CS259-L:
Performance Comparison of MapReduce and CUDA on the HG19 Human Genome Dataset

Chung-Hsuan Wu
Computer Science Department
San Jose State University
San Jose, CA 95192
408-924-1000
chung-hsuan.wu@sjsu.edu

ABSTRACT
Although both MapReduce and CUDA are efficient techniques for parallel processing, they differ fundamentally in design and application. MapReduce emphasizes distributed data processing through Mapper and Reducer operations, while CUDA exploits massive thread parallelism on GPUs. This project compares CPU-based MapReduce and GPU-based CUDA by implementing a k-mer counting task on a human gene dataset. Performance will be evaluated in terms of execution time, scalability, and resource utilization. This project will also focus on analyzing the factors driving the observed differences, and discussing the types of bioinformatics and data-intensive workloads best suited for each framework.

1. INTRODUCTION
Human genome analysis has been one of the most important areas in bioinformatic research. K-mer counting is one fundamental task in genome assembly, sequence comparison, taxonomic classification, and error estimation. However, as the size of human genomic datasets continues to grow, traditional methods based on a hash map will lead to memory exhaustion. Single-core processors are insufficient for such tasks, and even multi-core systems on a single machine cannot meet the computational demand. Therefore, distributed and parallel computing systems such as Hadoop or CUDA are becoming more and more popular in this field. Due to the essential characteristic in K-mer counting, MapReduce becomes extremely suitable for this task. This project will first use Hadoop to establish a cluster system for MapReduce to evaluate its efficiency and scalability. The same algorithm is also tested on a single CPU as a baseline. Finally, NVIDIA CUDA will be utilized to perform the same task, allowing for a direct comparison of performance and hardware utilization.

2. DATASET
The *hg19* human genome assembly is used in this project. *Hg19* is an older version of the human genome reference assembly with a total size of approximately 3.8 GB. There are only five characters in this dataset, which are A, C, T, G, and N. Although it is not widely used in recent bioinformatic research, it is still a foundational reference genome used for identifying genes, genetic variations, and understanding diseases.

3. METHODOLOGY
This project will use the MapReduce framework to perform the K-mer counting task. There are two types of phases in the MapReduce framework: Mapper and Reducer. The Mapper phase is for generating intermediate key-value pairs. The result for Mapper is, for example, ("GAT: 1") when K is 3. Notably, the value from Mapper will always be one, even if the keys are the same. The next step is shuffling, where all records with the same keys are grouped together. The final phase is Reducer, which is to group key-value pairs by summing the count for each key. This inherent property makes MapReduce suitable for execution on different machines or threads. In this project, two different architectures will be implemented and tested to analyze the efficiencies.

- Hadoop

Hadoop is a cluster system framework that is highly suitable for distributed processing. Hadoop also supports fault tolerance and is commonly used in big data analysis.

- CUDA

CUDA is a parallel computing platform and programming model developed by NVIDIA. It allows the use of an NVIDIA GPU for general-purpose processing. Although CUDA supports C and C++ programming, some advanced data structures, such as sets and maps, are not supported in GPU computation; only basic data structures, including pointers and arrays, can be used in CUDA.

Since a map cannot be used in this project, an array will be used to simulate a map to store the key-value pairs. The size of the array relies on the number of k. In this study, the value of k is set to 3, which means only three bits will be considered each time. There are only five choices for each digit, which are A, C, T, G, and N. The size of the array becomes 5 to the power of k, meaning the size becomes $5^3 = 125$ for this project.

In CUDA, a thread is the basic unit for parallel processing. Combining multiple threads becomes a block. Similarly, the blocks of parallel threads make up a grid. The following figure shows the relations between thread, block, and grid.

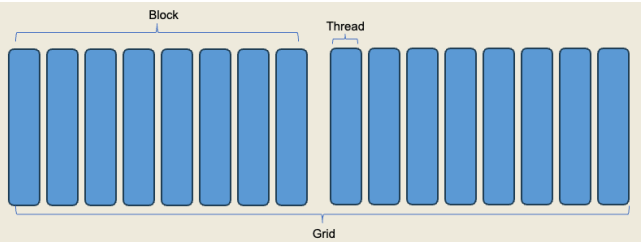

Figure 1. The relations between thread, block, and grid.

CUDA introduces a special feature known as unified memory. Unified memory provides a single virtual address space accessible by both CPU and GPU, meaning no consecutive transferring needs to be done during the process. Figure 2 shows how unified memory works.

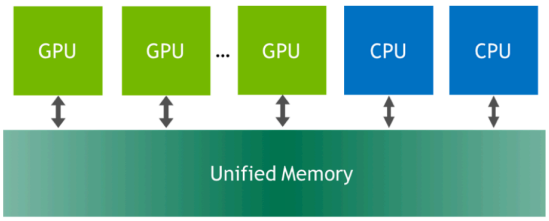

Figure 2. Unified Memory

4. HARDWARE CONFIGURATION

The following table describe the hardware configurations in this study.

Table 1. hardware configurations

CPU	i7-14700F
GPU	4060 8GB
RAM	16GB
System	Ubuntu 24.04

5. RESULTS

5.1 Hadoop

Figure 3 shows the results for Hadoop.

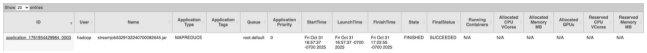

Figure 3. Results for Hadoop

Performance analysis focuses on the start time and finish time in this study. It took approximately 30 minutes to complete the MapReduce task. There are multiple reasons for it. First, both Mapper and Reducer are using standard output (STDOUT). However, it takes a long time to perform the printing command. Secondly, for Mapper, the intermediate results will be written to disk. As a result, Reducer must get the key-value pairs from disk. Disk reading and writing take a huge amount of time compared to RAM or cache. This will slow down the whole process. Besides, the RAM size in this project is set to 8GB. Page faults might occur and therefore hinder the process. Notably, in general, those different machines for Mappers and Reducers are usually connected through the internet, making the bandwidth of the internet a concern. In this study, only one machine is used in the Hadoop system, making this issue able to be ignored.

5.2 Sequential C++

Figure 4 shows the compiling and executing process, and the analysis for a normal C++ program.

```
hadoop@vincent-Ubuntu:~/CS259/Code/CUDA$ g++ -o test k-mer_sequential.cpp
hadoop@vincent-Ubuntu:~/CS259/Code/CUDA$ sudo /usr/bin/time -v perf stat ./test

 Performance counter stats for './test':

        104,944.84 msec task-clock                       #    1.000 CPUs utilized
               756      context-switches                 #    7.204 /sec
                 1      cpu-migrations                   #    0.010 /sec
         1,585,247      page-faults                      #   15.106 K/sec
     <not counted>      cpu_atom/cycles/                                              (0.00%)
   565,301,449,648      cpu_core/cycles/                 #    5.387 GHz
     <not counted>      cpu_atom/instructions/                                        (0.00%)
 1,771,930,211,090      cpu_core/instructions/           #    3.13  insn per cycle
     <not counted>      cpu_atom/branches/                                            (0.00%)
   337,913,234,414      cpu_core/branches/               #    3.220 G/sec
     <not counted>      cpu_atom/branch-misses/                                       (0.00%)
     1,402,632,197      cpu_core/branch-misses/          #    0.42% of all branches
           TopdownL1 (cpu_core)                 #   17.4 %  tma_backend_bound
                                                #    5.2 %  tma_bad_speculation
                                                #   23.0 %  tma_frontend_bound
                                                #   53.4 %  tma_retiring

     104.954862309 seconds time elapsed

     103.480279000 seconds user
       1.464862000 seconds sys


        Command being timed: "perf stat ./test"
        User time (seconds): 103.49
        System time (seconds): 1.47
        Percent of CPU this job got: 99%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 1:44.98
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 3279996
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 0
        Minor (reclaiming a frame) page faults: 1586226
        Voluntary context switches: 6
        Involuntary context switches: 761
        Swaps: 0
        File system inputs: 0
        File system outputs: 0
        Socket messages sent: 0
        Socket messages received: 0
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0
```

Figure 4. Analysis for normal C++ program.

For CPU, it shows the program took approximately 104 seconds to complete the task, with only 1 CPU, and merely 1 second is used for system calls. This program is running 3 instructions per cycle for pipelining, and almost achieves 99.5% of successful branch predictions. For RAM, there are 1,585,247 page faults in this program. There are two types of page faults: Major and Minor. A major page fault means the data is not in RAM. Minor page fault represents that the data is not in the process page table. Since in this program, the array is initialized to empty, minor page faults are unavoidable.

5.3 CUDA

5.3.1 Single Block

Figure 5 shows the analysis for single block with 256 threads.

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
76.1	260,979,072	61	4,278,345.4	4,347,544.0	9,976	4,455,027	563,875.1	cudaMemcpy
21.5	73,710,665	2	36,855,332.5	36,855,332.5	29,057	73,681,608	52,080,218.3	cudaMalloc
2.0	6,887,085	60	114,784.8	116,851.5	2,745	124,719	14,825.6	cudaDeviceSynchronize
0.3	1,042,740	2	521,370.0	521,370.0	76,254	966,486	629,489.1	cudaFree
0.1	323,655	60	5,394.3	2,848.5	2,256	158,124	19,808.1	cudaLaunchKernel
0.0	10,729	2	5,364.5	5,364.5	4,535	6,194	1,173.1	cudaEventRecord
0.0	7,938	1	7,938.0	7,938.0	7,938	7,938	0.0	cudaMemset
0.0	4,428	2	2,214.0	2,214.0	289	4,139	2,722.4	cudaEventCreate
0.0	1,405	1	1,405.0	1,405.0	1,405	1,405	0.0	cudaEventSynchronize
0.0	966	2	483.0	483.0	176	790	434.2	cudaEventDestroy
0.0	645	1	645.0	645.0	645	645	0.0	cuModuleGetLoadingMode

Figure 5. Analysis for single block in CUDA.

The longest time in a single block with CUDA is the memory copying, which is to transfer data into unified memory. “cudaDeviceSynchronize” is where the CPU waits for GPUs to finish their tasks. In a single block, the total amount of time for GPU work is 6,887,085 ns, and is approximately 2% of the total amount of time.

5.3.2 Multiple Blocks

Figure 6 shows the analysis for multiple blocks in CUDA.

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
48.7	310,714,948	60	5,311,915.8	5,259,765.0	4,543,054	7,648,677	617,860.1	cudaDeviceSynchronize
39.7	260,133,305	61	4,264,480.4	4,342,523.0	10,310	4,400,445	561,465.9	cudaMemcpy
11.4	74,371,866	2	37,185,933.0	37,185,933.0	33,274	74,338,592	52,541,794.2	cudaMalloc
0.1	947,211	2	473,605.5	473,605.5	88,510	858,701	544,607.3	cudaFree
0.0	310,214	60	5,170.2	2,658.0	2,358	146,376	18,544.0	cudaLaunchKernel
0.0	10,348	2	5,174.0	5,174.0	3,994	6,354	1,668.8	cudaEventRecord
0.0	7,859	1	7,859.0	7,859.0	7,859	7,859	0.0	cudaMemset
0.0	4,939	2	2,469.5	2,469.5	315	4,624	3,046.9	cudaEventCreate
0.0	1,623	1	1,623.0	1,623.0	1,623	1,623	0.0	cudaEventSynchronize
0.0	1,094	2	547.0	547.0	204	890	485.1	cudaEventDestroy
0.0	500	1	500.0	500.0	500	500	0.0	cuModuleGetLoadingMode

Figure 6. Multiples blocks in CUDA

For multiple blocks, the time GPUs actually work is 318,714,948 ns, and is 48.7% of the total amount of time. Surprisingly, the time increases when multiple blocks are used. This is due to a race condition. Although more threads are used to perform the task, they might intend to write the same key-value pair in the array, which means one of them should idle while the other is writing data. The following figure shows an example of when a race condition occurs.

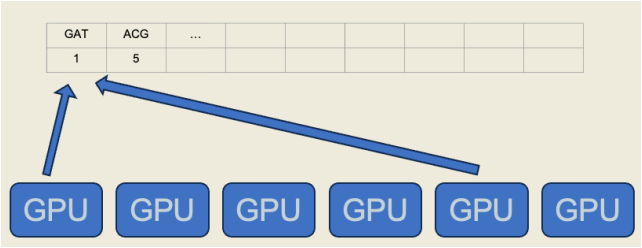

Figure 7. Race Condition

Since more workers will not necessarily speed up the task, this project aims to solve the latency for memory copying.

5.3.3 *Pipeline*

Two different approaches can solve the memory copying issue: prefetching and pipelining. Prefetching can be used when the size that needs to be allocated is known in advance. In this study, the HG-19 dataset is too huge to be entirely put in the unified memory, as well as the array with size 125. The figure below shows an example of prefetching.

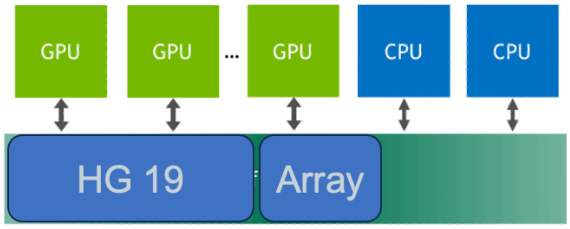

Figure 8. Prefetching

As a result, splitting data into different pieces is necessary. In this project, the data is split into 50 MB. Therefore, repetitively moving those chunks of data needs to be performed. This process is completed by the CPU. Figure 9 shows this process.

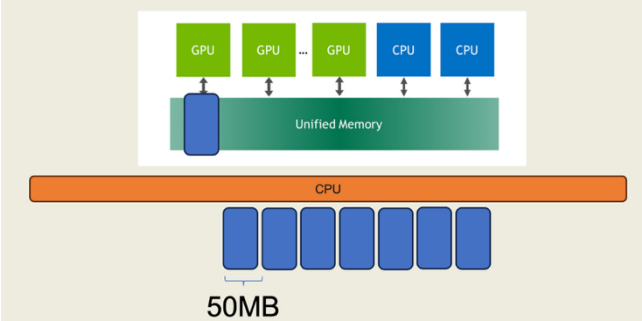

Figure 9. Splitting data

Since the transfer is executed by the CPU, GPUs can compute those chunks of data simultaneously. The figures below show the workflow with and without pipelining.

CPU	MOVE DATA		MOVE DATA		MOVE DATA	
GPU		COMPUTE		COMPUTE		COMPUTE

Figure 10. workflow without pipelining.

CPU	MOVE DATA	MOVE DATA	MOVE DATA	MOVE DATA	MOVE DATA	MOVE DATA
GPU		COMPUTE	COMPUTE	COMPUTE	COMPUTE	COMPUTE

Figure 11. workflow with pipelining.

5.3.4 *Pipeline with single block*

Figure 12 shows the analysis with single block utilizing pipelining.

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
55.1	125,476,237	1	125,476,237.0	125,476,237.0	125,476,237	125,476,237	0.0	cudaDeviceSynchronize
32.1	73,135,584	3	24,378,528.0	44,577.0	34,577	73,056,430	42,156,300.0	cudaMalloc
7.7	17,433,620	2	8,716,810.0	8,716,810.0	8,631,470	8,802,150	120,689.0	cudaHostAlloc
2.7	6,047,320	2	3,023,660.0	3,023,660.0	2,688,905	3,358,415	473,415.1	cudaFreeHost
2.0	4,479,245	60	74,654.1	2,908.5	2,320	4,296,085	554,222.2	cudaLaunchKernel
0.4	965,105	3	321,701.7	51,648.0	38,262	875,195	479,386.0	cudaFree
0.1	215,559	60	3,592.7	3,394.0	2,839	12,130	1,213.8	cudaMemcpyAsync
0.0	18,609	2	9,304.5	9,304.5	2,842	15,767	9,139.4	cudaStreamCreate
0.0	13,781	2	6,890.5	6,890.5	5,919	7,862	1,373.9	cudaEventRecord
0.0	12,371	1	12,371.0	12,371.0	12,371	12,371	0.0	cudaMemcpy
0.0	8,022	1	8,022.0	8,022.0	8,022	8,022	0.0	cudaMemset
0.0	5,318	2	2,659.0	2,659.0	1,548	3,770	1,571.2	cudaStreamDestroy
0.0	3,909	2	1,954.5	1,954.5	235	3,674	2,431.7	cudaEventCreate
0.0	1,972	1	1,972.0	1,972.0	1,972	1,972	0.0	cudaEventSynchronize
0.0	1,096	2	548.0	548.0	288	808	367.7	cudaEventDestroy
0.0	548	1	548.0	548.0	548	548	0.0	cuModuleGetLoadingMode

Figure 12. Single block with pipelining.

The longest time in this approach is “cudaDeviceSynchronize”, which means GPUs are working. Compared to the non-pipeline version of single block, the total amount of time decreased, and the percentage of time increased as well. Pipelining significantly makes the whole process much more efficient.

5.3.5 *Pipeline with multiple blocks*

The figure below shows the total time needed for each process using multiple blocks with pipelining.

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
66.6	219,580,414	1	219,580,414.0	219,580,414.0	219,580,414	219,580,414	0.0	cudaDeviceSynchronize
24.5	80,869,178	3	26,956,392.7	40,497.0	34,752	80,793,929	46,624,674.2	cudaMalloc
5.3	17,522,425	2	8,761,212.5	8,761,212.5	8,697,573	8,824,852	89,999.8	cudaHostAlloc
1.8	5,953,820	2	2,976,910.0	2,976,910.0	2,588,316	3,365,504	549,554.9	cudaFreeHost
1.4	4,465,802	60	74,430.0	2,845.0	2,245	4,294,153	553,997.4	cudaLaunchKernel
0.3	943,518	3	314,506.0	35,999.0	32,099	875,420	485,769.7	cudaFree
0.1	210,470	60	3,057.0	3,407.5	2,880	11,869	1,164.3	cudaMemcpyAsync
0.0	19,569	2	9,784.5	9,784.5	2,833	16,736	9,830.9	cudaStreamCreate
0.0	13,930	2	6,965.0	6,965.0	4,783	9,147	3,085.8	cudaEventRecord
0.0	11,316	1	11,316.0	11,316.0	11,316	11,316	0.0	cudaMemcpy
0.0	8,377	1	8,377.0	8,377.0	8,377	8,377	0.0	cudaMemset
0.0	4,331	2	2,165.5	2,165.5	1,422	2,909	1,051.5	cudaStreamDestroy
0.0	3,914	2	1,957.0	1,957.0	283	3,631	2,367.4	cudaEventCreate
0.0	2,377	1	2,377.0	2,377.0	2,377	2,377	0.0	cudaEventSynchronize
0.0	815	2	407.5	407.5	203	612	289.2	cudaEventDestroy
0.0	501	1	501.0	501.0	501	501	0.0	cuModuleGetLoadingMode

Figure X. Multiple blocks with pipelining.

With pipelining, multiple blocks can still achieve an efficient computing process. The total time needed decreased approximately 1/3 while the percentage of executing time increased.

These evaluations both show that the pipeline can obtain a more efficient computing approach.

6. CONCLUSION

In this study, Hadoop and CUDA are examined to determine which approach can outperform the other in the K-mer counting task using the HG19 human genome assembly dataset. Hadoop can achieve a more efficient computation with multiple machines when it comes to big data analysis, while on a single machine, CUDA can outperform Hadoop by fully utilizing GPUs. However, due to the limited RAM size in CUDA, data splitting may be necessary, meaning the transfer between RAM and VRAM in GPUs needs to be implemented. This process can be performed by the CPU, while GPUs can compute the results simultaneously. Pipelining can get a higher throughput. Nonetheless, more threads that can perform more parallel processing do not necessarily speed up the process due to a race condition issue. In future studies, the divide and conquer method can be examined to figure out if there is a way to avoid a race condition.

7. REFERENCES

[1] Mark Harris, “An Even Easier Introduction to CUDA, ” 2025. [Online] Available: https://developer.nvidia.com/blog/even-easier-introduction-cuda/

[2] Mark Harris, “Unified Memory, ” 2013 [Online] Available: https://developer.nvidia.com/blog/unified-memory-in-cuda-6/

[3] “MapReduce: Simplified Data Processing on Large Clusters,” research.google. https://research.google/pubs/mapreduce-simplified-data-processing-on-large-clusters/

[4] Apache Hadoop, “MapReduce Tutorial, ” 2025 [Online] Available: https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html