Bioinformatics
doi.10.1093/bioinformatics/xxxxxx
Advance Access Publication Date: Day Month Year
Application Note

Genome analysis

GPU accelerated KMC2

Huiren Li [1,*], Anand Ramachandran [1] and Deming Chen [1,*]

[1] Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA

Associate Editor: XXXXXXX

Received on XXXXX; revised on XXXXX; accepted on XXXXX

Abstract

Motivation: K-mer counting is a popular pre-processing step in many bioinforamtic algorithms. KMC2 is one of the most popular tools for k-mer counting. We want to exploit the computation power on this existing algorithm.
Results: We achieved 3.7x speedup using one GPU with one CPU thread and 5.4x speedup using one GPU with four CPU threads.
Availability: Freely available at url...
Contact: dchen@illinois.edu
Supplementary information: Supplementary data are available at *Bioinformatics* online.

1 Introduction

K-mer counting refers to counting the frequencies of all k-long strings in a collection of sequencing reads. K-mer counting is a fundamental step for many bioinformatic algorithms, such as BLESS 2 (Heo *et al*., 2016).

The idea behind k-mer counting is very simple. The most naive way is to count k-mers using brutal force, such as building a local histogram to count the frequencies of all k-mers. This method could work if we have thousands of genome reads.However, in real life, we would have millions of genome reads to process within a batch of job. It will take up lots of memory if you do it in the naive way. Therefore, k-mer counting is a time-consuming and memory-intense process.

Different k-mer counting algorithms have been developed to facilitate this process. KMC2 (Deorowicz *et al*., 2014) is a popular k-mer counting tool among them and it is also used in BLESS2. KMC2 is a memory frugal and fast k-mer counting tool while some other k-mer counting tools would use tens of gigabytes of memory. Due to the limitations of graphic card's video memory, it is not plausible to implement an algorithm that is too memory intense. The memory frugalness of KMC2 is the most important reason why we chose to implement this particular algorithm on GPU.

Even with tools like KMC2, k-mer counting would still take a substantial amount of time to process. The original KMC2 was implemented purely using CPU. We wanted to exploit the computation power of GPU and further optimize the running time of KMC2. Therefore, we developed a GPU implementation of KMC2. We achieved a substantial speedup over the original CPU implementation while maintaining the memory frugalness of KMC2.

2 Approach

There are two key concepts behind KMC2: signature and super k-mer(Deorowicz *et al*., 2014).

A signature is a lexicographically smallest m-mer where m is smaller or equal to k. Another constraint for a signature is that is can't contain substring sequence AA and it doesn't start with AAA or ACA(Deorowicz *et al*., 2014). The idea of signature is introduced to balance the number of super k-mers stored in each bin in the first stage. And it is used to extract super k-mers from the sequencing reads and distribute them to their corresponding bins.

A super k-mer is formed by consecutive k-mers which share the same signature. When we collect results in the first stage, instead of storing all k-mers separately, we store the super k-mers to save disk usage.

KMC2 is a disk based k-mer counting algorithms. It includes two major stages. The first stage is to extract super k-mers from the sequencing reads and distribute them to their corresponding bins on the disk. We have 512 bins in the first stage. The second stage is to process the bins produced during the first stage and put the statistics into result files.

3 Methods

GPU accelerated KMC2 is parallelized using CUDA and openmp progamming. Right now, we are only using a single GPU to perform the computation and it could be further accelerated using multiple GPU's.

The overall algorithm is shown in Figure 1. In the first stage, the host will read the sequencing reads from the fastq file and store it in a buffer. When enough reads are stored in the buffer, it will pass the sequencing reads to the device side for processing. Each GPU thread is a splitter responsible for one sequencing read. The splitters will extract the super k-mers from the reads store them in the temporary bin files.

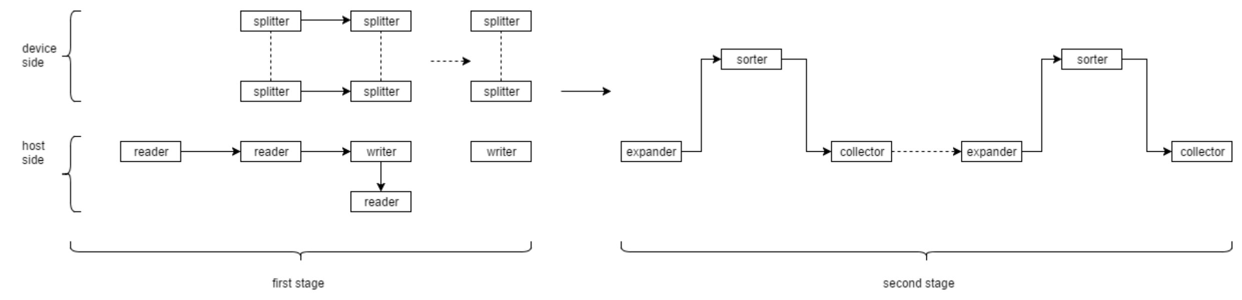

Fig. 1. Overview of GPU acclerated KMC2

On the device side, we have two options to implement the buffer for the bins. 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 every bin in the buffer. Although giving each thread its own buffer space avoids the pain of dealing with locks in CUDA, it limits the number of reads we could process during each batch. The later method turns out to be worse because of the large memory overhead it introduces. Moreover, since we could process fewer reads during each batch of work, we have to call cudaMemcpy more times which also introduces a larger overhead.

To minimize the running time of the first stage, we maximized the overlap between host and device computation. As shown in Figure 1, when the kernels for splitters are running on the device side, the host first writes the result of the previous batch to the disk if it exits and then prepares the reads for the next batch. Therefore, the device will never sit idle to wait for its data except for the first batch of work.

The other technique we used is dividing the kernels into steams. K-mer counting is a memory intense process. This means the time spent on copying data between host and device is not negligible. Currently, the kernels are divided into four steams to overlay the kernel executions and cudaMemcpy's to hide the latency.

For the second stage, we used openmp together with thrust library. And thrust is a high performance parallel algorithm library for GPU.

There are three major steps in the second stage. The first one is expander, it is used to 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 statistics from the sorted results and store it in the result file.

The expander and collector are serialized work. This makes their performance on the GPU not ideal. Therefore, instead of moving the computation to the device side, the expander and collector are implemented on the host side. The sorter uses radix sort which can be easily parallelized on GPU. We used radix sort implemented in thrust library to perform the sorting.

However, simply using the thrust library to parallelize the sorters didn't give us a satisfactory speedup. This is caused by the serialization between the expander, sorter and collector. The GPU remains idle when it is waiting for the data for the expander. And it could not begin to process the data from the next bin until the previous bin's collector finishes his job.

Luckily, there is one more level of parallelism to explore in KMC2. Since each bin is independent from each other, we can use openmp to process multiple bins at the same time. In our experimentation, we used four openmp threads to process four bins simultaneously. The drawback of using openmp together with thrust library is that only one sorting kernel can be launched at a time.

By using the computation power of GPU, we achieved a substantial speedup over the single thread version while maintaining the memory frugalness of the original KMC2(Deorowicz *et al.*, 2014).

4 Results and Conclusion

In order to evaluate the performance of the GPU implementation of KMC2, we tested our implementation against the original KMC2 (Deorowicz *et al.*, 2014).

All experiments are done using Xeon E5-2603 v2 CPU and GeForce GTX 770.

Table 1. K-mer counting results

	first stage(s)	second stage(s)	total running time(s)
1 thread CPU	127	307	435
4 thread CPU	62	85	148
GPU with 1 CPU thread	20	97	117
GPU with 4 CPU thread	20	60	80

The testing is done using mouse genomes read length equals to 100. And we count all k-mers with length 40. The results are summarized in Table 1.

Overall, the GPU implementation of KMC2 achieved a 3.7x speedup over single thread CPU version while using one CPU thread and achieved a 5.4x speedup while using four CPU threads.

One thing to notice here is that we achieved a 6.35x speedup in the first stage disregard the number of CPU threads we are using because we are not using openmp in the first stage. Moreover, the speedup we got using GPU with 4 CPU threads isn't very significant. The reason is that the sorting stage isn't parallelized by openmp. Even multiple CPU threads are running at the time, only one sorter kernel could be launched. There is a large overhead is this process due to the serialized cudaMemcpy's and kernel launched. Therefore, it wouldn't be able to achieve a better performance.

References

Heo,Y., Ramachandran,A., Hwu,W., Ma,J., Chen,D. (2016) BLESS 2: accurate, memory-efficient and fast error correction method, *Bioinformatics*, **146**.

Deorowicz,S.,Kokot,M.,Grabowski,S.,Debudaj-Grabysz,S. (2014) KMC 2: Fast and resource-frugal k-mer counting, *Bioinformatics*, **00**, 1-21.