Project A1: Advanced OS and CPU Feature Exploration

Modern processors and operating systems expose a rich set of mechanisms that influence performance, isolation, and scalability. In this project, you will design and implement your own micro-benchmarks to investigate selected OS/CPU features on your system (laptop, desktop, or cloud). Rather than following a fixed procedure, you will formulate questions, collect measurements, and interpret how modern hardware-software interfaces behave.

Learning Goals Your project should demonstrate your ability to:

- Identify and describe advanced OS or CPU features that impact performance
- Design controlled experiments to reveal their behavior and trade-offs
- Measure and interpret results using system tools
- Communicate your findings clearly and reproducibly

Feature Menu (choose any 4, get permission from TA if you would like to work on other features)

- I/O Data Path: zero-copy I/O; asynchronous I/O
- Scheduling & Isolation: CPU affinity; scheduler classes; control groups
- Memory Management: transparent huge pages or hugetlbfs; NUMA locality and remote memory latency
- Micro-architectural Behavior: SMT (Simultaneous Multithreading) interference; cache prefetcher effects
- DDIO (Data Direct I/O) on Xeon cloud instances; observe cache contention between I/O and compute

Notes

- Use whatever features your system supports; skip those unavailable
- Control environment variability: fix CPU frequency, pin threads, disable background tasks
- Use perf stat counters for cache/TLB behavior and CPU cycles
- Document all versions (compiler, kernel, CPU model)
- Each experiment should produce plots or quantitative tables

Grading

Category	Weight	Description
Technical Depth	25 pts	Ambitious and correct use of selected mechanisms; meaningful scope
Experimental Rigor	25 pts	Reproducible, well-controlled measurements with clear methodology
Analysis & Insight	25 pts	Logical interpretation of results; links to OS/hardware principles
Clarity & Presentation	25 pts	Organized code, readable plots, concise and clear writing

Project A2: Dense vs Sparse Matrix Multiplication (SIMD + Multithreading)

Matrix multiplication is a cornerstone of scientific computing and ML. **SIMD** and **multithreading** can accelerate dense GEMM dramatically, but when matrices are **sparse**, compressed formats like **CSR/CSC** (row/column compression) often win, depending on sparsity structure and size. This project quantifies the cross-over points and explains **when to use compressed formats**.

Learning Goals (What your experiments must reveal)

- Implement and compare **dense GEMM** vs **sparse × dense** multiplication using **CSR** (row-compressed) and **CSC** (column-compressed) formats.
- Demonstrate **SIMD** and **multithread** speedups on both dense and sparse kernels.
- Identify the **break-even density/structure** where compressed formats outperform dense.
- Understand **memory-bandwidth limits**, **arithmetic intensity**, and cache behavior using a **roofline** perspective.
- Discuss the **cost of format conversion** (dense→CSR/CSC) and when it is amortized.

Scope & Variants

- Required: **SpMM** (Sparse A × Dense B → Dense C) with **CSR** (and/or CSC) for A; **GEMM** baseline (Dense A × Dense B → Dense C).
- Optional (bonus): **SpGEMM** (Sparse × Sparse → Sparse).

Tools You'll Use

- **C/C++** compiler (GCC/Clang), with optimization and target ISA flags.
- **SIMD**: auto-vectorization and/or intrinsics (e.g., AVX2/AVX-512, NEON). Record flags and ISA used.
- **Multithreading**: OpenMP or std::thread; pin threads as needed.
- **Perf counters** (e.g., "perf") for instructions, cache misses, memory bandwidth; timers for runtime.
- **(Reference only)** BLAS library (OpenBLAS/BLIS/oneMKL) to sanity-check correctness/perf of dense GEMM.

Note: The implementation work must be **your own** for CSR/CSC kernels; BLAS is for reference/validation

Experimental Knobs (orthogonal axes)

1. **Matrix sizes (m, k, n):** include square (e.g., 1024/2048) and rectangular (tall-skinny, fat) cases.
2. **Sparsity level:** sweep from **0.1% → 50%**; include midpoints (0.5%, 1%, 2%, 5%, 10%, 20%).
3. **Thread count:** 1, 2, 4, 8, … up to physical cores.
4. **Data layout:** A in CSR/CSC; B and C in row-major vs column-major (measure both where helpful).
5. **Blocking/tiling:** cache tiling for dense GEMM; tile width for SpMM inner loops (columns of B processed per chunk).

Required Experiments & Plots

1. **Correctness & baselines**: Validate dense GEMM (vs BLAS) and CSR-SpMM results (relative error tolerance). Report scalar vs SIMD single-thread baselines.
2. **SIMD & threading speedup**: For dense GEMM and CSR-SpMM (at representative sizes), show speedup when using SIMD only, multi-threading only, and SIMD + multi-threading. Report **GFLOP/s** and **cycles per nonzero (CPNZ)** for SpMM.

3. **Density break-even (uniform random)**: Fix (m,k,n); sweep density p. Plot **runtime** (or **GFLOP/s**) for **dense GEMM** vs **CSR-SpMM** on the same axes. Mark the **break-even density** where sparse becomes faster; explain using arithmetic intensity and bandwidth.
4. **Working-set transitions**: Sweep sizes to cross **L1→L2→LLC→DRAM**. From the same runs, show **GFLOP/s** and measured memory bandwidth (from perf or a streaming microbench). Discuss where each kernel becomes memory-bound.
5. **Roofline interpretation**: Compute **arithmetic intensity** for dense GEMM and for CSR-SpMM. Place your achieved GFLOP/s on a roofline using measured memory bandwidth and estimated peak FLOPs. Conclude **compute- vs memory-bound** regimes.

Reporting & Deliverables (commit everything to GitHub)

- **Source code** (dense GEMM, CSR-SpMM, CSC-SpMM if implemented), build scripts, run scripts, and plotting code.
- **Setup/methodology:** CPU µarch, ISA, cores/threads, memory, compiler & flags, OS, governor/SMT, thread pinning, dataset generation (seeds), and BLAS version for references.
- **Plots/tables** with units and error bars (≥3 runs); axes clearly indicate fixed vs varied knobs; include p95 latency where relevant.
- **Analysis** tied to arithmetic intensity, cache behavior, thread scaling, SIMD width, and roofline predictions.
- **Limitations/anomalies** with hypotheses.

Grading Rubric (Total 210 pts)

Implementations (80)

- (20) Correct, performant **CSR-SpMM** (and/or CSC) with SIMD-aware inner loop
- (20) Dense GEMM with cache-aware tiling and SIMD
- (20) Robust multithreading
- (20) Correctness tests and tolerances documented

Experimental design/plotting (40)

- (20) Comprehensive sweeps over density, size, and structure
- (20) Clear separation of scalar vs SIMD vs threading effects

Break-even analysis (20)

- (20) Precise identification of break-even density; explanations grounded in arithmetic intensity and bandwidth

Roofline & counters (30)

- (15) Correct AI estimates; roofline placement using measured bandwidth & peak FLOPs
- (15) Use of perf counters (cache/TLB/mem BW) to support conclusions

Reporting quality (40)

- (20) Reproducible repo (scripts, configs, seeds), labeled plots with error bars
- (20) Clear discussion of anomalies/limits and practical recommendations

Project A3 — Approximate Membership Filters (XOR vs Cuckoo vs Quotient)

Approximate membership structures answer "is x probably in the set?" far faster and smaller than exact indexes. Modern designs—**XOR filter**, **Cuckoo filter**, and **Quotient filter**—extend classic Bloom filters with better space/speed trade-offs and, in some cases, **deletions** and more cache-friendly access. This project implements all three, builds a solid baseline (**blocked Bloom**), and maps when each design wins.

Learning Goals (What your experiments must reveal)

- Implement and validate **XOR**, **Cuckoo**, and **Quotient** filters (and a **blocked Bloom** baseline).
- Quantify **bits per entry (BPE)** vs **false positive rate (FPR)** and **lookup/insert/delete throughput**.
- Characterize **tail latency (p95/p99)** and **negative-lookup heavy** workloads.
- Understand **dynamic vs static** trade-offs (XOR is static; Cuckoo/Quotient support online inserts/deletes).
- Tie observations to **cache/TLB/branch** behavior and **load factor** effects.

Tools You Use

- **C/C++** (GCC/Clang) with `-O3 -march=native` (or NEON on ARM).
- A fast non-cryptographic hash (e.g., **xxHash**/Murmur3) with distinct seeds.
- `perf stat` (or similar) for uarch counters; timing harness & plotting scripts.

Scope: In-memory filters. Implement the core data structures **yourself** (no third-party libraries). SIMD is encouraged for bucket probes and small scans.

Scope & Requirements

- **XOR filter (static):** Build-time construction over a fixed key set; three hash positions; compact fingerprint array (8–16 bits). Must support **build** + **query**.
- **Cuckoo filter (dynamic):** Bucketized table (e.g., 2 choices, bucket size k=4), **fingerprints** per slot, **insert**, **query**, **delete**, bounded evictions, optional small **stash**. Measure behavior near high load factors.
- **Quotient filter (dynamic):** Single array with quotient/remainder; metadata bits (occupied, continuation, shifted). Support **insert**, **query**, **delete**. Emphasize contiguous, cache-friendly scans.
- **Blocked Bloom baseline:** Cache-line-blocked bitset; Supports **insert** + **query** (no deletes).

All structures use **64-bit keys** and a common API for apples-to-apples comparisons.

Experimental Knobs (orthogonal axes)

- **Set size (n):** 1M, 5M, 10M (scale up if RAM allows).
- **Target FPRs:** 5%, 1%, 0.1% (choose fingerprint sizes / table sizes accordingly).
- **Workloads:**
  - Read-mostly: 95% queries / 5% inserts (for dynamic filters).
  - Balanced: 50/50.
  - Read-only: 100% queries (all).
  - Negative-lookup share: 0%, 50%, 90%.
- **Load factor (dynamic only):** Sweep 0.4 → 0.95 (step 0.05).
- **Fingerprint width:** 8, 12, 16 bits (Cuckoo/XOR).

- **Threads:** 1, 2, 4, 8, … to physical cores (pinned)

Required Experiments & Plots

1. **Space vs accuracy:**
   - For each structure at the three target FPRs (false positive rate), measure **bits per entry** (including metadata) and achieved FPR on an independent negative set.
   - Show theory lines for Bloom (optional) to contextualize.

2. **Lookup throughput & tails:**
   - Queries/second vs negative-lookup share (0→90%).
   - Report **p50/p95/p99** latency for each structure and FPR.

3. **Insert/delete throughput (dynamic):**
   - Cuckoo & Quotient: ops/s across load factors; record **insertion failure rate** (cuckoo) and average probe length.
   - Cuckoo: bounded evictions, stash hits; Quotient: cluster length histograms.

4. **Thread scaling:**
   - Throughput vs threads for read-mostly and balanced mixes; document concurrency control (per-bucket locks, striped locks, or lock-free lookups) and contention points.

Reporting & Deliverables (commit everything to GitHub)

- **Source code** for XOR, Cuckoo, Quotient filters, plus blocked Bloom; uniform CLI and benchmark harness; plotting scripts.
- **Setup/methodology:** CPU/ISA, compiler & flags, OS, governor/SMT, pinning, hash choices/seeds, dataset generation.
- **Plots/tables** with units and error bars (≥3 runs). Clear labels showing fixed vs varied knobs.
- **Analysis** connecting BPE/FPR/throughput to structure internals (fingerprints, clusters, bucket size), counters, and load factor.
- **Limitations/anomalies** (e.g., rebuilds for XOR, stash growth, long quotient clusters) with hypotheses and mitigations.

Grading Rubric (Total 180 pts)

Implementation (70)

- (20) Correct **XOR filter** build & query; verifies target FPR.
- (20) Correct **Cuckoo filter** with insert/delete, bounded evictions, stash.
- (20) Correct **Quotient filter** with insert/delete (metadata handling).
- (10) **Blocked Bloom** baseline and common benchmark harness.

Experiment Design (40)

- (20) Complete sweeps (FPRs, load factor, negative rate, threads, distributions).

- (10) Fair, apples-to-apples parameters and independent test sets.
- (10) Tail-latency and cold/warm methodology.

5. **Performance Results (40)**

- (15) Clear **BPE vs FPR** and **throughput vs negative-rate** plots; error bars.
- (15) Insert/delete throughput vs load factor; failure/cluster statistics.
- (10) Thread scaling plots with discussion.

6. **Reporting Quality (30)**

- (30) Clean repo, reproducible scripts, labeled plots, and practical conclusions (when to choose which filter).

Project A4 — Concurrent Data Structures and Memory Coherence

Modern multicore programs rely on shared in-memory data structures (e.g., hash tables, trees, queues) that are simultaneously accessed by many threads. Hardware cache coherence ensures correctness at the cache-line level, but high-level correctness and scalability depend on synchronization design. This project explores how lock granularity and coherence traffic affect performance by comparing a simple coarse-grained design with a finer or atomic variant.

Learning Goals

- Implement a thread-safe index and reason about its invariants
- Compare coarse vs. fine synchronization in terms of scalability and overhead
- Measure the effects of contention and false sharing on throughput
- Explain results using cache-coherence and Amdahl's Law.

Scope & What to Build

Choose one data structure:

- Hash Table: open addressing or chaining
- Tree: Binary Search Tree or simplified B+-Tree

Each supports insert(key,value), find(key), and erase(key) for integer keys and fixed-size values.

Synchronization Requirements

- You must implement two versions:
  1. Coarse-grained locking (mandatory baseline): one global mutex guarding all operations
  2. One additional strategy of your choice: fine-grained locking, reader–writer locks, lock striping, or atomic CAS on shared pointers or counters
- Clearly document how you maintain consistency and prevent deadlocks.

Workloads & Evaluation

- Run multi-threaded benchmarks against a single shared instance
- Workload mixes:
  1. Lookup-only: read-dominated (e.g., database queries)
  2. Insert-only: pure writer stress test
  3. Mixed 70/30: realistic balance of reads/writes.
- Data set sizes: 10^4 – 10^6 keys
- Thread scaling: 1, 2, 4, 8, 16 threads (or up to available cores)
- Collect throughput (ops/s), speedup vs. 1 thread, and perf counters such as cycles, LLC-load-misses, and LLC-store-misses

Grading

Category	Points	Description
Correctness & Implementation Quality	20	Operations correct under concurrency; code clear and documented.
Synchronization Design & Comparison	20	Coarse + second strategy implemented; reasoning well explained.
Experimental Rigor	20	Controlled trials, multiple repetitions, reproducible scripts.
Analysis & Insight	20	Interprets scaling trends via coherence and contention principles.
Presentation Quality	20	Readable plots, concise narrative.