An Efficient GPU-based de Bruijn Graph Construction Algorithm for Micro-Assembly

Ren, Shanshan; Ahmed, Nauman; Bertels, Koen; Al-Ars, Zaid

DOI
10.1109/BIBE.2018.00020

Publication date
2018

Document Version
Accepted author manuscript

Published in
Proceedings - 2018 IEEE 18th annual IEEE International Conference on BioInformatics and BioEngineering (BIBE 2018)

An Efficient GPU-based de Bruijn Graph Construction Algorithm for Micro-Assembly

Shanshan Ren Nauman Ahmed Koen Bertels Zaid Al-Ars
Quantum & Computer Engineering Dept.
Delft University of Technology, 2628 CD Delft, The Netherlands
{s.ren, n.ahmed, k.l.m.bertels, z.al-ars}@tudelft.nl

***Abstract*—In order to improve the accuracy of indel detection, micro-assembly is used in multiple variant callers, such as the GATK HaplotypeCaller to reassemble reads in a specific region of the genome. Assembly is a computationally intensive process that causes runtime bottlenecks. In this paper, we propose a GPU-based de Bruijn graph construction algorithm for micro-assembly in the GATK HaplotypeCaller to improve its performance. Various synthetic datasets are used to compare the performance of the GPU-based de Bruijn graph construction implementation with the software-only baseline, which achieves a speedup of up to 3x. An experiment using two human genome datasets is used to evaluate the performance shows a speedup of up to 2.66x.**

***Index Terms*—GPU acceleration; de Bruijn graph construction; micro-assembly; repeat k-mers**

I. Introduction

Alignment-based variant discovery is a widely used approach to identify variants in genomic data. This approach first aligns the sequencing dataset of a sample genome to a reference genome using alignment tools. It then compares the aligned dataset to the reference genome and extracts the genome positions where the sample genome differs from the reference genome using variant callers. The variants found through this approach include single nucleotide variations (snv's), small insertions/deletions (indels) and structural variations (svs). However, reads with indels are easily misaligned during the alignment step, leading to low accuracy of indel detection.

In order to improve the accuracy of variant detection of indels in particular, various local assembly based variant callers are proposed to correct the misalignment errors of alignment-based variant discovery approach, such as Scalpel [1], Platypus [2] and GATK HaplotypeCaller (HC) [3], [4]. In local assembly based variant callers, reads aligned to a certain region of the reference genome are assembled into a long DNA sequence covering this region. This process is referred to as micro-assembly or local assembly. Assembly based variant callers not only improve the accuracy of indel detection, but also enhance the accuracy of snv detection by making use of linkage disequilibrium between nearby variants.

One of the challenges of genome assembly is repeats in the genome. In most of local assembly based variant callers, a popular method to handle repeats is to avoid cycles in the graph [5], used in Scalpel, GATK HC, and ABRA. If cycles are detected in the graph, the region is reassembled using higher k-mer sizes until there are no cycles in the graph. GATK HC, which is widely used in many large-scale sequencing project, takes more measures to handle repeats. In GATK HC, k-mers are classified into two groups: unique k-mers and repeat k-mers. If a k-mer occurs twice or more than twice in a read, it is a repeat k-mer; otherwise, it is a unique k-mer. During graph construction, unique k-mers are collapsed into single nodes, while repeat k-mers are not collapsed into single nodes. In this way, some cycles in the graph caused by repeats are avoided, which reduces the probability of reassembly with larger k-mer sizes.

Existing assembly algorithms used by many genome assemblers use de Bruijn graphs (DBGs) construction methods. However, since previous efforts to accelerate DBG construction on GPU (such as [6], [7] and [8]) collapse repeat k-mers into single nodes, these methods are not suitable for DBG construction of micro-assembly in GATK HC. In this paper, we propose a novel GPU-based algorithm for DBG construction for micro-assembly in GATK HC.

The rest of this paper is organized as follows. Section II describes the algorithm of DBG construction in GATK HC. Section III presents the proposed algorithm of DBG construction. Section IV presents and discusses the experimental results. Finally, Section V concludes this paper.

II. DBG construciton in GATK HC

The DBG construction for micro-assembly in GATK HC is divided into two main steps.

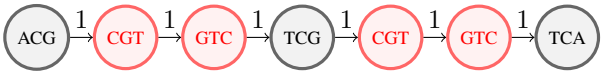
In step 1, each read aligned to a region is decomposed into multiple k-mers and each k-mer is checked to find repeat k-mers. A region is identified based on significant evidence of variation, which is referred to as active region.

In step 2, each read is then considered as a candidate to create new nodes, create new edges and increase edge weights. For a read, the first unique k-mer is identified ignoring the repeat k-mers before it. From this unique k-mer, the k-mers in this read and the overlap relationships between k-mers are used to construct the de Bruijn graph. This construction is performed as follows: (a) If the node mapped by this unique k-mer has not been created, a new node mapped by this unique k-mer is created. Otherwise, the node mapped by this unique k-mer is identified. Let the node obtained be Node A. (b) The program then checks whether Node A has an edge, which connects Node A and the node mapped by the next k-mer. If this edge is found, the weight of the edge is increased. Otherwise, the program checks whether the next k-mer is a unique k-mer or a repeat k-mer. If it is a unique k-mer, the program finds or creates a node mapped by the next k-mer. If it is a repeat k-mer, a node mapped by the next k-mer is

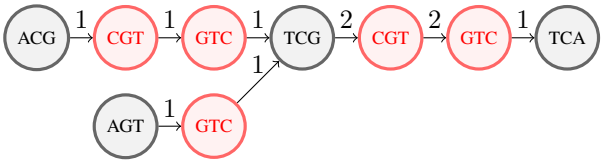
Read 1: ACGTCGTCA k-mers: ACG, CGT, GTC, TCG, CGT, GTC, TCA

Read 2: AGTCGTC k-mers: AGT, GTC, TCG, CGT, GTC

(a) Classification of k-mers



(b) DBG of Read 1



(c) DBG of Read 1 and Read 2

Fig. 1. Illustration of a simple DBG construction in GATK HC

created. An edge connecting this node and the node mapped by the next k-mer is then created and the weight is set to 1. In this step, the node mapped by the next k-mer is obtained. (c) Let the node obtained in step (b) be Node A and repeat step (b) until there are no k-mers in the read.

Fig. 1 shows a simple example of DBG construction in GATK HC for two reads using a k-mer size of 3. The two reads are first decomposed into multiple k-mers, which are then classified into unique k-mers and repeat k-mers. The repeat k-mers are marked in red. The two reads are then taken in turn to construct the graph. In the graph, black nodes and red nodes represent nodes mapped by unique k-mers and repeat k-mers, respectively. Edges represent overlap relationships between k-mers and the numbers above these edges represent occurrences of the overlap relationships, referred to as edge weights. As shown by Fig. 1, the unique k-mers with the same value are mapped to single nodes in the graph, such as "TCG'. However, the repeat k-mers with the same value are mapped to multiple nodes in the graph, such as "CGT" and "GTC". Moreover, the repeat k-mers with the same value from different reads may be collapsed into single node. For example, the second "CGT" in Read 1 and "CGT" in read 2 are mapped to one node. This is because the node mapped by the second "CGT" in Read 1 is created in Fig. 1 (b) and when Read 2 are taken to construct a graph, there is already an edge connecting the node mapped by "TCG" and the node mapped by "CGT" in the graph. As explained in the workflow, the node mapped by a repeat k-mer is created when there is no edge connecting Node A and the node mapped by the repeat k-mer in the graph. Thus, the repeat k-mer "CGT" in Read 2 do not create new nodes and the weight of the edge is increased to two. This example indicates that the creation of the node mapped by a repeat k-mer depends on the early computation results.

III. GPU-BASED DBG CONSTRUCTION

A. Algorithm idea

There are two kinds of nodes in the graph: unique nodes, which are nodes mapped by unique k-mers, and repeat nodes, which are nodes mapped by repeat k-mers. Hence, there are four kinds of edges in the graph: U-U, U-R, R-R and R-U. U-U edge stands for an edge that starts from a unique node and ends with a unique node, etc.

DBG construction can be divided into two parts. One part is to create repeat nodes and U-R, R-R and R-U edges, while the other part is to create unique nodes and U-U edges. The former can be calculated by handling the special subsequences of reads using the method in GATK HC. The special subsequences are defined as two kinds of subsequences: (a) the subsequence having at least three k-mers, among which the first and last k-mers are unique k-mers and the other k-mers are repeat k-mers, and (b) the subsequence having at least two k-mers, among which the first k-mer is a unique k-mer and the other k-mers are repeat k-mers and the last repeat k-mer is the last k-mer of the read where the subsequence comes from. The later is similar to the common way of DBG construction, where the same k-mers are collapsed into single nodes. One of the most popular acceleration methods of the common way of DBG construction is to calculate the occurrences of (k+1)-mers in parallel.

Thus, in this paper we propose the following GPU-based algorithm for DBG construction. First, we assume there are no repeat k-mers and calculate the occurrences of (k+1)-mers in parallel on the GPU. We then check whether there are repeat k-mers. If there are no repeat k-mers, (k+1)-mers and their occurrences are used to construct the graph. Otherwise, (k+1)-mers having one or two repeat k-mers are deleted and the special subsequences are identified. The remaining (k+1)-mers and their occurrences and the special subsequences are then used to construct the graph.

B. Workflow of GPU-based algorithm

Fig. 2 shows the workflow of the GPU-based DBG construction algorithm. The input data are the reads aligned to multiple active regions. The output data are the de Bruijn graphs stored using ReadThreadingGraph in GATK HC.

The DBG construction algorithm is implemented through a C program, a CUDA program and a Java program together. There are in total twelve steps. Since GATK HC is a Java-based program, the input data are transferred to the C program through JNI (Java Native Interface) and then transferred to GPU. On the GPU, the 64-bit values of k-mers and (k+1)-mers are generated. The 64-bit values of (k+1)-mers are processed to obtain their occurrences, while the 64-bit values of k-mers are handled to calculate the number of k-mers in each active region after reducing the repeat k-mers in each sequence, which is used to check whether there are repeat k-mers in each active region. The computation results on the GPU are then transferred back to the host. For each active region, if there are repeat k-mers in the active region, the repeat k-mers are found and the special subsequences are identified. Moreover, the 64-bit values of (k+1)-mers having repeat k-mers and their occurrence are deleted. The remaining 64-bit values of (k+1)-mers are transferred to GPU, then transformed into (k+1)-mers and transferred back to the host. The computation results of the C program are transferred to the Java program and used to construct the graphs.

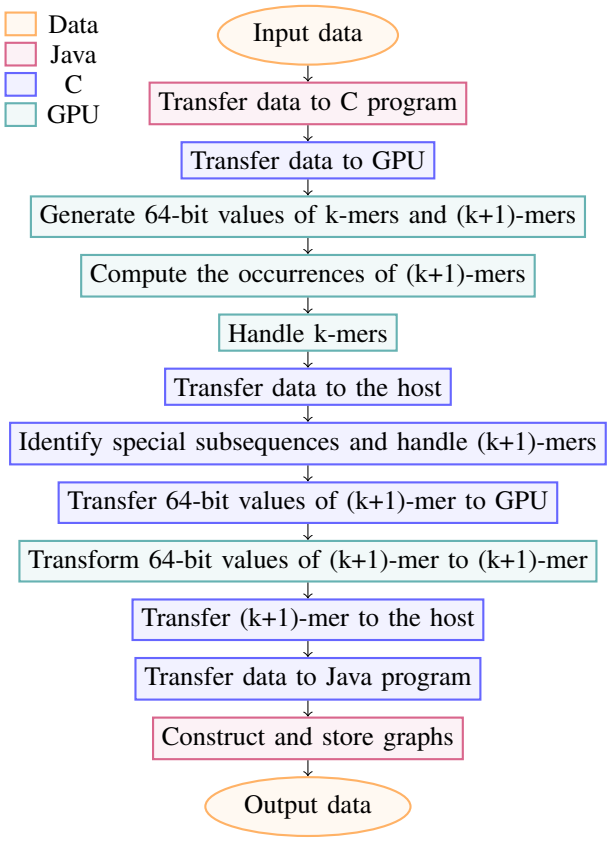


Fig. 2. Workflow of GPU-based DBG construction algorithm

C. Generate 64-bit values of k-mers and (k+1)-mers

This step is implemented on the GPU. Each thread block takes charge of one read to generate 64-bit values of its k-mers and (k+1)-mers, which is shown in Algorithm 1. Every 4 characters of the read are loaded by one thread and stored in the shared memory ($read\_s[]$). Every character of the read is transformed into one byte, which is 0, 1, 2, or 3, and then stored in the shared memory ($Rbyte[]$) by one thread. Finally, each thread performs bitwise operations to generate the 64-bit values of one k-mer and one (k+1)-mer and stores the 64-bit values in the global memory ($kmer\_64[]$ and $kmer\_add\_64[]$). Since reads are from multiple active regions, two arrays ($kmer\_active[]$ and $kmer\_add\_active[]$) are used to store the active region id of each k-mer and (k+1)-mer, respectively. Moreover, one more array ($kmer\_seq\_id[]$) is used to store the sequence id of each k-mer.

D. Compute the occurrences of (k+1)-mers

This step is implemented on the GPU. Functions from the Thrust library [9] are employed to compute the occurrences of (k+1)-mers. Since (k+1)-mers are from multiple active regions, the 64-bit value array of (k+1)-mers and the active region id array of (k+1)-mers are handled together. This step is implemented by three operations, which is shown with an example in Fig. 3.

(1) The elements in the 64-bit value array of (k+1)-mers are sorted in ascending order. In the meanwhile, the elements in the active region id array change their placements in correspondence to the sorting operations in the 64-bit value array of (k+1)-mers. This operation is implemented by the $sort\_by\_key()$ function.

(2) The elements in the active region id array are sorted in ascending order, leading to corresponding changes of element placements in the 64-bit value array of (k+1)-mers. This operation is implemented by the $stable\_sort\_by\_key()$ function.

(3) A constant array is used to store the current occurrence of each (k+1)-mer and the value of each element in this array is 1. The $Reduce\_by\_key()$ function is employed to calculate the number of the consecutive equal elements in the 64-bit value array of (k+1)-mers. The reduced results of the 64-bit value array of (k+1)-mers are stored in a new array and the occurrences of (k+1)-mers are stored in another new array.

The new 64-bit value array of (k+1)-mers and the occur-

Algorithm 1 Generating k-mers and (k+1)-mers

```
1: function GENERATE(read[], length,k_size active_id, seq_id,
   kmer_64[], kmer_active[], kmer_seq_id[], kmer_add_64[],
   kmer_add_active[])
2:     h ← (length + 4 − 1)/4
3:     t ← (h + blockDim.x − 1)/blockDim.x
4:     for i ← 0, t − 1 do                          ▷ Load and store a read
5:         j ← threadIdx.x + i ∗ blockDim.x
6:         if j < h then
7:             a ← read[j]                          ▷ type of a is char4
8:             read_s[j × 4] ← a.x
9:             read_s[j × 4 + 1] ← a.y
10:            read_s[j × 4 + 2] ← a.z
11:            read_s[j × 4 + 3] ← a.w
12:    end for
13:    __syncthreads()
14:    t ← (length + blockDim.x − 1)/blockDim.x
15:    for i ← 0, t − 1 do                          ▷ Transform into bytes
16:        j ← threadIdx.x + i ∗ blockDim.x
17:        if j < length then
18:            b ← read_s[j]
19:            c ← (b == 'A')?0 : ((b == 'C')?1 : ((b == 'G')?2 : 3))
20:            Rbyte[j] = c
21:    end for
22:    __syncthreads()
23:    kmer_number ← length − k_size + 1
24:    t ← (kmer_number + blockDim.x − 1)/blockDim.x
25:    for i ← 0, t − 1 do                          ▷ Generate 64-bit values
26:        j ← threadIdx.x + i ∗ blockDim.x
27:        val ← 0                                  ▷ type of val is uint64_t
28:        if j < kmer_number then
29:            for f ← 0, k_size − 1 do
30:                val ← (val << 2)|(Rbyte[j + f]&3)
31:            end for
32:            kmer_64[j] ← val
33:            kmer_active[j] ← active_id
34:            kmer_seq_id[j] ← seq_id
35:            if j! = kmer_number − 1 then
36:                val ← (val << 2)|(Rbyte[j + k_size]&3
37:                kmer_add_64[j] ← val
38:                kmer_add_active[j] ← active_id
39:    end for
40: end function
```

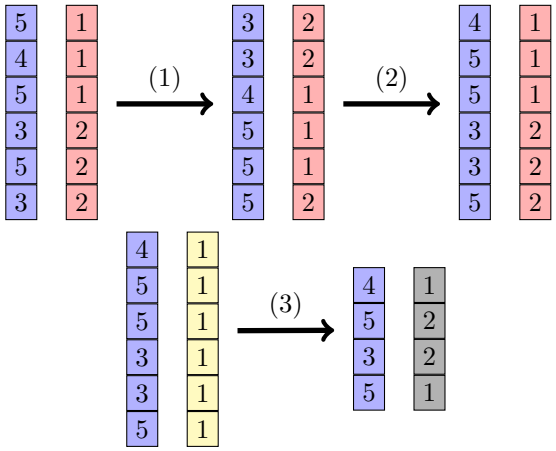


Fig. 3. An example of calculating the occurrences of (k+1)-mers. Purple array stores the 64-bit values of (k+1)-mers. Pink array stores the active region id of each (k+1)-mer. Grey array stores the occurrences of (k+1)-mers. Yellow array is a constant array.

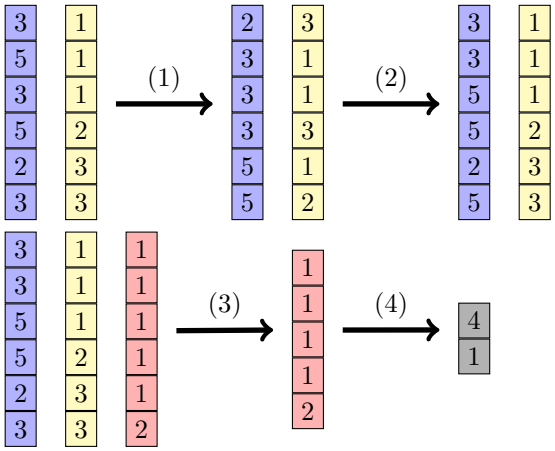


Fig. 4. An example of handling k-mers. Purple array stores the 64-bit values of k-mers. Yellow and pink array stores the sequence id and the active region id of each k-mer, respectively. Grey array stores the number of k-mers in each active region after reducing repeat k-mer in each sequence.

rence array of (k+1)-mers are the computation results of this step and will be used in the following steps.

E. Handle k-mers

This step is implemented on the GPU and also employs functions from the Thrust library. The 64-bit value array of k-mers, the sequence id array of k-mers and the active region id array of k-mers are handled together. This step is implemented by four operations, which is shown with an example in Fig. 4. The first two operations (1) and (2) are similar to the first two operations in Section III-D except the active region id array of (k+1)-mers is replaced by the sequence id array of k-mers. After the first two operations, the 64-bit values of the equal k-mers from the same sequence are consecutive in the 64-bit value array of k-mers.

The third operation (3) is to reduce the consecutive equal elements in the 64-bit value array of k-mers and the sequence id array is used to make sure only the consecutive equal elements from the same sequence are reduced, which is implemented by the $Reduce\_by\_key()$ function. The result of this operation is a new array, which stores the active region ids of the remaining k-mers.

The fourth operation (4) is to calculate the number of k-mers in each active region after reducing repeat k-mers in each sequence, which is implemented by the $Reduce\_by\_key()$ function. The result is stored in a new array, the length of which is equal to the number of active regions.

The array storing the number of k-mers in each active region after reducing repeat k-mers in each sequence is the computation result of this step and will be used in the following steps.

F. Identify special subsequences and handling (k+1)-mers

This step is implemented by a C program, which is shown in Algorithm 2. For each active region, if the number of k-mers is equal to the number of k-mers calculated in Section III-E, there are no repeat k-mers in the region. Otherwise, there are repeat k-mers in the region. Each read is then checked to find repeat k-mers. After all the repeat k-mers in the region are found, each read is checked again to identify the special subsequences.

Since operation (3) in Section III-D does not make sure that only the consecutive equal 64-bit values of (k+1)-mers from the same active region are reduced, extra operations are taken to remedy this, which are from line 11 to line 26 in Algorithm 2. For each active region, if the sum of the weights of existing (k+1)-mers of the region is smaller than the number of (k+1)-mers in the region, a (k+1)-mer has a chance to be added to the region and the weight of the (k+1)-mer added to the region is calculated from line 14 to line 19. If the 64-bit value of the (k+1)-mer has repeat k-mers, the (k+1)-mer will not be added to the region and neither will the calculated weight of the (k+1)-mer.

G. Transform 64-bit values of (k+1)-mer into (k+1)-mer

This step is implemented on the GPU, where each thread takes charge of one (k+1)-mer. Each thread first loads the 64-bit value of a (k+1)-mer into its registers and then transform every 2 bits to a character, which is then stored in the shared memory. After transformation, characters calculated by each thread in one thread block are stored in the consecutive addresses. Finally, each four characters composing a char4 value are stored in the global memory by one thread. In this way, the global memory accesses of the threads in a warp (32 consecutive threads) to store characters are coalesced. The computation result of this step is an array storing the characters of all (k+1)-mers.

H. Construct and store graphs

This step is implemented by a Java program. For each active region, (k+1)-mers are used to create unique nodes and U-U edges and occurrences of (k+1)-mers are the weights of U-U edges. If the active region has repeat k-mers, the special subsequences are handled using the method in GATK HC to create repeat nodes and U-R, R-R and R-U edges and increase the weights of these edges. All the nodes, edges and weights of these edges are stored using ReadThreadingGraph.

Algorithm 2 Identifying special subsequences and handling (k+1)-mers

```
1: function IDENTIFY(region_number, k_region[], k_add_region[], k_region_GPU[], kmer_add_64[], k_add_number[], new_64, new_number[])
2:     t ← 0, p ← 0
3:     for i ← 1, region_number do
4:         if k_region[i] == k_region_GPU[i] then
5:             for reads in the region do
6:                 find_repeat_kmer()
7:             end for
8:             for reads in the region do
9:                 find_subsequence()
10:            end for
11:        cur ← 0
12:        while cur < k_add_region[i] do
13:            cur+ = k_add_number[t]
14:            if cur > k_add_region[i] then
15:                sub ← cur − k_add_region[i]
16:                n ← k_add_number[t] − sub
17:                k_add_number[t] ← sub
18:            else
19:                n ← k_add_number[t]
20:            if Not_have_repeat(kmer_add_64[t]) then
21:                new_64[p] ← k_add_64[t]
22:                new_number[p] ← n
23:                p + +
24:            if cur <= k_add_region[i] then
25:                t ← t + 1
26:        end while
27:    end for
28: end function
```

IV. Results and Discussion

A. Experimental setup

We compare the GPU-based DBG construction implementation and DBG construction implementation in GATK HC (CPU benchmark) with both synthetic and real datasets. The CPU implementation is achieved by modifying GATK HC 3.7 to read input datasets, construct DBGs and output graphs.

The input datasets are reads of multiple active regions. In order to get the synthetic input datasets, we first use Wgsim [10] to generate reads from a reference sequence, which is chromosome 19 of the human genome (UCSC hg19), then follow the GATK best practices pipeline to get the input datasets for GATK HC and use GATK HC to produce reads of multiple active regions. As to the real datasets, the processing steps are the same as for the synthetic datasets except that we use reads produced by an NGS platform instead of the reads simulated by Wgsim.

The output datasets are DBGs of multiple active regions. A simple program is designed to sort the nodes and edges in each graph in order to compare the output and assure the correctness of the results.

A server-class machine is used to perform all the experiments. This machine has two Intel Xeon processors, each of which has 14 two-way hyper-threaded cores running at 2.4 GHz, 192 GB of RAM, and an NVIDIA Tesla K40 card, which consists of 2880 cores running at 745 MHz.

B. Impact of coverage on performance

We generated 8 synthetic datasets using Wgsim with different levels of read coverage, ranging from 10x to 80x. For all these synthetic reads, we used the Wgsim parameters of 100bp for the read length, 0.01 for the mutation rate, and 0.15 for the indel fraction. The k-mer size is 10.

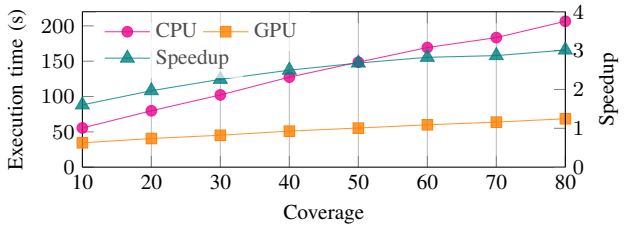


Fig. 5. Execution time and speedup vs coverage of the GPU and CPU implementations with the synthetic datasets

Fig. 5 shows that the execution time and speedup of the GPU and CPU implementations with respect to coverage of the synthetic datasets. The figure indicates that the execution time of both implementations increases with increasing coverage. However, the execution time of the CPU implementation increases faster than the GPU-based implementation. Thus, the speedup of the GPU-based DBG construction implementation increases with increasing coverage. When the coverage of the synthetic dataset is 80x, the speedup is the highest at 3x.

The execution time of the GPU-based implementation is divided into four parts: computation time on the GPU (including data transfer to/from GPU), computation time of the C program, computation time of the Java program and data transfer time using JNI. Fig. 6 shows the percentage of the four parts of the execution time of the GPU-based implementation with the synthetic datasets for different coverage levels. The computation time on the GPU occupies a large part of the total execution time, which increases when the coverage increases. This is because when the coverage increases, the number of reads aligned to each active region increase as well, which increases the number of k-mers and (k+1)-mers and in turn increases the computation time on the GPU.

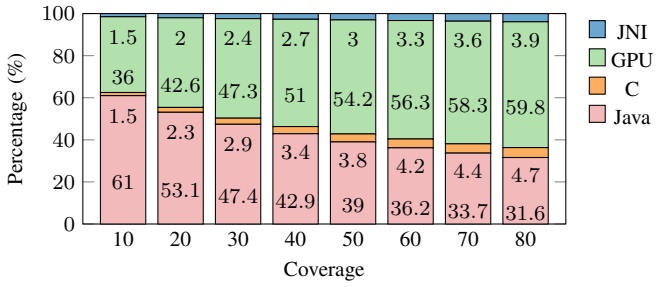


Fig. 6. Percentage of the four parts of the execution time of the GPU-based implementation with the synthetic datasets for different coverage levels

For the GPU-based implementation, the weights of edges are calculated in parallel on the GPU by calculating the occurrences of (k+1)-mers; while for the CPU implementation, the weights of edges are serially increased. Thus, the different methods of handling weights of edges make the execution time growth rates of the two implementations different when the coverage increases.

C. Impact of mutation rate on performance

We used a total of 16 synthetic datasets divided into 4 groups according to their coverage (20x, 40x, 60x and 80x). Each group consists of 4 synthetic datasets with a different mutation rate: 0.05%, 0.1%, 0.5% and 1%. The other parameters of Wgsim for each dataset are kept the same: read length is 100bp, indel fraction is 0.15. The k-mer size is 10.

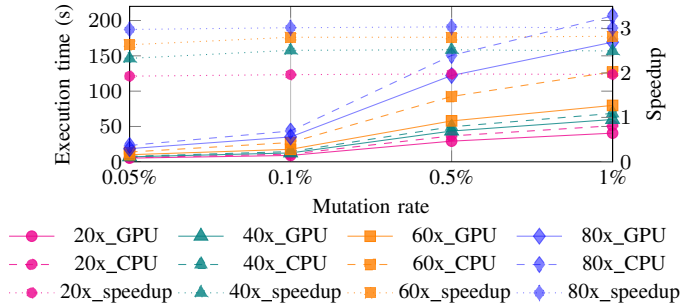


Fig. 7. Execution time and speedup of the two implementations with the synthetic datasets of different mutation rate

Fig. 7 shows the execution time and speedup of the two implementations with all 16 synthetic datasets. The execution time of the two implementations of different coverage increases when the mutation rate increases. This is because when the mutation rate increases, the number of regions which are chosen based on the significant evidence of variation increases and in turn the input data handled by the two implementations increases. In addition, Fig. 7 shows that the speedup of the GPU-based implementation does not change much when the mutation rate increases. To explain this behavior, we take the synthetic datasets with coverage 80x as an example. For this coverage level, Fig. 8 shows that the percentage of the computation time on the GPU changes a little when the mutation rate increases, resulting in little change of the speedup brought by GPU acceleration. In addition, the percentage of the computation time of the C program slightly increases when the mutation rate increases. This is because the number of different k-mers and different (k+1)-mers increase as well when the mutation rate increases.

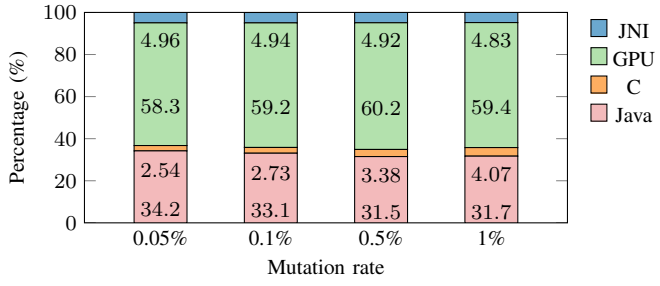


Fig. 8. Percentage of the four parts of the execution time of the GPU-based implementation with different mutation rates for coverage 80x

D. Performance with real datasets

We use 2 real datasets to compare the performance of the GPU and CPU implementations for a k-mer size of 10. The first dataset is chromosome 20 of NA12878 downloaded from the GATK resource bundle, coverage of which is ∼64x. The second dataset is chromosome 17 of G15512.HCC1954.1, coverage of which is ∼58x.

Table I shows the speedup of the two implementations with respect to the real datasets. The speedup of the first dataset is 2.66x and the speedup of the second dataset is 2.47x.

TABLE I
PERFORMANCE COMPARISON FOR REAL DATASETS

Dataset	CPU time (s)	GPU time (s)	Speedup
1	72.9	27.4	2.66x
2	53.8	21.7	2.47x

V. CONCLUSIONS

Micro-assembly is a widely used technique to increase the accuracy of variant callers, such as the popular GATK HC. This paper proposes a GPU-based DBG construction algorithm for micro-assembly in GATK HC. The proposed algorithm assumes that there are no repeat k-mers in the dataset and calculates the occurrences of (k+1)-mers in parallel on the GPU, thereby achieving high speedup. Then the dataset is inspected for repeat k-mers, and only these repeats are re-evaluated on the CPU. Experimental results show that the speedup of our implementation compared with the CPU benchmark implementation for synthetic datasets is up to 3x, while the speedup achieved for real human genome datasets can reach 2.66x.

REFERENCES

[1] H. Fang, E. A. Bergmann, K. Arora, V. Vacic, M. C. Zody, I. Iossifov, J. A. O'Rawe, Y. Wu, L. T. Jimenez Barron, J. Rosenbaum, M. Ronemus, Y. H. Lee, Z. Wang, E. Dikoglu, V. Jobanputra, G. J. Lyon, M. Wigler, M. C. Schatz, and G. Narzisi. Indel variant analysis of short-read sequencing data with Scalpel. *Nat Protoc*, 11(12):2529–2548, Dec 2016.

[2] A. Rimmer, H. Phan, I. Mathieson, Z. Iqbal, S.R.F. Twigg, WGS500 Consortium, A.O.M. Wilkie, G. McVean, and G. Lunter. Integrating mapping-, assembly- and haplotype-based approaches for calling variants in clinical sequencing applications. *Nat Genet*, 46(8):912–918, 08 2014.

[3] G. A. Van der Auwera, M. O. Carneiro, C. Hartl, R. Poplin, G. Del Angel, A. Levy-Moonshine, et al. From FastQ data to high confidence variant calls: the Genome Analysis Toolkit best practices pipeline. *Curr Protoc Bioinformatics*, 43:1–33, 2013.

[4] S. Ren, K.L.M. Bertels, and Z. Al-Ars. Efficient acceleration of the pair-hmms forward algorithm for gatk haplotypecaller on gpus. *Evolutionary Bioinformatics*, 14, March 2018.

[5] G. Narzisi and M. C. Schatz. The challenge of small-scale repeats for indel discovery. *Front Bioeng Biotechnol*, 3:8, 2015.

[6] Mian Lu, Qiong Luo, Bingqiang Wang, Junkai Wu, and Jiuxin Zhao. *GPU-Accelerated Bidirected De Bruijn Graph Construction for Genome Assembly*, pages 51–62. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[7] S. Qiu and Q. Luo. Parallelizing big de bruijn graph construction on heterogeneous processors. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1431–1441, June 2017.

[8] D. Li, C. M. Liu, R. Luo, K. Sadakane, and T. W. Lam. MEGAHIT: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de Bruijn graph. *Bioinformatics*, 31(10):1674–1676, May 2015.

[9] Thrust: A productivity-oriented library for cuda. In *GPU Computing Gems, Jade Edition*, pages 359 – 371. Morgan Kaufmann, Boston, 2012.

[10] Wgsim. https://github.com/lh3/wgsim. Accessed January 28, 2018.