GPU ACCELERATION OF ADVANCED K-MER COUNTING FOR COMPUTATIONAL GENOMICS

BY

HUIREN LI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Professor Deming Chen

ABSTRACT

k-mer counting is a popular pre-processing step in many bioinformatic algorithms. KMC2 is one of the most popular tools for k-mer counting. In this work, we leverage the computational power of the GPU to accelerate KMC2. Our goal is to reduce the overall runtime of many genome analysis tasks that use k-mer counting as an essential step. We achieved 4.03x speedup using one GTX 1080 Ti with one CPU (Xeon E5-2603) thread and 5.88x speedup using one GPU with four CPU threads over KMC2 running on a single CPU thread. This speedup is significant because accelerating k-mer counting is challenging due to reasons like serialized portions of code and overhead of disk operations.

To my parents, for their love and support.

ACKNOWLEDGMENTS

I thank my adviser, Deming Chen, for his help and guidance throughout my graduate school, and my colleague, Anand Ramachandran, for his guidance and help of this work.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The deoxyribonucleic acid (DNA) molecule found in the cell encodes information essential for the functioning of an organism. The DNA may be considered a sequence of molecular units called nucleotides or bases. Sub-sequences along the DNA, called genes, contain information regarding proteins that a cell may synthesize. Proteins are essential agents in cellular processes. During protein synthesis, cellular machinery reads the sequence of bases in a gene in the DNA, and maps this sequence of bases into a sequence of amino acids, which form a protein molecule [1]. Being the source of information regarding which proteins should be synthesized, and regulating cellular activities in other ways [2], the DNA sequence is of primary importance to the functioning of the organism, and knowledge of the specific sequence of bases in the DNA in an individual is essential in many biological and clinical applications.

Sequencing technologies such as Next Generation and Third Generation Sequencing (NGS, TGS) platforms provide a means to obtain large quantities of data regarding the sequences in the DNA. Data available from sequencing platforms are called sequencing reads. Sequencing reads are text sequences representing DNA segments sampled from some unknown location in the DNA. The reads from one of the most popular NGS platforms available today, Illumina [3], are short, up to a few hundred bases long, compared to the human genome that is around 3 billion bases long. Each base in the DNA is covered by more than one read on average, providing redundant information (Figure 1.1) guarding against errors introduced during the sequencing process. In order to get a better picture of the complete DNA from sequencing reads, additional algorithms are used such as read alignment [4], and read assembly [5]. Genomic analysis usually can proceed only after sequencing reads are processed using tools such as these. It is also possible to perform error-correction on the sequencing reads, by exploiting the inherent redundancy in

the data. In some applications such as read assembly, error-correction can be a very valuable operation.

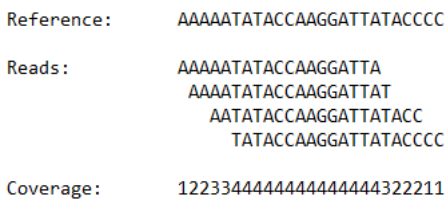

Figure 1.1: Coverage of sequencing reads calculated using the number of reads that are sequenced from a specific base in sequenced DNA

k-mer counting refers to counting the frequencies of all k-length substrings in a collection of sequencing reads, where the alphabet is $\Sigma = \{A, C, G, T\}$. k-mer counting is a fundamental step in many bioinformatic algorithms mentioned before, as many algorithms such as read assembly manipulate sequencing data at the level of k-mers [6]. A specific example is BLESS 2 [7] which is an error-correction tool for NGS data. BLESS 2 uses a k-mer occurrence histogram to determine a threshold for solid or error-free k-mers in sequencing reads, and the remaining k-mers are corrected by the algorithm based on the solid k-mers in the dataset. There are also other tools that use k-mer counting, e.g., MUSCLE [8], Arachne 2 [9], BFC [10] and Mash [11].

The basic idea behind k-mer counting is very simple. A naive method would be akin to building a hash table that maps a k-mer to its count. However, when the number of k-mers becomes very large, such as in the case of the human genome, this approach becomes inefficient. This method is also extremely slow and implementing multi-thread safe locking mechanisms also results in a worse performance than a single-threaded implementation [12].

To make k-mer counting efficient, different algorithms have been developed. Tools like Jellyfish [13] and BFCounter [14] rely on hash tables and Bloom filters. These structures require 10s of GBs of memory. Tools like DSK [15] and KMC2 [16] take a different approach by storing intermediate results on disk instead of keeping everything in memory. This results in a smaller memory footprint for these tools.

With recent drops in sequencing cost, high-coverage sequencing data is becoming abundant. For instance, the genome-in-a-bottle [17] database provides 300x coverage human genome sequences, which results in hundreds of billions of bytes of data. This abundance of data is desirable from the

perspective of gaining new knowledge about the DNA as well as improving the accuracy of analyses. However, as genomics becomes accepted more and more in the clinic, this puts stress on the execution times of computational tools that analyze the data, which need to run faster than before, analyzing higher coverage data from more patients. It then becomes paramount to accelerate core functionalities, such as k-mer counting, that are present in many bioinformatic pipelines.

Modern GPUs provide us with massive computational power, recently achieving 100 Teraflops of performance in Deep Learning applications [18]. While computing platforms such as GPUs are harder to program, and need expert knowledge to leverage performance, it is also attractive to offload core computations in a domain to these devices because they can impact many applications at once. This can be especially true of applications in big-data domains such as bioinformatics, where growing volumes of data necessitate faster execution of applications beyond what conventional computing platforms can facilitate. k-mer counting, as has been discussed, is a functionality leveraged by many tools in bioinformatics, and is hence a fit candidate for acceleration, capable of impacting many tools used in the field.

Motivated by these factors, in this work, we present our efforts in accelerating k-mer counting leveraging the GPU. We chose KMC2 as the algorithm because it is one of the most memory-efficient, and hence suitable for GPU computation. At the same time, there are some challenges posed by the algorithm in moving it to GPU. First, many portions of the code are not easily parallelized. There are other portions in the code that are data parallel and also involve a significant amount of the runtime, but since each data item may take a different amount of time to process, these chapters cause thread divergence, degrading the performance of the GPU. Managing these issues without impacting overall performance is important. Second, there is some I/O overhead in KMC2 in the form of disk-access. Using a GPU, in addition, involves data-transfer overheads through PCIe. Effectively hiding these overheads behind computations is key to extracting performance from KMC2.

To the best of our knowledge, this is the first work that attempts to accelerate KMC2 on the GPU. We achieved a significant speedup over the original CPU implementation while maintaining a small memory footprint. We believe that our implementation may be used by other applications that use

k-mer counting, such as those listed before, to further improve their performances.

The contributions of this work are as follows:

- An analysis of the issues in implementing KMC2 on the GPU architecture.
- The first implementation, to the best of our knowledge, on GPU, achieving accelerated performance over the original CPU version, potentially helping many bioinformatic applications that use k-mer counting.

The rest of the thesis is organized as follows: Background regarding GPU computation and KMC2 is in chapter 2, our implementation of KMC2 on GPU is described in chapter 3 and, finally, results and conclusion are presented in chapter 4 and chapter 5.

CHAPTER 2

BACKGROUND

2.1 GPU Computation

Recent advances in the development of GPUs provide us with unprecedented computational power in these devices. A GPU has thousands of cores to execute parallel work efficiently while a CPU only contains a few cores optimized for latency.

One of the most well-known use-cases for GPUs is machine learning. Data scientists and researchers have been using GPUs for machine learning purposes for a variety of different applications, such as image classification and natural language processing. Various open source machine learning libraries utilizing GPUs have been developed, such as PyTorch [19] and TensorFlow [20]. In addition to applications developed in machine learning, GPUs are also popular in general purpose computing. CUDA [21] is a parallel computing platform that makes the development of general purpose computing on GPU easy and elegant. CUDA provides users with the option of processing serial code on the host side while processing parallel workloads on the device.

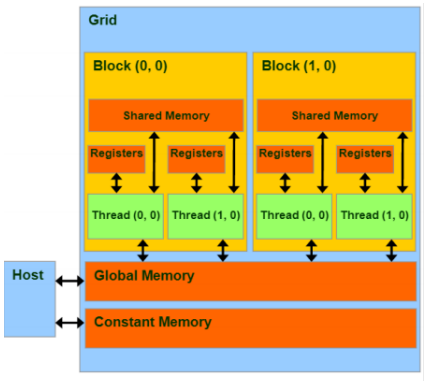

Figure 2.1: CUDA memory model

Figure 2.1 represents the typical memory model in a GPU. There are four memory levels in a GPU: register (R/W 1 cycle), shared memory (R/W 5 cycles), global memory (R/W 500 cycles) and constant memory (R only; 5 cycles). The data transfer between the CPU and GPU is through the PCIe port. GPUs have a limited amount of memory compared to CPUs which could have hundreds of gigabytes of RAM within a single system. Fast access memory like shared memory and constant memory are very limited in size, usually fewer than 100 kB. In addition to the limitation in size, there are additional limitations on shared memory access with regard to which threads may access a block of shared memory.

From the software programmer's view, GPU computation in CUDA is launched in the form of threads, and threads are organized into groups called thread blocks. Threads within a block are provided means to cooperate closely (e.g., access of the same shared memory space). Formally, threads in the same block execute in a SPMD (single program, multiple data) fashion and share memory and synchronization. From the hardware perspective, the scheduling unit in the GPU is called a warp, and each warp has machinery to perform computations of a certain number of threads in single instruction multiple data (SIMD) fashion. This means all threads executing within the same warp execute the same instruction at any clock cycle. It may be noted that for efficiency, the programmer may deem a block of software threads to span an integer multiple of warps in the hardware. A block is a convenient abstraction, allowing the programmer, in many cases, to express the application such that code executing in one block is largely independent of code executing in another, and the dependencies within a block are efficiently handled by the facilities provided by the GPU for threads in a block to cooperate. At the same time, when different threads executing within a warp take different execution paths due to branches, the different execution paths are serialized and the throughput of the warp is reduced. Therefore, GPUs are not suitable for computations with large degrees of such branching among the threads.

Many bioinformatics algorithms require substantial computation effort. In many cases, they are data-parallel due to certain assumptions such as that reads sequenced from a particular site in the DNA may be considered independent. However, there are also some features that discourage parallel execution. Careful consideration of dependencies in the application to de-

termine an appropriate level of abstraction in the code where the execution is parallel and independent, and the mapping of these abstractions to the parallel execution units on the GPU (such as blocks or threads), are important to extract performance from GPU computation in these applications. In addition, the memory required by the code mapped to each parallel execution unit on the GPU (a block or a thread) should not be too high since the GPU trails the CPU significantly in that department, or additional steps may be needed to enable efficient sharing of such resources across threads or even blocks. Various GPU-based bioinformatics algorithms have been developed in recent years, such as read alignment [22], read mapping for RNA sequence [23] and assembly [24].

In the context of k-mer counting, we need to process all the sequencing reads and collect statistics from them. Processing across reads may be largely independent, but there are dependencies within each read. Thus it is very attractive to assign a read to a thread or a block. At the same time, data-structures shared among such units induce external dependencies among them.

In the following chapters, we will discuss the original KMC2 and the methods we follow to implement the algorithm on GPU.

2.2 KMC2

There are two key concepts behind KMC2: signature and super k-mer.

A signature is an idea adopted from minimizers [25]. A signature is lexicographically the smallest m-mer where m is smaller or equal to k. Another constraint for a signature is that it cannot contain substring sequence AA except at the beginning of the signature and it does not start with AAA or ACA. Many sequencing reads have runs of As and a large amount of k-mers would share the same signature if the constraint is not applied. This constraint further reduces disk usage and memory usage.

A super k-mer is formed by consecutive k-mers in a read, which share the same signature.

Figure 2.2 is an example of splitting reads into super k-mer assuming the signature length is 4 and k is 7. First, KMC2 determines the signatures of the k-mers in a read. If consecutive k-mers share the same signature, they

will be merged into a super k-mer.

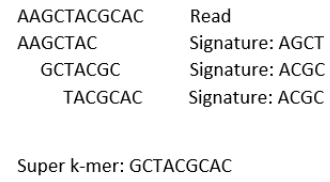

Figure 2.2: Example of signature and super k-mer

KMC2 includes two major stages and one pre-processing stage. The pre-processing stage obtains a well-balanced signature map. The first stage finds all the super k-mers and distributes them across bins. The second stage collects statistics from the super k-mers in the bins.

In the pre-processing stage, KMC2 samples a relatively small number of reads from the dataset and performs a profiling on these reads to get the distribution of signatures in the super k-mers. Based on the profiling result, KMC2 divides signatures into bins by balancing the number of super k-mers stored in each bin. Assuming the sampled reads fairly represent all the reads in the read dataset, this signature map will be well-balanced across the bins for the whole dataset as well.

In the first stage, KMC2 scans through each read to find the super k-mers. Then the super k-mers are distributed across the bins and stored on disk, based on the signatures of the super k-mers and the signature map obtained in the pre-processing stage. The usage of super k-mers rather than k-mers allows reduction in disk footprint. The fact that the bins are balanced helps reduce memory footprint when each bin is opened in memory for further processing in the second stage. Otherwise certain bins may be too large - the largest bin will set the required memory size for the application.

In the second stage, KMC2 processes the bins one at a time. For each bin, KMC2 expands the super k-mers stored in the bin by splitting them into (k, x)-mers. The idea of (k, x)-mers is introduced to reduce the number of strings to be sorted in the next sorting phase of the algorithm. The (k, x)-mer representation utilizes the concept of the canonical representation of k-mers. A k-mer can have two forms, one is directly the k-mer, the other is its reverse complement; whichever is lexicographically smaller is the canonical form of the k-mer. A (k, x)-mer is a (k, x')-mer, $0 \leq x' \leq x$. (k, x)-mers are split from the super k-mers in such a way that the canonical form of a k-mer belongs

to only one (k, x)-mer. Then we sort the expanded (k, x)-mers, collect the k-mer counts in it and write the results to the output file.

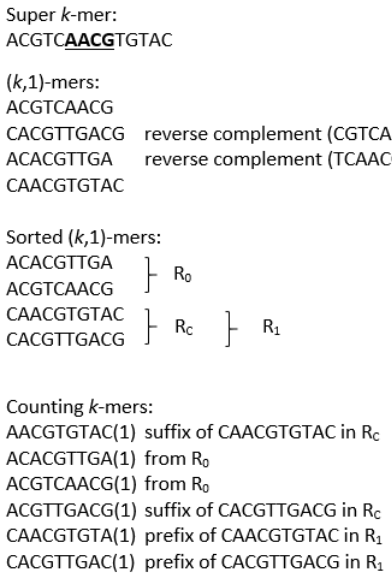

Figure 2.3: Example of how counts are extracted from one (k, x)-mer

The following is an example of how statistics are collected from the sorted (k, x)-mers. Assume x is set to 1. Then the extracted (k, x)-mers are either k-mers or $(k+1)$-mers. There are six possibilities in which we could find the canonical k-mer:

- Some $(k, 1)$-mer with length k
- A suffix of a $(k+1)$-mer preceded by A
- A suffix of a $(k+1)$-mer preceded by C
- A suffix of a $(k+1)$-mer preceded by G
- A suffix of a $(k+1)$-mer preceded by T
- A prefix of a $(k+1)$-mer

KMC2 sorts all the (k, x)-mers and then splits them into six subarrays. R_0 contains k-mers. R_A, R_C, R_G and R_T contain subarrays starting with A, C, G and T, and R_1 contains all the $(k+1)$-mers. KMC2 scans through these six arrays. For R_A, R_C, R_G and R_T, KMC2 uses the k-mer from its suffixes. For R_1, KMC2 uses the k-mer from its prefixes. Then KMC2 finds the next

lexicographically smallest k-mer from these six subarrays. If it is the same as the most recently added k-mer, KMC2 increases the counter associated with that k-mer. Otherwise, KMC2 adds this k-mer to the resulting array. In this fashion, KMC2 collects counts of all k-mers in the sorted arrays by going through all the k-mers in lexicographical order.

Figure 2.3 shows how k-mer counts are collected from a super k-mer. In this example, k is 9, x is 1 and the signature of this super k-mer is AACG. Recollect that a $(k, 1)$-mer is a $(k + 1)$-mer or a k-mer such that every k-mer in the $(k, 1)$-mer is in its canonical form. If such a $(k + 1)$-mer cannot be found for a part of the super k-mer, then simply a k-mer is used as the $(k, 1)$-mer. For instance, the first $(k, 1)$-mer is ACGTCAACG instead of ACGTCAACGT because the canonical form of the second k-mer, CGTCAACGT, is its reverse complement. This means the first two k-mers cannot be merged to form a single $(k, 1)$-mer. The canonical k-mers of the second and third k-mer are both their reverse complements. Therefore, the second $(k, 1)$-mer is the reverse complement of CGTCAACGTG. All the rest of the $(k, 1)$-mers are found in such a fashion. After all the $(k, 1)$-mers are found in the super k-mer, they will be sorted in lexicographical order and divided into six sub-arrays as mentioned above. In this example, we only have three sub-arrays because we do not have $(k + 1)$-mer preceded by A, G or T. To collect the results from the sorted $(k, 1)$-mers, we looked at the k-mers from R_0, suffixes from R_C and prefixes from R_1. The counts of all the k-mers are collected in such a manner.

CHAPTER 3

APPROACH

3.1 Analysis of KMC2

Before we started to implement KMC2 on GPU, we did an analysis of KMC2 to find out the most time-consuming portions of KMC2 using a single thread with k-mer length set to 44. Splitter is the function used to extract super k-mers from the reads in the first stage. Expander is the function used to find the (k, x)-mers from the super k-mers in the second stage. Sorter is the function used to sort the (k, x)-mers in the second stage.

Table 3.1: k-mer counting profiling result

Total run time	420 s
First stage	134 s
- splitter	129 s
Second stage	286 s
- expander	50 s
- sorter	217 s

Table 3.1 summarizes the results of this analysis. Not mentioned in this table are the functions reader, writer and collector. The reader reads input files, and the writer dumps the results of the first stage onto disk. The collector processes the final part of the 2nd stage and writes the final results to disk. As may be seen here, in the first stage, the majority of time is spent on the splitters. In the second stage, the most time-consuming function is the sorter. The expander takes some time to process, but it is still much shorter compared to the sorter. Both the splitter and the sorter, the most time-consuming parts in KMC2, are parallelizable. This analysis shows that it is plausible to implement KMC2 on GPU.

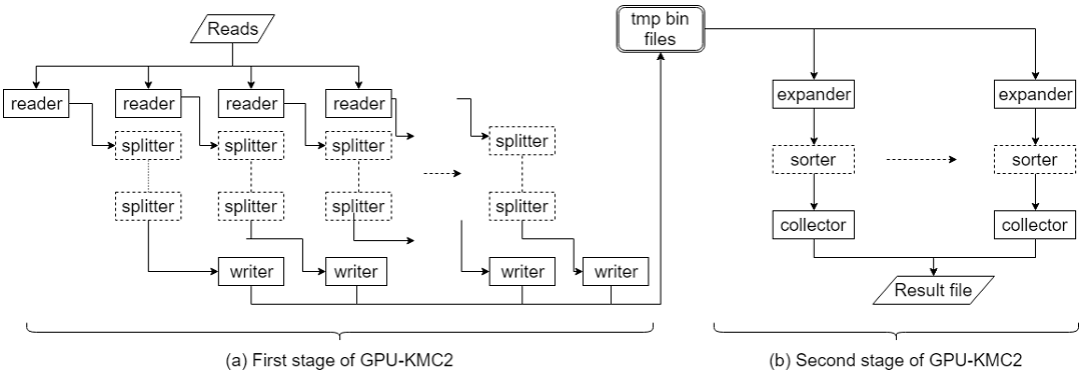

Figure 3.1: Overview of GPU accelerated KMC2. Rectangles with dotted lines are processes that are parallelized using CUDA.

3.2 Pre-processing Stage

In the pre-processing stage, a small number of reads are copied to the device side to calculate the signature map. Each thread is responsible for collecting statistics for one read. The signature map is later copied to the constant memory on the device side. The signature map will be accessed multiple times by every thread in the first stage and will not be changed later. Therefore, moving it to the constant memory reduces memory access latency and saves global memory on the GPU.

3.3 First Stage

Figure 3.1 (a) shows our implementation of the first stage. In the first stage, the host will read the sequencing reads from the input file and store them in a buffer. When enough reads are stored in the buffer, it will pass the sequencing reads to the GPU side for processing. Each GPU thread is a splitter responsible for one sequencing read in the read buffer. The splitter splits super k-mers from the read it processes and stores them in a return buffer in global memory on the device. The return buffer is organized into different bins mirroring the super k-mer bins that will be stored on the disk. After the return buffer is copied back to the host, it will be dumped into files representing super k-mer bins on disk. One CPU thread is used in the first stage to read sequencing reads from input file and write super k-mer bins on disk.

We have two options to implement the return buffer on the GPU side. We could have a global buffer shared by all threads, or each thread could have its own buffer. Using the global buffer is more memory efficient but it requires a lock (for synchronization among the threads) for every bin in the buffer. Another way to implement the return buffer is to give each thread its own buffer space, which means locks may be avoided. After every thread has finished its read, the return buffer is merged together before being copied back to the host. The drawback of this approach is that if we have 512 bins, we need to allocate enough space for every bin for each thread since we do not know the distribution of signatures within each sequencing read beforehand. Moreover, we need 512 counters for each thread to keep track of the number of super k-mers in each bin. This results in a large memory overhead and most bins will be empty in this case. Due to the limitation of global memory on GPUs, this approach also limits the number of reads we could process during each batch. Hence, the latter method turns out to be worse due to the large memory overhead it introduces.

Hence, we prefer the first method, which requires locks. Implementing locks and critical chapters in CUDA is tricky. A naive way to implement locks in CUDA would be the following:

```
__device__ void critical_chapter(int &lock)
{
    while(atomicCAS(&lock, 0, 1) != 0)
    {
                ⋮
          critical chapter
                ⋮
    }
    atomicExch(&lock, 0);
}
```

The idea is that a thread grabs the lock, finishes the critical chapter and releases the lock afterwards. However, this design could result in dead locks. If two threads within the same warp try to acquire the lock to the same bin, one of them will grab the lock but it will never pass the loop to get the chance to release the lock. This warp will be stuck in the while loop forever because this warp will keep executing the while loop until the second thread gets the

lock, but it never gets the chance to get the lock since the thread which has the lock does not get the chance to release it.

The following is the actual implementation for critical chapter used in this work:

```
__device__ void critical_chapter(int &lock)
{
    volatile bool leaveLoop = false;

    while(!leaveLoop)
    {
        if(atomicCAS(&lock, 0, 1) == 0)
        {
                    ⋮
            critical chapter
                    ⋮
            leaveLoop = true;
            atomicExch(&lock, 0);
        }
    }
}
```

The keyword "volatile" is important in CUDA 8.0 and above to avoid dead locks. Our implementation has 512 bins and has 512 locks within global memory to synchronize writes to bins from all the threads.

To minimize the running time of the first stage, we overlapped the host and device computations. The readers and writers (which could not be efficiently parallelized on the GPU) constitute the operations performed by the host. When the kernels for splitters are running on the device side, the host first writes the result of the previous batch to the disk and then prepares the reads for the next batch. Figure 3.2 shows the collaboration between GPU and CPU in the first stage.

Though the data movement between host and device is fast, it is still not negligible in our implementation. All kernels in CUDA launch in a stream. When no stream is specified by the user, a single default stream will be used. Figure 3.3 (a) shows an example of kernel launch without specifying a stream. The kernel launch has to wait until all the sequencing reads for the

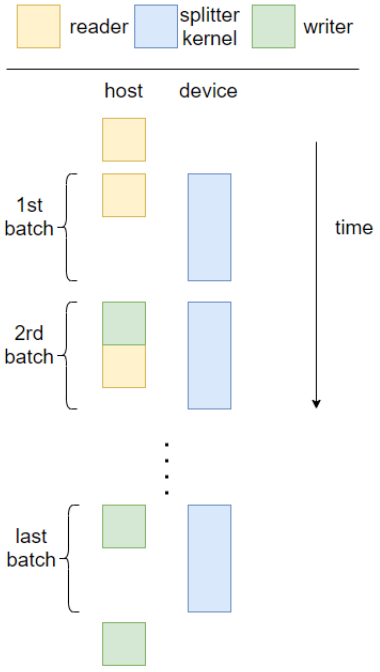

Figure 3.2: Collaboration between host and device side in the first stage

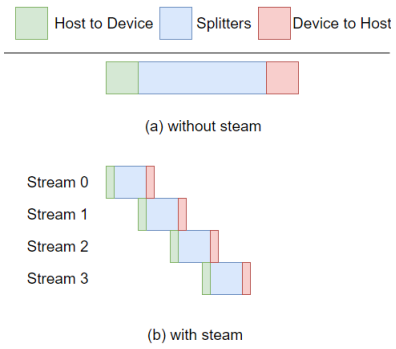

Figure 3.3: Speedup using CUDA streams

current batch are copied from host to device. After the splitter kernels have finished, the next batch cannot begin immediately and has to wait for the results from the previous batch to be copied from the device to the host. This results in a higher latency due to computations waiting for data movement. However, by specifying the number of streams to use (we used four streams in our implementation), we can hide some of the latency of data movement by overlapping data transfers with computations. Figure 3.3 (b) shows an example of a kernel launch with four different streams. By splitting into four different streams, we basically divided the splitter kernel into four smaller sub-kernels in each batch. Each sub-kernel would process one fourth of the sequencing reads in the current batch. This means the first sub-kernel can launch after a fourth of the data is transferred, overlapping the computations of the first sub-kernel with transfers for the second sub-kernel, and so on.

We may observe the challenges we mentioned before in action here. For instance, each thread may take a different amount of time to process its assigned read. This can cause divergence among the threads in the splitter kernels. However, analyzing the execution times of the implementation, we determined that the impact on performance due to thread divergence is not significant because the host functions take about the same amount of time to process as the kernels. We may also note the importance of pipelining the CPU and GPU tasks (Figure 3.2), as well as the use of streams (Figure 3.3) in minimizing the impact of I/O and PCIe overheads on overall performance.

3.4 Second Stage

There are three main steps in the second stage. The first one is the expander; it is used to read the bins from the disk and pre-process the super k-mers. The second step is a sorter to sort the result from the first step. The third step is a collector to collect counts from the sorted results and store them in the result file. Figure 3.1 (b) shows the implementation of the second stage.

The expander reads the super k-mers stored on disk and extracts the (k, x)-mers from them for the sorters to process them later. To implement the expander on the device side, we would need to first scan through the bin file to find out all the super k-mers in each bin. Each thread in the device is responsible for extracting the (k, x)-mers from one super k-mer. However,

experiments showed this approach to be slower than directly running the expander on the host side. According to our understanding, this is caused by the overhead of scanning of the bin file, transferring it to the device and thread divergence in the kernels. The super k-mers vary in length and the positions of the (k, x)-mers differ in these super k-mers. This causes divergence among the threads while extracting the (k, x)-mers from the super k-mers on the device side. Therefore, the expander is implemented on the host side with one thread for each bin file.

The sorter uses radix sort which can be easily parallelized on GPU. We used radix sort implemented in Thrust [26] provided by NVIDIA to perform the sorting on GPU.

The collectors count the number of occurrences of each k-mer and store the result on disk. To complete this process, each collector has to scan through the sorted array from the sorters and increase the counters corresponding to the k-mer counts accordingly. The collector has to operate in serial order following the order returned by the sorter, which forces serialization. Therefore, the collectors are implemented on the host side.

Overall, the second stage is organized as follows. While it is not attractive to parallelize the expander and collector within each bin on the GPU, it is possible to parallelize them across bins on the host side using OpenMP threads. The overall organization is shown in Figure 3.4. Each OpenMP thread launches the expander, a sorter kernel and the collector. While the expander and collector instances in the different OpenMP threads may operate independently, the sorter instances can only execute one after the other, because they need the GPU resource, and only one GPU is available. This also limits the amount of performance gain from the OpenMP parallelization.

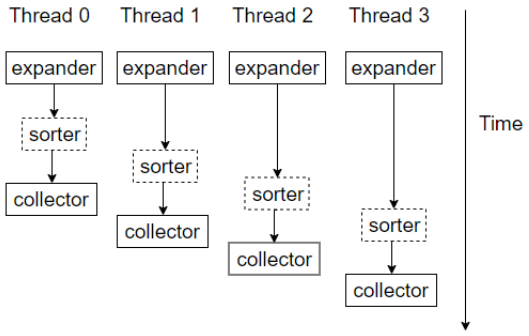

Figure 3.4: Multi-threading in second stage

CHAPTER 4

RESULTS

In order to evaluate the performance of the GPU acceleration for KMC2, we tested our implementation against the original KMC2 [16] which is implemented purely for CPU execution.

All experiments were done using Xeon E5-2603 v2 CPU and GeForce GTX 1080 Ti. For input data we used 40x coverage reads of length 100bp, simulated from the Mouse Chromosome Y reference (accession number: NC_000087.7) using the tool pIRS [27]. The data and additional details regarding it are available at https://github.com/gowthami19m/SPECTACLE. The results are summarized in Table 4.1 and Table 4.2.

Compared to the original KMC2 implementation [16] running on a single CPU thread, our implementation that uses one GPU achieved an overall speedup of 4.03x when using a single CPU thread and 5.88x when using four CPU threads for k=40. These accelerations are, respectively, 3.48x and 4.6x for $k = 50$. Compared to the original KMC2 implementation [16] running on 4 CPU threads, our implementation using a single GPU and 4 CPU threads achieves a speedup of 2x in the best case (k=40).

We also compared our implementation of KMC2 to two other popular k-mer counting tools, JellyFish2 and DSK for k=40. When the JellyFish2 and DSK implementations use 1 CPU thread, our implementation using 1 GPU and 1 CPU thread achieves 7.24x improvement over JellyFish2 and 5.85x over DSK. When all three tools use 4 CPU threads, our implementation achieves 4.32x speedup over JellyFish2 and 3.03x speedup over DSK.

From Tables 4.1 and 4.2, one may see the performance gains in the first and second stages individually. The performance gains from the first stage come mainly from accelerating the splitter which is parallelized on the GPU. As presented in Table 3.1, the splitter is the dominant function in the first stage. In the CPU implementation of KMC2, the splitter is parallelized across multiple CPU threads. While the serial performance of each CPU

thread is superior to that of a GPU thread, the GPU can launch many more threads than a CPU can. There is some thread divergence in our GPU implementation as we discussed before, but the performance of the device-side functionality currently matches that of the host-side functionality, constituted by the readers and writers, so this is not a detriment to overall performance. Also, pipelining the host-side functionality with the device-side functionality allows us to hide some overheads such as disk access in the first stage.

Table 4.1: k-mer counting results, baseline is KMC2 with one thread

k-mer length=40	First Stage		Second Stage		Total	
	time	speedup	time	speedup	time	speedup
KMC2 (1 thread)	127	1.0x	307	1.0x	435	1.0x
KMC2 (4 threads)	62	2.05x	85	3.61x	148	2.94x
GPU KMC2 (1 thread)	23	5.52x	85	3.61x	108	4.03x
GPU KMC2 (4 threads)	23	5.52x	51	6.02x	74	5.88x
DSK (1 thread)	N/A	N/A	N/A	N/A	866	0.50x
DSK (4 threads)	N/A	N/A	N/A	N/A	224	1.94x
JellyFish2 (1 thread)	N/A	N/A	N/A	N/A	1072	0.41x
JellyFish2 (4 threads)	N/A	N/A	N/A	N/A	320	1.36x

Table 4.2: k-mer counting results, baseline is KMC2 with one thread

k-mer length=50	First Stage		Second Stage		Total	
	time	speedup	time	speedup	time	speedup
KMC2 (1 thread)	101	1.0x	298	1.0x	400	1.0x
KMC2 (4 threads)	42	2.40x	80	3.73x	123	3.25x
GPU KMC2 (1 thread)	29	3.48x	86	3.47x	115	3.48x
GPU KMC2 (4 threads)	29	3.48x	58	5.16x	87	4.60x

The most time-consuming functionalities in the second stage are the expander and the sorter, as presented in Table 3.1. We parallelized the sorter on the GPU and the implementation using Thrust provides significant performance gains. In addition, we used OpenMP parallelization for the second stage, similar to the software version of KMC2, which allows the expander to process multiple bins in parallel, even though the OpenMP threads would have to wait for their turn to use the GPU for executing the sorter afterwards.

Note that the speedup in the first stage is the same regardless of the number of CPU threads we used because we are not using OpenMP in the first stage. The speedup we achieved in the second stage using four CPU threads is insignificant compared to the implementation using one CPU thread due to the overhead mentioned in the previous chapter.

CHAPTER 5

CONCLUSION

k-mer counting is an important bioinformatic function present in many algorithms that are used to extract information from raw sequencing reads. With increasing coverage and more data, the tools generally need to be faster. It is hence valuable to accelerate common core functionalities such as k-mer counting that can impact many such tools. We analyzed KMC2, one of the most popular k-mer counters, and determined that there are parallelizable chapters in it. We accelerated those functions on GPU, while dealing with challenges such as thread divergence, and disk and data-transfer overheads. Our method performs faster than KMC2 as well as a few other competing tools, which are implemented on CPU. We believe that our implementation, which is open-sourced, will help improve the performance of many tools that use k-mer counting.

At the same time, we will be looking to further improve the performance of our method in future work. For instance, it may be possible to launch multiple kernels when more than one GPU is available on the system. A detailed analysis of load-balancing among the host and the GPU devices may be performed in this case to optimize the overall performance.

CHAPTER 6

REFERENCES

[1] S. Clancy and W. Brown, "Translation: DNA to mRNA to protein," *Nature Education*, p. 101, 2008.

[2] L. Hoopes, "Introduction to the gene expression and regulation topic room," *Nature Education*, p. 160, 2016.

[3] "Illumina sequencing platforms." [Online]. Available: https://www.illumina.com/systems.html

[4] H. Li and R. Durbin, "Fast and accurate short read alignment with burrows-wheeler transform," *Bioinformatics*, pp. 1754–1760, 2009.

[5] D. Zerbino and E. Birney, "Velvet: Algorithms for de novo short read assembly using de Bruijn graphs," *Genome Res*, p. 821829, 2008.

[6] P. Pevzner, H. Tang et al., "An Eulerian path approach to DNA fragment assembly," *PNAS*, pp. 9748–9753, 2001.

[7] Y. Heo et al., "Bless 2: accurate, memory-efficient and fast error correction method," *Bioinformatics*, p. 146, 2016.

[8] S. Deorowicz, M. Kokot et al., "Kmc 2: Fast and resource-frugal k-mer counting," *Bioinformatics*, pp. 1–21, 2014.

[9] D. Jaffe et al., "Whole-genome sequence assembly for mammalian genomes: Arachne 2," *Genome Res.*, pp. 91–96, 2003.

[10] H. Li, "BFC: correcting illumina sequencing errors," *Bioinformatics*, pp. 2885–2887, 2015.

[11] B. Ondov, T. Treangen et al., "Mash: fast genome and metagenome distance estimation using minhash," *Genome Biology*, p. 132, 2016.

[12] M. Michael and M. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," *Proceedings of PODC '96*, 1996.

[13] G. Marcais and C. Kingsford, "A fast, lock-free approach for efficient parallel counting of occurrences of k-mers," *Bioinformatics*, pp. 764–770, 2011.

[14] P. Melsted and J. Pritchard, "Efficient counting of k-mers in DNA sequences using a Bloom filter," *BMC Bioinformatics*, p. 333, 2011.

[15] G. Rizk, D. Lavenier et al., "DSK: k-mer counting with very low memory usage," *BMC Bioinformatics*, pp. 652–653, 2013.

[16] R. Edgar, "Muscle: multiple sequence alignment with high accuracy and high throughput," *Nucleic Acids Res*, pp. 1792–1797, 2004.

[17] "Genome-in-a-bottle." [Online]. Available: www.genomeinabottle.org

[18] "NVIDIA TESLA V100." [Online]. Available: https://www.nvidia.com/en-us/data-center/tesla-v100/

[19] A. Paszke, S. Gross et al., "Automatic differentiation in PyTorch," 2017. [Online]. Available: https://pytorch.org/

[20] M. Abadi et al., "Tensorflow: Large-scale machine learning on heterogeneous systems," 2015. [Online]. Available: tensorflow.org

[21] J. Nickolls, I. Buck et al., "Scalable parallel programming," 2008. [Online]. Available: https://developer.nvidia.com/cuda-zone

[22] R. Wilton, T. Budavari et al., "Arioc: high-throughput read alignment with GPU-accelerated exploration of the seed-and-extend search space," *PeerJ*, p. 808, 2015.

[23] I. Media, J. Tarraga et al., "Highly sensitive and ultrafast read mapping for rna-seq analysis," *DNA Research*, pp. 93–100, 2016.

[24] D. Li, C. Liu et al., "Megahit: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de Bruijn graph," *Bioinformatics*, pp. 1674–1676, 2015.

[25] M. Roberts, W. Hayes et al., "Reducing storage requirements for biological sequence comparison," *Bioinformatics*, pp. 3363–3369, 2004.

[26] H. Jared and B. Nathan, "Thrust," 2015. [Online]. Available: Thrust:https://thrust.github.io/

[27] X. Hu, J. Yuan et al., "pIRS: Profile-based Illumina pair-end reads simulator," *Bioinformatics*, pp. 1533–1535, 2012.