CSC367 Assignment 1 report

Tianchen Xu, Xuan Du

September 2021

Part 1

A) Design an experiment to measure the memory bandwidth in your computer

Result: memory bandwidth measurement is 32 GB/S.

Since the MHZ of the lab computer is around 3000 HZ and data bandwidth is 64 bits, theoretically we can get the memory bandwidth is $x = \frac{3000}{1000} \times \frac{64}{8} \times 2 = 48GB/s$.

The experiment we designed is filling data to a 1GB array and measure the time taken. Then we can calculate the bandwidth by $x = \frac{1}{time}$ = memory bandwidth GB/s.

At the very beginning we used for loop to write and getting a result $\leq 20GB/S$. Then we used memset instead of for loop and result in $\geq 30GB/S$. Which is way better.

```
struct timespec measure_time_memset(int size) {

    memset(src, 1, size); // warm up cache/memory

    struct timespec start, end, diff;
    clock_gettime(CLOCK_MONOTONIC, &start);
    memset(src, 1, size);
    clock_gettime(CLOCK_MONOTONIC, &end);
    diff = difftimespec(end, start);

    return diff;
}
```

The trick we used here is to warm up the memory first then write to the array, it will prevent the page fault that cause efficiency issues.

Although the result is $\geq 30GB/S$, it still doesn't close to the theoretical result, the reason here is each time write, write will go to the cache line, and finally write the cache line to the memory. It is traversing access to the array, that is, writing 1B each time, computer need to read the array from the memory to the cache line first, and then write the cache line. Finally, the cache line is written back to the memory, which is equivalent to two memory accesses each time. If we can disable the cache operations while write, we can get more accurate result.

We can observe the bandwidth calculated by the program through 10 iterations of writing to memory through the file it generates, it showed a consistent result of around 32GB/s.

```
(base) duxuan1@dh2020pc20:~/CSC367/a1/group_0114/A1/part1$ make clean
rm -f *.o part1 *.txt *.csv
(base) duxuan1@dh2020pc20:~/CSC367/a1/group_0114/A1/part1$ make all
gcc -std=gnu11 -Wall -Werror -g3 -O3 -DNDEBUG -c part1.c -o part1.o
gcc part1.o -o part1
(base) duxuan1@dh2020pc20:~/CSC367/a1/group_0114/A1/part1$ ./part1
(base) duxuan1@dh2020pc20:~/CSC367/a1/group_0114/A1/part1$ cat memory_bandwidth.txt
memory bandwidth is 32.093539 GB/S
memory bandwidth is 28.524469 GB/S
memory bandwidth is 31.575143 GB/S
memory bandwidth is 32.022296 GB/S
memory bandwidth is 32.573168 GB/S
memory bandwidth is 31.455922 GB/S
memory bandwidth is 31.299822 GB/S
memory bandwidth is 32.150724 GB/S
memory bandwidth is 32.203658 GB/S
memory bandwidth is 31.411017 GB/S
```

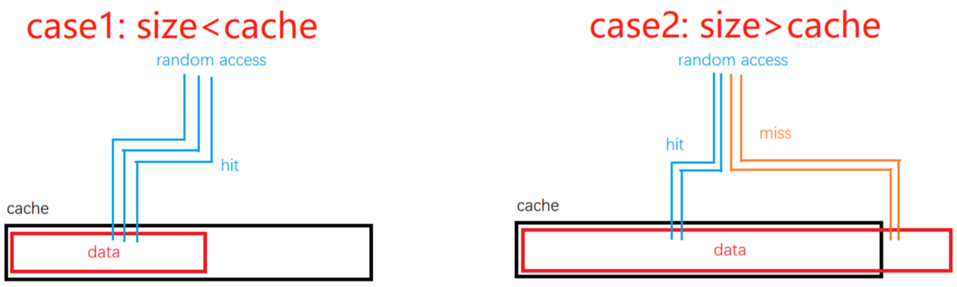
B) Design an experiment to determine the number of levels in the CPU cache hierarchy, and to measure cache sizes and cache (write) latencies for each level, as well as the write latency of main memory.

Result: L1 cache is around 32k, L2 cache is around 256K, L3 cache is around 8192k to 16384k.

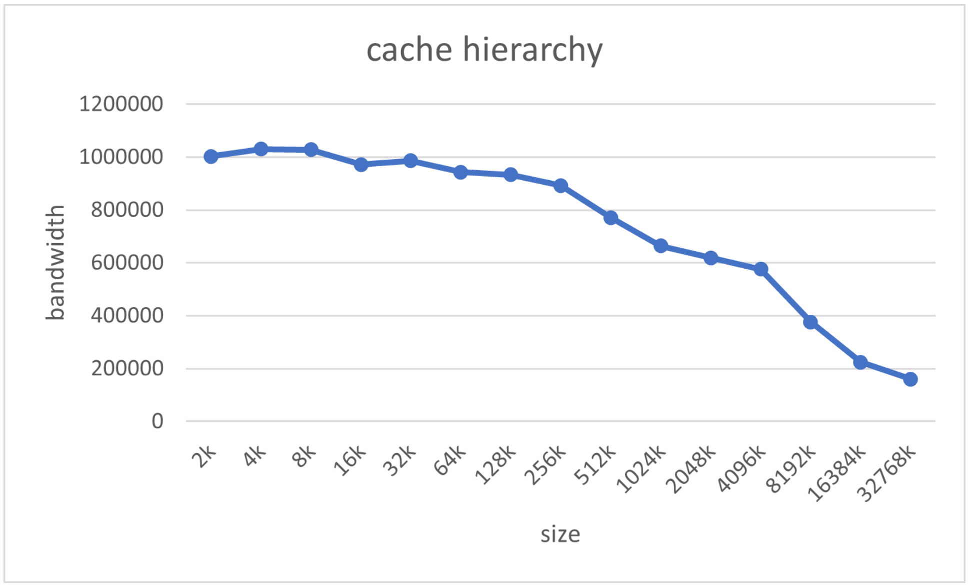
By using linux command. We can get the cache information. That is close to our measurement.

```
(base) duxuan1@dh2020pc20:~$ getconf -a | grep CACHE
LEVEL1_ICACHE_SIZE                 32768
LEVEL1_ICACHE_LINESIZE             64
LEVEL2_CACHE_SIZE                  262144
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  12582912
LEVEL3_CACHE_LINESIZE              64
```

Now this is the intuition of our cache hierarchy measurement. We warm up the cache, and randomly access the index and measure time taken. If the index is warmed up in cache, we can use little time to access, else would cause miss, taking much longer time. We measure access time from 2KB to 32MB.



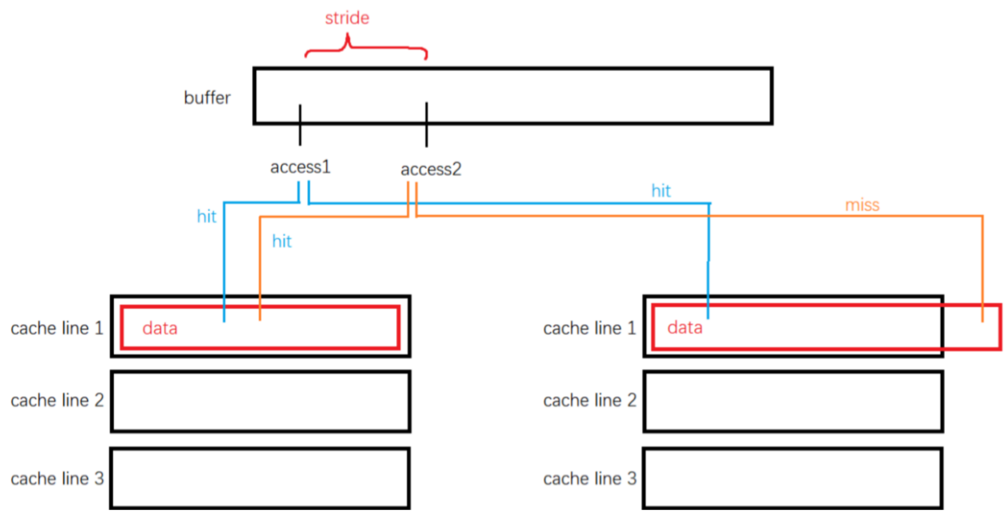
From the result that we parsed into excel. We can see there is a drop at 32k, which is L1. A drop at 256K, which is L2. A drop at 8192k to 16384k, which is L3, from getconf -a we can see the L3 cache size is 12582912, which is 12288K, in between 8192k to 16384k.



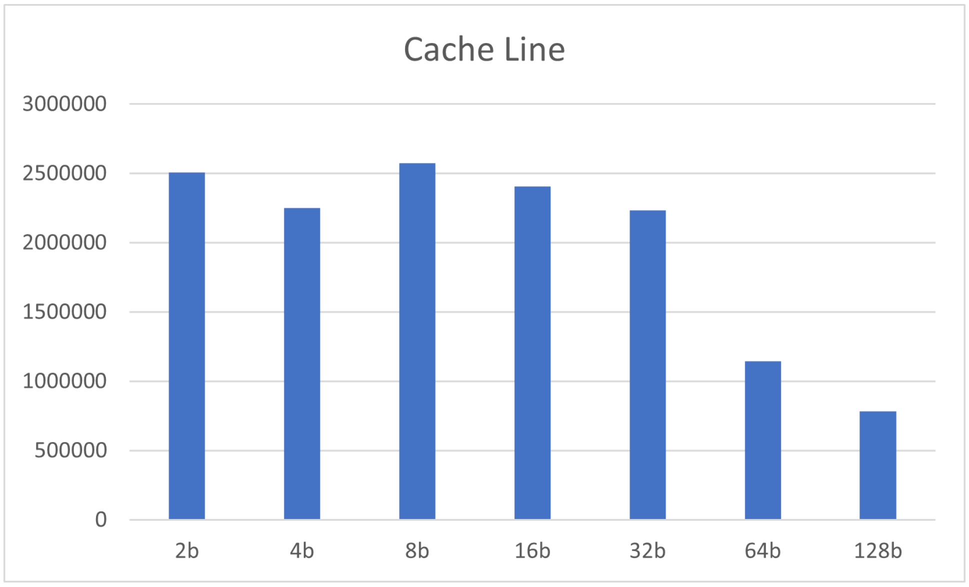
```
(base) duxuan1@dh2020pc20:~/CSC367/a1/group_0114/A1/part1$ make clean
rm -f *.o part1 *.txt *.csv
(base) duxuan1@dh2020pc20:~/CSC367/a1/group_0114/A1/part1$ make all
gcc -std=gnu11 -Wall -Werror -g3 -O3 -DNDEBUG -c part1.c -o part1.o
gcc part1.o -o part1
(base) duxuan1@dh2020pc20:~/CSC367/a1/group_0114/A1/part1$ ./part1
(base) duxuan1@dh2020pc20:~/CSC367/a1/group_0114/A1/part1$ make run
python parse.py
(base) duxuan1@dh2020pc20:~/CSC367/a1/group_0114/A1/part1$ cat cache_hierarchy.csv
2k,1020667.7
4k,1041506.4
8k,1043405.0
16k,1030527.4
32k,1045905.7
64k,1045814.0
128k,1042987.4
256k,1034231.7
512k,848374.6
1024k,706265.9
2048k,671676.2
4096k,644392.3
8192k,480621.5
16384k,267940.3
32768k,184859.8
```

BONUS: design an experiment to measure the cache line size.

Result: cache line size is 64KB.



Now this is the intuition of our cache line measurement. When our stride length is within the L1 cache line, we can hit the data line loaded in the L1 cache when we access the data last time, and when our step length exceeds the size of the L1 cache line, Misses will occur, which will reduce the time of access.



We can see a drop at 64KB, which is the size of cache line.

```
(base) duxuan1@dh2020pc20:~/CSC367/a1/group_0114/A1/part1$ make clean
rm -f *.o part1 *.txt *.csv
(base) duxuan1@dh2020pc20:~/CSC367/a1/group_0114/A1/part1$ make all
gcc -std=gnu11 -Wall -Werror -g3 -O3 -DNDEBUG -c part1.c -o part1.o
gcc part1.o -o part1
(base) duxuan1@dh2020pc20:~/CSC367/a1/group_0114/A1/part1$ ./part1
(base) duxuan1@dh2020pc20:~/CSC367/a1/group_0114/A1/part1$ make run\
> ^C
(base) duxuan1@dh2020pc20:~/CSC367/a1/group_0114/A1/part1$ make run
python parse.py
(base) duxuan1@dh2020pc20:~/CSC367/a1/group_0114/A1/part1$ cat cache_line.csv
2b,2787031.5
4b,2909696.3
8b,2042435.2
16b,2496431.2
32b,2214589.4
64b,1138635.2
128b,851417.5
```

Part 2

Result: We can observe all programs result in the same course averages. The parallel and non-parallel program used similar time, but parallel-opt program improved execution efficiency by a lot.

```
(base) duxuan1@dh2020pc20:~/CSC367/a1/group_0114/A1/part2$ ./part2
CSC243: 64.009901
CSC298: 56.000102
CSC486: 57.999518
CSC319: 65.000869
CSC179: 82.274746
CSC312: 81.494259
CSC293: 64.500386
CSC189: 72.499343
22.545857
(base) duxuan1@dh2020pc20:~/CSC367/a1/group_0114/A1/part2$ ./part2-parallel
CSC243: 64.009901
CSC298: 56.000102
CSC486: 57.999518
CSC319: 65.000869
CSC179: 82.274746
CSC312: 81.494259
CSC293: 64.500386
CSC189: 72.499343
22.094352
(base) duxuan1@dh2020pc20:~/CSC367/a1/group_0114/A1/part2$ ./part2-parallel-opt
CSC243: 64.009901
CSC298: 56.000102
CSC486: 57.999518
CSC319: 65.000869
CSC179: 82.274746
CSC312: 81.494259
CSC293: 64.500386
CSC189: 72.499343
5.771735
(base) duxuan1@dh2020pc20:~/CSC367/a1/group_0114/A1/part2$
```

In parallel version, we make n threads to compute average for each courses, but didn't improve efficiency compare to non parallel program.

```
void compute_averages(course_record *courses, int courses_count) {
    assert(courses != NULL);
    pthread_t tpool[courses_count];
    int err;
    for (int i = 0; i < courses_count; i++) {
        err = pthread_create(&tpool[i], NULL, compute_average, (void*)&courses[i]);
        if (err) {
            fprintf(stderr, "err pthread_create %d\n", err);
            exit(-1);
        }
    }
    for (int i = 0; i < courses_count; i++) {
        err = pthread_join(tpool[i], NULL);
        if (err) {
            fprintf(stderr, "err pthread_join %d\n", err);
            exit(-1);
        }
    }
}
```

In parallel-opt version, we used local average variable instead of accessing course.average, result of 4 times efficiency improvement.

```
void *compute_average(void *arg) {
    course_record* course = (course_record*)arg;
    assert(course != NULL);
    assert(course->grades != NULL);

    // use local average variable instead of course->average
    double average = 0.0;
    course->average = 0.0;
    for (int i = 0; i < course->grades_count; i++) {
        average += course->grades[i].grade;
    }
    course->average = average/course->grades_count;
    return NULL;
}
```

This is how we come up with the parallel-opt solution, by using perf.

```
(base) duxuan1@dh2020pc20:~/CSC367/a1/group_0114/A1/part2$ perf record ./part2-parallel
CSC243: 64.009901
CSC298: 56.000102
CSC486: 57.999518
CSC319: 65.000869
CSC179: 82.274746
CSC312: 81.494259
CSC293: 64.500386
CSC189: 72.499343
29.297861
[ perf record: Woken up 1 times to write data ]
[kernel.kallsyms] with build id 4a11316284e0020a24633b0f06ec126391bdb396 not found, continui
[ perf record: Captured and wrote 0.025 MB perf.data (609 samples) ]
(base) duxuan1@dh2020pc20:~/CSC367/a1/group_0114/A1/part2$ perf report
```

By opening the perf report, we can see most time are spent at compute average function, let us dig into it.

```
Samples: 609  of event 'cycles:uppp', Event count (approx.): 738977650
Overhead  Command         Shared Object       Symbol
  99.66%  part2-parallel  part2-parallel      [.] compute_average
   0.15%  part2-parallel  [kernel]            [k] 0xffffffffa52009e7
   0.04%  part2-parallel  ld-2.27.so          [.] _dl_fixup
   0.04%  part2-parallel  ld-2.27.so          [.] do_lookup_x
   0.04%  part2-parallel  ld-2.27.so          [.] _dl_runtime_resolve_xsavec
   0.04%  part2-parallel  libc-2.27.so        [.] malloc
   0.02%  part2-parallel  ld-2.27.so          [.] __GI___tunables_init
   0.01%  part2-parallel  libc-2.27.so        [.] __clone
   0.00%  part2-parallel  libpthread-2.27.so  [.] __GI___pthread_timedjoin_ex
   0.00%  part2-parallel  [kernel]            [k] 0xffffffffa520015c
```

We found 55.2 mov 0x10(rax),eax this assembly line is taking a lot of time that is just a simple mov. Then we take a guess on the there are a lot of cache misses happening. By tracing the code, we find course.average += course.grades[i].grade, the course.average is hitting misses, to avoid that, just simply replace course.average by a local variable and form spacial locality (cache friendly code).

```
          course->average = 0.0;
    mov    -0x8(%rbp),%rax
    pxor   %xmm0,%xmm0
```

```
              movsd  %xmm0,0x18(%rax)
                    for (int i = 0; i < course->grades_count; i++) {
              movl   $0x0,-0xc(%rbp)
            ↓ jmp    5f
                           course->average += course->grades[i].grade;
        26:   mov    -0x8(%rbp),%rax
 14.52        movsd  0x18(%rax),%xmm1
              mov    -0x8(%rbp),%rax
 16.31        mov    0x8(%rax),%rdx
              mov    -0xc(%rbp),%eax
  1.61        cltq
              shl    $0x3,%rax
  0.18        add    %rdx,%rax
  2.33        mov    0x4(%rax),%eax
  0.54        pxor   %xmm0,%xmm0
  2.33        cvtsi2sd %eax,%xmm0
  4.12        addsd  %xmm1,%xmm0
              mov    -0x8(%rbp),%rax
  2.15        movsd  %xmm0,0x18(%rax)
                    for (int i = 0; i < course->grades_count; i++) {
              addl   $0x1,-0xc(%rbp)
        5f:   mov    -0x8(%rbp),%rax
 55.20        mov    0x10(%rax),%eax
              cmp    %eax,-0xc(%rbp)
  0.72      ↑ jl     26
                    }
```