# Memory Accesses
# Assignment #1, CSC 746, Fall 2021

Arulselvan Madhavan[*]
San Francisco State University

## Abstract

This paper presents a study of the impact of memory accesses on the execution time(FLOPs) of a program. Modern processors are bound by memory accesses than CPU operations. So, it's important to understand the limits of memory layers to be predict the execution time of programs and to extract maximum performance out of the machine in which they run on. This paper used the summation function and tried three memory common access scenarios - no memory accesses, sequential memory accesses and random memory accesses. The results show the importance of reducing memory accesses and that sequential memory access should be preferred over random memory accesses, to get maximum performance out of the machines.

## 1 Introduction

This paper shows how the execution characteristics of a simple program to sum 'N' numbers depends on the memory access pattern of the implementation. Three different memory access patterns were studied as part of this work.

- Calculate sum without any memory access
- Calculate sum with sequential memory accesses
- Calculate sum with random memory accesses

Modern processors are bound by memory accesses than CPU. As memory accesses dominate the performance characteristics of programs it becomes increasingly important to understand the latencies and bandwidth associated with different layers of memory in the machine in which the program is run on. Understanding the memory access bounds of a machine helps in understanding and reasoning about the deviation that programs see from the roofline performance model(theoretical peak). This paper show how one such program's performance is directly correlated with its memory access patterns.

- This paper shows the correlation between execution characteristics of a program and its memory access pattern in the Intel Xeon Phi "Knight's Landing" (KNL) machine. The KNL machine was part of a 9688 node supercomputer(Cori) with a theoretical peak of 44.8 GFlops/core.
- Improving the clock cycle can only get us so far when the program is limited by memory accesses. Modern powerful processors are invariably limited by memory accesses than CPU performance or clock cycles.

The work on understanding the latencies and bandwidth for memory is important in predicting the performance of long running programs. This paper provides a systematic way of measuring such details which could lead to more accurate predictions of runtime of long running programs.

- The paper shows the stark difference between three different memory access patterns.
- The experiment and the observed results show the stark difference in performance between memory accesses from different layers of the memory hierarchy. Structured memory access(heavily utilizing L1 cache) beating unstructured memory accesses that has to reach L2 cache/DRAM far more often.

The rest of this paper first discusses related work in Section 2, and then describes our implementation in Section 3. Section 4 describes how we evaluated our system and presents the results. Section 5 presents our conclusions and describes future work. More focus is presented on analyzing the results which, I believe, will be the most important contribution of this paper.

## 2 Related Work

Most of the information and claims related to memory accesses, latencies, bandwidth and bounds found their roots in the textbook Hennessy and Patterson [1].

Similar efforts on analyzing machine characteristics have been performed in the past and is still widely employed to discuss and compare the performance profiles of different machines and their architectures. Similar analysis efforts are also employed in industries where performance is paramount like High Frequency Trading, large scale predictions etc.,

- The increase in commodity hardware and Infrastructure as a Service solutions has moved analysis efforts like this out of the in-house software engineers and into the hands of performance specialists. Large amount of tools have come out of the industry that do similar analysis. For example, perf, pprof, etc.,
- Intel's VTune analyzer and the likes have come very close to making actual suggestions on improving your code for better performance.
- A variety of performance visualization tools have also come into existence. Flamegraph being the most notable one.

There are performance profiling tools that use various techniques to profile the CPU usage, memory allocations when the program is running. Most of these techniques have some impact on the runtime even if the performance events were sampled. The performance measurements used in this paper were intentionally simple and no such runtime performance metrics were collected.

## 3 Implementation

The program that was studied as part of this paper was a simple summation function. However, the size of the sum was unbounded i.e leaving the problem size as an input from the command line. Three different memory access patterns were studied - No memory access, Structured memory access and Unstructured memory access. Since modern compilers have sophisticated compiler optimizations, two compilation modes were also studied - no compiler optimizations and full compiler optimizations.

[*]email:amadhavan@sfsu.edu

- Size of the summation function was studied with 7 different values - 100000, 500000, 1000000, 5000000, 10000000, 50000000 , 100000000
- The experiments below show the results of studying the summation function compiled with two different compilation modes and with each of these problem sizes under three different memory access patterns

### 3.1 No Memory Access (NM)

- Compute sum without any new memory accesses inside the loop
- The objective of this setup is to exclude memory accesses as a factor in the execution time of the summation program

```
for (int i = 0; i < problem_size; i++) {
  sum += i;
}
```

Listing 1: Performs summation without any memory accesses

### 3.2 Structured Memory Access(SM)

- Computes sum by accessing elements in order
- The objective of this setup is to study the impact of cache lines in the execution of the summation program

```
for (int i = 0; i < problem_size; i++) {
  sum += vect[i];
}
```

Listing 2: Performs summation by accessing memory sequentially

### 3.3 Unstructured Memory Access(UM)

- Computes sum by accessing elements in random order
- Unix utility function lrand48 was used with a seed set to the current time
- The objective of this setup is to study the impact of cache misses on the execution time of the summation program

```
for (int i = 0; i < problem_size; i++) {
  sum += vect[ptr];
  ptr = vect[ptr];
}
```

Listing 3: Performs summation by accessing memory in random order

## 4 Evaluation

The program was compiled using GCC 7.5 on a Intel Knight Landing Processor machine. Each experiment setup was compiled and evaluated under two modes.

- No compiler Optimization
- Full compiler Optimization

### 4.1 Experiment Setup

- The experiment was run on one of the Intel Knights Landing Processors connected to Cori supercomputer
- Flops (Theoretical Peak) - 44.8 GFlops/Core
- Bandwidth (Theoretical Peak) - 102 GiB/s
- Clock rate - 1.4 GHz
- L1 caches - 64 KB (32 KiB instruction cache, 32 KB data)
- L2 cache - 1MB

### 4.2 Findings and Discussion

The results showed the impact of memory accesses on the execution of the program

- The program without any memory accesses clearly outperformed other two implementations of summation
- The program with structured memory accesses outperformed the program that accessed memory randomly
- Max number of FLOPs achieved 0.12 GFlops. This is just 0.2 percent of the theoretical peak. The program that achieved this was the program that didn't have any memory accesses. Refer Table 1
- The Min, Max, Avg bandwidth realized in these experiment setups are given in Table 4
- The Max bandwidth realized was 1.5 GiB/sec. This was achieved by the program(NM) that didn't access any new memory in the loop. This is 1 percent of the peak bandwidth provided by KNL.
- Table 5 and Table 6 shows how latency and memory accesses correlated in the programs that did structured memory accesses and unstructured memory accesses. It's clear from this table that structured memory access provides lower latency. This is because of the increased cache hits that it gets by sequentially accessing information. Therefore, Maximum accesses(48000000) and minimum latency(0.21ns) is achieved in the program that accesses memory sequentially.
- The configuration that provides the best memory accesses and lowest latency is the one that doesn't do any memory access. Between sequential memory access and random memory access, sequential memory access performs better. The configuration that works the best is when it's compiled with full optimizations(O3) turned ON.
- The configuration that performs the worst is the unstructured memory access program with no optimizations(O0). Memory accesses dropped to 17000000 and latency increased to 1600ns
- Figures 1, 2, 3 shows the graph of the runtime over different problem sizes. The compiler optimizations start to show a difference when the size is really high, as evident in Figure 3

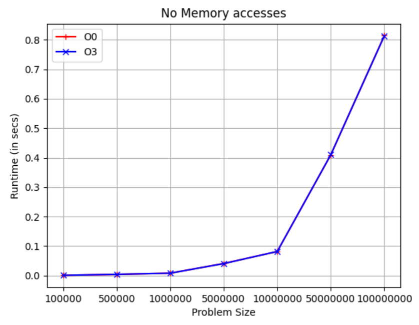

Figure 1: Comparison of compiler optimizations - No Memory accesses

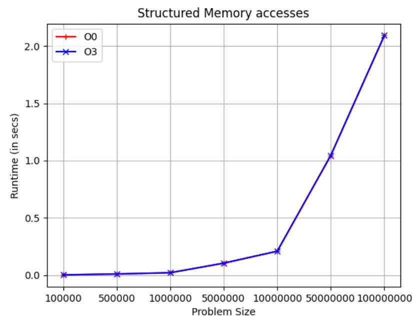

Figure 2: Comparison of compiler optimizations - Structured Memory accesses

| Size | NM (flops) | | SM (flops) | | USM (flops) | |
|---|---|---|---|---|---|---|
| | O0 | O3 | O0 | O3 | O0 | O3 |
| 1 | 1.2e+08 | 1.2e+08 | 4.8e+07 | 4.8e+07 | 4.6e+07 | 4.4e+07 |
| 5 | 1.2e+08 | 1.2e+08 | 4.8e+07 | 4.8e+07 | 4.5e+07 | 2.7e+07 |
| 10 | 1.2e+08 | 1.2e+08 | 4.8e+07 | 4.8e+07 | 2.5e+07 | 4.6e+07 |
| 50 | 1.2e+08 | 1.2e+08 | 4.8e+07 | 4.8e+07 | 1.8e+07 | 2.1e+07 |
| 100 | 1.2e+08 | 1.2e+08 | 4.8e+07 | 4.8e+07 | 1.9e+07 | 9.5e+06 |
| 500 | 1.2e+08 | 1.2e+08 | 4.8e+07 | 4.8e+07 | 1.5e+07 | 8.3e+06 |
| 1000 | 1.2e+08 | 1.2e+08 | 4.8e+07 | 4.8e+07 | 3.2e+06 | 1.5e+07 |

Table 1: Comparison of FLOPS. Size is expressed in $10^5$

| Size | NM (bytes/sec) | | SM (bytes/sec) | | USM (bytes/sec) | |
|---|---|---|---|---|---|---|
| | O0 | O3 | O0 | O3 | O0 | O3 |
| 1 | 1.9e+04 | 1.9e+04 | 3.9e+08 | 3.9e+08 | 7.3e+08 | 7.0e+08 |
| 5 | 4.0e+03 | 3.8e+03 | 3.8e+08 | 3.8e+08 | 7.3e+08 | 4.3e+08 |
| 10 | 2.0e+03 | 2.0e+03 | 3.8e+08 | 3.8e+08 | 4.0e+08 | 7.4e+08 |
| 50 | 3.9e+02 | 3.9e+02 | 3.8e+08 | 3.8e+08 | 2.8e+08 | 3.3e+08 |
| 100 | 2.0e+02 | 2.0e+02 | 3.8e+08 | 3.8e+08 | 3.1e+08 | 1.5e+08 |
| 500 | 3.9e+01 | 3.9e+01 | 3.8e+08 | 3.8e+08 | 2.4e+08 | 1.3e+08 |
| 1000 | 2.0e+01 | 2.0e+01 | 3.8e+08 | 3.8e+08 | 5.0e+07 | 2.4e+08 |

Table 2: Comparison of bytes accessed. Size is expressed in $10^5$

| Size | NM (secs) | | SM (secs) | | USM (secs) | |
|---|---|---|---|---|---|---|
| | O0 | O3 | O0 | O3 | O0 | O3 |
| 1 | 4.1e-04 | 4.1e-04 | 2.1e-08 | 2.1e-08 | 1.1e-08 | 1.1e-08 |
| 5 | 2.0e-03 | 2.1e-03 | 2.1e-08 | 2.1e-08 | 1.1e-08 | 1.9e-08 |
| 10 | 4.0e-03 | 4.0e-03 | 2.1e-08 | 2.1e-08 | 2.0e-08 | 1.1e-08 |
| 50 | 2.1e-02 | 2.0e-02 | 2.1e-08 | 2.1e-08 | 2.8e-08 | 2.4e-08 |
| 100 | 4.1e-02 | 4.1e-02 | 2.1e-08 | 2.1e-08 | 2.6e-08 | 5.3e-08 |
| 500 | 2.0e-01 | 2.0e-01 | 2.1e-08 | 2.1e-08 | 3.3e-08 | 6.0e-08 |
| 1000 | 4.1e-01 | 4.1e-01 | 2.1e-08 | 2.1e-08 | 1.6e-07 | 3.4e-08 |

Table 3: Comparison of latencies. Size is expressed in $10^5$

| | NM (bytes/sec) | | SM (bytes/sec) | | USM (bytes/sec) | |
|---|---|---|---|---|---|---|
| | O0 | O3 | O0 | O3 | O0 | O3 |
| Min | 1.5e+09 | 7.6e+08 | 8.8e+07 | 1.4e+09 | 7.6e+08 | 2.3e+08 |
| Max | 1.5e+09 | 7.7e+08 | 1.3e+09 | 1.5e+09 | 7.7e+08 | 1.3e+09 |
| Avg | 1.5e+09 | 7.6e+08 | 6.8e+08 | 1.5e+09 | 7.7e+08 | 6.8e+08 |

Table 4: Comparison of Bandwidths

| Structured Mem | | | |
|---|---|---|---|
| Accesses/Sec | | Latency | |
| O0 | O3 | O0 | O3 |
| 4.8e+07 | 2.1e-08 | 4.8e+07 | 2.1e-08 |
| 4.8e+07 | 2.1e-08 | 4.8e+07 | 2.1e-08 |
| 4.8e+07 | 2.1e-08 | 4.8e+07 | 2.1e-08 |
| 4.8e+07 | 2.1e-08 | 4.8e+07 | 2.1e-08 |
| 4.8e+07 | 2.1e-08 | 4.8e+07 | 2.1e-08 |
| 4.8e+07 | 2.1e-08 | 4.8e+07 | 2.1e-08 |
| 4.8e+07 | 2.1e-08 | 4.8e+07 | 2.1e-08 |

Table 5: Comparison of Accesses and Latency

## 5 Conclusions and Future Work

This paper studied the impact of memory accesses on the execution time of programs and showed that memory accesses are one of the major bottlenecks to realizing the maximum FLOPs

- Programs can increase number of FLOPs by reducing the number of memory accesses.
- Accessing memory sequentially provides more cache hits and improves bandwidth and drops latency

| Unstructured Mem | | | |
|---|---|---|---|
| Accesses/Sec | | Latency | |
| O0 | O3 | O0 | O3 |
| 9.2e+07 | 1.1e-08 | 8.8e+07 | 1.1e-08 |
| 9.1e+07 | 1.1e-08 | 5.3e+07 | 1.9e-08 |
| 5.0e+07 | 2.0e-08 | 9.3e+07 | 1.1e-08 |
| 3.5e+07 | 2.8e-08 | 4.1e+07 | 2.4e-08 |
| 3.8e+07 | 2.6e-08 | 1.9e+07 | 5.3e-08 |
| 3.0e+07 | 3.3e-08 | 1.7e+07 | 6.0e-08 |
| 6.3e+06 | 1.6e-07 | 3.0e+07 | 3.4e-08 |

Table 6: Comparison of Accesses and Latency

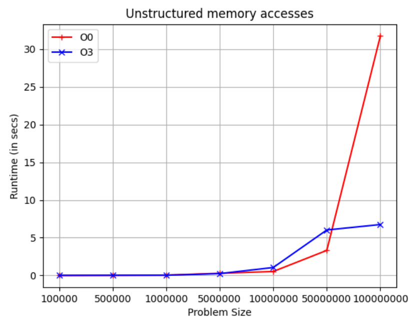

Figure 3: Comparison of compiler optimizations - Unstructured Memory accesses

- Compiler optimizations are limited but can certainly help in improving the FLOPs depending on the program and the problem size

Understanding the maximum limits of a machine is important in predicting the runtime of programs. This paper presents one such way to understand the impacts of memory accesses on FLOPs in Intel's KNL processor. Once we understand the behavior of the architecture, we can write code targeting the architecture behavior to extract maximum performance.

## References

[1] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th ed., 2011.