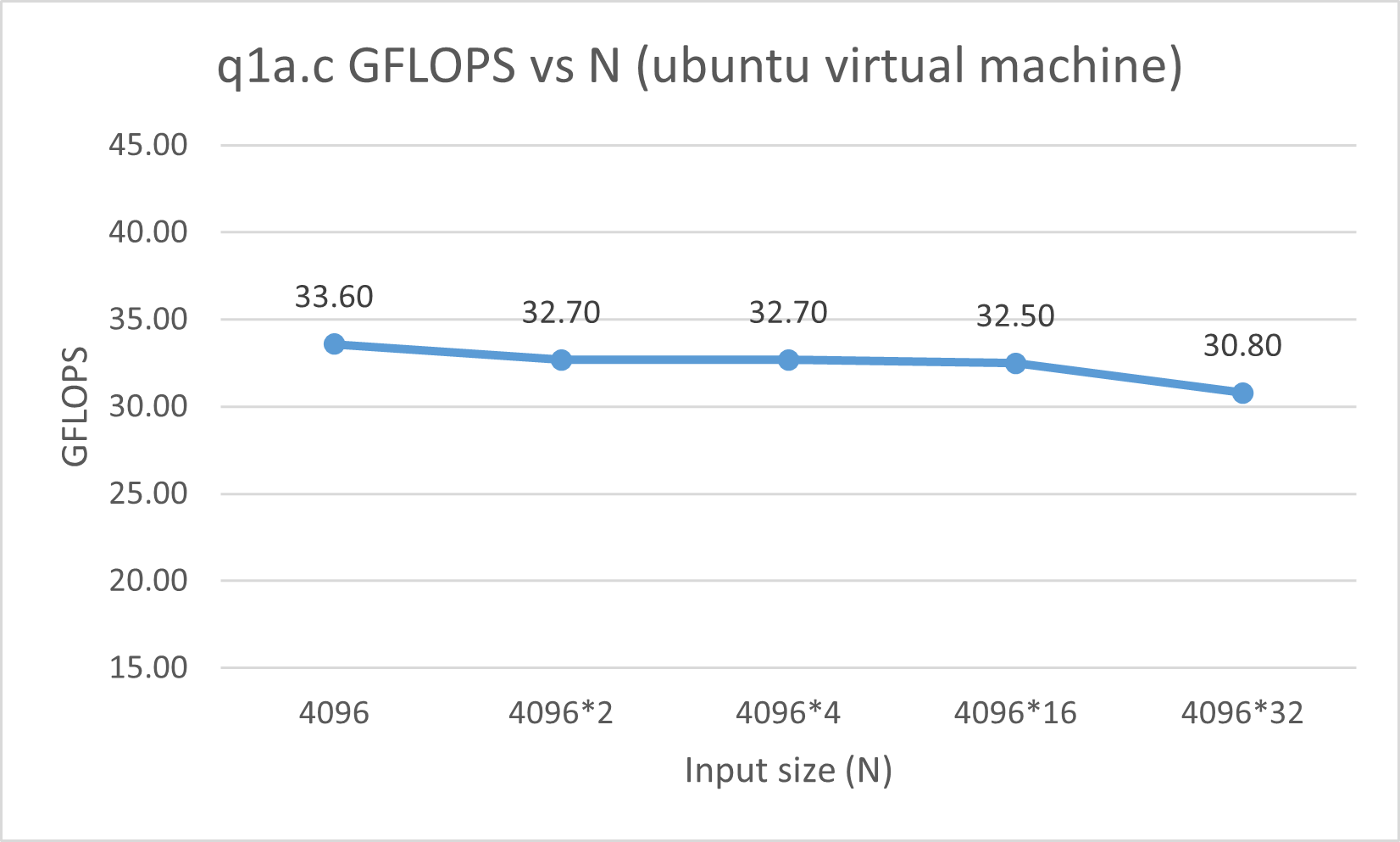
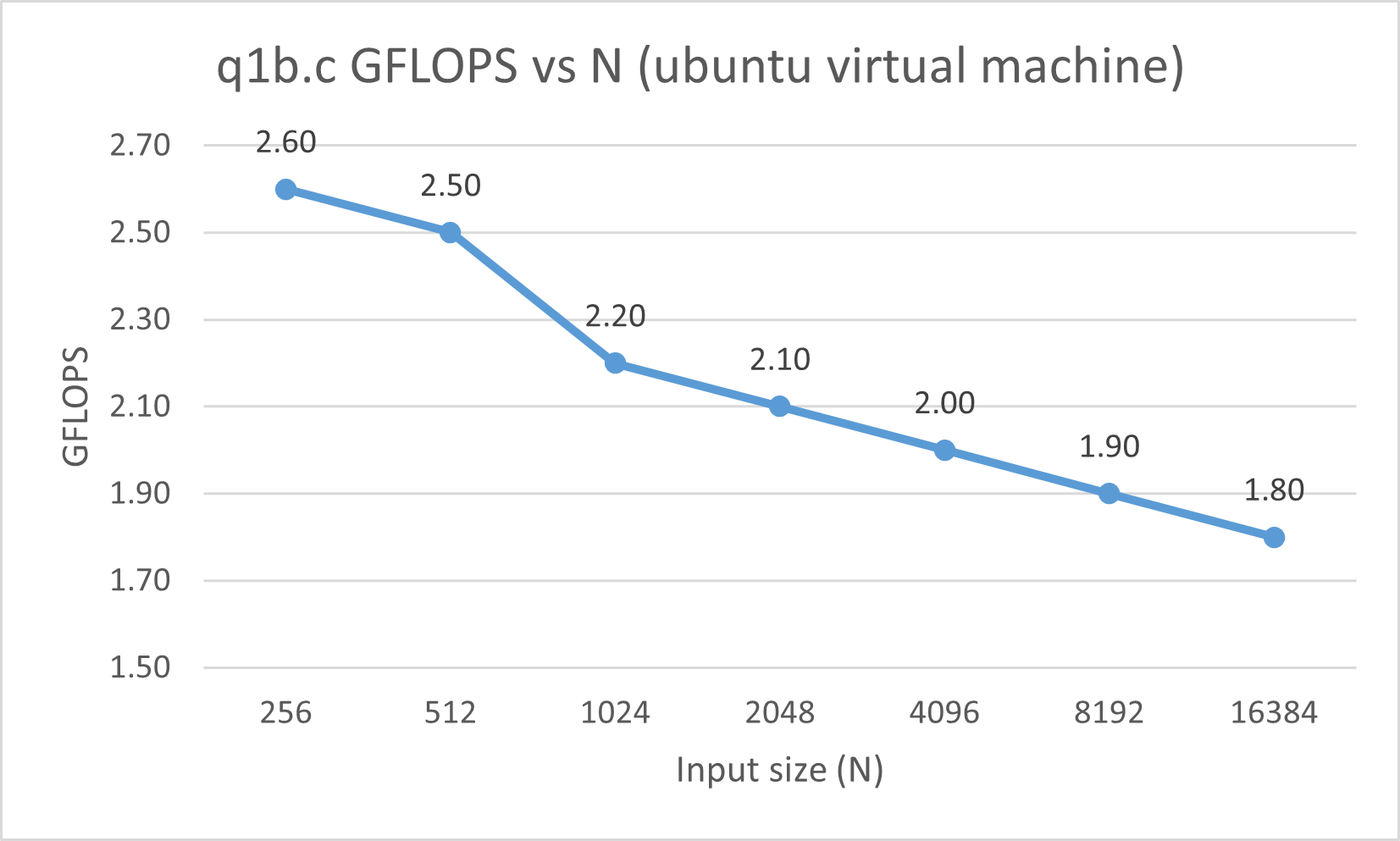
Question 1A.





Question 1B.

Graph A shows that when the routine in q1a.c is run with increasingly higher input sizes, the calculated GFLOPS value roughly stays the same. This indicates that the program is compute-bound, meaning that the program has a high arithmetical intensity. The GFLOPS value does not increase because the program is running at peak FLOPS, due to the balanced fmadd intrinsic instructions and minimal loads/stores.

Graph B’s GFLOPS value decreases slightly with higher input sizes; q1b.c is memory bound and has a low arithmetical intensity. Increasing the input size in this program increases the amount of cache misses, resulting in the CPU spending more time in slower memory trying to find and load the data.

Assuming that variables are being stored as 4 bytes each the arithmetical intensity of each routine is as follows:

* Q1a AI: 16/(2\*4) = **2.0**
* Q1b AI: 3/(5\*4) = **0.15**

Using the following equation, the peak FLOPS of the computer running the code has been calculated:

*Peak.FLOPS = num.CPU.cores x CPU.freq. x simd.length.in.bits / 32 x num.FMA.units x 2*

**345.6 FLOPS** = 4 cores x 2.7 GHz \* 256 bits / 32 \* 2 FMA units \* 2

Why are the measured values for q1a.c and q1b.c lower than the theoretical peak FLOPS of the computer?

q1a.c is a compute-bound program and has a high AI but the recorded GFLOPS values are still lower than the theoretical peak GFLOPS value because the program is being run on a single core, just like q1b.c. Even when it’s considered that the peak performance of one core is **86.4** GFLOPS(345.6/4), the code runs at much less than this, which is likely because of Operating System interrupts and also because the program is being run within a virtual machine that uses less system resources than what is available. Furthermore, this suggests that the code could potentially be further optimized.

q1b.c is a memory-bound program and has a low AI and therefore is bottlenecked by memory. The routine within q1b.c that is being measured has 4 loads and 1 store and is therefore repeatedly accessing memory. Because of Memory Hierarchy, the CPU will spend a large amount of time trying to find the data within L1, L2 and L3 cache and then eventually RAM. As the CPU goes through the hierarchy the number of cycles taken to fetch data increases, therefore the program takes longer to execute and the recorded GFLOPS value is significantly lower. This is all because the arrays become too big to fit within cache.

Question 1C.

Total D1 cache references/accesses: **6,381,601** = 4,248,917 D1 reads + 2,132,684 D1 writes.

Total D1 misses: **2,123,067** = 1,069,201 D1 read misses + 1,053,866 D1 write misses.

Total D1 miss rate: **33.3%** = 6,381,601 / 2,123,067

From running q1b.c through Valgrind it is shown that the CPU accessed the L1 data cache 6,381,601 times, and 2,123,967 of these accesses resulted in cache misses (the remaining 4 million were therefore cache hits). This shows that the program has a L1 data cache miss rate of 33.3%.

The routine being run within q1b.c has N^2 iterations, and each iteration is executing the following statement:

Y[i] += alpha \* A[i][j] \* X[j]

When the CPU runs this code, it will need to load three arrays, a floating-point number, and the loop iteration/array index integers. These variables will be loaded into different places within the memory hierarchy.

*Alpha, i,* and *j* are all very small variables, so they will be stored within CPU registers so that they can be accessed as quickly as possible when needed, and as they are always needed in the inner most loop (will be accessed N^2 times), they demonstrate temporal locality. However, the three arrays are all too large to fit into the small CPU registers and must therefore be stored within the larger but slower cache memories.

The arrays that are initialised in the program are stored contiguously in Main Memory. When the program is executed, the CPU will need to initially find and load the data from Main Memory into their appropriate location within the memory hierarchy (of which has been discussed above). However, the CPU is designed to first access the registers and cache memory to look for the data it needs. When the data is not found within the registers, it will look at L1 cache and when the data is not found it will result in a D1 cache miss. This will repeat for L2 and L3 cache memories, and then the Main Memory will be accessed, the data found, and then loaded into registers or cache if possible. It can be assumed that after the first iteration *alpha, i and j* are stored within the registers until the program terminates and they are no longer needed.

In the context of this program, the CPU will look for a particular array index. However, when the CPU loads this array data from main memory to the cache it will also load additional contiguous array values as a block within a cache line and store them too. Therefore, when the CPU finishes one iteration and tries to start the next, the data will already be ready within the L1 data cache, resulting in a D1 cache hit.

The cache on the device running the program is 32KB but cache lines can’t move large amounts of data and therefore the arrays cannot be fully loaded at once and therefore there is a large number of misses – they are moved in blocks that are typically 64 bytes. As a float is 4-bytes large, a 64-byte cache line can move 8 float array elements from main memory at a time. Therefore, 3 cache lines will be needed to move 8 contiguous elements from each of the arrays to D1 cache, and after processing 8 iterations, another cache miss will occur upon which the next 8 elements from each array is loaded. This results in an estimated 3x speedup of the program.