CSC 367 Assignment 1 Report

Muyuan Cao Shuting Zhang
caomuyua zhan4145

Part 1 - Know the Scinet machine's memory system!

Part 1.1: Memory Bandwidth

1.1.1 Idea

Memory write bandwidth: the amount of data written to memory in a fixed period of time.
In order to measure the memory bandwidth, we created large sample array sizes, and measured the accessing (writing) time for each array size.

1.1.2 Experimental setup

The `part_a()` function in part1.c does experiment to measure the memory bandwidth. It does test on 11 large sample array sizes. The sizes are 1MB, 2MB, 4MB, 8MB, 16MB, 32MB, 64MB, 128MB, 256MB, 512MB, and 1GB.

We first allocated a large array of 1GB, and used memset() to fill this array with a constant byte. We find that this process—filling constant bytes—is necessary because if we don't, the time it takes to write to the array is longer, causing the measured memory bandwidth to be too small and inaccurate.

During a single test for each testing size, we used memset() to fill the entire space of the testing size with a constant byte, and timed the operation. We ran tests for each array size 50 times and averaged the results, for the purpose of eliminating outliers and obtaining a more accurate measurement.

In order to collect data and generate graphs, we wrote a python script.
We added

```
`run: all
        python3 plot.py`
```

to Makefile, and added the command

```
`make run`
```

to the job-a1-p1.sh.

1.1.3 Result:

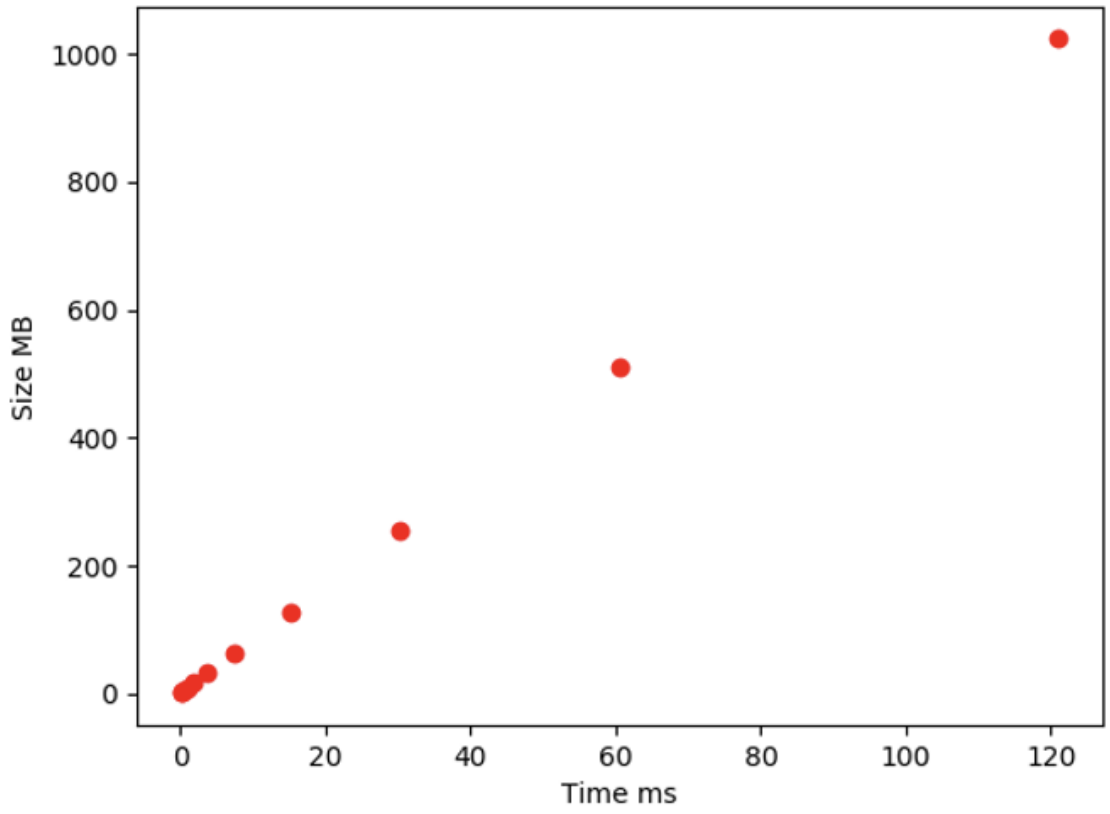

Figure 1: Accessed memory size VS Time

Figure1 has the 11 different accessed memory sizes on the y axis, and the access time on the x axis. The scattered graph has a trend of a straight line, which is a desired result. The slope of the line is the measured memory bandwidth.

MB	Second	MB/s
1	0.000124	8043.752868
2	0.000239	8353.481932
4	0.000486	8237.471475
8	0.000955	8374.367209
16	0.001912	8366.309228
32	0.003806	8408.141753
64	0.007600	8420.969486
128	0.015167	8439.495928
256	0.030316	8444.344773
512	0.060555	8455.084798
1024	0.121066	8458.230796

Table 1: Accessed memory size, time, and the corresponding calculated memory bandwidth

According to the calculations presented in Table 1, the calculated memory bandwidth is around 8400 MB/s, which is approximately 8 GB/s.

Part 1.2: CPU Cache Hierarchy

1.2.1 Idea:

The idea of this experiment is to measure the speed of memory access (write). If the speed significantly slows down when a larger chunk of space is being accessed to, we know that the amount of data does not fit into the current cache and must be stored in a lower level cache, which slows down the speed.

1.2.2 Experimental Setup:

The ` part_b_levels()` function in part1.c does experiment to measure the CPU Cache Hierarchy. It does test on 16 various sample array sizes. The sizes are 4KB, 8KB, 16KB, 32KB, 64KB, 128KB, 256KB, 512KB, 1MB, 2MB, 4MB, 8MB, 16MB, 32MB, 64MB, and 128MB.

For each array size, we iterated through it using a stride of 128 bytes for smaller arrays (under 1MB) and 512 bytes for larger arrays (1MB and above) while we accessed it by writing integers into it. The reason we picked 128 bytes and 512 bytes is because the cache line size is 64 bytes; 128 bytes and 512 bytes are both larger than the cache line size.

Moreover, since the time consumed by a single memory access is very short and therefore the measurement of it is very difficult and inaccurate, we repeatedly accessed the array for 50000 – 100000 times, and then divided the measured time by the number of accessing in total. This entire process is repeated for 500 times. We then averaged the results to eliminate the effects of outliers.

1.2.3 Results:

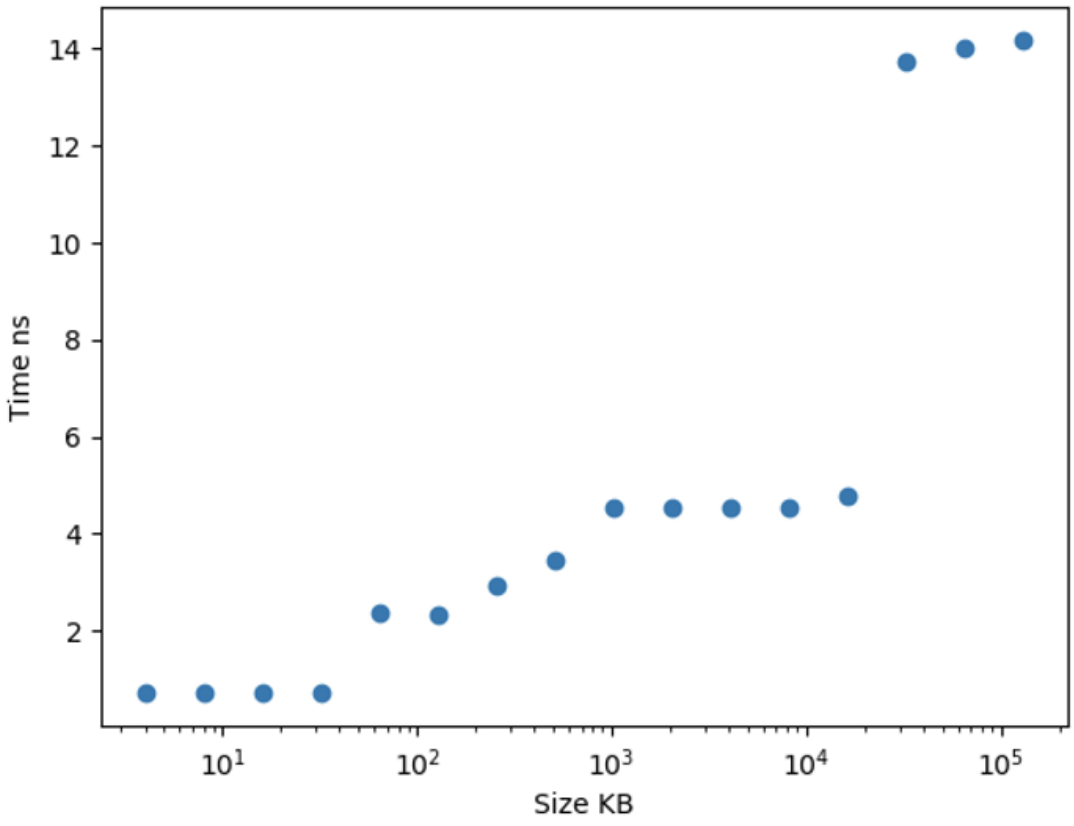

Figure 2: CPU cache hierarchy measurement. Time to write an integer vs. array size.

Size	Time (ns)
4 KB	0.717453
8 KB	0.716338
16 KB	0.716396
32 KB	0.716654
64 KB	2.352793
128 KB	2.341236
256 KB	2.930206
512 KB	3.449254
1 MB	4.544598
2 MB	4.549494
4 MB	4.554142
8 MB	4.557843
16 MB	4.789736
32 MB	13.747505
64 MB	14.035740
128 MB	14.193540

Table 2: accessed array size vs. accessing time (for an integer)

As shown in Figure 2, there are 4 different levels of speed.

- Level 1 speed is, which can be read from Table 2, around 0.71ns (per writing an integer) with an array size of 4KB ~ 32KB. Level 1 corresponds to L1 cache of size 32KB
- Level 2 speed is around 2.5ns with an array size of 64KB ~ 256KB. Level 2 corresponds to L2 cache of size 256KB
- Level 3 speed is around 4.5ns with array size 512KB ~ 16MB. Level 3 corresponds to L3 cache of size 16MB
- Level 4 speed is around 14ns with size larger than 16MB. Level 4 corresponds to the main memory.

Part 1.3: Cache (write) latencies for each level

1.3.1 Idea:

In order to measure the write latency for each level of the memory hierarchy, we allocated an array that's large enough to span through the entire memory hierarchy (L1, L2, L3 cache and the main memory). We then measured the write latency by measuring the speed of memory access for each cache size measured from the previous part.

1.3.2 Experimental Setup:

The ` part_b_latency()` function in part1.c does an experiment to measure the write latencies for each level. We first allocated an array of 128MB, which is large enough that if we fill it, it's data would need to be stored in L1, L2, L3 cache and the main memory.

Then we did test on 4 various sample array sizes. The sizes are 32KB, 256KB, 16MB, and 128MB, each corresponding to a level of the memory hierarchy.

The rest of the experimental setup is the same as the previous part (1.2.2). For each array size, we iterated through it using a stride larger than 64 bytes, accessed data, timed the code, and repeated for many times for more accurate results.

1.3.3 Results:

L1 Cache: 1.069309 ns
L2 Cache: 4.554166 ns
L3 Cache: 5.447833 ns
L4 Cache: 14.790218 ns

Part 2 - Performance and Profiling

2.1 Implement part2-paralle.c

The original program (part2.c) computes the historic average grades for a set of courses. We profiled the code by using gprof. After checking the gprof details, we found that

compute_average() almost took 100% of the entire runtime. This represents that compute_average() is called the most frequently in this program. The program will go through each course one by one; the next call must wait for the current one to finish. Therefore, paralleling this part by creating different threads executing compute_average() should speed up the program.

However, during the experiment, we found that if compute_average() was not called many times, the runtime of part2-parallel.c won't always be shorter than the runtime of part2.c. In our opinion, this should be because of overhead introduced, which is caused by initialized threads, and false sharing.

2.2 Implement part2-parallel-opt.c

We used perf to analyze the cache performance by the following command in our script.
And we forced on cache-references and cache-misses of L1 cache specifically.

perf result of part2-parallel.c:

```
Performance counter stats for './part2-parallel':

        5,785,680      L1-dcache-load-misses:u   #    5.19% of all L1-dcache hits    (57.49%)
      111,462,525      L1-dcache-loads:u                                             (39.37%)
       19,921,822      L1-dcache-stores:u                                            (21.46%)
            1,551      L1-icache-load-misses:u                                       (21.21%)

      0.088332271 seconds time elapsed
```

After several times, we found that the cache-miss rate on L1 is about 5.19%.

Then we read the part2-parallel.c and analyzed it. In this program, every thread will repeatedly read and write on struct course_record. We checked the size of course_record by printf, which has a size of 32bytes. This means that, with the (assumed) cache line size 64 bytes, there might be 2 structs in the same cache line. However, simultaneous updates of individual elements in the same cache line coming from different processors invalidates entire cache lines, even though these updates are logically independent of each other. Each update of the individual element will mark the cache line as invalid. This will cause false sharing that when another thread tries to access another individual element in the same cache line will see this line is invalid. Then they are forced to fetch a more recent copy of the line from memory or elsewhere, even though the element accessed has not been modified.

Therefore, we decided to optimize this part through 2 approaches:

First approach: Store the data as local variables.
We stored the course->grade_count, course->average, and course->grades as local variables in compute_average() to avoid repeatedly accessing or modifying the struct course_record. This will avoid a huge amount of false sharing, which is caused by many iterations.

Second approach: Sharded parallel implementation.
Since the (assumed) cache line size is 64 bytes, there will be 2 course_record structs on the same line. So we decided to let each thread process an equal amount(2 structs) of consecutive elements from the array of all course_record structs. By assigning 2 consecutive structs to each thread, the situation of different threads accessing the same cache line for different elements will be avoided.

Finally we combined the 2 approaches and analyzed the program by perf command.

perf result of part2-parallel-opt.c:

```
Performance counter stats for './part2-parallel-opt':

        1,223,239      L1-dcache-load-misses:u   #    2.19% of all L1-dcache hits    (51.64%)
       55,805,966      L1-dcache-loads:u                                             (32.81%)
       22,602,209      L1-dcache-stores:u                                            (24.90%)
            9,952      L1-icache-load-misses:u                                       (29.02%)

      0.101733493 seconds time elapsed
```

In this program, cache-misses rate on L1 decreases to 2.19%.

2.3 study the effect of the gcc optimization level (-O0, -O1, -O2, -O3)

We compiled para2-parallel.c with different gcc optimization levels and used perf to analyze the cache performance by focusing on cache-references and cache-misses.

To better understand the effect of gcc, we compiled the program by the following command:

```
gcc -c -g  data.c -o data.o -pg
gcc -c -g  part2-parallel.c -o part2-parallel.o -pg -pthread
gcc -o part2-parallel data.o part2-parallel.o -pg  -pthread
```

perf result with -O0:

```
Performance counter stats for './part2-parallel':

         3,045,794      cache-references:u                                          (23.02%)
         1,264,164      cache-misses:u            #   41.505 % of all cache refs    (34.97%)
       925,349,956      cycles:u                                                    (46.22%)
       184,188,496      instructions:u            #    0.20  insn per cycle
        10,217,718      branches:u                                                  (57.09%)
               242      faults:u
                 0      migrations:u
         3,826,663      L1-dcache-load-misses:u   #    3.55% of all L1-dcache hits  (55.69%)
       107,754,926      L1-dcache-loads:u                                           (41.54%)
        19,607,414      L1-dcache-stores:u                                          (21.29%)
             9,512      L1-icache-load-misses:u                                     (21.32%)

      0.169792212 seconds time elapsed
```

perf result with -O1:

```
Performance counter stats for './part2-parallel':

           218,011      cache-references:u                                          (14.08%)
            84,417      cache-misses:u            #   38.721 % of all cache refs    (27.33%)
        92,377,414      cycles:u                                                    (42.17%)
        65,948,652      instructions:u            #    0.71  insn per cycle
        11,122,808      branches:u                                                  (65.77%)
               241      faults:u
                 0      migrations:u
         2,279,630      L1-dcache-load-misses:u   #   15.47% of all L1-dcache hits  (70.34%)
        14,736,357      L1-dcache-loads:u                                           (37.73%)
         7,745,933      L1-dcache-stores:u                                          (15.58%)
            19,409      L1-icache-load-misses:u                                     (10.60%)

      0.107410624 seconds time elapsed
```

perf result with -O2:

```
Performance counter stats for './part2-parallel':

         1,297,197      cache-references:u                                          (31.88%)
           368,210      cache-misses:u            #   28.385 % of all cache refs    (40.53%)
        54,888,158      cycles:u                                                    (49.89%)
        60,813,747      instructions:u            #    1.11  insn per cycle
         4,245,110      branches:u                                                  (49.00%)
               241      faults:u
                 0      migrations:u
             3,198      L1-dcache-load-misses:u   #    0.11% of all L1-dcache hits  (39.56%)
         3,006,926      L1-dcache-loads:u                                           (34.63%)
            44,872      L1-dcache-stores:u                                          (28.56%)
             5,977      L1-icache-load-misses:u                                     (33.93%)

      0.108382966 seconds time elapsed
```

perf result with -O3:

```
Performance counter stats for './part2-parallel':

          371,002      cache-references:u                                          (21.48%)
           94,376      cache-misses:u            #   25.438 % of all cache refs    (29.42%)
       12,836,899      cycles:u                                                    (36.82%)
       16,767,065      instructions:u            #    1.31  insn per cycle
          969,832      branches:u                                                  (40.93%)
              244      faults:u
                0      migrations:u
          595,968      L1-dcache-load-misses:u   #    4.87% of all L1-dcache hits  (46.19%)
       12,239,468      L1-dcache-loads:u                                           (49.13%)
           83,666      L1-dcache-stores:u                                          (32.33%)
           25,839      L1-icache-load-misses:u                                     (26.54%)

      0.107242285 seconds time elapsed
```

From the above results, we can find that the cache-miss rate of -O0 and -O1 are higher than the cache-miss rate of -O2 and -O3. And -O0 and -O1 have similar rates, while -O2 and -O3 ‘s data are similar.

Which makes sense because -O0 is the default and most optimizations are completely disabled. With -O1, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.

With -O2, GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. And with -O3, it turns on all optimizations specified by -O2 and also turns on more flags.