

Lab: Matrix Sum

姓名: 盛爾葳
學號: 111032042

Questions

- Why is the program so slow?

*You have to explain how the program utilizes the CPU cache, **step by step***

The bad performance of the initial program is mainly due to how the CPU cache is utilized, particularly related to spatial locality and memory access.

How CPU cache works

The CPU cache is a smaller, faster memory located close to the CPU designed to speed up the access to frequently used data. It works optimally when the program accesses memory locations that are physically close to each other (spatial locality), as a single cache line fetch can preload a block of contiguous memory addresses.

→ fetch data時會從同個cache開始(cache hit), 若沒找到(cache miss)就會從更慢的記憶體層級(通常是RAM)取得並存到cache(形成cache lines), 速度較直接存取cache慢很多; 此外因為space locality, (高頻率)跨cache line讀取資料也會比較耗時。

*space locality: if a program accesses a certain memory location, it's likely to access nearby memory locations soon after.

How the program utilizes CPU cache

```
for (int i = 0; i < M; i++) {  
    for (int j = 0; j < N; j++) {  
        sum += mat[j][i];  
    }  
}
```

- We can see that in the initial program, the inner loop accesses the array elements **column-wise** ($sum += mat[j][i];$) when doing calculations.
- Since the 2D array is stored in **row-major order** (the C++ default, where consecutive elements of a row are adjacent in memory), this access pattern leads to poor cache utilization.
- Each time the program accesses an element in a new column, it likely causes a cache miss because the required data is not in the cache line that was most recently fetched. This is because elements in the same column are not contiguous in memory; instead, they are separated by the entire length of a row.

- Verify your thoughts with the **perf** profiler using the command **perf stat -d ./mat_sum**

```

student@lab:~/www$ perf stat -d ./mat_sum
took: 521 ms
sum: 160000000

Performance counter stats for './mat_sum':

      1019.37 msec task-clock                #    0.973 CPUs utilized
         1      context-switches            #    0.981 /sec
         0      cpu-migrations              #    0.000 /sec
      156377    page-faults                 #   153.405 K/sec
         0      cycles                      #    0.000 GHz
    5109221884  instructions                #
    474223472  branches                    #   465.211 M/sec
      8080250  branch-misses                #    1.70% of all branches
    1819900517 L1-dcache-loads              #    1.785 G/sec
    173488470  L1-dcache-load-misses        #    9.53% of all L1-dcache accesses
<not supported> LLC-loads
<not supported> LLC-load-misses

      1.047582347 seconds time elapsed

      0.838862000 seconds user
      0.208694000 seconds sys
  
```

- The performance log from the initial run shows a high number of Level 1 data cache load misses (*173,488,470 L1-dcache-load-misses and 9.53% of all L1-dcache accesses*), indicating that a significant portion of memory accesses could not be served by the cache, causing slower accesses to main memory.
 - The high number of cache misses is a direct consequence of the column-wise access pattern, which does not align with the principle of spatial locality. The CPU is forced to fetch new cache lines frequently as it jumps from one memory location to another far apart in the memory layout, instead of efficiently using the data preloaded in the cache.
- How to improve the performance? How is it faster?
You have to explain how your modified code utilizes the CPU cache

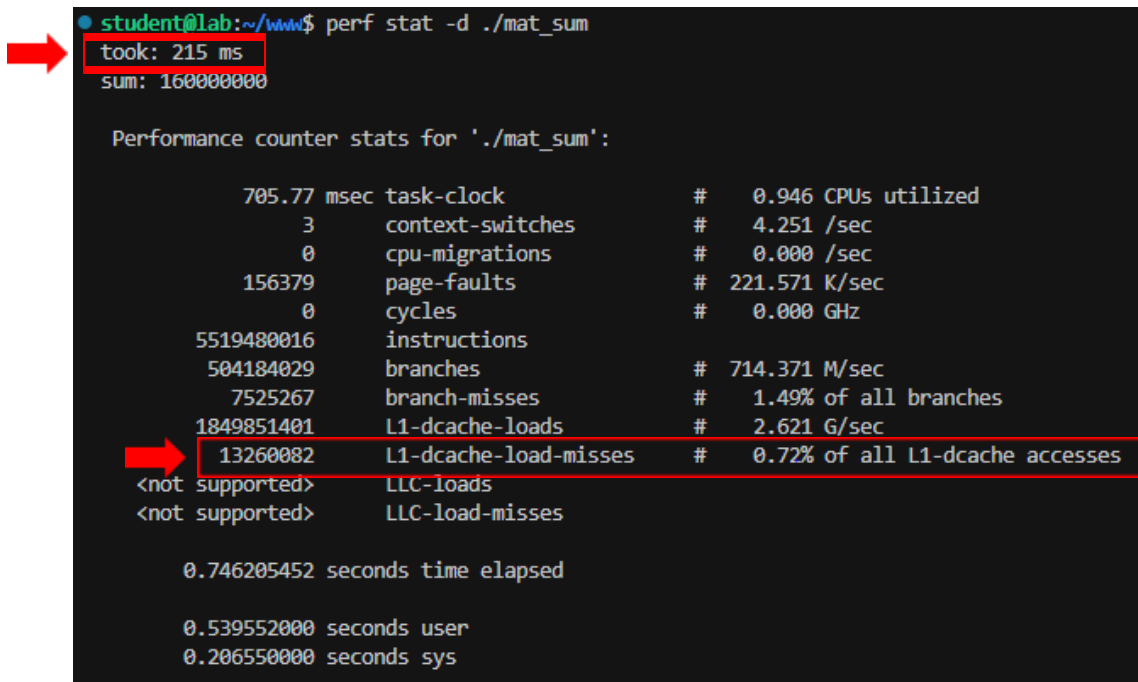
```

for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        sum += mat[j][i];
    }
}
  
```

By changing the loop to iterate over rows inside the inner loop (`mat[j][i]` where `j` iterates over rows), we can align the access pattern with the layout of the 2D array in memory (**row-major order**). This means that when iterating over `j`, the program is accessing consecutive elements in memory, making better use of the data loaded into the cache with each cache line fetch. As a result, there's **fewer cache misses**, lower memory bandwidth usage, and overall faster for computations.

- Verify that your program runs faster

Provide comparisons of both time & perf output



```

student@lab:~/www$ perf stat -d ./mat_sum
took: 215 ms
sum: 160000000

Performance counter stats for './mat_sum':

    705.77 msec task-clock                #   0.946 CPUs utilized
         3      context-switches          #    4.251 /sec
         0      cpu-migrations            #    0.000 /sec
    156379      page-faults               #   221.571 K/sec
         0      cycles                    #    0.000 GHz
    5519480016   instructions              #    714.371 M/sec
    504184029    branches                 #    1.49% of all branches
    7525267      branch-misses            #    2.621 G/sec
    1849851401   L1-dcache-loads           #    0.72% of all L1-dcache accesses
    13260082     L1-dcache-load-misses     #
<not supported>    LLC-loads
<not supported>    LLC-load-misses

    0.746205452 seconds time elapsed

    0.539552000 seconds user
    0.206550000 seconds sys
  
```

- The performance improvement is obvious: the *execution time* dropped from **521 ms to 215 ms**. This indicates a more efficient use of the CPU cache, as less time is wasted fetching data from main memory.
- This massive reduction in cache misses, *L1-dcache-load-misses* also dropped from **9.53% to 0.72%**, means that a significantly higher percentage of the memory accesses were served directly from the cache, reducing the need for slow memory accesses, thus reducing the overall execution time.