Lab 2 – KMeans with CUDA

Jerry Reinoehl

CS380P Fall 2023

1 OS and Hardware Details

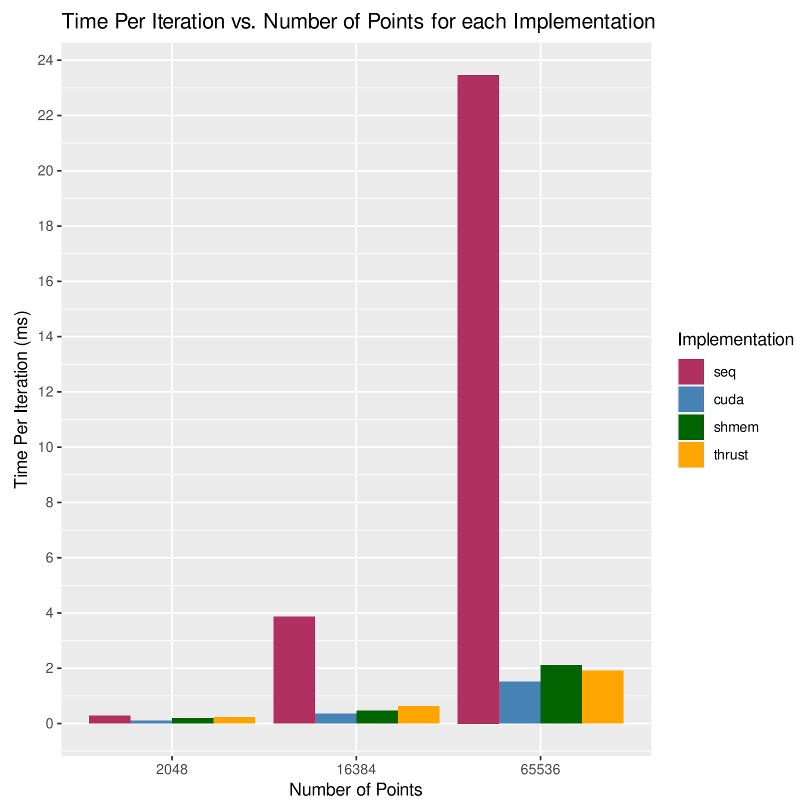
OS Details	
OS	Arch Linux
Kernel	6.5.4-arch2-1
Architecture	x86_64
Memory	80 GiB
CUDA version	12.2
gcc version	13.2.1

CPU Hardware Details	
Name	AMD Ryzen 5 1400 Quad-Core Processor
Cores	4
Threads	8
Clock rate	3.2 GHz
Cache L1	96 KB (per core)
Cache L2	512 KB (per core)
Cache L3	8 MB (shared)

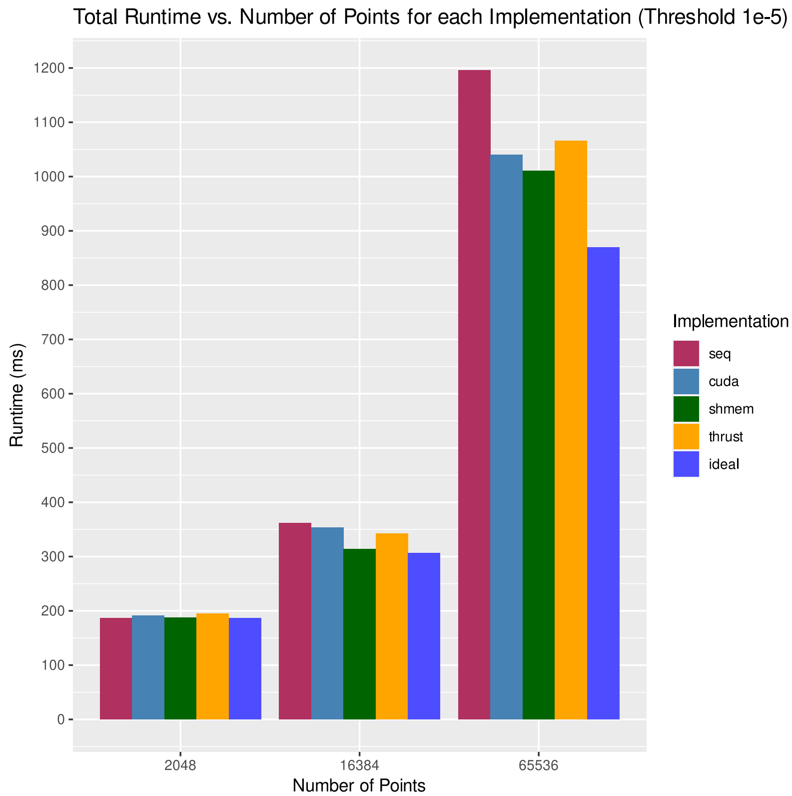
GPU Hardware Details	
Name	NVIDIA GeForce RTX 2060
Architecture	Turing
Compute capability	7.5
Clock rate	1,710 MHz
Global memory	6 GiB
Constant memory	64 KiB
Memory bus width	192
L2 cache size	3 MiB
Shared memory per block	48 KiB
Multiprocessor count	30
Number of cores	1920
Max threads per multiprocessor	1024
Max threads per block	1024
Registers per block	65536
Warp size	32

2 Performance Comparison

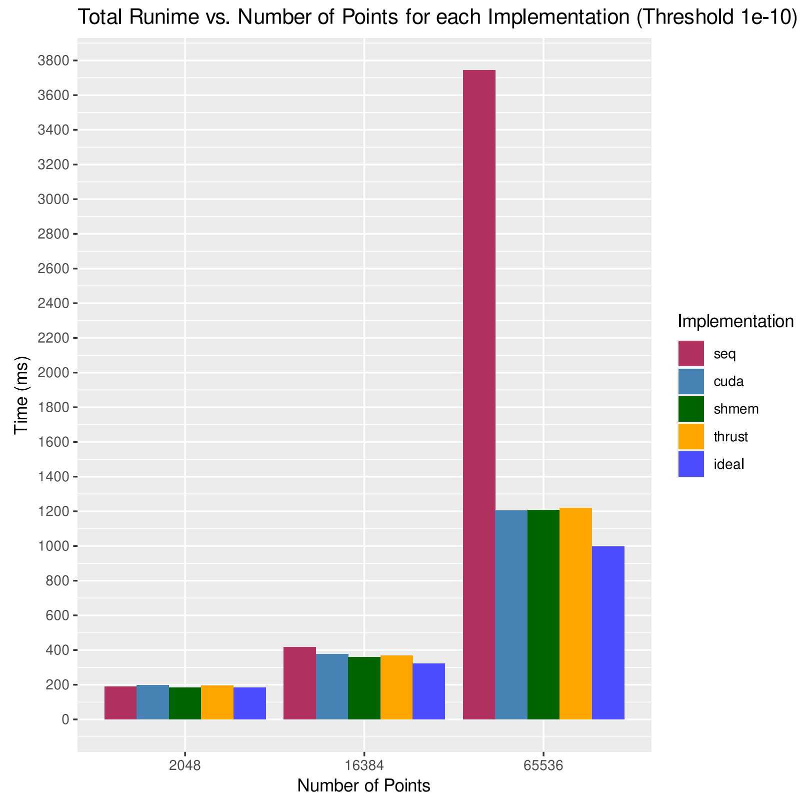
First we compare the time per iteration for each implementation. We see that as the number of points increases, the time for the sequential loop grows very quickly compared to the parallel implementations. This shows that the more iterations that are required for the solution to converge, the more speedup we can expect from the parallel implementations.



Next we compare the end-to-end runtimes of each implementation. We see that for a smaller number of points the sequential implementation is faster than any of the parallel implementations. As we increase the number of points or decrease the error threshold, we gain more from the parallel implementations because more time is spent in the kmeans loop (the portion that we can parallelize).



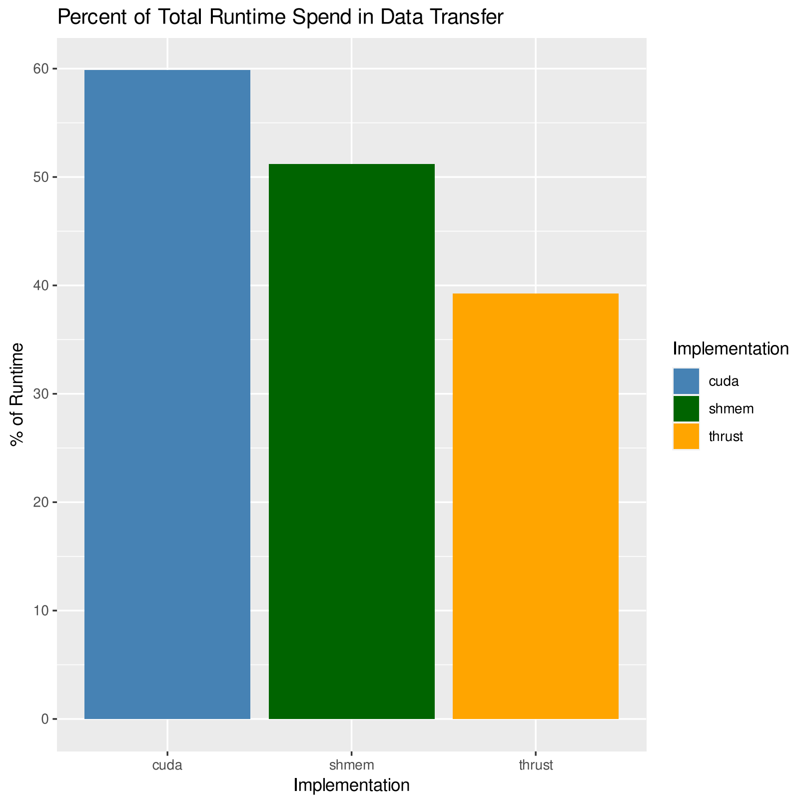
When we decrease the threshold to 1e-10, we see a drastic reduction in runtime for 65,536 points, relative to the sequential implementation. The table below shows that for this problem, 117 iterations are required to converge to a solution. As the number of iterations increases we can expect to see much better relative performance of the parallel implementations.



The ideal runtime is calculated by taking the sequential time of the sequential implementation and adding the ideal loop time. For example, we see that for problem size 65,536 with an error threshold of 1e-5, the sequential implementation time is 1195.56 ms and the time per iteration is 23.2777 ms. To get the sequential time we take $total\_time - time\_per\_iter * num\_iters = 1195.56 - 23.2777 * 14 = 869.6722ms$. The ideal loop time is calculated by taking the sequential implementation loop time and dividing by the maximum number of current threads of the GPU. For this case that is 30 (multiprocessor count) * 1024 (max threads per multiprocessor) = 30720 maximum concurrent threads. So the best kmeans loop iteration time we could expect would be 23.2777 / 30720 = 0.0008 ms. The total ideal time is then seq_time + ideal_iter_time * num_iters = 869.6722 + 0.0008 * 14 = 869.68 ms. The CUDA with shared memory implementation is the closest to this ideal time, being 140.5 ms (14% of its total time) slower.

impl	points	threshold	iters	iter_time	time
seq	2048	1e-5	18	0.2886	186.37
cuda	2048	1e-5	18	0.1101	191.30
shmem	2048	1e-5	18	0.1610	187.71
thrust	2048	1e-5	18	0.2507	194.57
ideal	2048	1e-5	18	0.0000	186.37
seq	16384	1e-5	15	3.6963	361.44
cuda	16384	1e-5	15	0.3697	353.39
shmem	16384	1e-5	15	0.4794	313.91
thrust	16384	1e-5	15	0.6245	342.47
ideal	16384	1e-5	15	0.0001	306.00
seq	65536	1e-5	14	23.2777	1195.56
cuda	65536	1e-5	14	2.0839	1040.30
shmem	65536	1e-5	14	2.7562	1010.18
thrust	65536	1e-5	14	2.6303	1065.77
ideal	65536	1e-5	14	0.0008	869.68
seq	2048	1e-10	18	0.2877	188.44
cuda	2048	1e-10	18	0.1064	197.88
shmem	2048	1e-10	18	0.1883	184.41
thrust	2048	1e-10	18	0.2331	193.27
ideal	2048	1e-10	18	0.0000	183.26
seq	16384	1e-10	25	3.8654	416.97
cuda	16384	1e-10	25	0.3565	377.38
shmem	16384	1e-10	25	0.4650	358.13
thrust	16384	1e-10	25	0.6241	368.85
ideal	16384	1e-10	25	0.0001	320.34
seq	65536	1e-10	117	23.4670	3742.55
cuda	65536	1e-10	117	1.5167	1203.70
shmem	65536	1e-10	117	2.1053	1207.19
thrust	65536	1e-10	117	1.9130	1219.80
ideal	65536	1e-10	117	0.0008	997.00

The parallel implementations are not as fast as the ideal time due to time spent in data transfer and accessing memory. In fact, we see that the majority of the total runtime of the parallel implementations is spent in data transfer in the graph below.



In the graphs above we can see that the fastest parallel implementation is the CUDA with shared memory implementation, while the slowest is the thrust implementation. It makes sense that the slowest would be thrust due to having to create additional arrays to hold the indices of the points and centroids. This results in additional memory lookups and data transfers. However, the thrust implementation is still significantly faster than the sequential implementation and very close to the other parallel implementations. The thrust library allows the programmer to work at the algorithm level, without having to worry about shared memory, block size, grid dimensions, etc., while still achieving great performance.

It also makes sense that we see an improvement of runtime of the shared memory implementation over the basic CUDA implementation. By using shared memory we reduce the number of memory accesses to the slower global memory in exchange for the faster shared memory. However, when using shared memory we must be extra careful to load our shared memory and synchronize our threads. Much like the thrust implementation, there is a tradeoff between implementation complexity and performance.

3 Time Spend on Lab

Time Breakdown	
Sequential implementation	7 hrs
CUDA basic implementation	14 hrs
CUDA shmem implementation	5 hrs
Thrust implementation	12 hrs
Report	8 hrs
Total	**46 hrs**