dict[str, float]:

arr = make\_aos(N)

ke = 0.0

md = 0.0

for \_ in range(steps):

ke, md = baseline\_aos\_step(arr, dt)

return {"ke": ke, "mean\_dist": md}

# --------------

# SoA Optimized

# --------------

def make\_soa(N: int) -> Tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray, np.ndarray, np.ndarray]:

"""Create a Structure-of-Arrays layout as 6 separate contiguous arrays."""

x = np.random.randn(N).astype(np.float64)

y = np.random.randn(N).astype(np.float64)

z = np.random.randn(N).astype(np.float64)

vx = (np.random.randn(N) \* 0.01).astype(np.float64)

vy = (np.random.randn(N) \* 0.01).astype(np.float64)

vz = (np.random.randn(N) \* 0.01).astype(np.float64)

return x, y, z, vx, vy, vz

# Vectorized NumPy version

def numpy\_soa\_step(x, y, z, vx, vy, vz, dt: float) -> Tuple[float, float]:

x += vx \* dt

y += vy \* dt

z += vz \* dt

ke = 0.5 \* (vx\*vx + vy\*vy + vz\*vz)

ke\_sum = float(np.sum(ke))

dist = np.sqrt(x\*x + y\*y + z\*z)

mean\_dist = float(np.mean(dist))

return ke\_sum, mean\_dist

def numpy\_soa\_run(N: int, steps: int, dt: float) -> Dict[str, float]:

x, y, z, vx, vy, vz = make\_soa(N)

ke = 0.0

md = 0.0

for \_ in range(steps):

ke, md = numpy\_soa\_step(x, y, z, vx, vy, vz, dt)

return {"ke": ke, "mean\_dist": md}

# Numba JIT kernels

@njit(fastmath=True)

def \_numba\_soa\_step(x, y, z, vx, vy, vz, dt):

N = x.shape[0]

ke\_sum = 0.0

dist\_sum = 0.0

for i in range(N):

x[i] += vx[i] \* dt

y[i] += vy[i] \* dt

z[i] += vz[i] \* dt

vi2 = vx[i]\*vx[i] + vy[i]\*vy[i] + vz[i]\*vz[i]

ke\_sum += 0.5 \* vi2

xi = x[i]; yi = y[i]; zi = z[i]

dist\_sum += math.sqrt(xi\*xi + yi\*yi + zi\*zi)

return ke\_sum, dist\_sum / N

@njit(fastmath=True, parallel=True)

def \_numba\_soa\_step\_parallel(x, y, z, vx, vy, vz, dt):

N = x.shape[0]

ke\_sum = 0.0

dist\_sum = 0.0

# Parallel reduction; simple form for demonstration. In practice, use

# local accumulators to reduce contention, then combine.

ke\_local = 0.0

dist\_local = 0.0

for i in prange(N):

x[i] += vx[i] \* dt

y[i] += vy[i] \* dt

z[i] += vz[i] \* dt

vi2 = vx[i]\*vx[i] + vy[i]\*vy[i] + vz[i]\*vz[i]

ke\_local += 0.5 \* vi2

xi = x[i]; yi = y[i]; zi = z[i]

dist\_local += math.sqrt(xi\*xi + yi\*yi + zi\*zi)

ke\_sum += ke\_local

dist\_sum += dist\_local

return ke\_sum, dist\_sum / N

def numba\_soa\_run(N: int, steps: int, dt: float, use\_parallel: bool = False) -> Dict[str, float]:

x, y, z, vx, vy, vz = make\_soa(N)

ke = 0.0

md = 0.0

step\_fn = \_numba\_soa\_step\_parallel if use\_parallel else \_numba\_soa\_step

# Warmup to trigger JIT

ke, md = step\_fn(x, y, z, vx, vy, vz, dt)

for \_ in range(steps - 1):

ke, md = step\_fn(x, y, z, vx, vy, vz, dt)

return {"ke": ke, "mean\_dist": md}

# -----------------

# Benchmark harness

# -----------------

@dataclass

class BenchResult:

impl: str

N: int

steps: int

mean\_ms: float

std\_ms: float

last\_ke: float

last\_mean\_dist: float

def timeit\_ms(fn, trials: int) -> Tuple[float, float]:

"""Return mean and std dev in ms over given number of trials."""

times = []

for \_ in range(trials):

t0 = time.perf\_counter()

out = fn()

t1 = time.perf\_counter()

times.append((t1 - t0) \* 1000.0)

return float(np.mean(times)), float(np.std(times))

def run\_one\_size(N: int, steps: int, dt: float, trials: int, use\_parallel: bool) -> List[BenchResult]:

results: List[BenchResult] = []

# Baseline AoS

def run\_baseline():

return baseline\_aos\_run(N, steps, dt)

mean\_ms, std\_ms = timeit\_ms(run\_baseline, trials)

out = run\_baseline()

results.append(BenchResult("baseline\_aos\_python", N, steps, mean\_ms, std\_ms, out["ke"], out["mean\_dist"]))

# NumPy SoA

def run\_numpy():

return numpy\_soa\_run(N, steps, dt)

mean\_ms, std\_ms = timeit\_ms(run\_numpy, trials)

out = run\_numpy()

results.append(BenchResult("optimized\_soa\_numpy", N, steps, mean\_ms, std\_ms, out["ke"], out["mean\_dist"]))

# Numba SoA

def run\_numba():

return numba\_soa\_run(N, steps, dt, use\_parallel=use\_parallel)

# Ensure JIT warmup is not counted: numba\_soa\_run already warms within

# but to be safe, call once before timing

\_ = run\_numba()

mean\_ms, std\_ms = timeit\_ms(run\_numba, trials)

out = run\_numba()

tag = "optimized\_soa\_numba\_parallel" if use\_parallel else "optimized\_soa\_numba"

results.append(BenchResult(tag, N, steps, mean\_ms, std\_ms, out["ke"], out["mean\_dist"]))

return results

def plot\_speed(results\_df: pd.DataFrame, outdir: Path) -> None:

pivot = results\_df.pivot(index="N", columns="impl", values="mean\_ms")

plt.figure()

pivot.plot(marker="o")

plt.xlabel("N (particles)")

plt.ylabel("Mean time per run (ms)")

plt.title("AoS vs SoA Performance")

plt.grid(True)

plt.tight\_layout()

out = outdir / "speed\_vs\_N.png"

plt.savefig(out, dpi=140)

plt.close()

def plot\_speedup(results\_df: pd.DataFrame, outdir: Path) -> None:

# Compute speedup relative to baseline for each N and impl

base = results\_df[results\_df["impl"] == "baseline\_aos\_python"][["N", "mean\_ms"]]

base = base.rename(columns={"mean\_ms": "baseline\_ms"})

merged = results\_df.merge(base, on="N")

merged["speedup"] = merged["baseline\_ms"] / merged["mean\_ms"]

merged = merged[merged["impl"] != "baseline\_aos\_python"]

pivot = merged.pivot(index="N", columns="impl", values="speedup")

plt.figure()

pivot.plot(marker="o")

plt.xlabel("N (particles)")

plt.ylabel("Speedup vs baseline")

plt.title("Speedup of SoA approaches over AoS baseline")

plt.grid(True)

plt.tight\_layout()

out = outdir / "speedup\_vs\_N.png"

plt.savefig(out, dpi=140)

plt.close()

# -----------

# Main entry

# -----------

def parse\_args() -> Config:

p = argparse.ArgumentParser(description="AoS vs SoA optimization demo")

p.add\_argument("--sizes", nargs="+", type=int, default=[20000, 50000, 100000],

help="Problem sizes N to benchmark")

p.add\_argument("--steps", type=int, default=50, help="Time steps per run")

p.add\_argument("--trials", type=int, default=5, help="Trials per implementation")

p.add\_argument("--dt", type=float, default=1.0, help="Timestep delta")

p.add\_argument("--seed", type=int, default=42, help="Random seed")

p.add\_argument("--outdir", type=Path, default=Path("results"), help="Output directory")

p.add\_argument("--no-plots", action="store\_true", help="Disable chart generation")

p.add\_argument("--parallel", action="store\_true", help="Use Numba parallel kernel")

args = p.parse\_args()

return Config(

sizes=args.sizes,

steps=args.steps,

trials=args.trials,

dt=args.dt,

seed=args.seed,

outdir=args.outdir,

no\_plots=args.no\_plots,

use\_parallel=args.parallel,

)

def main() -> None:

cfg = parse\_args()

set\_seed(cfg.seed)

cfg.outdir.mkdir(parents=True, exist\_ok=True)

rows: List[Dict[str, float]] = []

print("Running benchmarks...\n")

print(f"sizes={cfg.sizes} steps={cfg.steps} trials={cfg.trials} dt={cfg.dt} parallel={cfg.use\_parallel}")

for N in cfg.sizes:

print(f"\nN={N}")

res = run\_one\_size(N, cfg.steps, cfg.dt, cfg.trials, cfg.use\_parallel)

for r in res:

print(f" impl={r.impl:26s} mean={r.mean\_ms:10.2f} ms std={r.std\_ms:7.2f} ms ke={r.last\_ke:.3e} md={r.last\_mean\_dist:.3f}")

rows.append({

"impl": r.impl,

"N": r.N,

"steps": r.steps,

"mean\_ms": r.mean\_ms,

"std\_ms": r.std\_ms,

"last\_ke": r.last\_ke,

"last\_mean\_dist": r.last\_mean\_dist,

})

df = pd.DataFrame(rows)

csv\_path = cfg.outdir / "results.csv"

df.to\_csv(csv\_path, index=False)

print(f"\nSaved CSV to {csv\_path}")

if not cfg.no\_plots:

plot\_speed(df, cfg.outdir)

plot\_speedup(df, cfg.outdir)

print(f"Saved plots to {cfg.outdir}")

# Save a small README with run configuration

with open(cfg.outdir / "README.txt", "w", encoding="utf-8") as f:

f.write("AoS vs SoA Optimization Demo\n")

f.write(f"sizes={cfg.sizes}\nsteps={cfg.steps}\ntrials={cfg.trials}\ndt={cfg.dt}\nseed={cfg.seed}\n")

f.write(f"parallel={cfg.use\_parallel}\n")

f.write("\nFiles produced:\n")

f.write("- results.csv\n")

if not cfg.no\_plots:

f.write("- speed\_vs\_N.png\n- speedup\_vs\_N.png\n")

if \_\_name\_\_ == "\_\_main\_\_":

main()

## 

## Execution Screenshots

The following figures show the program’s runtime outputs and generated performance plots.

A black screen with white text

AI-generated content may be incorrect.

**Figure 1.** Execution output of the benchmark script.

**A graph with a line and numbers

AI-generated content may be incorrect.**

**Figure 2.** Runtime performance of AoS vs SoA implementations.

**A graph with blue and orange lines

AI-generated content may be incorrect.**

**Figure 3.** Relative speedup of SoA implementations over AoS baseline.

## GitHub Repository Link

The complete project code and files are hosted at:  
https://github.com/mkchintha/Final-Project-Part-1-Optimization-Technique-and-Implementation-Project-Report.git

## Observations

* The baseline AoS implementation was several orders of magnitude slower than the optimized SoA versions.
* The Numba JIT-compiled version provided the greatest speedup, aligning with theoretical expectations.
* Screenshots and plots confirm correctness and performance improvements discussed in the main report.