**Final Project Part 1: Code and Screenshots**

Safal Poudel

Algorithms and Data Structures (MSCS-532-B01)

University of the Cumberlands

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

#!/usr/bin/env python3

"""

HPC Data-Structure Optimization Benchmarks

------------------------------------------

Focus: Data locality & layout (AoS -> SoA), strided vs contiguous access,

and cache-friendly blocking for matrix multiply.

This aligns with common fixes in the MSR'23 empirical study of HPC performance bugs

(e.g., data-locality optimizations, micro-architecture aware changes, and guiding compilers).

USAGE (examples):

python hpc\_locality.py

python hpc\_locality.py --n-dot 600000 --repeats 5

python hpc\_locality.py --no-plots

python hpc\_locality.py --mat-n 192 --mat-repeats 3

python hpc\_locality.py --force-py-mm # force pure-Python mm (can be slow)

python hpc\_locality.py --no-numba # ignore numba even if installed

Outputs:

- CSV: results\_hpc\_locality\_<timestamp>.csv

- PNGs: dot\_bench\_<timestamp>.png, stride\_bench\_<timestamp>.png, mm\_bench\_<timestamp>.png (if plotting enabled)

"""

import argparse

import csv

import math

import os

import platform

import sys

import time

from datetime import datetime

from typing import Callable, Dict, List, Tuple

# Optional deps

try:

import numpy as np

except Exception as e:

print("ERROR: This script requires NumPy. Please install with `pip install numpy`.", file=sys.stderr)

raise

try:

import psutil

except Exception:

psutil = None

# Optional plotting

try:

import matplotlib.pyplot as plt

HAVE\_MPL = True

except Exception:

HAVE\_MPL = False

# Optional numba

HAVE\_NUMBA = False

if "--no-numba" not in sys.argv:

try:

from numba import njit, prange

HAVE\_NUMBA = True

except Exception:

HAVE\_NUMBA = False

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

# Helpers

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

def secs\_fmt(x: float) -> str:

if x < 1e-3:

return f"{x\*1e6:.1f} µs"

if x < 1:

return f"{x\*1e3:.2f} ms"

return f"{x:.3f} s"

def time\_once(fn: Callable, \*args, \*\*kwargs) -> float:

t0 = time.perf\_counter()

fn(\*args, \*\*kwargs)

return time.perf\_counter() - t0

def time\_repeat(fn: Callable, repeats: int, warmup: int = 1, \*args, \*\*kwargs) -> Tuple[float, float, List[float]]:

# Warmup

for \_ in range(warmup):

fn(\*args, \*\*kwargs)

# Timed runs

samples = []

for \_ in range(repeats):

t = time.perf\_counter()

fn(\*args, \*\*kwargs)

samples.append(time.perf\_counter() - t)

mean = sum(samples) / len(samples)

# simple stdev

var = sum((s - mean)\*\*2 for s in samples) / (len(samples) - 1) if len(samples) > 1 else 0.0

stdev = math.sqrt(var)

return mean, stdev, samples

def system\_info() -> Dict[str, str]:

info = {

"python": sys.version.split()[0],

"numpy": np.\_\_version\_\_,

"platform": platform.platform(),

"processor": platform.processor(),

"cpu\_count\_logical": str(os.cpu\_count()),

}

if psutil:

try:

info["cpu\_count\_physical"] = str(psutil.cpu\_count(logical=False))

info["memory\_gb"] = f"{psutil.virtual\_memory().total / 1e9:.1f}"

except Exception:

pass

return info

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

# Bench 1: AoS vs SoA dot products

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

def gen\_dot\_data(n: int, seed: int = 42):

rng = np.random.default\_rng(seed)

A = rng.random((n, 3), dtype=np.float64)

B = rng.random((n, 3), dtype=np.float64)

# AoS: list of tuples; SoA: separate arrays

aos = [(float(A[i,0]), float(A[i,1]), float(A[i,2]),

float(B[i,0]), float(B[i,1]), float(B[i,2])) for i in range(n)]

return A, B, aos

def dot\_aos\_py(aos) -> float:

s = 0.0

for (ax, ay, az, bx, by, bz) in aos:

s += ax\*bx + ay\*by + az\*bz

return s

def dot\_soa\_numpy(A: np.ndarray, B: np.ndarray) -> float:

return float(np.sum(A \* B))

if HAVE\_NUMBA:

@njit(fastmath=True)

def dot\_aos\_numba(aos\_arr: np.ndarray) -> float:

s = 0.0

# aos\_arr shape: (n, 6)

for i in range(aos\_arr.shape[0]):

ax, ay, az, bx, by, bz = aos\_arr[i, 0], aos\_arr[i, 1], aos\_arr[i, 2], aos\_arr[i, 3], aos\_arr[i, 4], aos\_arr[i, 5]

s += ax\*bx + ay\*by + az\*bz

return s

@njit(fastmath=True, parallel=True)

def dot\_soa\_numba(A: np.ndarray, B: np.ndarray) -> float:

s = 0.0

for i in prange(A.shape[0]):

s += A[i,0]\*B[i,0] + A[i,1]\*B[i,1] + A[i,2]\*B[i,2]

return s

def run\_dot\_bench(n: int, repeats: int, use\_numba: bool) -> List[Dict]:

A, B, aos = gen\_dot\_data(n)

results = []

# Verify equivalence

ref = dot\_soa\_numpy(A, B)

# AoS Python

m, s, \_ = time\_repeat(dot\_aos\_py, repeats, 1, aos)

results.append({"bench": "Dot AoS (Python loop)", "N": n, "mean\_s": m, "stdev\_s": s})

# SoA NumPy

m, s, \_ = time\_repeat(dot\_soa\_numpy, repeats, 1, A, B)

results.append({"bench": "Dot SoA (NumPy vectorized)", "N": n, "mean\_s": m, "stdev\_s": s})

if use\_numba and HAVE\_NUMBA:

# Prepare contiguous AoS array for Numba

aos\_arr = np.asarray(aos, dtype=np.float64).reshape(-1, 6)

# compile

dot\_aos\_numba(aos\_arr)

dot\_soa\_numba(A, B)

# timed

m, s, \_ = time\_repeat(dot\_aos\_numba, repeats, 1, aos\_arr)

results.append({"bench": "Dot AoS (Numba JIT)", "N": n, "mean\_s": m, "stdev\_s": s})

m, s, \_ = time\_repeat(dot\_soa\_numba, repeats, 1, A, B)

results.append({"bench": "Dot SoA (Numba JIT, parallel)", "N": n, "mean\_s": m, "stdev\_s": s})

# quick sanity check

assert abs(ref - dot\_soa\_numpy(A, B)) < 1e-6, "Ref mismatch"

return results

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

# Bench 2: Strided vs Contiguous reductions

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

def gen\_stride\_data(n: int, seed: int = 123):

rng = np.random.default\_rng(seed)

X = rng.random(n, dtype=np.float64)

return X

def sum\_strided(X: np.ndarray, stride: int) -> float:

return float(np.sum(X[::stride]))

def sum\_contiguous\_copy(X: np.ndarray, stride: int) -> float:

# materialize to contiguous buffer, then reduce

view = X[::stride]

contig = np.ascontiguousarray(view)

return float(np.sum(contig))

def sum\_chunked(X: np.ndarray, chunk: int) -> float:

# reduce in cache-friendly chunks

s = 0.0

for i in range(0, X.size, chunk):

s += float(np.sum(X[i:i+chunk]))

return s

def run\_stride\_bench(n: int, stride: int, chunk: int, repeats: int) -> List[Dict]:

X = gen\_stride\_data(n)

results = []

# Strided sum

m, s, \_ = time\_repeat(sum\_strided, repeats, 1, X, stride)

results.append({"bench": f"Strided sum (stride={stride})", "N": n, "mean\_s": m, "stdev\_s": s})

# Contiguous copy then sum

m, s, \_ = time\_repeat(sum\_contiguous\_copy, repeats, 1, X, stride)

results.append({"bench": f"Contiguous copy+sum (stride={stride})", "N": n, "mean\_s": m, "stdev\_s": s})

# Chunked sum (no stride) to show simple blocking

m, s, \_ = time\_repeat(sum\_chunked, repeats, 1, X, chunk)

results.append({"bench": f"Chunked sum (chunk={chunk})", "N": n, "mean\_s": m, "stdev\_s": s})

# Full contiguous baseline

m, s, \_ = time\_repeat(np.sum, repeats, 1, X)

results.append({"bench": "Contiguous full sum", "N": n, "mean\_s": float(m), "stdev\_s": float(s)})

return results

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

# Bench 3: Matrix multiply (naive vs blocked vs NumPy)

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

def gen\_mm\_data(n: int, seed: int = 99):

rng = np.random.default\_rng(seed)

A = rng.random((n, n), dtype=np.float64)

B = rng.random((n, n), dtype=np.float64)

return A, B

def mm\_numpy(A: np.ndarray, B: np.ndarray) -> np.ndarray:

return A @ B

def mm\_naive\_py(A: np.ndarray, B: np.ndarray) -> np.ndarray:

# WARNING: very slow in pure Python for large N

n = A.shape[0]

C = np.zeros((n, n), dtype=np.float64)

for i in range(n):

for k in range(n):

aik = A[i, k]

for j in range(n):

C[i, j] += aik \* B[k, j]

return C

def mm\_blocked\_py(A: np.ndarray, B: np.ndarray, bs: int = 32) -> np.ndarray:

# Still Python loops; uses blocking to improve locality

n = A.shape[0]

C = np.zeros((n, n), dtype=np.float64)

for ii in range(0, n, bs):

for kk in range(0, n, bs):

for jj in range(0, n, bs):

i\_max = min(ii+bs, n)

k\_max = min(kk+bs, n)

j\_max = min(jj+bs, n)

for i in range(ii, i\_max):

for k in range(kk, k\_max):

aik = A[i, k]

for j in range(jj, j\_max):

C[i, j] += aik \* B[k, j]

return C

if HAVE\_NUMBA:

@njit(fastmath=True)

def mm\_naive\_numba(A: np.ndarray, B: np.ndarray) -> np.ndarray:

n = A.shape[0]

C = np.zeros((n, n), dtype=np.float64)

for i in range(n):

for k in range(n):

aik = A[i, k]

for j in range(n):

C[i, j] += aik \* B[k, j]

return C

@njit(fastmath=True, parallel=True)

def mm\_blocked\_numba(A: np.ndarray, B: np.ndarray, bs: int) -> np.ndarray:

n = A.shape[0]

C = np.zeros((n, n), dtype=np.float64)

for ii in prange(0, n, bs):

for kk in range(0, n, bs):

for jj in range(0, n, bs):

i\_max = min(ii+bs, n)

k\_max = min(kk+bs, n)

j\_max = min(jj+bs, n)

for i in range(ii, i\_max):

for k in range(kk, k\_max):

aik = A[i, k]

for j in range(jj, j\_max):

C[i, j] += aik \* B[k, j]

return C

def run\_mm\_bench(n: int, repeats: int, block: int, force\_py: bool, use\_numba: bool) -> List[Dict]:

A, B = gen\_mm\_data(n)

results = []

# NumPy BLAS baseline

ref = mm\_numpy(A, B)

m, s, \_ = time\_repeat(mm\_numpy, repeats, 1, A, B)

results.append({"bench": "MatMul NumPy (BLAS)", "N": n, "mean\_s": m, "stdev\_s": s})

# Pure Python loops are extremely slow; only run if forced or if numba absent and n is small

if force\_py:

# naive

m, s, \_ = time\_repeat(mm\_naive\_py, repeats, 0, A, B)

C1 = mm\_naive\_py(A, B)

assert np.allclose(C1, ref, atol=1e-8)

results.append({"bench": "MatMul naive (Python loops)", "N": n, "mean\_s": m, "stdev\_s": s})

# blocked

m, s, \_ = time\_repeat(mm\_blocked\_py, repeats, 0, A, B, block)

C2 = mm\_blocked\_py(A, B, block)

assert np.allclose(C2, ref, atol=1e-8)

results.append({"bench": f"MatMul blocked (Python loops, bs={block})", "N": n, "mean\_s": m, "stdev\_s": s})

if use\_numba and HAVE\_NUMBA:

# compile first

mm\_naive\_numba(A, B)

mm\_blocked\_numba(A, B, block)

# naive numba

m, s, \_ = time\_repeat(mm\_naive\_numba, repeats, 0, A, B)

C3 = mm\_naive\_numba(A, B)

assert np.allclose(C3, ref, atol=1e-8)

results.append({"bench": "MatMul naive (Numba JIT)", "N": n, "mean\_s": m, "stdev\_s": s})

# blocked numba

m, s, \_ = time\_repeat(mm\_blocked\_numba, repeats, 0, A, B, block)

C4 = mm\_blocked\_numba(A, B, block)

assert np.allclose(C4, ref, atol=1e-8)

results.append({"bench": f"MatMul blocked (Numba JIT, bs={block})", "N": n, "mean\_s": m, "stdev\_s": s})

return results

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

# Plotting

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

def save\_barplot(rows: List[Dict], title: str, fname: str):

if not HAVE\_MPL:

return

labels = [r["bench"] for r in rows]

vals = [r["mean\_s"] for r in rows]

plt.figure(figsize=(10, 5))

plt.bar(labels, vals)

plt.ylabel("Seconds (mean)")

plt.title(title)

plt.xticks(rotation=20, ha="right")

plt.tight\_layout()

plt.savefig(fname, dpi=200)

plt.close()

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

# Main

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

def main():

parser = argparse.ArgumentParser(description="HPC data locality/layout benchmarks")

parser.add\_argument("--n-dot", type=int, default=300\_000, help="N for dot-product AoS vs SoA")

parser.add\_argument("--n-stride", type=int, default=4\_000\_000, help="N for strided/contiguous reductions")

parser.add\_argument("--stride", type=int, default=8, help="Stride for strided reduction")

parser.add\_argument("--chunk", type=int, default=131072, help="Chunk size for chunked reduction")

parser.add\_argument("--mat-n", type=int, default=192, help="Matrix size N (NxN) for mm benchmark")

parser.add\_argument("--block", type=int, default=32, help="Block size for blocked matmul")

parser.add\_argument("--repeats", type=int, default=3, help="Repeats per benchmark")

parser.add\_argument("--mat-repeats", type=int, default=1, help="Repeats per matrix multiply variant")

parser.add\_argument("--no-plots", action="store\_true", help="Disable PNG plots")

parser.add\_argument("--no-numba", action="store\_true", help="Ignore Numba even if installed")

parser.add\_argument("--force-py-mm", action="store\_true", help="Force pure-Python matmul loops (slow)")

args = parser.parse\_args()

timestamp = datetime.now().strftime("%Y%m%d\_%H%M%S")

csv\_name = f"results\_hpc\_locality\_{timestamp}.csv"

out\_rows: List[Dict] = []

# Header: system info

info = system\_info()

print("\n=== System Info ===")

for k, v in info.items():

print(f"{k:>20}: {v}")

print("\n=== Bench 1: Dot Products (AoS vs SoA) ===")

dot\_rows = run\_dot\_bench(args.n\_dot, args.repeats, use\_numba=(not args.no\_numba))

for r in dot\_rows:

print(f"{r['bench']:<36} N={r['N']:<8} mean={secs\_fmt(r['mean\_s'])} stdev={secs\_fmt(r['stdev\_s'])}")

out\_rows.extend([{"group":"dot", \*\*r} for r in dot\_rows])

print("\n=== Bench 2: Strided vs Contiguous ===")

stride\_rows = run\_stride\_bench(args.n\_stride, args.stride, args.chunk, args.repeats)

for r in stride\_rows:

print(f"{r['bench']:<36} N={r['N']:<8} mean={secs\_fmt(r['mean\_s'])} stdev={secs\_fmt(r['stdev\_s'])}")

out\_rows.extend([{"group":"stride", \*\*r} for r in stride\_rows])

print("\n=== Bench 3: Matrix Multiply (naive/blocked/NumPy) ===")

mm\_rows = run\_mm\_bench(args.mat\_n, args.mat\_repeats, args.block, args.force\_py\_mm, use\_numba=(not args.no\_numba))

for r in mm\_rows:

print(f"{r['bench']:<36} N={r['N']:<8} mean={secs\_fmt(r['mean\_s'])} stdev={secs\_fmt(r['stdev\_s'])}")

out\_rows.extend([{"group":"mm", \*\*r} for r in mm\_rows])

# Save CSV

with open(csv\_name, "w", newline="") as f:

w = csv.writer(f)

w.writerow(["group", "bench", "N", "mean\_s", "stdev\_s"])

for r in out\_rows:

w.writerow([r.get("group",""), r["bench"], r["N"], f"{r['mean\_s']:.9f}", f"{r['stdev\_s']:.9f}"])

print(f"\nSaved CSV: {csv\_name}")

# Save plots

if not args.no\_plots and HAVE\_MPL:

save\_barplot(dot\_rows, f"Dot Products AoS vs SoA (N={args.n\_dot})", f"dot\_bench\_{timestamp}.png")

save\_barplot(stride\_rows, f"Strided vs Contiguous (N={args.n\_stride}, stride={args.stride})", f"stride\_bench\_{timestamp}.png")

if mm\_rows:

save\_barplot(mm\_rows, f"Matrix Multiply (N={args.mat\_n})", f"mm\_bench\_{timestamp}.png")

print("Saved plots (PNG).")

elif args.no\_plots:

print("Plotting disabled (--no-plots).")

else:

print("matplotlib not available; skipping plots.")

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

main()

# **Results**

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

A blue and white bar graph

AI-generated content may be incorrect.

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

A graph of blue rectangular bars

AI-generated content may be incorrect.

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

A blue rectangular object with white text

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Fig 4: Console output