CS 525   Project 4 Report

Name: Ali Necat Karakuloğlu
ID: 22101543

Implementation Details

i)*Serial*

In this part I tried to encode the reference sequence by the hash function. This function converts the characters in reference string to 2 bit integer. This 2-bit integers are combined as a 32-bit integer. Note that, maximum k-mer length is 14. Therefore, maximum k-mer string can be represent with 32-bit integer using this encoding. This combination represents a k-mer. With this operation the k-mers in reference string are obtained. The rest is to encode read strings and compare the integers(k-mer ids) with reference. If they are equal, store the index and keep a count. This operation decreases the number of nested for loops. In brute force approach, we would need 4 nested loops but with this approach, multiple 3 nested for loops solves the problem.

ii)*Parallel*

In parallel approach I didn’t use a hash function. It simply checks the equivalence of string until n-th character n-times (I modified the given function in util.cu). In my implementation only the first k element of the reference string is compared (k is k-mer). Each computation unit checks a k-mer from a read. This computation unit checks the whole reference string. Note that, there may be repeated computations because a compute unit is responsible from a k-mer. Because of the thread limitation per block (1024), I distributed the computation over multiple blocks. A result array stores the minimum index of the k-mer matches and a count array is used for storing the number of repetitions of a k-mer match. Then, count array is reduced on the cpu to find the matches in a read. Finally, the results are printed to the output file. This implementation creates fixed memory by using the maximum reference string length, read length etc. Therefore, one can say that it is not memory friendly. However, the program gives the same results with the serial implementation. I think since the data is not too large, the memory doesn’t create a bottleneck for the execution of the program.

Analysis

As I mentioned before, the memory allocation of the parallel program is fixed an it is made for the largest values of parameters. This can be altered in a more developed program. Furthermore, thread per block is set to 1. The block and thread assignment of the GPU kernel can be revisited for more performance. When I looked at the nvprof outputs (It is provided with the zip file), percentage of malloc and memcpy increases when the number of reads get smaller. However, usually, the computation of the kernel dominates the execution time. Furtermore, when the read number gets smaller, malloc operation percentage increases within the memory operation. This is related to the fixed memory allocation that mentioned previously. Other operations take negligible time.

Experiments&Discussion

To check the performance, I divided 9216 by 2 4 times and obtained 5 read files. Furthermore, I added a small read file with 32 reads. Since the file with 2304 reads has short read string length, it executes faster than the one with 1152 reads. In general, the execution time increases linearly with the number of reads and parallel implementation is faster in any of cases. The results are given in Table 1 and Table 2 for parallel and serial execution, respectively. The speedup plot is provided in Figure 1. With different reads, the speedup is almost constant. In this circumstances, we can say that the program is scalable.

Read Count/ Length	Run 1	Run 2
9216/100	20.2041s	20.3720s
4608/102	10.1474s	10.1762s
2304/39	1.87525s	1.86312s
1152/165	4.14133s	4.12887s
576/115	1.45550s	1.45696s
32/126	117.67ms	117.32ms

Table 1: Execution Time Obtained From nvprof Tool for Parallel Program

Read Count/ Length	Execution Time
9216/100	3m40,333s
4608/102	1m52,057s
2304/39	0m20,456s
1152/165	0m45,708s
576/115	0m15,849s
32/126	0m0,974s

Table 2: Execution Time of Serial Program for Different Inputs

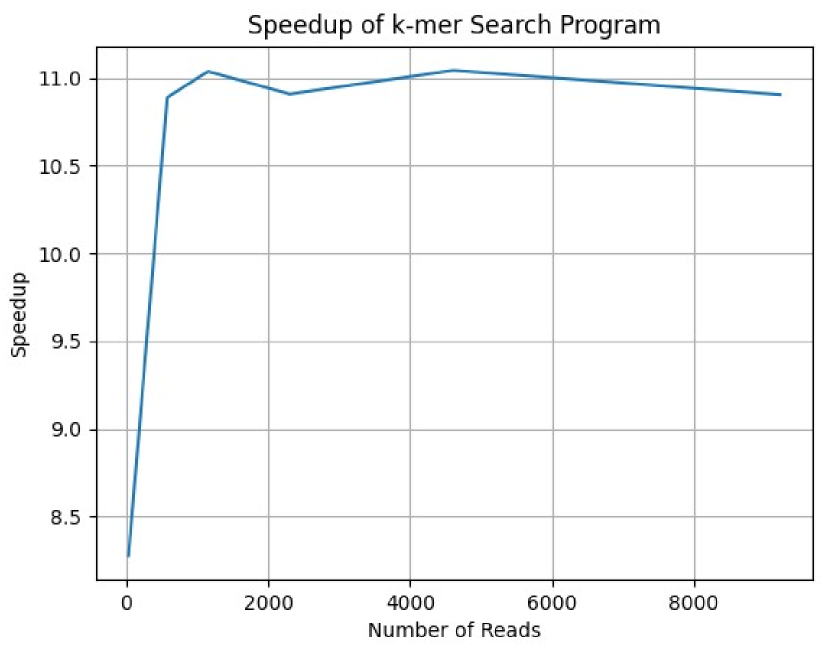

Figure 1: Speedup Plot of the Program with Different Number of Reads

Platform Specifications:
◦ Operating System: Ubuntu 20.04.3 LTS
◦ GPU: Nvidia RTX 2080 SUPER
◦ CPU: AMD Ryzen 7 3700X 8-Core Processor (16-threads)
◦ Number of Logical Cores: 16
◦ RAM: Corsair Vengeance 8 GB 3200 MHz DDR4, x2
◦ HDD/SSD: SanDisk SSD PLUS 120GB
◦ Compiler: g++ (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0

Note: Serial program gives “munmap_chunk(): invalid pointer” error. I looked at it but couldn’t solve it. It occurs when I tried to free the memory allocated with “malloc”

References

1)https://developer.nvidia.com/cuda-toolkit

2)https://github.com/uni-halle/gerbil

3)https://moodle.bilkent.edu.tr/2021-2022-fall/pluginfile.php/101629/mod_resource/content/1/set10.pdf (Lecture Slides)