Performance Analysis on Cache Effects on Modern Processor Architectures
Assignment #1, CSC 746, Fall 2021

Kory Callanan[*]
SFSU

Abstract

The problem being studied in this paper is cache effects on modern processor architectures. We used summation as major computational operation, and compare the elapsed time between only arithmetic operations (without accessing memory), structured memory accesses operations (using data structure) and unstructured memory accesses operations (pointer chasing). Our results indicated that structured memory accesses operations may use a higher bandwidth and achieve lower latency, as opposed to an unstructured memory access approach. Arithmetic operations without accessing memory result in the highest FLOPs/sec. The results prove our assumption about better performances using a linear structure for computing.

1 Introduction

The objective for this assignment is to study cache effects on modern processor architectures. The problem being studied is performance and optimization level in different configurations for gcc/g++ compiler. We use *-O0* no optimization compared with *-O3* full optimization as the two configurations using a gcc/g++ compiler . Experiments are made as simple and straightforward as possible to show our intention.

The chosen approach for studying this problem is to compare operational methods under no memory access, sequential memory access and random memory access. We experiment using elapsed time for one for-loop as the major measurement. In each iteration, we add up numbers a)Directly one by one b)Sequentially one after another c)Randomly assigned by pointers using the current number as the next number's location indicator.

The main results from the experiments are a) Optimization level affects simple operations more than complicated operations. b) Operations without accessing memory has higher FLOPs rate than with memory access c) On average, roughly around 3% bandwidth being used for summation done in the experiments.

2 Implementation

We used three approaches for the experiments. Each approach is conducted with both no optimization and full optimization setting. Here are the details of each method:

2.1 Arithmetic Approach

This approach uses a simple for-loop with iteration given by the user to compute a sum. We recorded the time it takes for the machine to finish the task so we can get a general idea of what percentage of its resources this machine would allocate for future experiments. We also included a hard-coded version of the same experiment in which the iterations are written in code instead of given by user. 1.

[*]email:lzhou4@mail.sfsu.edu

```
unsigned long long sum = 0;
for (int j=0; j<N; j++)
    sum += j;
```

Listing 1: Pseudo code for pure summation with no memory access.

2.2 Structured Memory Accesses Approach

This approach uses a data structure vector to store the values of a given size, and then uses a for loop to compute the sum. We record the time it takes for the machine to finish the task for different problem sizes and different configurations. 2.

```
// vector<unsigned long long> v, init with
    sequential values
unsigned long long sum = 0;
for (int j=0; j<N; j++)
    sum += v[j];
```

Listing 2: Pseudo code for summation with sequential memory access.

2.3 Unstructured Memory Accesses Approach

In this approach we randomly assign values to a vector with a user given size N. Then we start looping from index 0. We use a pointer *p* to keep track of the current value in v[index], then we assign v[index] to *p* and add up the value at index *p*. Eventually, the loop ends at N iterations. 3.

```
// vector<unsigned long long> v, init with random
    values
unsigned long long sum = 0, p = 0;
for (int j=0; j<N; j++){
    sum += v[p];
    p = v[p];
}
```

Listing 3: Pseudo code for summation with random memory access.

3 Evaluation

Details about each experiment are listed in this section. We performed summations with no memory access via two methods. In the first, we have user input as the problem size, and in the second method we hardcoded the problem size into our program. For experiments with memory accessing, we used user input to specify the problem size.

3.1 Computational platform and Software Environment

We use Cori, the Cray XC40 system which is the NERSC's supercomputer to run our experiment. The Cori system is comprised of 2388 Intel Xeon "Haswell" processor nodes and 9688 Intel Xeon Phi

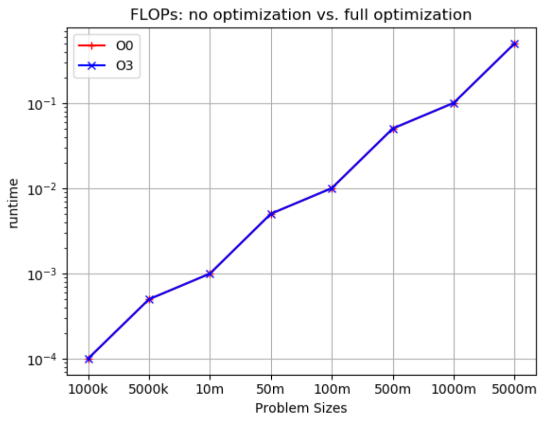

Figure 1: Comparison of no optimization and full optimization runtime with increasing problem sizes without memory access.In this case, we see the observed correlation between is strong. Results from the two configurations are nearly identical. This figure was produced by the sample `plot_config1r.py` file.

Processor 7250 "Knights's Landing"(KNL) nodes. We used KNL nodes to perform the tests. Each node has 68 cores at 1.4 GHz clock rate. Each node also has its own L1 caches, where 64KB consists of 32KB instruction cache and 32KB data. Each node has 96GB DDR-2400 MHz memory with 102GB/s peak bandwidth. [1]

The supercomputer runs on SUSE Linux Enterprise Server 15. For our experiment, we used g++ 7.5.0 to compile and execute the programs. We used the "-DCMAKE" compilation flag to indicate on/off optimization mode. [2]

3.2 Methodology

The procedures we used to test our system are to have our elapsed time library tool wrap around the main computational code for each program, and measure the time difference between the start point and the end point.

We ran tests over a set of prescribed problem sizes, these being 100000, 500000, 1000000, 5000000, 10000000, 50000000, 100000000, 500000000. Observable differences are found from problem sizes.

3.3 Experiment 1: Arithmetic

Using the arithmetic method we tried to answer the question of whether there is a difference between no optimization and full optimization when the computation does not involve memory access. We determined both optimization modes return similar results when the size is provided by user input 1.

However, the runtime for static iteration, i.e. hardcoded, of the two configurations indicates a huge gap between each other. 2.

3.4 Experiment 2: Sequential Memory Access

Using the sequential memory method, we tried to answer the question of whether there is a difference between no optimization and full optimization using sequential computation. We determined both optimization modes result in similarly increasing amounts of run time to compute the program. 3.

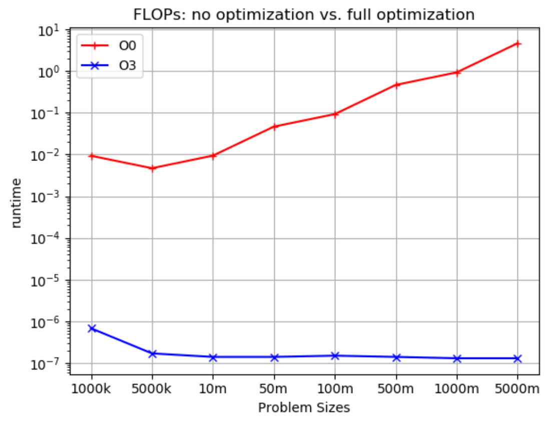

Figure 2: Comparison of no optimization and full optimization runtime with increasing problem sizes without memory access(hardcoded). In this case, we see the observed no optimization configuration has a significant difference than the full optimization runtime. This figure was produced by the sample `plot_config1.5r.py` file.

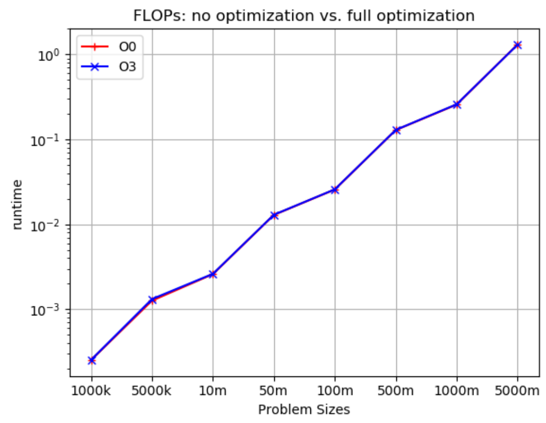

Figure 3: Comparison of no optimization and full optimization runtime with increasing problem sizes with sequential memory access. In this case, we see the observed correlation between is strong. Results from the two configurations are nearly identical. This figure was produced by the sample `plot_config2r.py` file.

[1] https://docs.nersc.gov/systems/cori/

[2] https://caiorss.github.io/C-Cpp-Notes/compiler-flags-options.html

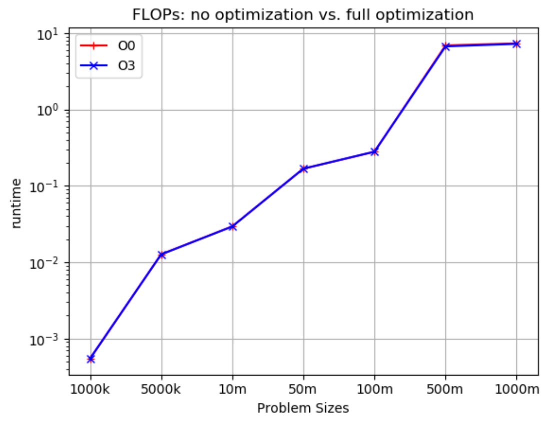

Figure 4: Comparison of no optimization and full optimization runtime with increasing problem sizes with random memory access. In this case, we see the observed correlation between is strong. Results from the two configurations are nearly identical. This figure was produced by the sample `plot_config3r.py` file.

Problem Size (N)	-O0 FLOPs/sec	-O3 FLOPs/sec
100000	993433405.2	993630826.4
500000	998633868.9	998733605.8
1000000	999150721.9	999320462.1
5000000	994318464.3	990862268.2
10000000	992654357.8	993808572.6
50000000	995127854	991268903.5
100000000	994174139.5	994510303.1
500000000	996484403	994941716.3

Table 1: Comparison of FLOPs/sec in O0 and O3 optimization for different problem sizes in arithmetic method.

Problem Size (N)	-O0 FLOPs/sec	-O3 FLOPs/sec
100000	396083526.1	394614304
500000	395050803.5	378813707
1000000	385074511.9	382043935.1
5000000	387260672.9	385139767.2
10000000	388557751.3	386748451.1
50000000	387681047	386165989.6
100000000	388281659.5	385328241.9
500000000	380048950.3	382216242.7

Table 2: Comparison of FLOPs/sec in O0 and O3 optimization for different problem sizes in sequential memory access method.

3.5 Experiment 3: Random Memory Access

Using the random memory method we tried to answer the question of whether there is a difference between no optimization and full optimization when using pointer chasing, non-sequential computation. We determined both optimization modes result in similarly increasing amounts of run time to compute the program. 4.

3.6 Findings and Discussion

3.6.1 FLOPs/second

The number of FLOPs/second we were able to realize in this paper is between 0.14 and 0.99 GFlops/s. We used the arithmetic program to derive this figure as it produces the highest FLOPs/second. In the measurement of FLOPs, memory access is a significant factor. Experiments without memory access significantly increased the FLOPs/sec. 1 Having to access memory, each floating point operation takes longer time to complete as opposite to simply rendering

Problem Size (N)	-O0 FLOPs/sec	-O3 FLOPs/sec
100000	182763567.9	183165279.2
500000	39448665.45	39541320.68
1000000	34010135.02	34094667.25
5000000	29838098.48	29834003.6
10000000	35837926.56	35761669.93
50000000	7262227.412	7484424.912
100000000	13638787.62	13866125.33

Table 3: Comparison of FLOPs/sec in O0 and O3 optimization for different problem sizes in random memory access method.

Problem Size(N)	s-O0(MB/s)	s-O3(MB/s)	r-O0(MB/s)	r-O3(MB/s)
100000	3168	3157	2924	2930
500000	3160	3030	631	632
1000000	3080	3056	5441	545
5000000	3098	3081	477	477
10000000	3108	3093	573	572
50000000	3101	3089	116	119
100000000	3106	3082	218	221
500000000	3040	3057	-	-

Table 4: Comparison of bandwidth in O0 and O3 optimization for different problem sizes in sequential memory access method and random memory access method.

Problem Size(N)	# mem(GB)/s	-O0 latency(ps)	-O3 latency(ps)
100000	3.17	39.4	39.6
500000	3.16	39.6	41.2
1000000	3.08	40.6	40.9
5000000	3.09	40.3	40.6
10000000	3.10	40.2	40.4
50000000	3.10	40.3	40.5
100000000	3.11	40.2	40.5
500000000	3.04	41.1	40.9

Table 5: Comparison of latency in O0 and O3 optimization for different problem sizes in sequential memory access method.

numbers sequentially on its own. 2 Different configurations do not result in differentiated FLOPs/second in random memory access method. At this point, run time (FLOPs speed) is highly affected by the amount of time spent on accessing memory.

My FLOPs/second rate compare to the theoretical peak for the Intel Knights Landing processors on Cori is 0.99/44.8 GFlops which is 2.23%.

3.6.2 Memory bandwidth

As listed 4, we observed that sequential method at problem size 100000 without optimization reaches maximum memory bandwidth at about 3.2GB/s. Also with no optimization, the pointer chasing method obtains the minimum memory bandwidth at about .1GB/s. Since no optimization is designed for debugging, it's theoretically faster in compilation and better at utilizing resources. Compare to the maximum theoretical peak on Cori KNL nodes, which is 102GB/s, the max utilization of our data makes up 3%, which is also the average memory bandwidth our experiments perform.

3.6.3 Latency

Memory latency has a direct impact on memory access speed, the lower the latency the faster memory access. In the sequential method, every iteration takes half of the times of memory access since there is only one operation. 5 However, the random method reassigns the pointer at each iteration. So theoretically, latency for the random method should be at least twice as high as the sequential method.

From both table 5 and 6, we can observe that no optimization produces the best number of memory accesses per second/lowest

Problem Size(N)	# mem(GB)/s	-O0 latency(ps)	-O3 latency(ps)
100000	2.92	42.7	42.7
500000	0.63	198	198
1000000	0.54	230	229
5000000	0.48	262	262
10000000	0.57	218	218
50000000	0.12	1080	1040
100000000	0.22	573	563

Table 6: Comparison of latency in O0 optimization and O3 for different problem sizes in random memory access method.

latency at problem size 100000 using the sequential method. Additionally, no optimization produces the worst number of memory accesses per second/highest latency at problem size 50000000 using the random method. These result suggests that no optimization configuration shows the direct behavior of the machine whereas full optimization configuration, in the majority of cases, helps reduce latency to improve performance.

4 Conclusions and Future Work

- The question we were trying to answer was how different configurations for gcc/g++ compilers affect the performance and optimization of computational operations, with major factors of memory access and user-input vs hard-coded runtime.

- We found that user input vs hard-coded runtime has a significant impact on speed, and also that there are significant differences in speed with different optimizations.

- The results we achieved serve as a guide for understanding how to optimize any future programs on the Cray XC40 supercomputer for speed. In future papers, we can further explore details of high performance computing with this base knowledge.