Homework Report
Assignment #1, CSC 746, Fall 2021

Peter Ijeoma *

SFSU ID: 921373073

Abstract

Evaluate the run-time for summing sequential integers from 0 to N-1; N is the specified problem size. The run-times are recorded for two compiler optimization levels: O0 (no optimization) and O3 (maximum optimization) and also for different modes of accessing memory namely sequential versus random memory access. The results are tabulated and plotted to compare performance at both optimization levels and also to compare the same summation with no memory access.

1 Introduction

The problem we have solved

- sum integers between 0 and N. N is the problem size. The user provides N as an argument to the program. If no argument is provided, a default problem size is used.Eight problem sizes are of interest: 100000, 500000, 1000000, 5000000, 10000000, 50000000, 100000000, 500000000
- Solve the problem in 3 memory models: using no memory, apply sequential memory access, randomly access memory. For random memory access, the value of each memory location is the address of the next location to access.
- Compile and run the solutions with zero optimization and maximum optimization.
- Plot the performance of each solution for zero optimization and maximum optimization for each of the 3 memory models

From the results, I will attempt to answer the following questions:

- What is the number of FLOPs/second you are able to realize in this assignment? Which of the 3 programs are you using to derive this figure, and why?
- What is the min/max/average memory bandwidth you are able to realize in this assignment? And in which of the 3 programs does this happen, and why?
- Produce a table showing the number of memory accesses/second (one column) and an estimate of memory latency (a second column) for the codes in Part 2 and Part 3 at different problem sizes.
- Which configuration produces the best number of memory accesses per second/lowest latency? Why?
- Which configuration produces the worst number of memory accesses per second/highest latency? Why?

This report presents what I have done to solve the problem, my implementation in Section 2. In Section 3, I present the results, and provide answers to the problem questions in Section 4 and finally, a conclusion.

*e-mail: pijeoma@mail.sfsu.edu

2 Implementation

The solution is composed of three code files; one for each memory model. For each code file, I have accessed the command line argument. If no argument is provided, I used a problem size of 10000 (hard coded). The code is in C++ and code files are:

2.1 Part 1

simple_sum_timer.cpp - there is no memory access for this code. I have used a for loop to iterate from 0 to "problem size - 1" and add the each value (0 through problem size - 1) to an accumulator variable.

```
    // sum from 0 to N - 1 (loop size) in a loop
    for (long i = 0; i < loop_size; i++)
    {
        sum_accm += i;
    }
```

Listing 1: No Memory Access

2.2 Part 2

array_sum_timer.cpp - Here I have used an array as the data structure to store in memory. The array is dynamically allocated memory (to avoid stack overflow). Each array element is assigned the value of its index. That is the ith element has value i. Then sequentially, I accessed the array from element at index 0 to last element and add the value in each element to an accumulator variable.

```
    // sum array elemnts in sequence
    for (long i = 0; i < array_size; i++)
    {
    //std::cout << " adding array value to sum for
     index " << i << std::endl;
        sum_accm += a_var[i];
    }
    //std::cout << " done with summation " << std
    ::endl;
```

Listing 2: Sequential Memory Access

2.3 Part 3

random_sum_timer.cpp - Here again, I used an array as the data structure. The array is dynamically allocated memory and each array element initialized with a random random integer between 0 and problem size. Then I randomly accessed the array from element at index 0 until I have made problem size iterations.The value in each array element is the index of the next array element to access and added to an accumulator variable.

```
    // sum array elemnts at random starting at
    index 0
    long arr_ind = 0;
    for (long i = 0; i < arr_itn_size; i++)
    {
    //std::cout << " random access of array value
    for index " << arr_ind << std::endl;
        sum_accm += a_var[arr_ind];
        arr_ind = a_var[i]; // next index to
    access
    }
    //std::cout << " done with sum " << std::endl;
```

Listing 3: Random Memory Access

3 Evaluation and Results

The three programs in this solution were tested separately for each problem size including the default problem size of 10000. Each program was run once for each optimization level. The results collected during the test on my laptop are presented below: a table showing the raw data and a plot of elapsed time by problem size.

3.1 Computational platform and Software Environment

The test was done on a Linux VM on my laptop. The VM is configured as follows:

- OS: Ubuntu (64 - bit)
- 1 CPU 2.70 GHz (6 cores)
- Memory: 8192 MB DDR4
- 12 MB cache

The programs were compiled using C++ compiler version 9.3.0 on Ubuntu. For maximum optimization, no compilation flags were specified. But for zero optimization the flags were set to -DCMAKE_CXX_FLAGS_RELEASE="-O0".

3.2 Methodology

I used a basic shell script to run the programs in the sequence simple_sum_timer, array_sum_timer, and random_sum_timer (the order has not real meaning to the test); for each of the problem sizes mentioned in 1 above.

Performance metrics: The performance metrics applied is elapsed time. The main computational code statements are wrapped between statements that record the start and end times and the difference is the elapsed time.

3.3 Experiment

The key experiment of this exercise was to find out:

- the effect of memory latency on performance
- the effect of how memory is accessed - sequential versus random

on performance.

The problem sizes tested and the questions to be answered are as stated in 1 above.

The results obtained are presented below in tables and graphically (for easy visualization) for each mode of accessing memory.

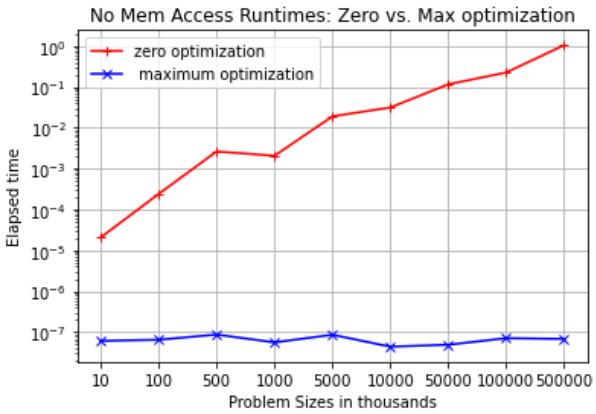

Figure 1: Plot for no memory access

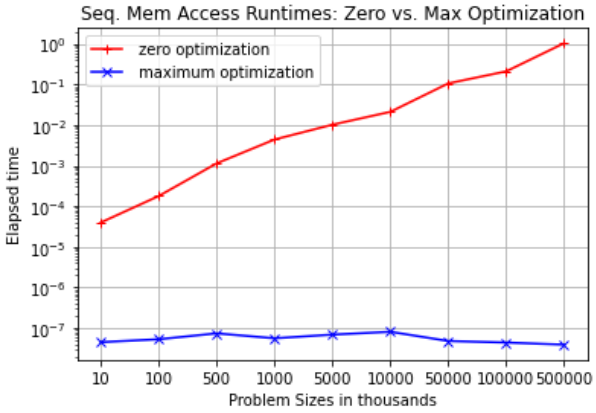

Figure 2: Plot for sequential memory access

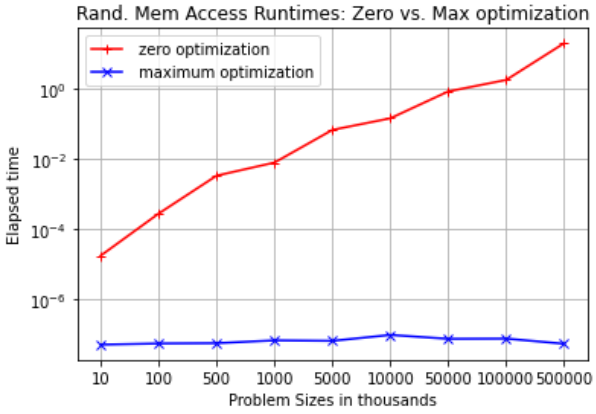

Figure 3: Plot for random memory access

Problem Size (N)	No Opt runtime (sec)	Max Opt runtime (sec)
10000	2.0823e-05	6.1e-08
100000	0.000241323	6.5e-08
500000	0.00264828	8.7e-08
1000000	0.00208102	5.6e-08
5000000	0.0192129	8.6e-08
10000000	0.0316001	4.4e-08
50000000	0.116772	4.9e-08
100000000	0.227564	7.1e-08
500000000	1.06203	6.8e-08

Table 1: No memory access runtimes

Problem Size (N)	No Opt runtime (sec)	Max Opt runtime (sec)
10000	3.9647e-05	4.5e-08
100000	0.000177994	5.3e-08
500000	0.00113441	7.4e-08
1000000	0.00439684	5.6e-08
5000000	0.010198	6.9e-08
10000000	0.0211224	8.1e-08
50000000	0.105929	4.8e-08
100000000	0.20986	4.4e-08
500000000	1.0366	3.9e-08

Table 2: Sequential memory access runtimes

Problem Size (N)	No Opt runtime (sec)	Max Opt runtime (sec)
10000	1.7191e-05	4.9e-08
100000	0.000274277	5.3e-08
500000	0.00332322	5.4e-08
1000000	0.00788109	6.5e-08
5000000	0.0676813	6.3e-08
10000000	0.145201	9.3e-08
50000000	0.851943	7.2e-08
100000000	1.81534	7.3e-08
500000000	20.3939	5.2e-08

Table 3: Random memory access runtimes

3.4 Findings and Discussion

There were specific questions that needed to be answered for this report. Below are the answers to each question.

- Question: What is the number of FLOPs/second you are able to realize in this assignment? Which of the 3 programs are you using to derive this figure, and why?
- Answer: there are 4 (1 addition, 1 comparison, 1 increment, 1 assignment) times problem size plus 1 initialization operations. That is, 4 * ¡problem size¿ + 1 operations for each run of the sequential memory access program. This gives on average, 1.72E+09 FLOPS with the no optimization and 7.27E+15 FLOPS with maximum optimization. I have used the sequential memory access program for this analysis because it is representative of the other programs.
- Question: What is the min/max/average memory bandwidth you are able to realize in this assignment? And in which of the 3 programs does this happen, and why?
- Answer: the minimum memory bandwidth is: 1.13E+14; the maximum memory bandwidth is: 9.81E+13; the average memory bandwidth is: 8.69E+16. The minimum bandwidth occurs in the random memory access program while the maximum memory bandwidth occurs in the sequential memory access program. Interestingly, the minimum and maximum bandwidths occurred for the same problem size.
- Produce a table showing the number of memory accesses/second (one column) and an estimate of memory latency (a second column) for the codes in Part 2 and Part 3 at different problem sizes.

Problem Size (N)	Mem access per sec	Estimated latency
10000	1.56E+12	6.43E-13
100000	1.32E+13	7.57E-14
500000	4.73E+13	2.11E-14
1000000	1.25E+14	8.00E-15
5000000	5.07E+14	1.97E-15
10000000	8.64E+14	1.16E-15
50000000	7.29E+15	1.37E-16
100000000	1.59E+16	6.29E-17
500000000	8.97E+16	1.11E-17

Table 4: Sequential access: Mem access per sec and Estimated latency

Problem Size (N)	Mem access per sec	Estimated latency
10000	2.04E+12	4.90E-13
100000	1.89E+13	5.30E-14
500000	9.26E+13	1.08E-14
1000000	1.54E+14	6.50E-15
5000000	7.94E+14	1.26E-15
10000000	1.08E+15	9.30E-16
50000000	6.94E+15	1.44E-16
100000000	1.37E+16	7.30E-17
500000000	9.62E+16	1.04E-17

Table 5: Random access: Mem access per sec and Estimated latency

- Question: Which configuration produces the best number of memory accesses per second/lowest latency? Why?
- Answer: Random memory access for the largest problem size produced the best memory access per second of 9.62E+16 (lowest latency of 1.04E-17). The reason for this may be that certain hardware optimizations become more apparent. Optimizations like pre-fetch and caching would have great impact as the problem size increases.
- Question: Which configuration produces the worst number of memory accesses per second/highest latency? Why?
- Answer: Sequential memory access for the least problem size has the worst memory access per second of 1.56E+12 (highest latency of 6.43E-13). We may think that sequential memory access would perform better with caching - in which case we could cache contiguous memory locations and improve performance. This would be the case only if memory allocation is contiguous. in these programs I used dynamic memory allocation which in this case might not have been contiguous.

4 Conclusions

- This experiment test the effect of optimization and memory access pattern (random or sequential) on performance.
- It is my observation that memory access pattern does not impact performance as much as optimization.