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

OXFORD

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. In this work, we leverage the computation power of 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.
Results: We achieved 3.7x speedup using one GTX 770 with one CPU (Xeon E5-2603) thread and 5.4x speedup using one GPU with four CPU threads over a single thread CPU.
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) and Gerbil (Marius *et al*., 2017).

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. Only, in real life, we would have millions of genome reads to process. With a naive algorithm, k-mer counting can take a large amount of memory and take a long time to run.

Different k-mer counting algorithms have been developed. KMC2 (Deorowicz *et al*., 2014) is a one of the most popular k-mer counting tool. KMC2 is designed to be memory frugal and fast while some other k-mer counting tools would use tens of gigabytes of memoryï¼Œ like Jellyfish (Marcais *et al*., 2011) and BFCounter (Melsted *et al*., 2011). 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. In this work, we would like to exploit the computation power of GPU and further optimize the running time of KMC2. We developed an effective GPU implementation of KMC2 for acceleration because GPU usually does not have a very large memory. We achieved a substantial speedup over the original CPU implementation while maintaining a small memory footprint.

2 Approach

There are two key concepts behind KMC2: signature and super k-mer. A signature is a lexicographically smallest m-mer where m is smaller or equal to k. Another constraint for a signature is that it cannot contain substring sequence AA and it does not start with AAA or ACA. The idea of signature is introduced to balance the number of super k-mers stored in each bin 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.

KMC2 is a disk based k-mer counting algorithms. It includes two major stages. There is also a pre-processing stage to calculate the signature map for distribution of super k-mers.

The first stage is to extract super k-mers from the sequencing reads and distribute them to their corresponding bins on the disk according the signature map. Super k-mers are used in this stage instead of naive k-mers to reduce disk footprint. In our implementation, we have a total of 512 bins in the first stage.

In the second stage, we process the bins one at a time. For each bin, we expand the super k-mers stored in the bin. Then we sort the expanded k-mers, collect the statistics of it and store the result.

3 Methods

Our GPU accelerated KMC2 is parallelized using CUDA and OpenMP. Currently, we are only using a single GPU to perform the computation and it could be further accelerated using multiple GPU.

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

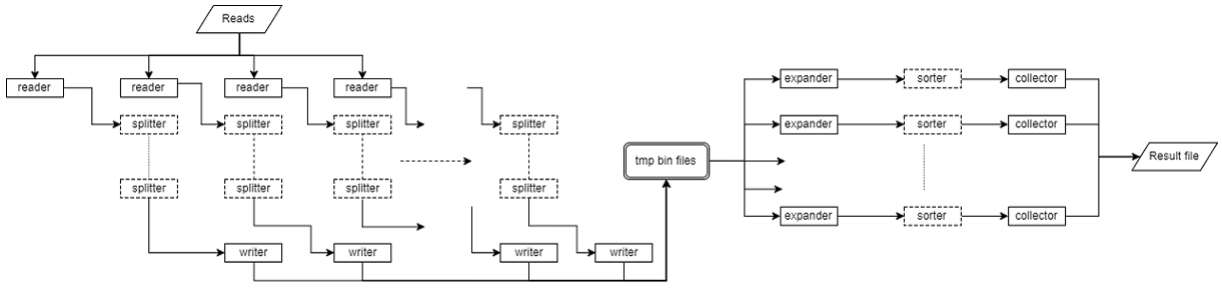

Fig. 1. Overview of GPU acclerated KMC2. Rectangles with dotted lines are processes that are parallelized using CUDA.

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 and store them in the buffer in gloabl memory on the device. After the result is copied back to the host, it will be dumped into the temporary bin files.

There are two key challenges in exporting KMC2 to GPU. The first limitation is that even though KMC2 is a memory frugal algorithm, it still requires a considerable amount of memory. We have to identify the limitations of GPU computation and utilize all the memory we have in GPU. The second challenge is that reading fastq file is time consuming and it is essential to hide the latency of file IO in our implementation.

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. Dealing with locks on the device side among different versions of CUDA is challenging but it could be resolved by the keyword "volatile". 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 would 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 that batch 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 streams. 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 streams to overlay the kernel executions and cudaMemcpy's to hide the latency. If we are only using a single stream, at the beginning of each batch, we have to wait for cudaMemcpy to copy all the reads from host to device. At the end of each batch, we have to wait for cudaMemcpy to copy all the results from device to host. The kernel would be idle during this process. By dividing the kernel into four streams, we only pays for one fourth of the overhead because each cudaMemcpy needs to copy one fourth of the previous workload. After the first cudaMemcpyHostToDevice finishes, the kernel for the first stream can be launched and cudaMemcpyHostToDeice for the second stream can be launched concurrently.

For the second stage, we used OpenMP together with Thrust. 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 non-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. Therefore, we used radix sort implemented in Thrust to perform the sorting.

However, simply using the Thrust 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 its job.

However, 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 is that only one sorting kernel can be launched at a time because Thrust doesn't allow concurrent kernel launching.

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.

4 Results and Conclusion

In order to evaluate the performance of the GPU acceleration for 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 whose 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.

Note that we achieved a 6.35x speedup in the first stage disregard the number of CPU threads we used because we are not using OpenMP in the first stage. Moreover, the speedup we got using GPU with 4 CPU threads is not very significant. The reason is that the sorting stage is not 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 prevented further performance gain.

References

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

Marius, E., Steffen, R., Matthias, M. (2017) Gerbil: a fast and memory-efficient k-mer counter with GPU-support, *Algorithms for Molecular Biology*.

Marcais,G., Kingsford,C. (2011) A fast, lock-free approach for effi- cient parallel counting of occurrences of k-mers, *Bioinformatics*, **146**, 764-770.

Melsted,P., Pritchard,J.K. (2011) Efficient counting of k-mers in DNA sequences using a bloom filter. *BMC Bioinformatics*, **12**, 333.

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