CS 610 Semester 2024–2025-I: Assignment 1

14th August 2024

Due Your assignment is due by Aug 24, 2024, 11:59 PM IST.

General Policies

- You should do this assignment ALONE.
- Do not copy or turn in solutions from other sources. You will be PENALIZED if caught.

Submission

- Submission will be through Canvas.
- Submit a compressed file called "⟨`roll`⟩`-assign1.tar.gz`". The compressed file should have the following structure.

```
-- roll
-- -- roll-assign1.pdf
-- -- <problem4-dir>
-- -- -- <source-files>
```

  The PDF file should contain solutions for the first three problems, and your description and results for the fourth problem. Show your computations where feasible and justify briefly.
- We encourage you to use the LaTeX typesetting system for generating the PDF file. You can use tools like Tikz, Inkscape, or Draw.io for drawing figures if required. You can alternatively upload a scanned copy of a handwritten solution, but MAKE SURE the submission is legible.
- You will get up to TWO LATE days to submit your assignment, with a 25% penalty for each day.

Problem 1 **[20 marks]**

Consider the following loop.

```
1    float s = 0.0, A[size];
2    int i, it, stride;
3    for (it = 0; it < 1000 * stride; it++) {
4        for (i = 0; i < size; i += stride) {
5            s += A[i];
6        }
7    }
```

Assume an 8-way set-associative cache with a capacity of 256 KB, line size of 64 B, and word size of 4 B (for `float`). The cache is empty before execution and uses an LRU replacement policy. Given `size`=32K, determine the total number of cache misses on `A` for the following access strides: 1, 4, 16, 64, 2K, 8K, 16K, and 32K. Consider all the three kinds of misses: cold, capacity, and conflict.

Problem 2

[40 marks]

Consider a cache of size 64K words and lines of size 8 words. The matrix dimensions are 1024×1024. Perform cache miss analysis for the *kij* and the *jik* forms of matrix multiplication (shown below) considering direct-mapped and fully associative caches. The arrays are stored in row-major order. To simplify the analysis, ignore misses from cross-interference between elements of different arrays (i.e., perform the analysis for each array, ignoring accesses to the other arrays).

Listing 1: kij form

```
for (k = 0; k < N; k++)
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      C[i][j] += A[i][k] * B[k][j];
```

Listing 2: jik form

```
for (j = 0; j < N; j++)
  for (i = 0; i < N; i++)
    for (k = 0; k < N; k++)
      C[i][j] += A[i][k] * B[k][j];
```

Your solution should have a table to summarize the total cache miss analysis for each loop nest variant and cache configuration, so there will be four tables in all. Justify your computations.

Problem 3

[30 marks]

Consider the following code.

```
#define N (4096)
double y[N], X[N][N], A[N][N];
for (k = 0; k < N; k++)
  for (j = 0; j < N; j++)
    for (i = 0; i < N; i++)
      y[i] = y[i] + A[i][j] * X[k][j];
```

Assume a direct-mapped cache of capacity 16 MB, with 32 B cache lines and a word of 8 B. Assume that there is negligible interference between the arrays `A`, `X`, and `y` (i.e., each array has its 16 MB cache for this question), and arrays are laid out in the row-major form.

Estimate the total number of cache misses for `A`, `X`, and `y`.

Problem 4

[40 marks]

Assume a naïve $O(n^3)$ sequential matrix multiplication kernel (*ijk* form) for matrices of dimensions 4096×4096 (given).

(i) Implement an optimized version of the sequential matrix multiplication implementation using loop blocking. Experiment with different block dimensions (for e.g., 4×4 to 32×32).

Use a table to report the speedup of your optimized implementation over the sequential implementation for the different block sizes that you have tried. The first three columns will be the block sizes for matrices A, B, and C, and the last column will report the speedup compared to the sequential version.

(ii) Note the block size that works best for your setup. Remember that the cache hierarchy will influence the optimal block dimensions. Justify the improved performance by tracking performance counters with PAPI. List the PAPI version and the performance counters you have used.

- Do not change the `ijk` loop order.
- You may experiment with different block sizes for the three matrices.
- Create separate functions for your optimized variants for ease of debugging and evaluation.
- You should time your code on a noiseless system (i.e., no concurrent applications are running). Take times for 3–5 runs, and use the arithmetic mean.
- Report the cache hierarchy for the system you will use.
- Ensure that you are using relatively recent versions of PAPI (v5.7+).
- We will try to reproduce your results.