Project #1: SIMD Advantage Profiling

Modern CPUs provide **SIMD** (Single Instruction, Multiple Data) vector units (e.g., SSE/AVX/AVX-512, NEON) that can process multiple data elements per instruction. This class-wide project quantifies the **real-world speedup** SIMD brings to simple numeric kernels and explains **when** and **why** vectorization helps (or doesn't), using principled measurement and analysis.

Learning Goals (What your experiments must reveal)

- **Baseline vs. SIMD speedup** for common kernels under realistic conditions.
- When kernels are **compute-bound** vs **memory-bound**, and how that limits SIMD gains.
- Effects of **data type**, **alignment**, **stride/tail handling**, and **working-set size** on performance.
- How to **verify vectorization** (compiler reports / disassembly) and relate results to a **roofline** model using measured memory bandwidth.

Tools You'll Use

- A modern **C/C++ compiler** (GCC or Clang) with optimization enabled (e.g., high-level auto-vectorization). Record exact versions and flags used (e.g., optimization level, target ISA, fast-math options, FTZ/DAZ settings).
- **Disassembly or compiler vectorization reports** to confirm SIMD (e.g., opt reports; check for vector instructions in the binary).
- **System timers and statistical scripts** to measure runtime, compute **GFLOP/s** and **cycles per element (CPE)**, and generate plots.
- (Recommended) OS performance counters (e.g., perf) to report instructions retired, vector instruction count, and memory traffic.

Scope: Keep experiments **single-threaded**. The goal is to isolate SIMD effects, not multithreading.

Kernel Set (choose at least three)

1. **SAXPY / AXPY**: $y \leftarrow a\,x + y$ (streaming FMA).
2. **Dot product / reduction**: $s \leftarrow \Sigma\, x_i\, y_i$ (reduction).
3. **Elementwise multiply**: $z_i \leftarrow x_i \cdot y_i$ (no reduction).
4. **1D 3-point stencil**: $y_i \leftarrow a\,x_{i-1} + b\,x_i + c\,x_{i+1}$ (neighbor access).

Experimental Knobs (orthogonal axes)

1. **Data type:** float32 vs float64 (and optionally int32).
2. **Alignment & tail:** aligned arrays vs deliberately misaligned; sizes that are multiples of the vector width vs sizes with remainders (tail handling).
3. **Stride / access pattern:** unit-stride (contiguous) vs strided (e.g., 2, 4, 8) or gather-like index patterns where applicable.
4. **Working-set size:** within **L1**, within **L2**, within **LLC**, and **DRAM-resident** (increase N across these regimes).
5. **Compiler / ISA flags:** scalar-only (vectorization disabled), default auto-vectorized, and a build targeted to your CPU's widest available vectors. Record any fast-math, FMA, FTZ/DAZ settings.

Required Experiments & Plots

1. **Baseline (scalar) vs auto-vectorized**
   Build a scalar-only baseline and an auto-vectorized version for each selected kernel. Measure runtime across sizes spanning L1→L2→LLC→DRAM. Report **speedup = scalar_time / simd_time** and achieved **GFLOP/s**.

2. **Locality (working-set) sweep**
   For one kernel, sweep N to cross cache levels. From the same runs, produce **GFLOP/s** (or GiB/s for purely streaming) and **CPE**; annotate cache transitions. Discuss where SIMD gains compress as the kernel becomes memory-bound.
3. **Alignment & tail handling**
   Compare aligned vs misaligned inputs and sizes with/without a vector tail. Quantify the throughput gap and explain (prologue/epilogue cost, unaligned loads, masking).
4. **Stride / gather effects**
   Evaluate unit-stride vs strided/gather-like patterns (where meaningful). Show the impact on effective bandwidth and SIMD efficiency; explain prefetcher and cache-line utilization effects.
5. **Data type comparison**
   Compare float32 vs float64 (and optionally int32). Report how vector width (lanes) and arithmetic intensity affect speedup and GFLOP/s.
6. **Vectorization verification**
   Provide evidence that SIMD occurred (compiler vectorization report and/or disassembly snippets identifying vector ops). Summarize—not full dumps.
7. **Roofline interpretation**
   For at least one kernel, compute arithmetic intensity (FLOPs per byte moved) and place your achieved GFLOP/s on a **roofline** using your **measured memory bandwidth** (from Project #2, if available) and an estimate of your CPU's peak FLOP rate. Explain whether you're **compute-bound** or **memory-bound** and how this predicts the observed SIMD speedup.

Reporting & Deliverables (commit everything **to GitHub)**

- **Source code and build scripts** (scalar & SIMD builds), run scripts, raw measurements, and plotting code (re-runnable).
- **Setup/methodology:** CPU model & ISA support, compiler & flags, OS, frequency policy (governor), SMT state, measurement method (timer, repetitions), and data initialization (to avoid trivial zeros/denormals).
- **Clearly labeled plots/tables** with units and error bars (≥3 runs when feasible). Speedup plots should show median with variability.
- **Analysis** tying results to cache locality, alignment, vector width, arithmetic intensity, and roofline predictions.
- **Limitations/anomalies** with hypotheses (e.g., denormals, thermal effects, frequency scaling, page faults, TLB behavior).

Grading Rubric (Total 110 pts)

1. **Baseline & correctness (10)**
   - (5) Scalar baseline established; results validated against a reference (relative error tolerance documented).
   - (5) Reproducible timing methodology with repetitions and error bars.
2. **Vectorization verification (15)**
   - (10) Clear evidence of SIMD via compiler reports or targeted disassembly.
   - (5) Correct interpretation (e.g., vector width, FMA usage).
3. **Locality sweep (15)**
   - (10) L1→L2→LLC→DRAM transitions identified and annotated.
   - (5) Discussion of where SIMD gains compress due to memory-bound behavior.
4. **Alignment & tail study (10)**
   - (10) Quantified impact of misalignment and tail handling with explanation.
5. **Stride / gather effects (10)**

    - (10) Impact of non-unit stride or gather-like access on SIMD efficiency analyzed.
6. **Data type comparison (10)**
    - (10) float32 vs float64 (and/or int32) results with lane-width reasoning.
7. **Speedup & throughput plots (10)**
    - (10) Clear scalar vs SIMD plots (speedup, GFLOP/s or GiB/s) with proper units/legends.
8. **Roofline analysis (20)**
    - (10) Correct arithmetic intensity and placement on a roofline using measured bandwidth and peak FLOPs.
    - (10) Sound conclusions on compute- vs memory-bound and expected SIMD gains.
9. **Reporting quality (10)**
    - (10) Clean repo structure, thorough methodology, labeled plots, and clear discussion of anomalies/limits.

Tips for Successful Execution

- **Fix CPU frequency** (performance governor) and **pin to a core** to reduce run-to-run variance; document SMT state.
- **Warm up** data to populate caches for "hot" runs; also test **cold** runs where relevant and state which you report.
- **Avoid denormals** and constant-zero paths; initialize with non-trivial data. Consider FTZ/DAZ if supported (and document).
- **Check vector width** available on your machine (e.g., 128-/256-/512-bit) and target that ISA with your compiler flags.
- **Verify alignment** of arrays; test deliberately misaligned variants to see the cost.
- **Measure more than time**: compute GFLOP/s, CPE, and memory traffic estimates; use multiple repetitions and report variability.
- **Keep it single-threaded** unless you explicitly evaluate threading; otherwise results conflate SIMD with parallelism.
- **Record everything**: compiler versions/flags, environment, thermal state; randomize run order to mitigate drift.