DUE DATE: **By the deadline posted on canvas**

CSE-443/543: High Performance Computing

Exercise #9

Max Points: 20

You should save/rename this document using the naming convention MUid.docx (example: `raodm.docx`).

Objective: The objective of this exercise is to review the following hazards in Instruction Level Parallelism (ILP):

1. Structural hazard arising due to accessing RAM and resolving it using caches.
2. Understanding performance impact of caching
3. Developing programs with cache awareness.

Fill in answers to all of the questions. You may discuss the questions with your instructor.

Name: **Maciej Wozniak**

Background: Reducing structural hazards using caches

Modern CPUs operate in nanoseconds but main memory (aka RAM) operates in 10s of nanoseconds resulting in significant slow down for accessing memory and reduces the throughput from the CPU. Consequently, hierarchies of small but high-speed memory called Caches are used to hide the latency of accessing RAM. Caches are designed to hide latency to access RAM by utilizing three key characteristics of programs, namely: (1) temporal, (2) spatial, and (3) sequential locality of reference. When programs exhibit good locality of reference, they can effectively use caches. On the other hand, if programs do not have good locality of reference, then they suffer from cache misses and performance degrades.

Task 1: Empirical analysis

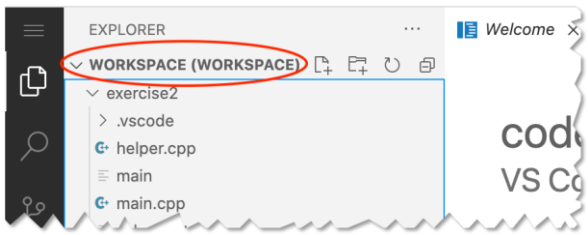
Download and save the supplied `MatrixColumnSum.cpp` program to your work folder. This program is based on the Example 2.5 discussed in your textbook in Chapter 2. This program is used to demonstrate that memory access patterns can have a significant impact on performance of program as they impact cache access, specifically cache misses. Note that time complexity $O(n^2)$ does not change but the implementation details make a big difference.

This part of the exercise expects you to create and submit a SLURM job for measuring performance using the following tasks.

Task 1.1: Project setup steps:

Estimated time: 5 minutes

1. Log into OSC's OnDemand portal via https://ondemand.osc.edu/. Login with your OSC id and password.
2. Startup a VS-Code server and connect to VS-Code. Ensure you switch to your workspace. Your VS-Code window should appear as shown in the adjacent screenshot.



3. Next, create a new VS-Code project in the following manner:
    a. Start a new terminal in VS-Code
    b. In the VS-Code terminal use the following commands:

```
$ # First change to your workspace directory
$ cd ~/cse443
$ # Use ls to check if workspace.code-workspace file is in pwd
$ # Next copy the basic template for a C++ project
$ cp -r /fs/ess/PMIU0184/cse443/templates/basic exercise9
$ # Copy the starter code for this exercise
$ cp /fs/ess/PMIU0184/cse443/exercises/exercise8/* exercise9
```

4. Add the newly created project to your VS-Code workspace to ease studying the source code and the supplied SLURM scripts.

Task #1.2: Submit the SLURM job

Estimated time: 5 minutes

5. Review the supplied SLURM script for this project. Note how the script compiles and runs the supplied microbenchmark for this exercise. Submit your SLURM job as

```
$ sbatch Exercise9_slurm.sh
```

The above job will be queued and then take several seconds to start and run. Once the job ends, you will see an output text file in the format `slurm-<JobID>.out` (e.g., `slurm-12345.out`) with the results.

> **Note**: While you are waiting for your job to run complete the next part of this exercise.

Task #1.3: Study the supplied source code

Estimated time: 10 minutes

While you are waiting for the job to complete study the supplied code. The supplied C++ program computes the sum of all columns in the matrix in the following manner:

1. The `columnSum1` method provides the first (cache-inefficient) approach for computing sum of each column of the given matrix. The code is structured as:
    a. For each column in the matrix
        i. Compute sum of all rows in the column of the matrix

This version is inefficient because Matrices are stored in a row-major form in memory while the method access entries in a column-major order that hinders effective use of caches.

2. The `columnSum2` method provides the second (relatively more cache-efficient) approach for computing sum of each column of the given matrix. The code is structured as:
    a. For each row in the matrix
        i. Update current sum for each column in the current row.

Study the two versions of the program. Note the subtle difference in the structuring of the `for`-loops.

Task 1.4: Measure timings for the two approaches

Once your job completes, you will see an output text file in the format `slurm-<JobID>.out` (e.g., `slurm-12345.out`). Record the runtimes in the Runtime Comparison Template Google Sheet and copy-paste the results for row-major and column-major methods below:

Change tile for cases and the type of data you are entering		
Replicate#	**Column Major**	**Row Major**
1	106.6721407	8.97700799
2	105.9378832	8.967898031
3	106.7327138	8.963750262
4	106.2778217	8.973781263
5	103.8205507	8.973373495
Average:	105.888222	8.971162208
SD:	1.199709752	0.005280014239
95% CI Range:	1.489636408	0.006556003591
Stats:	**105.888222 ± 1.49**	**8.971162208 ± 0.01**
T-Test (H_0: μ1=μ2)	0	

Observe that the runtimes for exactly the same program vary with each run. This is a typical and expected behavior. However, the key question is what the correct runtime is and what value to report? The answer is that all of the runtimes are correct and each user may get a different runtime. However, to ensure that numbers are comparable you must report the mean (average) value along with 95% confidence interval (CI). The 95% CI value essentially defines a range around the average that essentially indicates that you are confident that 95% of the time, the observed execution time would lie within the range you report. In other words there is a 5% chance that the numbers you report are wrong. In practice researchers often perform 10 to 20 runs and report statistics to obtain better CI values.

- Using the above formula record your timings for column major approach in the form $\mu \pm$ CI seconds below (don't forget the unit):

  Time for `column major`: 105.888222+-1.1999s

- Compute the average and 95% CI for running the program using row major version. (don't forget the unit):

  Time for `row major`: 8.971162208+-0.00528s

You should notice that the runtime of the two versions (both $O(n^2)$ time complexity) is vastly different. To explain the difference we are going to dig deeper into the `perf` statistics to determine hazards that could be impacting performance:

- Data hazards: In order to detect data hazards you will need to study the source code to determine if there are dependencies between iterations of the loop to see if you can restructure the code. In this example, the loops have just one operation to perform and there are not a whole lot of code-differences that can explain the significant difference in performance.
- Control hazards: The next important hazard that we studied is control hazards. Recollect that control hazards arise due to branching or loops in the code. In this example we do have nested loops with slightly different ranges. So it is important to record and compare branching behaviors (similar to the data from previous experiment).

  From Linux `perf` output record the **average** number of branches and branch-misses in the two versions of the microbenchmark below:

	Branches	**Branch misses**
Row major	10,239,553,174	1,286,627
Column major	10,433,783,298	2,409,595

- Structural hazards/cache-behavior: The next major hazard that arises in programs is typically due to difference in cache behaviors. In order to compare cache access patterns (this is access to L3 or Last Level Cache on CPU) the supplied PBS job script uses `perf` with `cache-references,cache-misses` flags. Using the cache access statistics reported by `perf` complete the following table:

	Cache references	**Cache Misses**
Row major	864,732,688	637,297,478
Column major	22,910,102,049	765,183,775

DUE DATE:	By the deadline posted on canvas

Task 2.5: Inferences

Briefly describe your inferences from the experiment in the space below:

We can see that because of *row major* does not have to go so deep into the cache references, it works way faster than column. In the *column major* we can see that there is multiple cases when the program has to *enter* references in L3 cache and miss it (so has to enter RAM in order to succeed).

Task 2: Optional: Exam-style Quantitative exercise

A 1 GHz 2-way superscalar CPU has a CPI of 0.75 with each instruction fetched from RAM. In addition, 25% of the instructions requiring write back to RAM (to store results of instruction processing). The CPU is coupled to a RAM operating at 50 ns. What is the effective number of FLOPS realized by the CPU?

What is the peak number of FLOPS that the CPU can provide (1/0.75):

Frequency of instruction retrieval from RAM (1/50ns):

Instructions that require write back to RAM (20*0.25):

Net number of FLOPS realized (20÷(1+0.25)):

Submit files to Canvas

Upload the following files to Canvas:

1. Upload this MS-Word document (duly filled with the necessary information) saved as PDF using the naming convention **`MUid.pdf`**.