Comparing Compiler Optimizations and Memory Access Assignment #1, CSC 746, Fall 2022

Christopher Humphrey*
SFSU

ABSTRACT

We are curious to study the nature of memory access and caches as they relate to a program's performance. Namely, the problem introduced is to understand compiler optimizations and structured vs unstructured memory access via the metrics FLOPs, bandwidth, and memory latency.

The core idea is that memory access (both the nature of the access, and the number of accesses) is a major element in computational speed. Our solution points to an understanding that structured memory access is superior partially because it can be better optimized by the compiler. We learned that it is important to be mindful of how memory access is structured, and to reduce the number of accesses generally. Furthermore, we learned that compiler optimization is a crucial tool in writing code for high-performance computing.

1 INTRODUCTION

This being a high-performance computing course, we are interested in the increasing computational speed. Memory can be divided colloquially into "fast" and "slow" memory, and over-access of "slow" memory can act as a limit upon a computer's theoretical speed.

We are comparing the time to sum numbers 0...N-1 using three different algorithms (via a typical for-loop summation, accessing a vector's elements in a "structured" way, and accessing a vector's elements in an "unstructured" way) each tested with both optimized and unoptimized compiler flags. We then compare the elapsed time of the six methods.

We find that by far the most dramatic speed gains are from compiler optimization, regardless of algorithm used or structured vs unstructured memory access, though generally structured memory access is faster.

2 IMPLEMENTATION

We compare the time to sum numbers using three different algorithms, each corresponding to a Part subsection.

2.1 Part 1

Objective: to establish a control experiment by summing up numbers with a for-loop and in the most traditional fashion.

We coded up a for-loop which sums unsigned long longs from 0....N-1, with each N being a user inputted number

[100000, 500000, 1000000, 5000000, 10000000, 50000000 , 100000000].

We clocked the elapsed real-time of the loop of each N. We repeated this process for C++ compiler flag

`${CMAKE_CXX_FLAGS_RELEASE} -O3`

`${CMAKE_CXX_FLAGS_RELEASE} -O0`

corresponding to no compiler optimization and full-optimization, respectively.

2.2 Part 2

```
1    unsigned long long int sum1 = 0;
2    for ( int i = 0; i < target; i++)
3        sum1 += myVector[i];
```

Objective: to time structured memory access.

We first stored each integer from 0...N-1 into a vector. The values for N were the same as in Part 1. We then clocked the elapsed real-time of a for-loop which iterated the vector[index] and summed each element. Just as in Part 1, we repeated the experiment for both an optimized and unoptimized compiler.

It is important to point out that we did not use an iterator, but instead accessed the elements of the vector using an index. This was to maintain consistency with Part 3.

2.3 Part 3

```
1    unsigned long long int sum2 = 0;
2    int zero = 0; // Declaring 0 pointer
3    int* pointer = &zero;
4    for ( int i = 0; i < target; i++)
5    {
6        sum2 += myVector2[*pointer];
7        pointer = &myVector2[*pointer];
8    }
```

Objective: to time unstructured memory access.

We stored pseudo-random seeded integers in the range [0, N-1] in a vector. The values of N were the same as in Part 1. We then initialized a pointer to zero, and measured the elapsed of a for-loop which kept a cumulative sum of vector[pointer] and then set vector[pointer] to be the new pointer. Just as in Part 1 and 2, we repeated the experiment for both an optimized and unoptimized compiler.

3 EVALUATION

We ran our three experiments in C++ using Dr. Bethel's CMakeLists.txt with minimal edits.

3.1 Computational platform and Software Environment

We ran our tests primarily on our personal Macbook Air M1, but we also ran our tests on the NERSC Cori supercomputer. Our Macbook runs macOS 12.3 and has an ARM-based M1 processor with 4 cores with 12MB L2 cache and a clock-rate of 3.2Hz, and 4 cores with 4MB L2 cache and a clock-rate of 2.0GHz. Each core has at least 128+64KB of L1 cache. The Macbook Air M1 has 8GB of 128-bit Dual Channel LPDDR4X-4266 Unified Memory.

*email:chumphrey@sfsu.edu

3.2 Methodology

We measured real elapsed time using the chrono library. That is, chrono used the system clock to measure timeStart - timeEnd of the three different loops (and did not measure other codeblocks of the program). The chrono library high resolution clock measures ticks in units at least as small as a nanosecond. We then plotted the results in the figures below using Google Sheets.

3.3 Experiment 1

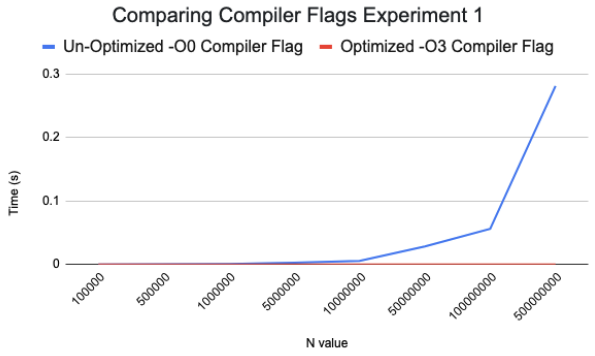

Figure 1: Comparing optimized vs un-optimized program speeds for experiment 1.

To us, experiment 1 (Figure 1) acts more as a control. It is a simple sum that should only make repeated memory accesses to the ”sum” variable. The un-optimized version more or less follows a simple O(N) trajectory in terms of elapsed time. The optimized version is highly constant in time-complexity, with little to no increase in the range of values for N that we inputted.

3.4 Experiment 2

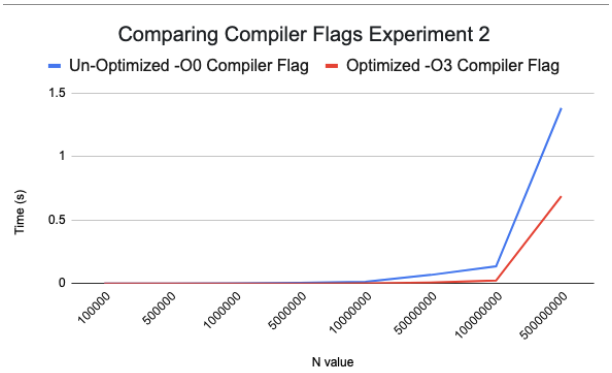

Figure 2: Comparing optimized vs un-optimized program speeds for experiment 2.

Experiment 2 (Figure 2) is a study on structured memory access. For consistency with experiment 3 we do not use an iterator/pointer, but access the index via the operator[]. The indexes of the vector were accessed in increasing order, i++, and summed together. The results are such that both the optimized and un-optimized versions have a similar elapsed time. trajectory, but the un-optimized version has a steeper time-complexity and performs slower, especially as N goes above 50000000.

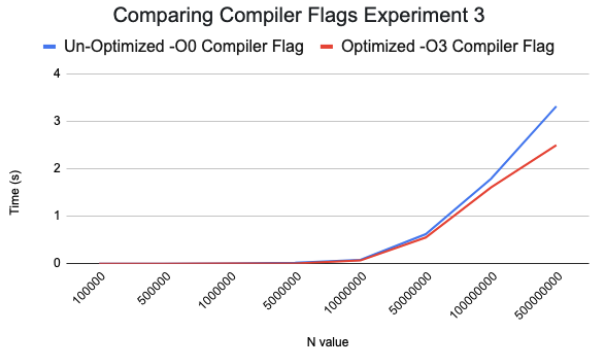

Figure 3: Comparing optimized vs un-optimized program speeds for experiment 3.

3.5 Experiment 3

Experiment 3 (Figure 3) is a study on unstructured memory access. We essentially jumped to random indexes in the vector, and summed the contents of those indexes N times. The elapsed time of the un-optimized and optimized versions of experiment 3 followed a nearly identical trajectory until N reached 50000000.

3.6 Cross-Experiment Comparison

Figure 4 overlays the elapsed time results from the un-optimzed versions of all three experiments. In order of increasing speed: experiment 1 was the most performant, followed by experiment 2 and then experiment 3. Memory access clearly has an outsized role on performance.

Table 1 is the calculated FLOPs at N = 500000000 of both the optimized and un-optimized compiler flags for each of the three experiments. this was calculated as the number of operations per second (500000000) divded by the elapsed time in seconds. Clearly optimized compilation outperforms. Out most performant iteration of the experiments was the optimized version of experiment 1.

**I should note that our suspicion is that this is a ”rounding” error of sorts, and that the resolution of the chrono clock is not such that we can get accurate measurements (specifically only for -O3 of experiment 1).

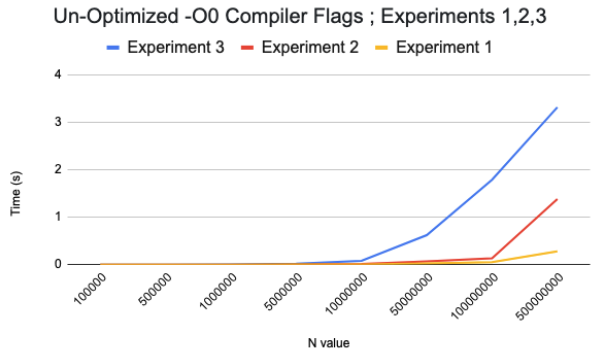

Figure 4: Overlaying results from all three experiments in their un-optimized compiled form.

3.7 Findings and Discussion

In Figure 5 and Table 1, our most performant experiment in terms of FLOPs was the optimized compiled version of experiment 1,

Experiment	-O0 Optimization (FLOPs)	-O3 Optimization (FLOPs)
1	1.78E+09	1.19E+16**
2	3.62E+08.	7.27E+08
3	1.51E+08.	2.00E+08

Table 1: Comparing FLOPs at N = 500000000

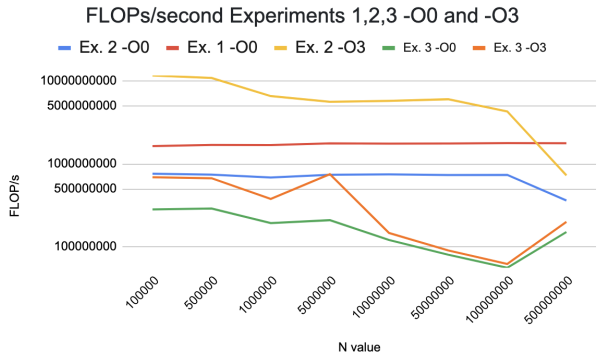

Figure 5: Comparing FLOPs of experiments 1,2,3 (omitting experiment 1 -O0)

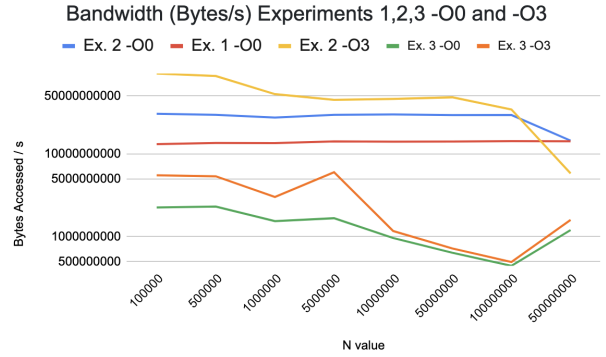

Figure 6: Comparing bandwidth of experiments 1,2,3 (omitting experiment 1 -O0)

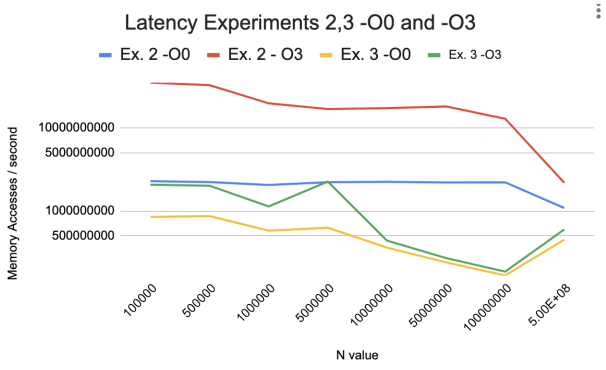

Figure 7: Comparing latency of experiments 1,2,3 (omitting experiment 1 -O0)

but because we suspect that this was a measurement error due to insufficient resolution of the chrono library we will dismiss it. On our Macbook Air M1, we did achieve 1.8 GFLOPs in the un-optimized compiled version of experiment 1 (which was our best reasonable result). This is substantially less than CORI's KNL 3 TFLOPs theoretical node peak.

The sum variable is 8 bytes in memory, the vectors in experiments 2 and 3 are variables are 24 bytes of memory, and the elements of the vectors are 8 bytes each. Thus, for experiment 1, each iteration of the for-loop accessed 8 bytes of memory; for experiments 2 and 3 each iteration of the for-loop accessed 40 bytes of memory.

Ignoring the optimized version of experiment 1 for suspected measurement issues, the highest average memory bandwidth (Figure 6) occurs in the optimized version of experiment 2 (5.15E+10). However, at large N we see that the highest bandwidth is in experiment 2's unoptimized compiled version. Of interest is that at large N both experiment 1 and 2 have similar bandwidth, regardless of optimization type. This is because the optimized version of experiment 1 has a bandwidth that is on a downward trend with respect to N. The lowest memory bandwidth consistently is in both compiled versions of experiment 3, which also has a downward trend. CORI has a bandwidth of 95.4 GB/s (after conversion GiB to GB), while we have a average bandwidth of 51.5 GB/s.

Experiment 2 and 3 access memory three times per for-loop iteration. Once to get the sum, once to get the vector, and once to get the vector element. Per Figure 7, Memory latency follows a very similar trend to bandwidth, and so we hypothesize that the majority of memory access bottleneck comes from the mere act of initiating a memory access.

Clearly, the optimized version of experiment 2, where we access data in a structured manner, is superior in terms of minimizing memory latency. The unoptimized version of experiment 3 (where we access unstructured data) is a magnitude worse in terms of latency. In fact, we could conclude that the overall greatest explanation for the time differential of experiments 2 and 3 is how much slower it is to access the unstructured data from experiment 3.

What are the differences between -O0 and -O3? Generally compiler optimization netted superior results. However, in experiment 3 the optimizations did change performance as dramatically, which suggests that the compiler had trouble optimizing the unstructured memory accesses.

This is all conjecture: one difference may be the use of the "register" whereby the "sum" variable is kept in register memory. This reduces the number of accesses significantly. Furthermore, for ascending data a pointer is a superior way to iterate a vector, so perhaps the compiler automatically turns the for-loop into some sort of pointer-based iteration in experiment 2, which it cannot do in experiment 3. Lastly, and this is beyond our understanding, but we must also consider the manner in which a compiler may optimize memory so that it can be physically "crawled" in the cache's array's rows.

4 Conclusions and Future Work

We studied how memory access is a major element in a program's performance. Specifically, we learned that memory access is (in these examples) the main factor in performance. We learned that it is important to be mindful of how memory access is structured, and to reduce the number of accesses generally. Furthermore, we learned that compiler optimization is a crucial tool in writing code for high-performance computing.

Our solution, to use structured memory access and to reduce memory accesses, is part of a larger understanding that structured memory access is superior partially because it can be better optimized by the compiler, among other reasons.

We would personally be interested to know more about the physics of exactly why structured memory is superior as it related to the cache.