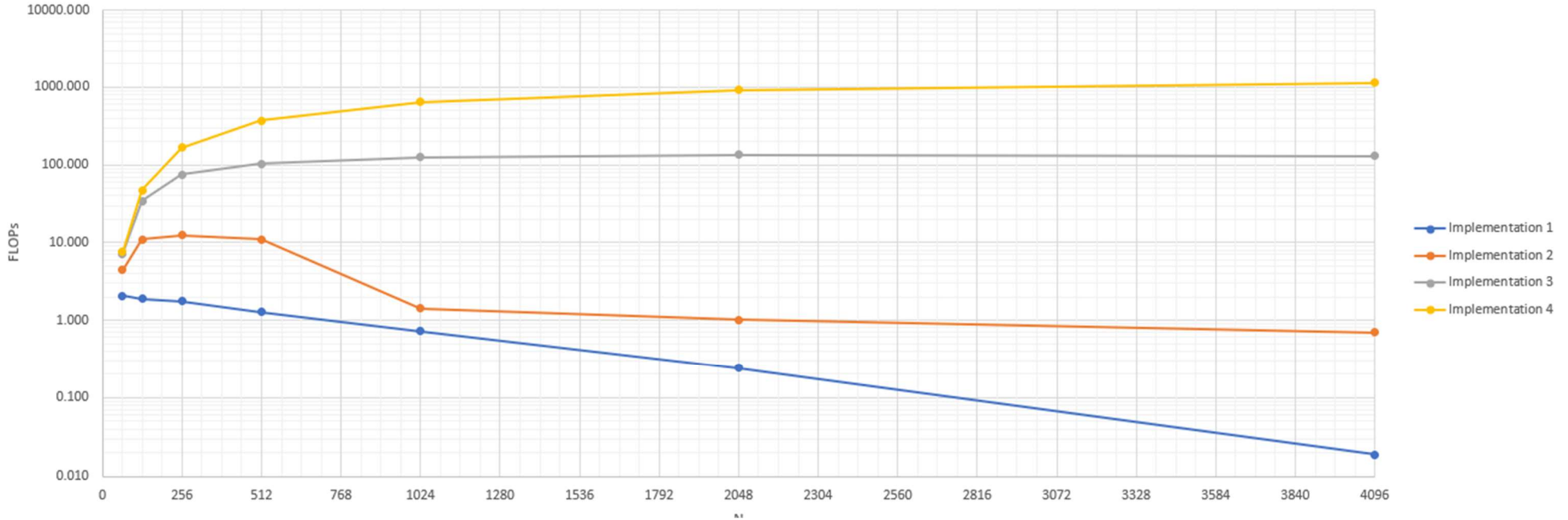
Question 2

Part 1 results

	NVIDIA GeForce GTX 1080 Ti															
	Implementation 1					Implementation 2				Implementation 3				Implementation 4		
N				Average				Average				Average				Average
64	2.068	2.077	2.071	**2.072**	4.413	4.407	4.424	**4.414**	7.172	7.184	7.071	**7.142**	7.653	7.657	7.487	**7.599**
128	1.901	1.903	1.909	**1.904**	10.897	10.897	10.917	**10.904**	34.328	34.154	34.335	**34.272**	47.187	47.752	46.674	**47.204**
256	1.757	1.759	1.759	**1.758**	12.432	12.469	12.492	**12.465**	75.977	75.160	75.102	**75.413**	165.130	168.726	170.283	**168.047**
512	1.287	1.283	1.289	**1.286**	11.051	11.060	11.060	**11.057**	102.520	103.169	103.833	**103.174**	367.628	370.260	378.479	**372.123**
1024	0.726	0.726	0.726	**0.726**	1.431	1.423	1.421	**1.425**	125.720	125.793	125.266	**125.593**	642.084	643.202	641.479	**642.255**
2048	0.249	0.231	0.238	**0.239**	1.007	1.012	1.024	**1.014**	134.122	134.264	133.657	**134.014**	915.798	911.733	915.234	**914.255**
4096	0.020	0.016	0.021	**0.019**	0.701	0.688	0.694	**0.694**	129.996	129.620	129.794	**129.804**	1141.878	1143.404	1148.879	**1144.720**



Part 2

The following is an explanation of the results achieved by each implementation.

Implementation 1:

For implementation 1, as N increase the size of arrays A, B and C (all of size N * N) grows exponentially. This means that for larger N values the data can no longer fit into the cache, meaning that data is pushed out of the cache and can't be reused, leading to frequent cache misses. Cache misses mean that data needs to be accessed from the slower main memory, significantly slowing down the execution.

The result of these increasing cache misses, can be seen on the graph as it shows, that as N increase the achieved FLOPs value decreases exponentially.

Implementation 2:

For implementation 2, OpenMP is utilised to create multiple threads to divide the workload between. While N is small the overhead of creating and managing threads is significant relative to the number of computations each thread performs, however as N increase each thread handles a larger workload, making the overhead less significant. This is the reason that the FLOPS value initially increases.

As N gets significantly larger the arrays A, B & C can no longer fit into the cache resulting in cache misses like in implementation 1. This increase in frequent cache misses explain why the graph shows a dramatic decrease in achieved FLOPs values after N is significantly large.

The reason implementation 2 still achieves higher FLOPs values than implementation 1 is because, the cache lines fetched by one thread can be reused by others that require the data. This reduces the total number of cache misses attained. The other reason is that while one thread is fetching data another can be performing computations, this overlap combined with the data reuse results in the achieved FLOPs values decreasing slower than that of implementation 1, as N increases.

Implementation 3:

Implementation 3 utilises the CUDA to use the GPU to perform computations. This requires that the data needed is transferred from the CPU memory to the GPU memory, resulting in an overhead as it takes time to transfer the data.

For small values of N this overhead is significant and leads to lower achieved FLOPs values. However, as N increases the overhead of the data transfer becomes less significant compared to the computational workload.

The plateau on the graph, at large values of N, is caused by the GPU is achieving its peak computational ability for this implementation.

Implementation 4:

Implementation 4 utilises CUDA similar to implementation 3, however it also utilises shared memory and tiling. Shared memory can be quickly accessed by the GPU and by having frequently accessed data inside the shared memory it ensures that data can be quickly fetched decreasing execution time. Tiling divides the arrays into smaller blocks that can fit inside of the shared memory, improving the data locality. This data inside the shared memory can be reused multiple times before new data has to be fetched from the slower global memory. Using both shared memory and tiling greatly improves the memory access pattern overall decreasing the execution time and improving the achieved FLOPs.

Similar to implementation 3, the overhead of transferring data to the GPUs memory is significant at small values of N but becomes less significant as N increases.

By utilising efficient memory management, of the shared memory and tiling, with the GPUs parallelisation, implementation 4 achieves the highest FLOPs values for any given N.

In summary the CPU implementations show that as N increases the achieved FLOPs decreases, due to the limited cache size and frequent cache misses as the array sizes grow. In contrast the GPU implementations achieve higher FLOPs values as N increase because the overhead or transferring data to the GPU becomes less significant compared to the computational workload. The GPUs parallel processing design means that it is much better as scaling for increased workloads than the CPU. All of this is clearly shown by the graph of the results obtained by each implementation.