Srinivas Prasad Prabhu

UT EID - sp55629

CS380P – Lab 2 – Kmeans with CUDA

Platform Details

CPU - AMD EPYC 7232P 8-Core Processor, 1.5 GHz
RAM - 128 GB RAM
OS - Ubuntu 22.04.3 LTS
GPU – Nvidia Hopper PCIe H100

- 41:00.0 3D controller: NVIDIA Corporation Device 233f (rev a1)
- Clock – Base – 1.125, Boost - 1.755 GHz
- Memory Clock – 1593 MHz
- Total GPU Memory – 80GB
- Memory Bus Width – 5120 Bits
- Max Registers Per Block – 64k
- Max Registers Per Thread – 255
- Max Thread Blocks Per SM – 32
- MaxThreadsPerBlock – 1024
- WarpSize – 32
- Concurrent warps per SM – 64
- SM's per GPU – 114
- Cuda Cores per SM – 128
- Cuda Compute Capability – 9.0

Driver – Nvidia GPU Driver version r535.00
CUDA Toolkit – Cuda Toolkit version 12.2
NVCC Version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2023 NVIDIA Corporation
Built on Tue_Aug_15_22:02:13_PDT_2023
Cuda compilation tools, release 12.2, V12.2.140
Build cuda_12.2.r12.2/compiler.33191640_0

g++ (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0

Which of your implementations is fastest? Does it match your expectations of which should be fastest? Estimate the best-case performance speedup your CUDA implementations should have based on the number of threads in your program and the number of processing contexts supported by your hardware. How far of that prediction is your best-case performance?

Ans –

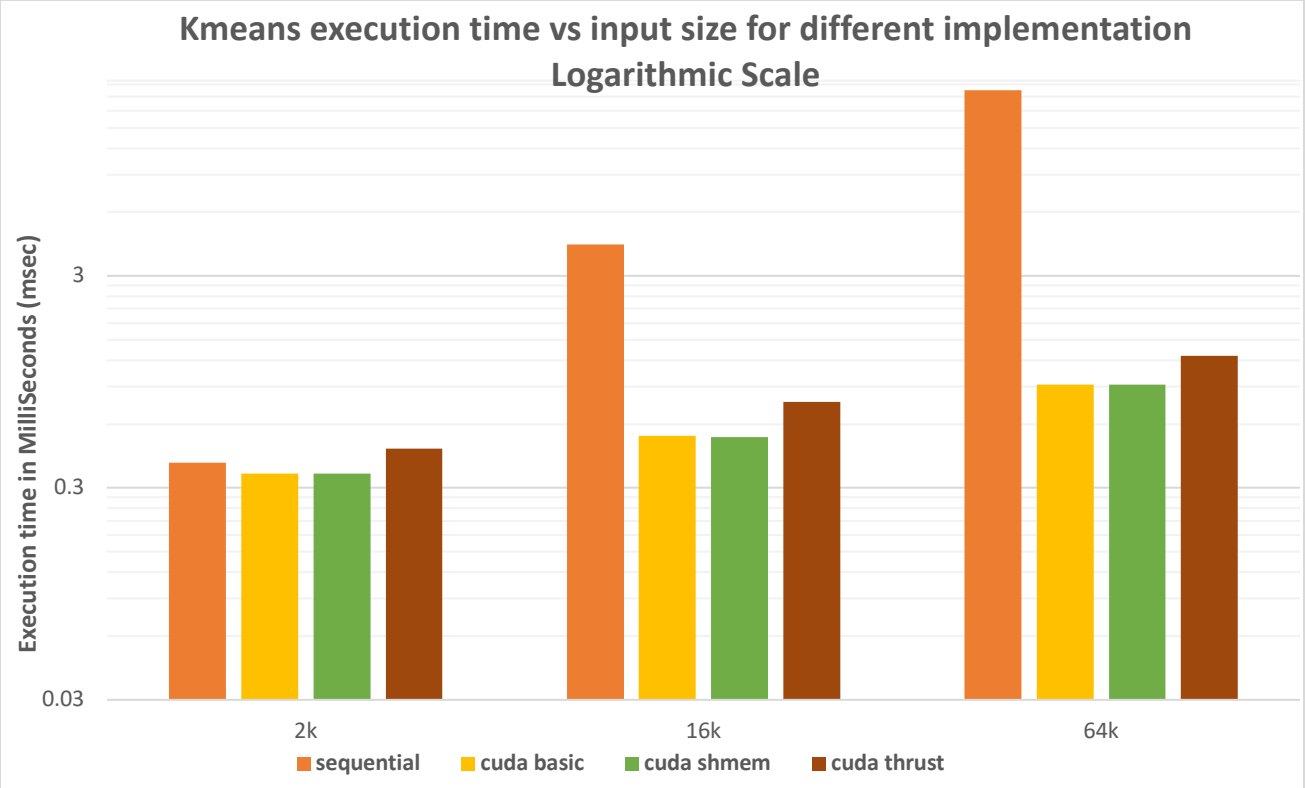
Among all my implementations the basic cuda version seems to be the fastest for inputs 16k and above. For lesser sized inputs, the difference between the cuda and sequential versions is not very significant. This can be observed from the plot below.

Note:

1) For cuda basic/shared versions number of threads = maxThreadsPerBlock = 1024
2) The below times were calculated using the command -

bin/kmeans -k 16 -t 0.00000001 -d<dims> -i <file> -m 150 -s 8675309 -c -a <alg>

Msec (per Iteration)	sequential	cuda basic	cuda shmem	cuda thrust
2k	0.393444	0.350526	0.351182	0.459445
16k	4.22568	0.526767	0.520145	0.76062
64k	22.499385	0.919002	0.91941	1.262012



Yes it matches my expectation that the CUDA version is the fastest.

The asymptotic work that needs to be done remains the same in each case.

By being able to convert the problem of kmeans from sequential to a parallel definition we can use the inherent parallelism of the GPU to complete this work in less time.

This also shows that as the input size increases, the cuda versions scale very well.

Other observations –

1) In sequential, cycles consumed increases exponentially with increase in input size. There is a lot of penalty paid due to constant thrashing of the caches as the algorithm keeps trying to access different points.
2) In my cuda implementation I have used maxThreadsPerBlock (1024) as my thread size. This utilizes a single block well; however overall occupancy is very low. This limits the amount of parallelism as the concurrent execution on an SM is limited by the warp size.

   I ran some experiments so that I can increase the amount of parallelism with different blocks and thread count. I ran these experiments with 64k input and basic cuda implementation. The times below indicate that for my

implementation on my hardware, thread count of 128 gives the optimal performance.

Thread Count	Time per Iteration (msec)
1024	0.919002
512	0.959761
256	0.801223
128	0.749300
64	0.776310
32	0.770931

This experiment shows that selecting the right block, thread tuple is a carefully crafted exercise based on the implementation and hardware. **My thread selection resulted in the implementation being 18% slower on my hardware compared to the optimal output.**

Theoretically the hardware I am using has -
GPU Memory BW = (1593*(10^6)) * (5120/8) * (2/(10^9)) = 2000 GB/s and upto 26 TFLOPs of FP64 operations.

My kernel is only using a fraction of this available BW. On running NCU, I could see very limited occupancy and BW usage. Below output is from the profiler for 64k input.

*cudaKmeans2(options_t *, double *, double *, double *, int *, int *, double) (64, 1, 1)x(1024, 1, 1), Context 1, Stream 7, Device 0, CC 9.0*

Section: GPU Speed Of Light Throughput

Metric Name	Metric Unit	Metric Value
...		
Memory Throughput	%	21.52
DRAM Throughput	%	0.60
Duration	msecond	1.37
....		
Compute (SM) Throughput	%	27.58

OPT This kernel grid is too small to fill the available resources on this device, resulting in only 0.5 full waves across all SMs. Look at Launch Statistics for more details.

Section: Launch Statistics

Metric Name	Metric Unit	Metric Value
Block Size		1,024
Cluster Scheduling Policy		PolicySpread
Cluster Size		0
Function Cache Configuration		CachePreferNone
Grid Size		64
Registers Per Thread	register/thread	40
Shared Memory Configuration Size	Kbyte	8.19
Driver Shared Memory Per Block	Kbyte/block	1.02
Dynamic Shared Memory Per Block	byte/block	0
Static Shared Memory Per Block	byte/block	0
Threads	thread	65,536
Waves Per SM		0.52

OPT Estimated Speedup: 48.39%

The grid for this launch is configured to execute only 64 blocks,

which is less than the GPU's 124

Section: Occupancy

```
   ------------------------------ ----------- ------------
   Metric Name                    Metric Unit Metric Value
   ------------------------------ ----------- ------------
    ....
   Max Cluster Size                     block           8
   Block Limit SM                       block          32
   Block Limit Warps                    block           2
   Theoretical Active Warps per SM       warp          32
   Theoretical Occupancy                    %          50
   Achieved Occupancy                       %       50.00
   Achieved Active Warps Per SM          warp       32.00
   ------------------------------ ----------- ------------
```

*cudaNewCentroids2(options_t *, double *, double *, int *, int *) (1, 1, 1)x(16, 1, 1), Context 1, Stream 7, Device 0, CC 9.0*

Section: GPU Speed Of Light Throughput

```
   ----------------------- ------------ ------------
   Metric Name             Metric Unit Metric Value
   ----------------------- ------------ ------------
   Memory Throughput                  %         1.67
   ...
   SM Active Cycles                cycle       239.61
   Compute (SM) Throughput            %         0.01
   ----------------------- ------------ ------------
```

OPT This kernel grid is too small to fill the available resources on this device, resulting in only 0.0 full waves across all SMs. Look at Launch Statistics for more details.

Section: Launch Statistics

```
   -------------------------------- -------------- --------------
   Metric Name                      Metric Unit    Metric Value
   -------------------------------- -------------- --------------
   Block Size                                                  16
   Grid Size                                                    1
   Registers Per Thread             register/thread            36
   ....
   Threads                                  thread           16
   Waves Per SM                                              0.00
   -------------------------------- -------------- --------------
```

3) Cuda shared memory implementation is not able to be better than cuda basic version. Based on my understanding, this is because
   a) We spend time initially waiting in many threads until the data is loaded into shared memory.
   b) We are only loading the centroids into shared memory. To access the inputs, we are still going to global memory.

   Only if we can reduce the number of global memory access on an aggregate considerably between the basic cuda version and the shared memory version, we will be able to see significant improvement in the shared memory time measurements.

Which of the parallel implementations is slowest, and does it match your expectations? Why or why not?

Ans -

Cuda thrust version is the slowest among all the parallel implementations. It does match my expectations as thrust is a high-level library which abstracts all the memory handling etc. in GPU's.

Typically, GPU performance is optimized by -
a) Having good parallel algorithm
b) Having efficient data access patterns for optimal results.

Thrust abstracts the data access patterns from the user completely. Hence can't expect thrust to provide the most optimal performance always.

Thrust provides a generic C++ STL like interface for ease of GPU programming. However, I believe under the hood it is doing additional temporary memory allocations, copies etc which adds to the time. Also, its data access patterns, placement of data into shared memory might not be optimal for a particular implementation. These factors contribute to additional time taken in the thrust implementation.

Even though thrust is not providing the best time, its ease of use and standard library of efficient algorithms makes it a good choice for quick development.

What fraction of the end-to-end runtime in your CUDA versions is spent in data transfer?

Ans-

As can be seen below, different cuda versions use less than 5% of the end-to-end time in data transfers. Most of the time is spent in actual running of the algorithm.

Basic Cuda version

Input Size	End to End Time(msec)	Algorithm Time(msec)	Difference	%age
2k	6.538848	6.309088	0.22976	3.5%
16k	13.767616	13.175296	0.59232	4.3%
64k	109.654205	107.799522	1.854683	1.7%

Cuda Shared Mem version

Input Size	End to End Time(msec)	Algorithm Time(msec)	Difference	%age
2k	6.573792	6.334016	0.239776	3.6%
16k	13.588384	12.991840	0.596544	4.3%
64k	109.619011	107.773788	1.845223	1.7%

How much time did you spend on the lab?

I spent approximately about 40 hours on this lab. It was a lot of learning.
I was startled when I first saw the cuda execution time measurements. It was a learning that parallelizing algorithms can result in exponential increase in performance using GPU.