TEXT HISTOGRAM: LAB ASSIGNMENT

# PREPARED BY: **ISHAN GAUTAM (email: iamishan9@gmail.com)**

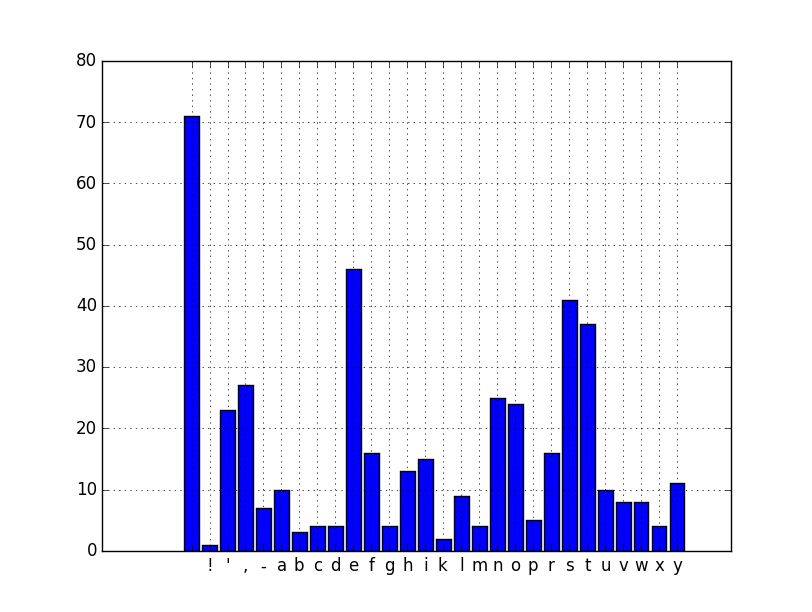


Figure 1 Example histogram

## Readme for the code:

### File Description:

1. ***template.cu*** contains the main program with all the calling codes and CPU codes
2. ***kernels.cu*** contains the GPU kernels
3. ***config.h*** contains the config details for the program to run

All important functions and codes are commented properly.

* Were there any difficulties you had with completing the optimization correctly?
* Here are some of the difficulties faced while completing the optimization correctly:

**File read process**:

First difficulty was the way I had to read the given file. First choice that was difficult was to store in an array or a vector. Since a vector’s size can be dynamically increased, I chose that. After that, another choice was to read all characters or just the eligible ones like ignoring ‘**$**‘ and ‘**\***‘. After reading the characters and storing them in vector, I had to copy them into a character array as it used less memory on the system.

**Number of blocks/threads:**

Another difficulty was to determine the number of blocks and threads to be used. The combination that was most suitable for what I wanted to achieve was 32 blocks and 1024 threads. This was achieved after a numerous test. This also provided a suitable outcome for the kernel using shared memory where the threads were performing quite well giving out acceptable processing time.

**Using shared memory:**

I created multiple kernels. One which doesn’t use the shared memory and the other one which uses it. I was trying to perform the kernel that uses shared memory in one go but it was not possible. The outcome was working for only the first block and afterwards, the histogram data was all wrong. So, I had to debug and separate the kernel where the addition of all the blocks data was to be done separately.

**Managing the constants:**

My code contains a lot of comparisons with constants. So, managing them and the code itself was a challenge. For which, I resorted to having 3 files. One file specifically for the constants so that they could be changed as well as new ones could be added in the future easily.

* Which optimizations gave the most benefit?
* I performed the task using two different optimization methods. One using just the global memory and the other one using the shared memory. By numerous tests, here are the results for the provided **words\_alpha.txt** file.

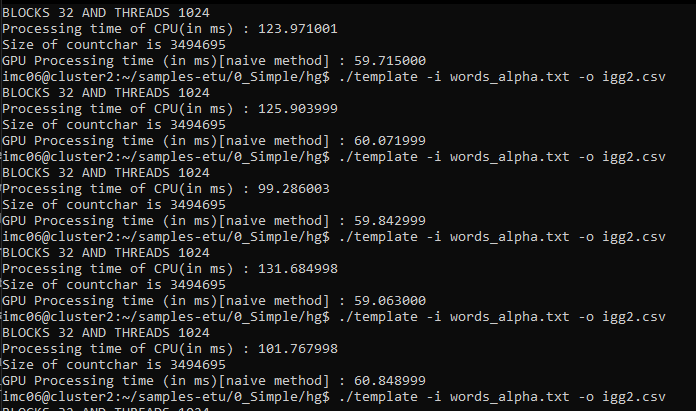


Figure Naive method (without using shared memory)

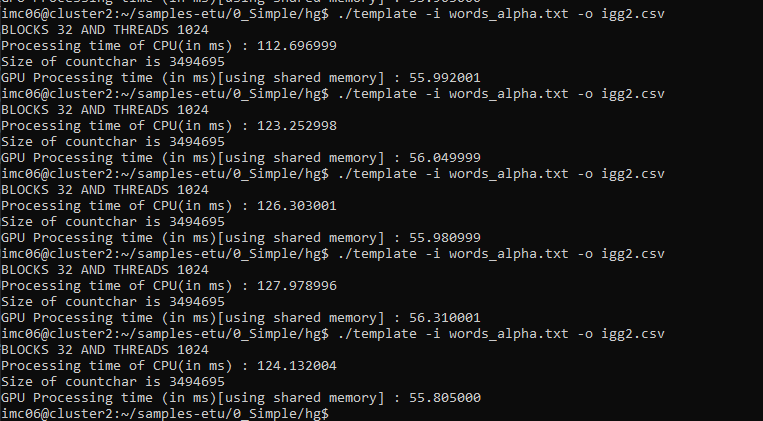


Figure Using shared memory

Hence, by the results itself, the optimization using shared memory was more beneficial as it utilized the resources in more efficient manner and performed the task in less time with ~5 outputs each for both the methods.

However, if both are run one after the other, the processing time shown will not be accurate.

* For the histogram kernel, how many global memory reads are being performed by your kernel? explain.
* The number of **global memory reads** performed by a kernel can be easily determined with the use of a profiler. But without access to it, I have to assume theoretically. Here, the **number of global memory reads** depends upon the **size of the file** to be read as well as the number of **blocks** and **bins** (num of characters) to be used. 67 in our case.

For example, if a file has size of 1,123,456 and we are using 5 blocks and 67 bins, then the total **number of global memory reads** would be the sum of all those 3 +- other access to some constants defined globally.

* For the histogram kernel, how many atomic operations are being performed by your kernel? explain.
* An **atomic operation** is capable of reading, modifying, and writing a value back to memory without the interference of any other threads, which guarantees that a race condition won't occur. **Atomic operations** in **CUDA** generally work for both shared memory and global memory.

The most authentic way to determine the exact number of atomic operations would be through a profiler like **nvprof**. But since the access to such profilers in our case was restricted, we must go to that theoretically. Using count for a single block where the count was increased every time an atomic operation was carried out, we can resort to the conclusion that

**no. of atomic operation per block = number of characters / (nb\_blocks \* nb\_threads)**

* Most text files will consist of only letters, numbers and whitespace characters. What can we say about atomic access contention regarding the number of threads that are simultaneously trying to atomically increment a private histogram?

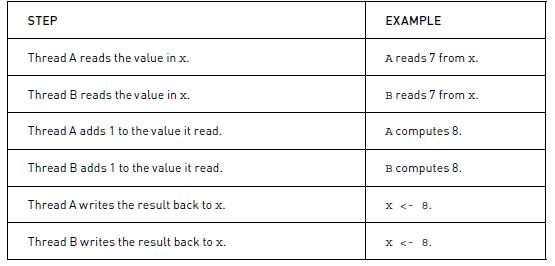


Figure Example of multiple threads accessing same value

As shown in above table, if our threads get scheduled unfavorably, we end up computing the wrong result. There are many other orderings for these six operations, some of which produce correct results and some of which do not. When moving from a single-threaded to a multithreaded version of this application, we suddenly have potential for unpredictable results if multiple threads need to read or write shared values.

Hence, we can say that **number of atomic access contention increases with the rise in the number of threads** trying to atomically increment a private histogram.