Lab: Matrix Sum

111062117, Hsiang-Sheng Huang

March 10, 2025

1 Why is the Original Program Slow?

The original program suffers from poor CPU cache utilization due to cache locality issues arising from an inefficient access pattern. Although the matrix, defined as `mat[N][M]`, is stored in row-major order, the summation loop accesses elements in a column-first manner (using `mat[j][i]`). This causes each jump in the `j` index to access different cache lines, resulting in frequent cache misses and high cache miss rates, which substantially slow down the computation.

2 Verification with `perf`

```
[hpci@apollo31 111062117]$ perf stat -d ./mat_sum
took: 846 ms
sum: 160000000

 Performance counter stats for './mat_sum':

          1,581.02 msec task-clock                       #    1.000 CPUs utilized
                 7      context-switches                 #    4.428 /sec
                 0      cpu-migrations                   #    0.000 /sec
           156,375      page-faults                      #   98.908 K/sec
     4,582,860,388      cycles                           #    2.899 GHz                      (39.28%)
     2,239,839,419      stalled-cycles-frontend          #   48.87% frontend cycles idle     (39.29%)
       970,797,558      stalled-cycles-backend           #   21.18% backend cycles idle      (39.29%)
     4,918,023,263      instructions                     #    1.07  insn per cycle
                                                  #    0.46  stalled cycles per insn  (49.41%)
       436,320,711      branches                         #  275.974 M/sec                    (50.04%)
         1,028,542      branch-misses                    #    0.24% of all branches          (50.59%)
     1,772,350,001      L1-dcache-loads                  #    1.121 G/sec                    (50.60%)
       172,158,835      L1-dcache-load-misses            #    9.71% of all L1-dcache accesses  (50.59%)
       128,298,850      LLC-loads                        #   81.149 M/sec                    (40.47%)
       126,442,556      LLC-load-misses                  #   98.55% of all LL-cache accesses  (39.85%)

       1.581721992 seconds time elapsed

       1.325458000 seconds user
       0.249586000 seconds sys
```

Figure 1: Performance before optimization

We measure the program's efficiency using `perf`:

- **L1 data cache misses**: Indicates how often requested data is not found in L1 cache.
- **LLC (Last Level Cache) misses**: Shows how frequently memory accesses result in expensive main memory fetches.
- **Branch mispredictions and CPU cycles**: To measure overall efficiency.

The high cache miss rates and CPU cycles suggest poor cache utilization and memory access patterns, leading to slow performance.

The further analysis will be shown in the Section 4.

3 Optimizing Performance

To improve performance, we modify the program to utilize **row-major access**:

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        sum += mat[i][j]; // Access matrix elements in row-major order
    }
}
```

By accessing matrix elements in row-major order, we ensure that consecutive elements are stored in contiguous memory locations. This enhances **cache locality** and reduces cache misses, leading to improved performance.

4 Verify Performance Improvements

We compare execution times and cache performance using `perf` before and after optimizations:

```
[hpci@apollo31 111062117]$ perf stat -d ./mat_sum
took: 846 ms
sum: 160000000

 Performance counter stats for './mat_sum':

          1,581.02 msec task-clock                       #    1.000 CPUs utilized
                 7      context-switches                 #    4.428 /sec
                 0      cpu-migrations                   #    0.000 /sec
           156,375      page-faults                      #   98.908 K/sec
     4,582,860,388      cycles                           #    2.899 GHz                      (39.28%)
     2,239,839,419      stalled-cycles-frontend          #   48.87% frontend cycles idle     (39.29%)
       970,797,558      stalled-cycles-backend           #   21.18% backend cycles idle      (39.29%)
     4,918,023,263      instructions                     #    1.07  insn per cycle
                                                  #    0.46  stalled cycles per insn  (49.41%)
       436,320,711      branches                         #  275.974 M/sec                    (50.04%)
         1,028,542      branch-misses                    #    0.24% of all branches          (50.59%)
     1,772,350,001      L1-dcache-loads                  #    1.121 G/sec                    (50.60%)
       172,158,835      L1-dcache-load-misses            #    9.71% of all L1-dcache accesses  (50.59%)
       128,298,850      LLC-loads                        #   81.149 M/sec                    (40.47%)
       126,442,556      LLC-load-misses                  #   98.55% of all LL-cache accesses  (39.85%)

       1.581721992 seconds time elapsed

       1.325458000 seconds user
       0.249586000 seconds sys
```

Figure 2: Performance before optimization

```
[hpci@apollo31 111062117]$ perf stat -d ./mat_sum
took: 481 ms
sum: 160000000

 Performance counter stats for './mat_sum':

          1,216.11 msec task-clock                       #    1.000 CPUs utilized
                 1      context-switches                 #    0.822 /sec
                 0      cpu-migrations                   #    0.000 /sec
           156,378      page-faults                      #  128.588 K/sec
     3,513,196,523      cycles                           #    2.889 GHz                      (39.57%)
       933,471,290      stalled-cycles-frontend          #   26.57% frontend cycles idle     (40.38%)
       876,811,749      stalled-cycles-backend           #   24.96% backend cycles idle      (40.79%)
     5,443,600,755      instructions                     #    1.55  insn per cycle
                                                  #    0.17  stalled cycles per insn  (50.66%)
       506,647,536      branches                         #  416.612 M/sec                    (50.67%)
         1,184,352      branch-misses                    #    0.23% of all branches          (50.57%)
     1,829,903,674      L1-dcache-loads                  #    1.505 G/sec                    (49.75%)
        22,302,602      L1-dcache-load-misses            #    1.22% of all L1-dcache accesses  (49.33%)
           799,336      LLC-loads                        #  657.287 K/sec                    (39.48%)
           893,944      LLC-load-misses                  #  111.84% of all LL-cache accesses  (39.47%)

       1.216665233 seconds time elapsed

       0.932127000 seconds user
       0.280771000 seconds sys
```

Figure 3: Performance after optimization

To evaluate the impact of cache optimization, we compare the performance metrics using `perf stat -d ./mat_sum`. Below are the key observations:

4.1 Total Execution Time

Before optimization, the program took **1.5817 seconds**, whereas after optimization, it was reduced to **1.2167 seconds**, yielding an improvement of approximately **23.1%**.

4.2 L1 Data Cache (L1-dcache)

- **Before Optimization:**
  - `L1-dcache-loads`: 1,772,350,001
  - `L1-dcache-load-misses`: 172,158,835 (**9.71% miss rate**)
- **After Optimization:**

  - `L1-dcache-loads`: 1,829,903,674
  - `L1-dcache-load-misses`: 22,302,602 (**1.22% miss rate**)
- **Improvement:** L1 cache miss rate dropped significantly from **9.71% to 1.22%**, indicating better memory locality and efficient cache utilization.

4.3 Last Level Cache (LLC)

- **Before Optimization:**
  - `LLC-loads`: 128,298,850
  - `LLC-load-misses`: 126,442,556 (**98.55% miss rate**)
- **After Optimization:**
  - `LLC-loads`: 799,336
  - `LLC-load-misses`: 893,944 (**111.84% miss rate**)
- **Observation:** LLC accesses have decreased significantly, reducing expensive memory accesses. The increase in LLC miss rate is due to fewer total accesses.

4.4 CPU Stalls and Efficiency

- **Before Optimization:**
  - `stalled-cycles-frontend`: 48.87%
  - `stalled-cycles-backend`: 21.18%
- **After Optimization:**
  - `stalled-cycles-frontend`: 26.57%
  - `stalled-cycles-backend`: 24.96%
- **Improvement:** The reduction in `frontend stalls` indicates that the CPU spends less time waiting for instructions, thus improving execution efficiency.

4.5 Conclusion

The optimization significantly improved cache utilization, reducing execution time by **23%**. The key takeaways are:

- **Lower L1 cache miss rate** from **9.71% to 1.22%**.
- **Fewer LLC accesses**, reducing main memory latency.
- **Better CPU efficiency**, with lower frontend stalls.