CSE240B WI20

Characterize the Cache Performance of a CPU

Swapnil Aggarwal – A53271788 – swaggarw@ucsd.edu

Introduction

The project aims at characterizing the performance of the memory hierarchy of a CPU, right from the caches down to the main memory. There are typically three levels of caches – L1, L2 and L3. Cache Memory is a very high-speed memory which is used to speed up and synchronize with high-speed CPU. It is costlier than main memory or disk memory but more economical than CPU registers. It is an extremely fast memory type that acts as a buffer between RAM and the CPU. It holds frequently requested data and instructions so that they are immediately available to the CPU when needed. L1 cache is the closest and fastest cache for the CPU, but also with the smallest memory size, usually only few KB (16, 32 or 64KB). Furthermore, it is divided into instruction and data cache, for caching instructions and data separately. We will only focus on data cache for this project. The next level is L2 cache which is usually bigger than L1 and unified (not separated into I Cache and D cache). Moreover, each core has a L1 and L2 cache, but the next level, i.e. L3 cache, is a unified shared cache across all processors. The next level is the main memory (RAM) which handles most of the bulk data storage. This is usually very large in size (few GB).

Documentation

In order to measure memory performance, we utilize the pointer chasing technique. Pointer chasing refers to a common sequence of instructions that involves a repeated series of regular/irregular memory access patterns that require the accessed data to determine the subsequent pointer address to be accessed, forming a serially dependent chain of loads. The program initially allocates a chunk of memory of defined size and then tries to access the memory as randomly as possible, so that accesses end in all levels of the cache. The data structure used is a *circularly chained linked list*, in which the respective address indices are shuffled and linked randomly to prevent sequential access. Afterwards, the entire memory is accessed a fixed number of times and the access time is recorded, and from this, the average access time for each operation is calculated. As the memory allocation is increased, the average access time changes as more reads and/or writes start ending up in the L3 cache and/or main memory.

In order to ensure complete randomness, a Mersenne Pseudo-Random Generator is used to generate the seed for the uniform random integer distribution, which generates the final address mapping.

Algorithm:

I. Allocate memory of the fixed buffer size (in B) of ‘void pointer’ type.
II. Allocate memory for the buffer indices of same length of linked list.
III. Initially, sequentially assign buffer indices to their respective integer position values, i.e. index[0] = 0, index[1] = 1, etc.

IV. Shuffle the index values randomly or in a stride fashion:
   a. **Random Shuffling by iteration**
      i. Use Mersenne generator (mt19937) and uniform distribution to generate an index between 0 and remaining elements.
      ii. Swap the i'th index with the random index, if they are not equal.
      iii. Iterate for all indices
   b. **Sequential Strides by iteration**
      i. Assign index of i'th element to an index which is 'stride' pointer locations apart (stride is an integer). Use '% {buffer length}' for linking circularly.

V. Link memory elements by assigning (i+1)'th buffer indexed element to i'th element as follows:

```
for(size_t i=0; i < buffLength - 1; i++) {

bufferMem[buffIdx[i]] = (void*) bufferMem[buffIdx[i+1]];}
```

VI. Link last pointer to first pointer to make it a circularly linked list:

```
bufferMem[buffIdx[buffLength - 1]] = (void *) &bufferMem[buffIdx[0]];
```

VII. Utilize pointer chasing for memory access:
   a. Record start time before pointer chasing.
   b. If doing read-modify-write access, do the following:
      i. Read current pointer and next pointer.
      ii. Modify current pointer as explained in the project writeup
      buffPtr = (void**) ((uintptr_t)temp & 0x7fffffffffffffff);
      iii. Assign the current pointer the next pointer value.
      iv. Iterate until the entire memory has been accessed
   c. Else if doing only read access:
      i. Assign the current pointer the next pointer value.
      ii. Iterate until the entire memory has been accessed.
   d. Record end time and return total time difference in seconds.
   e. Assign volatile variable to memory buffer to prevent compiler optimization:
      noPrefetch = *buffPtr;
      where *noPrefetch* is a global volatile void pointer.

VIII. Calculate and record average total time from all memory access runs.

Working and Compiler Options

The given Makefile can be used to compile the C++ file, cachetime.cpp. The given python script 'cachetime.py' calls the 'cachetime.out' executable file with the appropriate compiler options as follows:

Option	Default	Description
-t	1	Number of threads
-s	-	Sequential access
-ss	64	Stride size in Bytes
-r	-	Random Access
-m	-	RMW operation
-p	-	Make All Plots

The '-t' option specifies the number of threads (1, 2, 4, 8, 16). The '-s' option specifies sequential access and '-r' option specifies random access. If none of them is specified, default option is random access. The '-ss' option specifies stride size in Bytes and is applicable only in the case of sequential access. The '-m' option specifies RMW (Read-Modify-Write) operation. The '-p' options generates plots after all CSV files have been generated.

In order to generate all the results, you can execute the script file '*runs.sh*'. This runs all the possible configurations, stores latency data in folder '*csvData'* and displays output of each run. After the run is completed, you can run cachetime.py with the '-p' option to generate all plot files in 'plots' folder.

Two compile time options have been used: '-std=c++11' (C++ 2011 standard) and '-fopenmp' (OpenMP multithreading directive). For multithreaded access, each thread accesses the memory independently but accumulates the final total time atomically.

The following shows the CPU configuration. Note that since the project was moved to personal computers, the configuration available was of 4 cores (supporting a maximum of 8 threads). However, results with 16 threads have also been shown for completeness, although 16 threads undergo thread scheduling and therefore are not indicative of true cache performance.

Table 1: CPU Configuration

Specs	Core/Thread Configuration
Processor	Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
L1 Cache	32KB (Per Core, Non-Unified)
L2 Cache	256KB (Per Core, Unified)
L3 Cache	8MB (Shared, Unified)
Number of Cores	4
Maximum number of threads	8 (with SMT)

The script file '*doit.sh*' runs the program for random access for read only and read-write cases with 1, 2, 4, 8 and 16 threads, and displays the output.

Limitations

There are several limitations of this code. Firstly, the random access may end up hitting only 1 level of the cache, if a memory location points to itself. In this case, the latency will not be an ideal representation of the entire memory hierarchy, since the code may only hit lower levels of the cache. Secondly, the program may not scale in a similar fashion for a configuration greater than 8 threads, as should be visible in the following graphs with 16 thread configurations. The results will become irregular due to extensive thread scheduling, which may be non-deterministic in terms of memory access. Therefore, only a machine with enough main memory will work well with many threads.

Graphical Analysis

The following Figure 1 and Figure 2 show the read latency with 1 threads and multiple threads respectively.

Figure 1

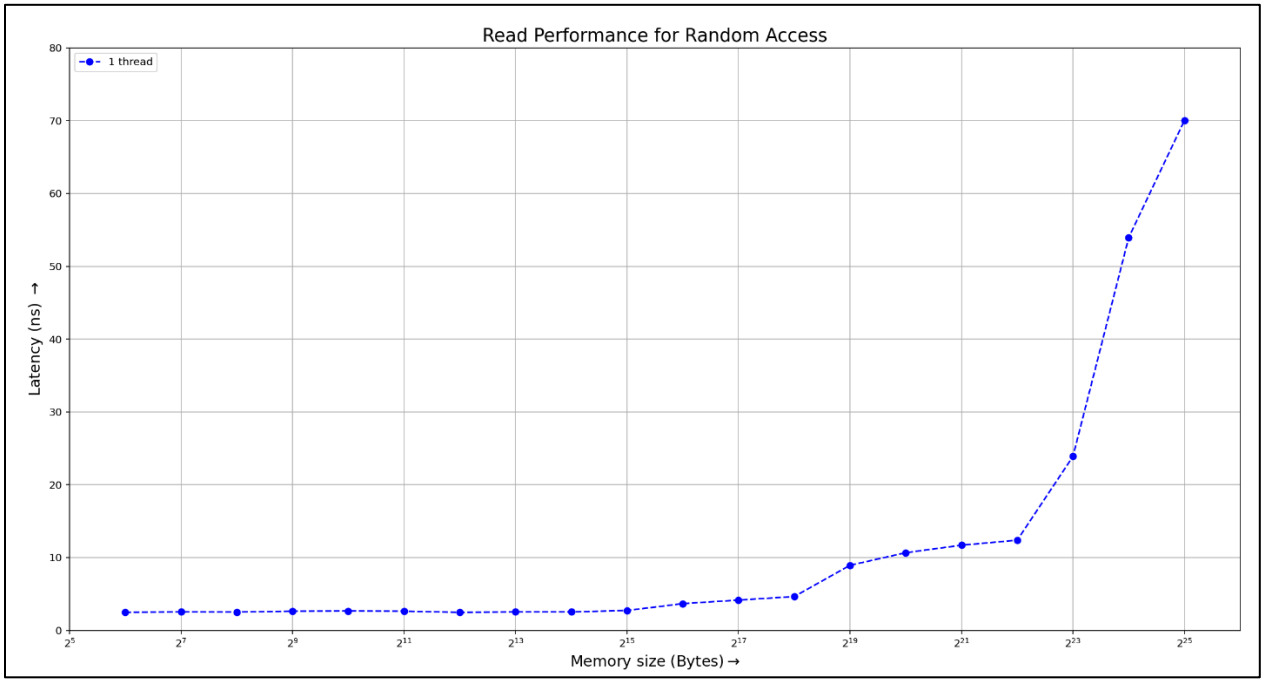
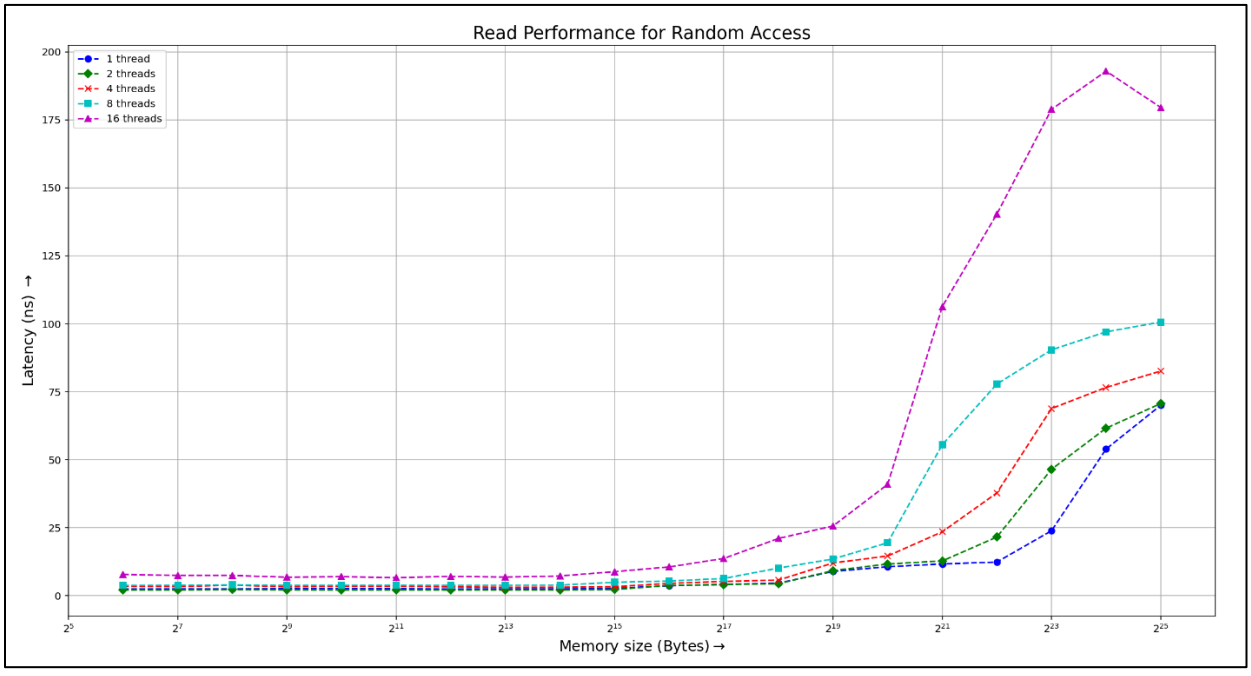


Figure 2



The following Figure 3 and Figure 4 show the read-modify-write latency with 1 threads and multiple threads respectively.

Figure 3

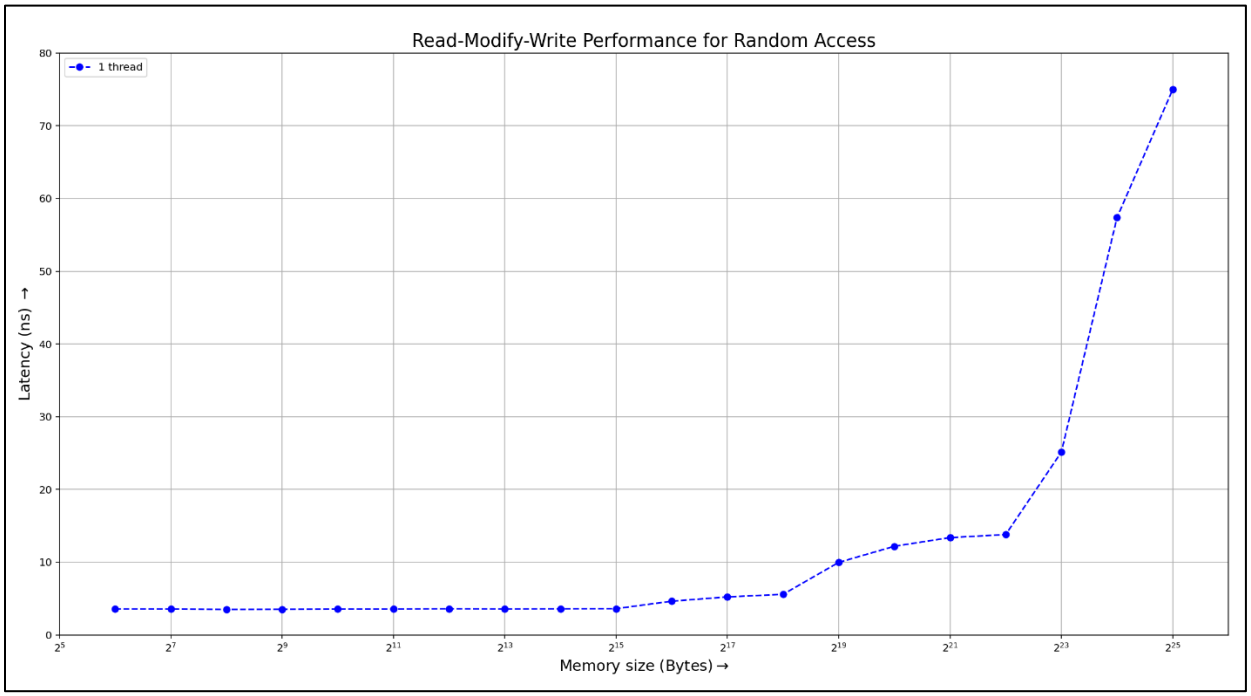
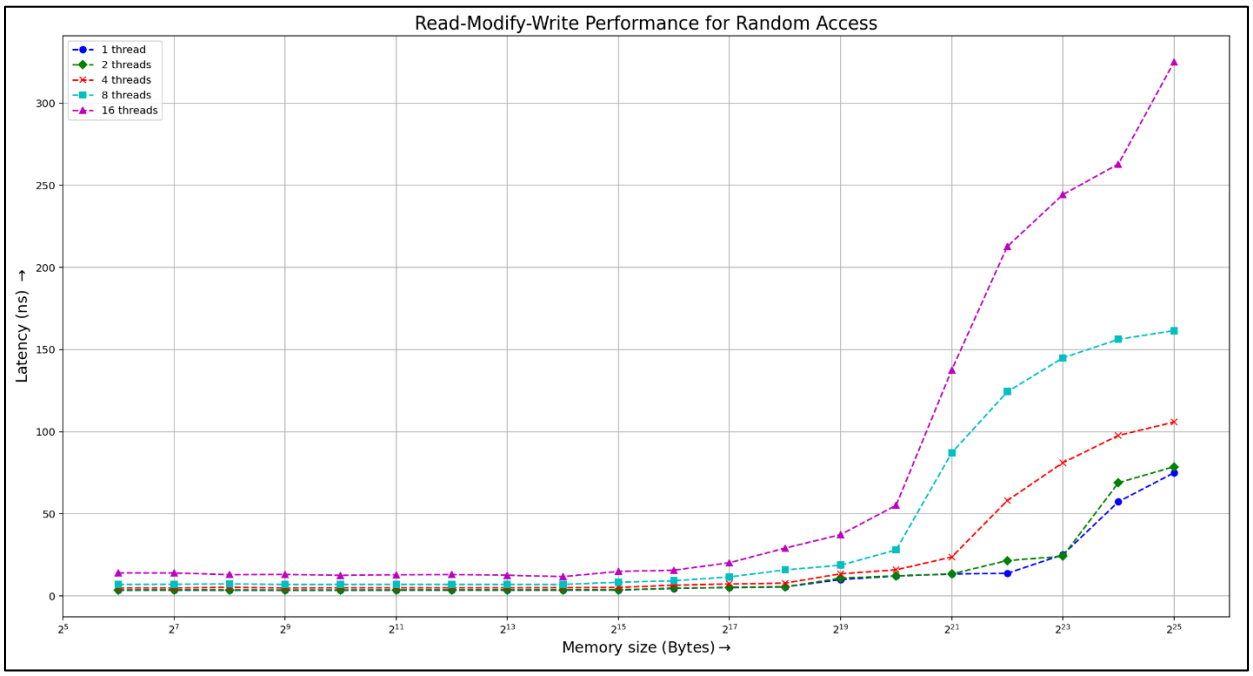


Figure 4



Analysis

Read-Only Analysis

In the figures 1-4, the significant regions of interest are the places where we observe an abrupt increase in the latency from the normal trend. These regions signify the spilling over of memory to the next level of cache (or main memory), implying that more memory has been allotted than can be handled by that memory component. In Figure 1, the jump is observed at 32KB (2^{15} Bytes) from 2.5ns to 3.5ns, denoting that the L1 cache is full and the subsequent access went to L2 cache, thus increasing latency. Thus we can conclude that the L1 cache is 32KB. Similarly, the next jump is observed at 256KB (2^{18} Bytes) from 4.5ns to 9ns, denoting that the L1 and L2 caches are full and the subsequent access went to L3 cache, thus further increasing latency. Thus, we can conclude that the L2 cache is 256KB. Consequently, the next and final jump is observed at 8MB (2^{23} Bytes) from 23ns to 54ns, denoting that the all the caches got full and a significant portion of total latency went to the main memory, thus increasing latency even further. Hence, we can conclude that the shared L3 cache is 8MB. In conclusion, L1 is 32KB, L2 is 256KB and L3 is 8MB. This matches exactly with the original configuration of the system.

When using multithreaded access, the memory components will get filled more quickly because more threads are reading and writing using their private memory simultaneously. Figure 3 shows the read latency for multithreaded access. The access time remain almost the same until 512KB or 1MB, mainly because each core has its own private L1 and L2 caches and therefore multithreaded access doesn't result in significant latency until that point. However, the latency increases many folds afterwards, mainly because of each thread trying to access different lines in the shared L3 cache lines and therefore leading to more eviction and main memory access. The access time with 16 threads doesn't scale because of hardware limitations – the hardware only allows 8 threads at a time with SMT. Therefore, the higher latency scaling in 16 threads is mainly because of thread scheduling overhead.

Read-Modify-Write Analysis

Figure 3 and 4 show the latencies for Read-Write access. Although the overall shape and scale of the graph are similar, the latency for Read-Write access is higher than that of Read access. This is because writing to memory is more time consuming than just reading. Because of higher latencies, the significant regions of interest are also more clearly demarcated and prominent. Even in multithreaded access, the results are similar but show much more latency than single-thread access. Because of false sharing in the caches, the multithreaded access makes the cache invalidate other copies and update memory constantly, thus increasing latency. Also, the latency with 16 threads stands very high, more so because of thread scheduling in the 8-threaded system.

As far as the stability of the results is concerned, both sets show stable results even for multithreaded access. However, the read-write curve is slightly more stable, mainly because of clear demarcation of the higher latencies and their differences in both single and multithreaded access. Moreover, the cache performance hints at write-back caches because of a minor latency difference in read only and read-modify-write operation. For sanity check, we can verify the results from the actual CPU configuration, as well as compare the curves for read-only and read-modify-write operations. This proves us that the results have been correctly compiled.

Extra Credit

The following figures 5-8 show the latency vs stride sizes for sequential-stride access. As elucidated in the limitations section, a randomly linked data memory chain may suffer from excessive local cache hits, thus giving us incorrect performance results. Therefore, strided access makes sure that the next access is a fixed number of bytes (and locations) apart. This disables the spatial locality optimization that caches may make use of in random access. As we increase the stride access, the latency increases and saturates after a fixed point, as all accesses come from a higher level in the memory hierarchy. Multithreaded performance also scales in a similar way for sequential access, saturating after a point. It is to be noted that the curve with 16 threads again shows abnormally higher latency, mainly because of the excessive thread scheduling in the 8-threaded machine.

Figure 5

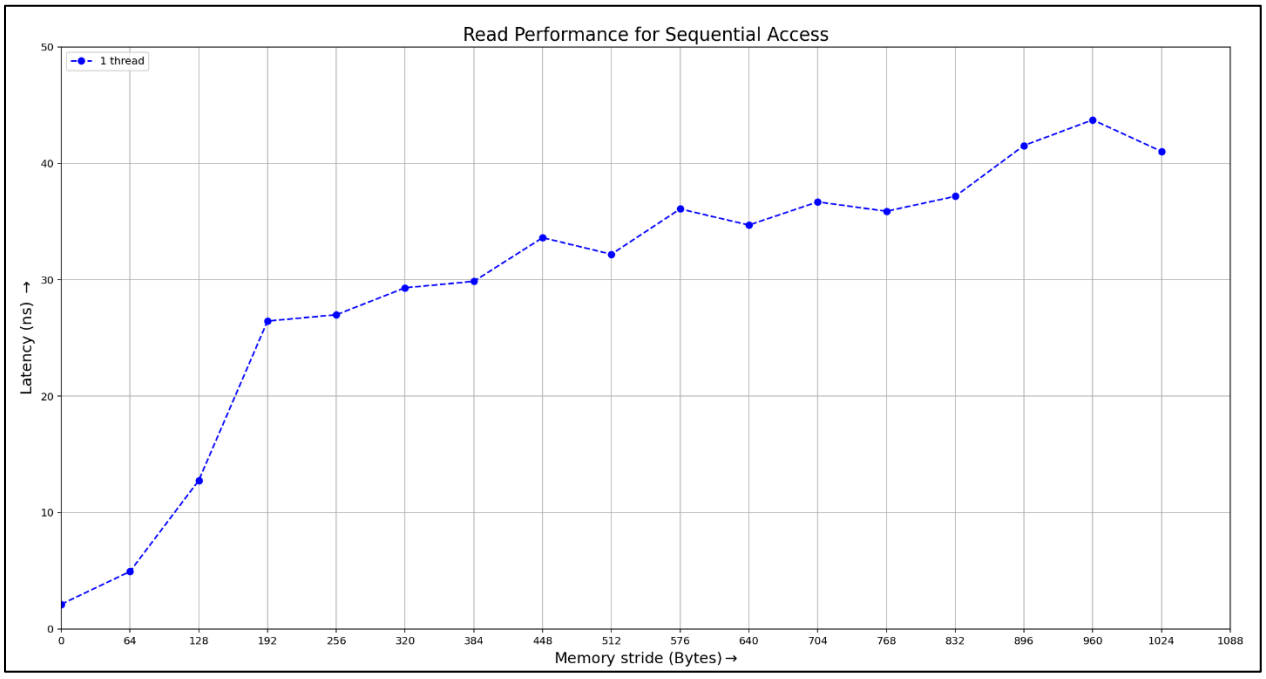


Figure 6

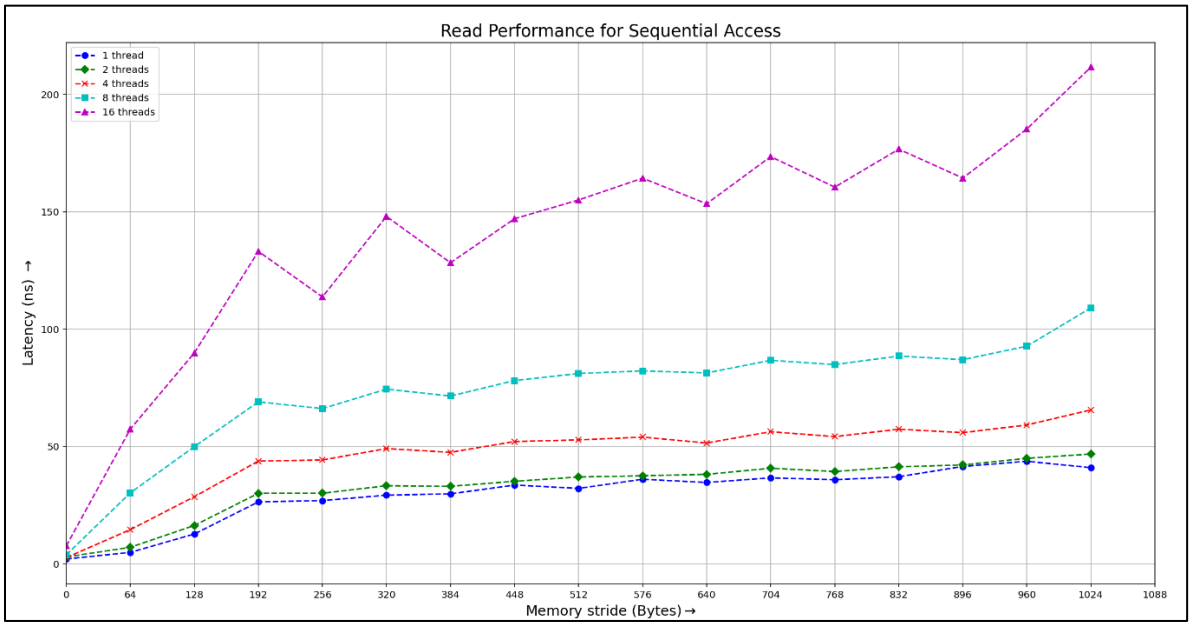


Figure 7

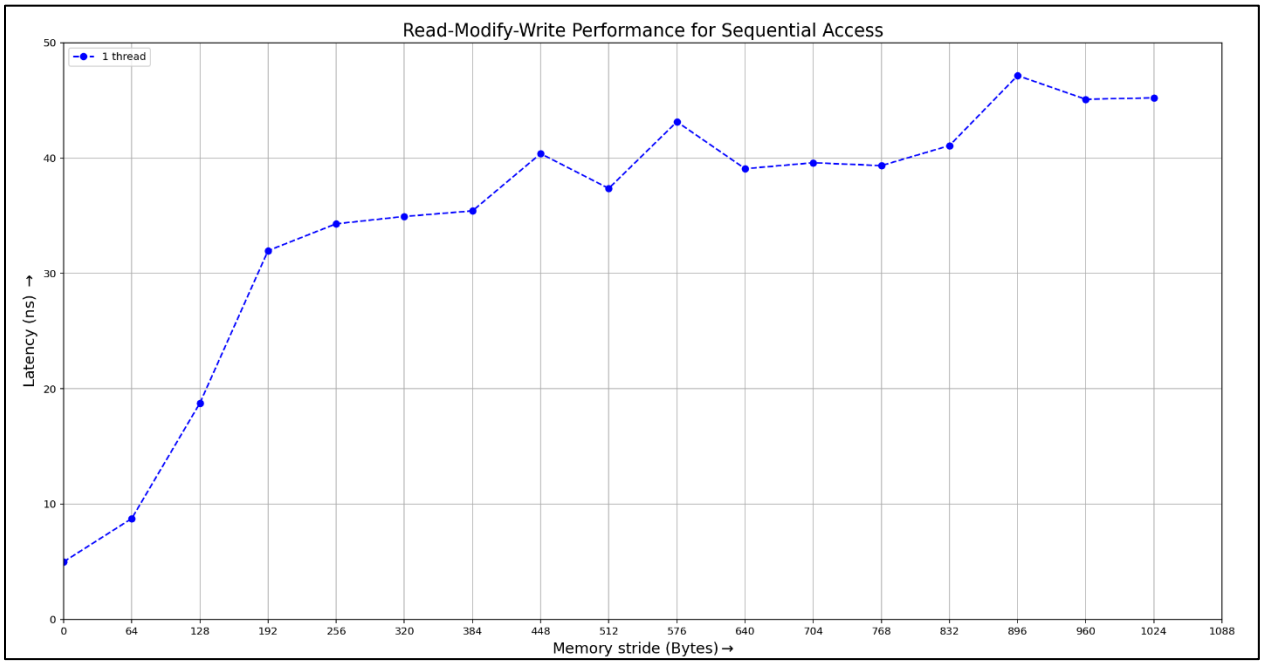
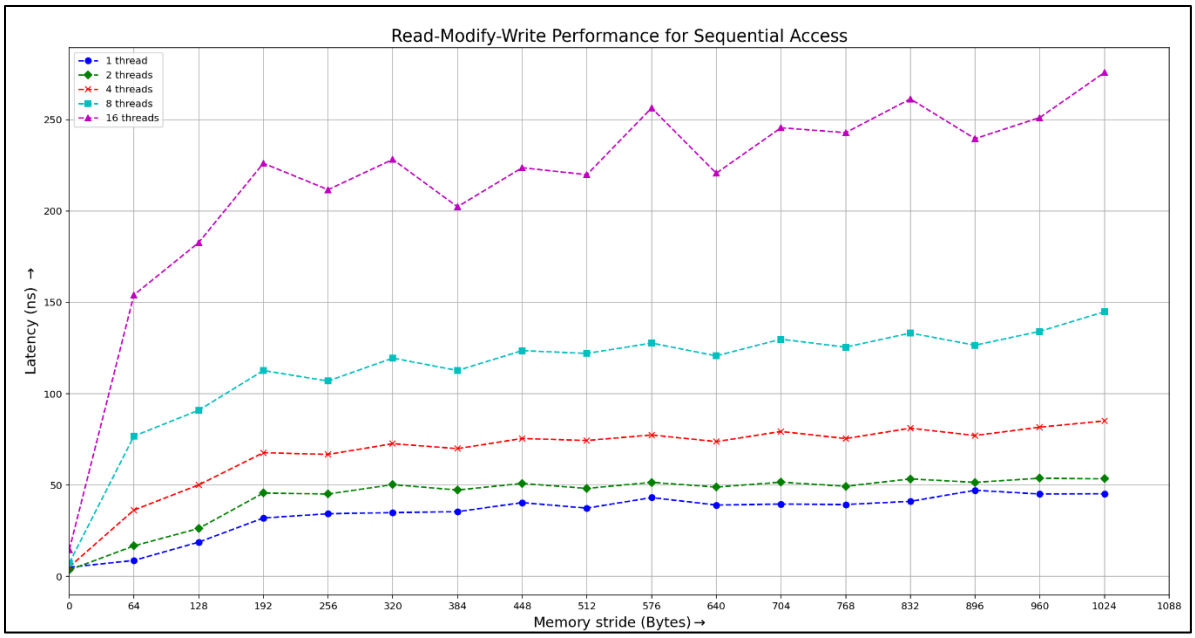


Figure 8



References

[1] Generating random numbers in C++: https://diego.assencio.com/?index=6890b8c50169ef45b74db135063c227c

[2] Pointer Chasing: https://en.wikichip.org/wiki/pointer_chasing

[3] Memory access pattern: https://en.wikipedia.org/wiki/Memory_access_pattern