Lab: Accelerating Code: From Python Baseline to Architecture-Aware Optimization

Objective

You (in teams of 2) start from a given **Python baseline** of a real-world numerical algorithm and iteratively accelerate it using **architecture-level design optimizations** — up to **AVX-512 vectorization** and multicore exploitation. The idea is to study the performance impact and understand Measurement and reproducibility of real performance (GFLOPs, speedup, scaling) using:

- **Multicore parallelism** (threads, task scheduling, synchronization)
- **Cache-friendly data layouts** and blocking
- **SIMD / VLIW-style vectorization (SSE, AVX2, AVX-512)**
- **Memory hierarchy & prefetching**
- **NUMA-aware memory allocation**
- **Branch prediction & instruction-level parallelism**
- **Compiler and build optimizations (-O3, -march, -funroll-loops)**

[Leaderboard will rank submissions by *speedup over baseline*.]

We will do two challenge problems --- one from Linear Algebra [Dense Matrix-Matrix Multiplication (GEMM)] and another from Computational Biology/Smith–Waterman string alignment OR Optimization / ML Gradient Descent on synthetic dataset. We will give a base line implementation for both challenges.

```python
# baseline_gemm.py
import numpy as np
from multiprocessing import Pool

def multiply_block(args):
    A_block, B_block = args
    return np.dot(A_block, B_block)

if __name__ == "__main__":
    N = 1024
    A = np.random.rand(N, N)
    B = np.random.rand(N, N)
    with Pool(4) as p:
        results = p.map(multiply_block, [(A[i::4], B) for i in range(4)])
    C = np.vstack(results)
```

You can *reimplement* in **C / C++ /optimized Python** — but must maintain the same functionality and correctness. All optimizations must be documented and reproducible.

You may do one or many of the following:

You may **NOT**

- Call pre-optimized BLAS routines directly (like *cblas_dgemm*) — unless they implement similar optimizations manually or via library introspection.
- Change the algorithm's semantics (must compute same result)

Measurement

I have given you the directory structure with the associated scripts. Please use that to create your submission tarball which we will autograde.

```
kolin@mosaic:~/col7418/labFinal$ tree
.
├── baseline
│   └── gemm_baseline.py
├── leaderboard
├── Makefile
├── optimized
│   ├── cpp
│   │   └── gemm_opt.cpp
│   └── gemm_opt
├── README.md
├── report_template.md
├── run.sh
└── submit.sh

3 directories, 9 files
```

You will measure wall-clock time for both baseline and optimized code. You will report Runtime (sec), GFLOPs or GB/s throughput, Speedup factor, Core count, Instruction vector width (SSE, AVX2, AVX-512) and L1/L2/L3 miss rates with appropriate graphs/visualization. You can use perf for to get performance counter stats.

```
Serial sum = 100000000, time = 0.244 sec
Parallel sum = 100000000, time = 0.017 sec
Number of logical CPUs available: 32
Speedup = 14.75x

 Performance counter stats for './a.out':

          1,393.94 msec task-clock                       #    1.887 CPUs utilized
                59      context-switches                 #   42.326 /sec
                 1      cpu-migrations                   #    0.717 /sec
           1,95,451      page-faults                      #  140.215 K/sec
   4,25,73,83,614      cycles                           #    3.054 GHz
   4,48,34,89,980      instructions                     #    1.05  insn per cycle
     51,04,34,834      branches                         #  366.183 M/sec
          1,47,317      branch-misses                    #    0.03% of all branches

       0.738770142 seconds time elapsed

       1.159144000 seconds user
       0.234859000 seconds sys
```

Grading

The assignment will be graded on three counts:

Metric	Description	Weight
Speedup	Baseline time / Optimized time	60%
Scalability	Speedup vs #cores (1–N scaling curve)	20%
Technical Depth	Demonstrated architecture-level reasoning in report	20%

Submission

Code repository

- baseline/ and optimized/ directories
- run.sh to build and test reproducibly
- Documentation of compiler flags, cores used, and assumptions

Report (4–6 pages)

- Description of optimizations and reasoning (with diagrams of data blocking)
- Profiling evidence (cache miss reductions, IPC improvements)
- Performance vs. baseline plots
- Reflection: what architectural factor dominated performance?

To standardize things, I have give you a script to create the tarball. You will run ./submit.sh entryNumbers inside the directory to create the tarball.

```
kolin@mosaic:~/col7418/labFinal$ ./submit.sh mcs259999mcs258888
Created submission package: submit_mcs259999mcs258888_20251019_070704.tar.gz
Please upload submit_mcs259999mcs258888_20251019_070704.tar.gz in Moodle
```

Problem 2:

The **Smith–Waterman algorithm** performs **local sequence alignment** — it finds the most similar region between two biological sequences (e.g., DNA, RNA, or proteins).

It uses **dynamic programming** to fill a scoring matrix, where each cell represents the best alignment score up to that point.

Recurrence relation:

$$H(i,j) = \max \begin{cases} 0, \\ H(i-1, j-1) + s(a_i, b_j), \\ H(i-1, j) + \text{gap}, \\ H(i, j-1) + \text{gap} \end{cases}$$

Reading if necessary:

https://en.wikipedia.org/wiki/Smith%E2%80%93Waterman_algorithm

https://www.geeksforgeeks.org/dsa/sequence-alignment-problem/

Implement, optimize, and benchmark the **Smith–Waterman (local) alignment** algorithm for synthetic DNA-like sequences. Start from a **pure Python baseline** (provided) and progressively optimize through **blocking, vectorization (SSE/AVX/AVX2/AVX-512)**, and **multi-threading**.

You will measure speedups, memory usage, and correctness relative to the baseline.

Scoring Scheme

Event	Score
Match	+2
Mismatch	−1
Gap	−2

DP recurrence:

$$H[i][j] = \max(0, H[i-1][j-1] + s(a_i, b_j), H[i-1][j] + \text{gap}, H[i][j-1] + \text{gap})$$

All other steps remain same as for gemm.