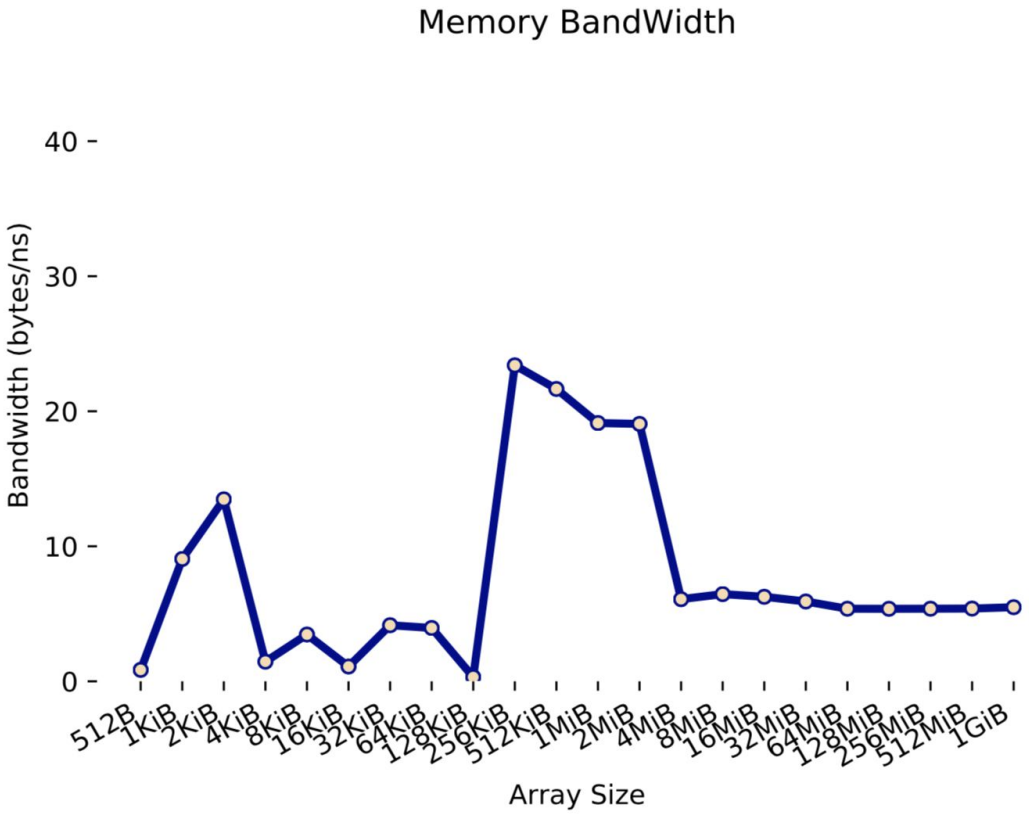
CSC367 Assignment 1 Report

Farzaneh Tabandeh
Sitao Wang

Part 1

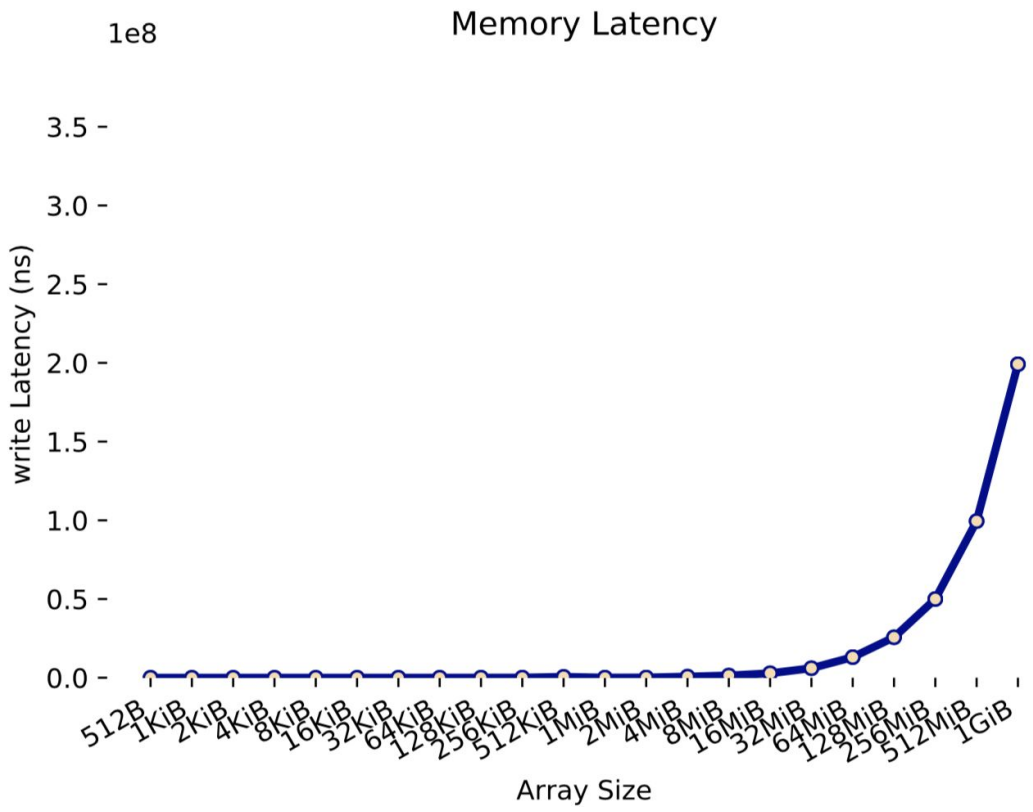
1.1 Measuring memory bandwidth:

We define the memory bandwidth to be the number of bytes that can be written to memory per ns. So memory bandwidth is in bytes/ns in our calculations.
In function calculateMemoryBandWidth() in part1.c, we calculate the memory bandwidth by defining an array of size 1 GiB and then iteratively issuing writes to every portion of it in one go. Starting from size 512 bytes we issue a write to initial 512 bytes of this array using memset() and time it. We increase the write size in powers of 2, and time each memset() in each iteration. Our goal is to measure how much write in bytes we can issue in one go before time going up exponentially. The resulting data gathered shows the memory bandwidth is around ~ 23.4 bytes/ns (equivalent to ~ 22315.98 MiB/s) at maximum which occurs at the write size of around 256 KiB.



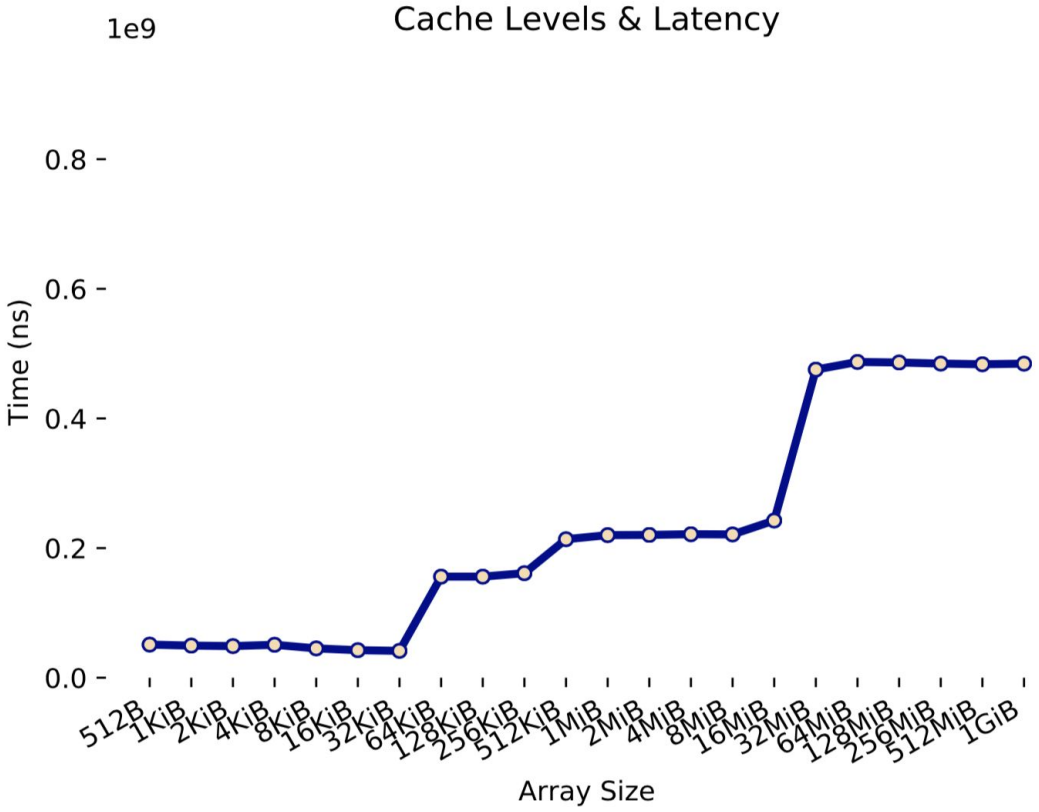
We can observe the memory bandwidth from another point of view using our gathered data in file memoryBandWidth.csv. The graph below shows the write size vs total time for that

write size. We can see writes size up to ~ 256 KiB take around the same amount to be done, while the write sizes of more than ~ 256 KiB makes the time of completion to peek exponentially. Again we can conclude we are at the max bandwidth limits of memory at the size of ~ 256 KiB.



1.2 Measuring cache sizes and their latencies

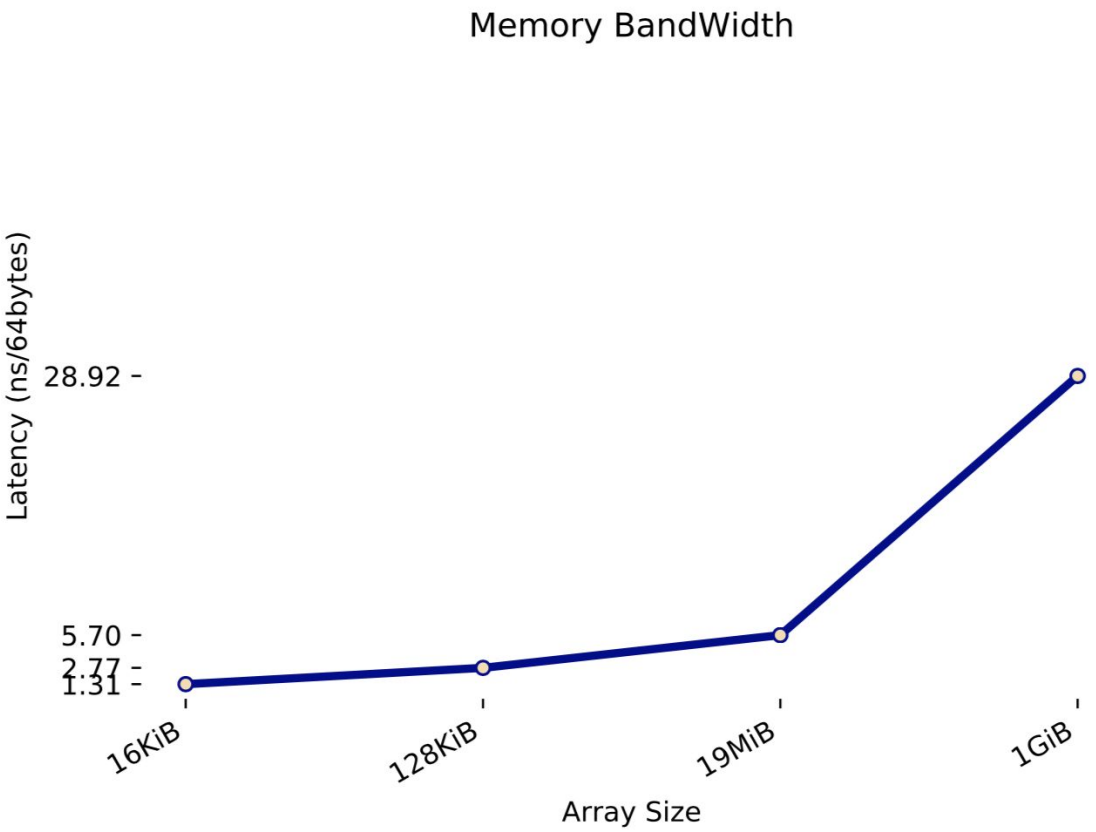
We measure the caches sizes (L1, L2, L3) in the function calculateCacheSizes() in part1.c. To measure the cache sizes, we define a large constant int number as:
$CONST\_ACCESS = 64 \times 1024 \times 1024$
We try to access arrays of different sizes, exactly $CONST\_ACCESS$ number of times. The idea is that, for each array, if the whole array can be held by the cache at the level whose size we are measuring, the time of every single access should be roughly constant, as shown in the results below.
We accessed each array $CONST\_ACCESS$ number of times and obtained the time of one single access by diving runtime by $CONST\_ACCESS$. We gathered the data in the file cache_sizes.csv. The graph below shows the results of arrays with different sizes vs total time of $CONST\_ACCESS$ accesses to them.



From the graph, we can see that there are 3 sections within which the total time of $CONST\_ACCESS$ accesses are roughly constant. These sections represent access to L1, L2, L3, and main memory.
So we can conclude the sizes of the caches are as follows:
L1: 32 KiB
L2: 256 KiB
L3: 16 MiB
Note that, from hardware info on SciNet, the L3 cache size is actually 20 MiB, but since we are using powers of 2 array size here, we do not catch the exact size of the L3 cache.

To measure the latencies of each cache level, we have the function calculateCacheLatencies() in part1.c. In this function, we use the information of cache sizes and use them to calculate the latencies. To calculate the latency of cache level i, we define a test array whose size is within the size of the cache level L_i. Then we define another array as an invalidator whose size is within cache at level L_{i-1}. Accessing the invalidator array causes all lines of the test array to be pushed down the hierarchy levels of caches, so that the test array is placed at the cache level L_i and not at any level above it. Then we go over the test array again to calculate the time of access of it which shows the latency of access to L_i cache.
As an example assume we want to calculate the latency of L2. We define a test array of size 128 KiB, which makes the whole array to be extended to the L2 cache when it is accessed in whole. Then we define another array as invalidator with size 32 KiB such that accessing it

causes all lines of the test array to be pushed down the hierarchy levels of caches and be located in L2 downwards. Now we can access the test array and the time of accessing it is in fact the latency of L2 cache.
We define the latency of a cache to be the time it takes for the CPU to retrieve one cache line (64 bytes) from that cache. We define Latency in (ns/cache line). We gathered the obtained data in cacheLatency.csv file and the graph below is the visualization of it.

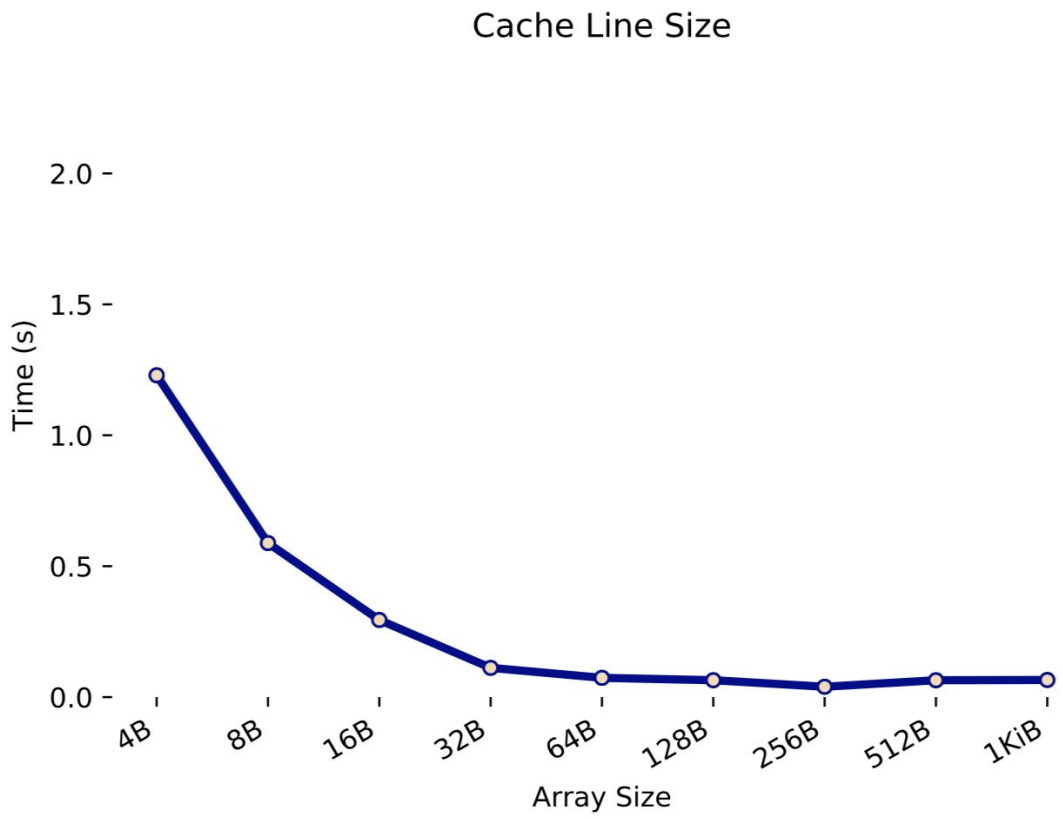


Upon several runs of the function, we obtained a range of numbers for latencies. The graph above is only the result of one of the runs. Based on this specific run, we can see that the latency for the caches can be approximated to be:
L1 ~ 1 ns
L2 ~ 3 ns
L3 ~ 6 ns (max we got among all runs was ~ 8 ns)
main memory ~ 30 ns (max we got among all runs was ~ 80 ns)

This results clearly shows the increase of latency as we go down the levels of cache hierarchy in the system.

1.3 Measuring cache line size

We measured cache line size in a similar method as measuring cache level sizes. This experiment is implemented in the function calculateCacheLineSize(). We time a constant number of accesses (8×1024) to different sizes of arrays starting from a 4 byte array. Since we access each array with different size a constant number of times, we can measure the approximate time of each single access to it, by dividing total runtime over the constant 8×1024. We gathered the data of his experiment in cache_line.csv and below demonstrates the results.



As this visualisation suggests, the time of 8×1024 accesses to the arrays with different sizes starts high but then stabilizes at a lower value when the size reaches to 64 Bytes. The reason is that in the case of arrays with size smaller than the cache line size of the system, CPU fetches one cache line size regardless. However, in these cases, some portion of the fetched line is unusable. As CPU instructions are only done on word aligned sizes, this scenario causes the overhead of data alignment which increases runtime. This happens for all sizes below 64 bytes. One other issue that could potentially cause overhead is if this unused data is shared between CPU cores, this possibly causes false sharing. So based on these data, we can conclude the cache line size is 64 bytes.

Part 2

In this part, we are parallelizing a computation-heavy code that computes the average of some large arrays. We first parallelize the code in a naive way which runs slower than the serial code. We then optimize the code to achieve better performance.

2.1 Naive Task Parallelism

Our first attempt is to divide the task and parallelize them. Because we need to compute the average of each array, we define each smaller task as computing the average of one array.

The result of this implementation is even worse than the serial code. In our experiment that runs on two arrays, the average runtime of 5 executions increases from 20.895511 seconds to 27.444531 seconds.

There are several reasons that count for the reduction of the performance.

First, the tasks divided in this naive way are unbalanced. Each task computes on its own array, which has different length, and thus different runtime. The tasks with shorter runtime end earlier and wait for tasks with longer runtime to finish.

Another reason for the deduction of performance is the cache misses. Each task works on different arrays, so the computation has low spatial locality and higher cache miss rate. From the experiment result shown in the table below, we can see that the naive parallelism increases the L1 data cache miss rate from 0.76% to 3.45%, which induces a large amount of runtime overhead.

	serial	naive parallel	optimized parallel
L1 dcache load	67,544,046	51,353,190	51,792
L1 dcache miss	515,126	1,769,365	10
L1 dcache miss rate	0.76%	3.45%	0.02%

2.2 Optimized Data Parallelism

To tackle the problems from the previous implementation, we decide to separate the computation by dividing the input data for computation of each array. In our implementation, we allocate 4 threads, each computing the sum of one-fourth of the array in a shared way, and adding them up with synchronization, and divided by array length to get the average. The computation of all arrays are serialized, that is, we do not compute the next array until we finish all computation on the current array.

In this way, all the tasks running at the same time are balanced. We segment the array uniformly in 4 sections, thus each task has nearly the same amount of computation

exception for the last thread having some more computation if the array size is not divisible by 4, so all the running threads ends at nearly the same time and no computation power is wasted to wait for a long running thread.

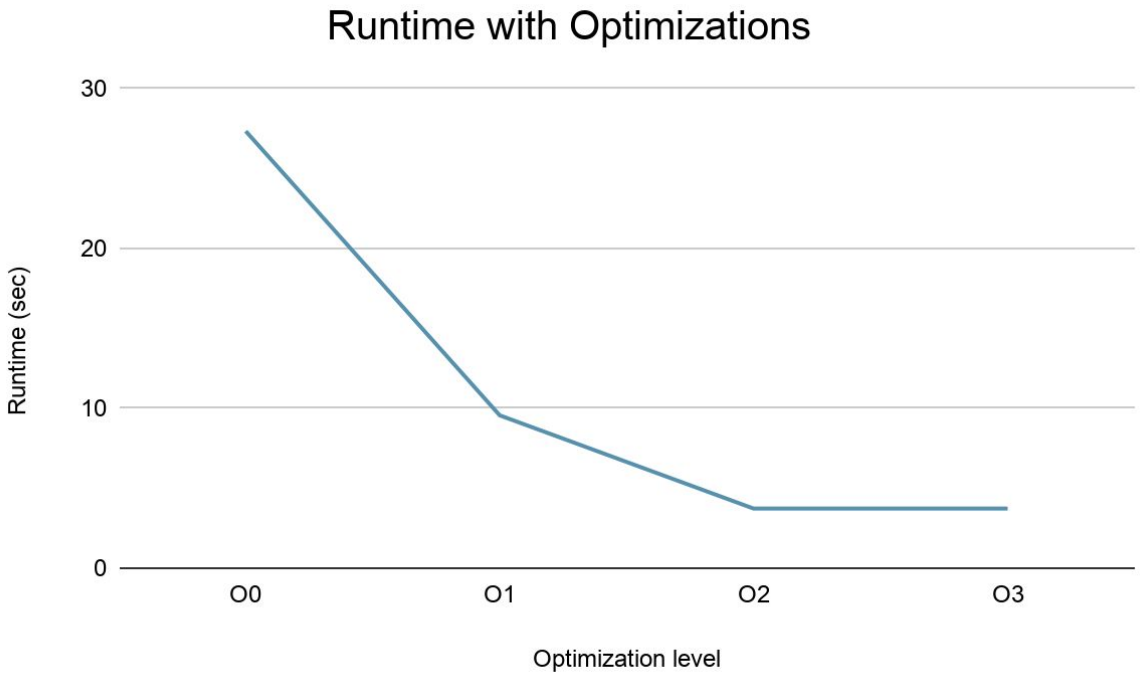
Because we compute one array after another, and each array is segmented in a shared way, we have higher spatial locality, and lower cache miss rate. From the experiment result, we can see that the L1 data cache miss rate drops to 0.02%, much lower than both serial implementation and naive parallel implementation.

With the optimization, the runtime of 5 average executions on the same machine as previous experiments drops to 5.24419 seconds.

2.3 Compiler Optimization for Naive Parallelism

All the previous experiments are conducted with no compiler optimization from gcc. We also analyze how compiler optimization will affect our naive implementation of parallelism.

We compile our implementation with different optimization flags, namely -O0, -O1, -O2 and -O3, run the generated binary 5 times and compute the average runtime. The results are shown in the graph below.



From the results, we can see that the optimization from O1 decreases the runtime from 27.33 seconds to 9.56 seconds. The main reason is that optimization like auto-inc-dec decreases the runtime of each thread, resulting in a decrease of overall runtime.

O2 also offers runtime optimization from 9.56 seconds to 3.73 seconds. Applying O2 optimization, we can see that L1 data cache miss rate, one bottleneck of the original naive implementation without compiler optimization, drops to 0.02%, same as our manually optimized code. O2 offers multiple optimizations on scheduling, i.e. interblock scheduling and speculative scheduling, that fixes the high cache miss rate.

O3 offers some critical optimization like loop unrolling, but has little effect on our code, which is computation heavy and already heavily optimized.