Course Project #1: Cache and Memory Performance Profiling

Due date: Sept. 25

The objective of this project is to gain deeper understanding of cache and memory hierarchy in modern computers. You should design a set of experiments that will quantitatively reveal the following:

(1) the read/write latency of cache and main memory when the queue length is zero (i.e., zero queuing delay)
(2) the maximum bandwidth of the main memory under different data access granularity (i.e., 64B, 256B, 1024B) and different read vs. write intensity ratio (i.e., read-only, write-only, 70:30 ratio, 50:50 ratio)
(3) the trade-off between read/write latency and throughput of the main memory to demonstrate what the queuing theory predicts
(4) the impact of cache miss ratio on the software speed performance (the software is supposed to execute relatively light computations such as multiplication)
(5) the impact of TLB table miss ratio on the software speed performance (again, the software is supposed to execute relatively light computations such as multiplication)

The Intel Memory Latency Checker is a useful tool: Google or ask ChatGPT about "Intel Memory Latency Checker"
The Linux "perf" command can gather lots of CPU runtime information such as cache miss ratio and TLB miss ratio, and you can Google or ask ChatGPT to learn more.

Create a Github site to host all your projects through the semester. Post your code/script and detailed results and analysis on Github, and make sure your Github page is clear and self-explanatory.

Course Project #2: Dense/Sparse Matrix-Matrix Multiplication

Due date: TBD

1. Introduction

The objective of this design project is to implement a C/C++ module that carries out high-speed dense/sparse matrix-matrix multiplication by explicitly utilizing (i) multiple threads, (ii) x86 SIMD instructions, and/or (iii) techniques to minimize cache miss rate via restructuring data access patterns or data compression (as discussed in class). Matrix-matrix multiplication is one of the most important data processing kernels in numerous real-life applications, e.g., machine learning, computer vision, signal processing, and scientific computing. This project aims to help you gain hands-on experience of multi-thread programming, SIMD programming, and cache access optimization. It will help you develop a deeper understanding of the importance of exploiting task/data-level parallelism and minimizing cache miss rate.

2. Requirement

Your implementation should be able to support configurable matrix size that can be much larger than the on-chip cache capacity. Moreover, your implementation should allow users to individually turn on/off the three optimization techniques (i.e., multi-threading, SIMD, and cache miss minimization) and configure the thread number so that users could easily observe the effect of any combination of these three optimization techniques. Other than the source code, your Github site should contain

(1) Readme that clearly explains the structure/installation/usage of your code
(2) Experimental results that show the performance of your code under different matrix size (at least including 1,000x1,000 and 10,000x10,000) and different matrix sparsity (at least including 1% and 0.1%)
(3) Present and compare the performance of (i) native implementation of matrix-matrix multiplication without any optimization, (ii) using multi-threading only, (iii) using SMID only, (iv) using cache miss optimization only, (v) using all the three techniques together
(4) Present the performance of (1) dense-dense matrix multiplication, (2) dense-sparse matrix multiplication, and (3) sparse-sparse matrix multiplication
(5) Thorough analysis and conclusion (include discussions under what matrix sparsity you would like to enable matrix compression)

3. Additional Information

C does not have built-in support for multithreaded application, hence must rely on the underlying operating systems. Linux provides the pthread library to support multithreaded programming. You can find a very nice tutorial on pthread at https://computing.llnl.gov/tutorials/pthreads/. Microsoft also provides support for multi-thread programming (e.g., see https://docs.microsoft.com/en-us/windows/win32/procthread/multiple-threads). Since C++11, C++ has built-in support of multithreading programming, feel free to utilize it. You are highly encouraged to program on Linux since Linux-based programming experience will help you most on the job market.

The easiest way to use SIMD instructions is to call the intrinsic functions in your C/C++ code. The complete reference of the intrinsic functions can be found at https://software.intel.com/sites/landingpage/IntrinsicsGuide/, and you can find many on-line materials about their usage.

Moreover, matrix-matrix multiplication has been well studied in the industry, and one well-known library is the Intel Math Kernel Library (MKL), which can be a good reference for you.

Course Project #3: SSD Performance Profiling

Due date: TBD

This project helps you to gain first-hands experience on profiling the performance of modern SSDs (assuming the SSD in your computer is modern enough). The task is simple: Use the Flexible IO tester (FIO), which is available at https://github.com/axboe/fio and may already be included in the OS on your machine, to profile the performance of your SSD. Its man page is https://linux.die.net/man/1/fio. FIO is a storage device testing tool widely used in the industry. Like Project 1, you should design a set of experiments to measure the SSD performance (latency and throughput) under different combinations of the following parameters: (1) data access size (e.g., 4KB/16KB/32KB/128KB), (2) read vs. write intensity ratio (e.g., read-only, write-only, 50%:50% and 70%:30% read vs. write), and (3) I/O queue depth (e.g., 0~1024). Note that throughput is typically represented in terms of IOPS (IO per second) for small access size (e.g., 64KB and below), and represented in terms of MB/s for large access size (e.g., 128KB and above).

WARNING: FIO may overwrite the entire drive partition, so you should create an empty partition on your SSD just for FIO testing. Carelessly running FIO on your existing drive partition **will destroy all your data**!

Again, you should observe a clear trade-off between access latency and throughput (as revealed by queueing theory discussed in class): As you increase the storage access queue depth (hence increase data access workload stress), SSD will achieve higher resource utilization and hence higher throughput, but meanwhile the latency of each data access request will be longer.

The specification of Intel Data Center NVMe SSD D7-P5600 (1.6TB) lists a random write-only 4KB IOPS of 130K. Compare your results with this Intel enterprise-grade SSD, and try to explain any unexpected observation (e.g., your client-grade SSD may show higher IOPS than such expensive enterprise-grade SSD, why?). Explain your reasoning in the report on Github.

Post all your results and analysis on Github.

Course Project #4: In-Memory Key-Value Store with Concurrency Support

Due date: TBD

1. Introduction

The goal of this project is to design and implement an in-memory key-value store that supports concurrent operations, ensuring data consistency and efficiency. The store will handle multiple read and write operations simultaneously, employing appropriate synchronization mechanisms to avoid race conditions and ensure correct behavior in a multithreaded environment. In-memory key-value stores are widely used for high-performance applications where low-latency data retrieval is crucial. They are commonly used in caching systems, real-time analytics, and distributed systems. In this project, you will build a simplified version of an in-memory key-value store from scratch, focusing on efficient data storage, retrieval, and concurrency control.

2. Requirement

Your implementation should support the following operations:

(1) Put(key, value): Inserts or updates a key-value pair
(2) Get(key): Retrieves the value associated with a given key
(3) Delete(key): Removes a key-value pair from the store
(4) Support for multiple concurrent read and write operations
(5) Optional (as a bonus): Support lossless in-memory data compression at small speed performance loss

Feel free to choose your preferred in-memory indexing data structure (hash, tree, …). You should also write a testbench to test your in-memory key-value store under (1) different operational concurrency (one user, two users, 4 users, …), (2) different number of internal working threads (4, 8, 16, …), (3) different read vs. write ratio, and (4) different value size (8B, 64B, 256B) (you may keep the key size as 8B).

Again, post all your code, readme, report and analysis on your GitHub site.